Fine-tuning et RAG

EMSI - Université Côte d'Azur
Richard Grin
Version 1.3 - 12/12/24

1

Plan du support

2

Principaux problèmes des LLMs

- Ils ne disent jamais qu'ils ne peuvent répondre à une question et ils préfèrent halluciner
- Ils n'ont aucune connaissance sur les données privées des entreprises car ils n'y ont pas eu accès pendant leur apprentissage
- Leurs connaissances s'arrêtent à la date de fin de leur apprentissage
- Ils ne peuvent donc pas être utilisés tels quels dans une application d'entreprise, sans améliorations ni garde-fou

3

Pour atténuer les problèmes

- Prompt engineering (déjà étudié)
- Fine-tuning qui modifie le LLM (change ses paramètres)
- RAG qui ne modifie pas le LLM mais qui, pour chaque question, recherche des informations supplémentaires pertinentes pour y répondre, et les ajoute au prompt pour que le LLM en prenne connaissance

4

Fine-tuning

5

Fine-tuning (réglage fin)

- Entrainement complémentaire d'un modèle d'IA pré-entraîné (OpenAi, Gemini,...), sur de nouvelles données (tâche spécifique, nouveau domaine, ...)
  - Le pré-entrainement avait permis d'apprendre les structures de langage, la syntaxe, la sémantique et des connaissances générales
  - Le fine-tuning modifie les paramètres du modèle ; le nouveau modèle a des connaissances dans un domaine particulier, sur une tâche spécifique, ou pour converser avec un style ou un ton défini

6

Entrainement pour fine-tuning

- On fournit à l'API des paires de texte (entrée et sortie attendue) pour entrainer le modèle
- Chaque LLM a son propre format pour les données d'entrainement
- Pour avoir un impact significatif sur le modèle de base pré-entrainé, il faut l'entrainer sur au moins plusieurs centaines d'exemples (pour une tâche très spécifique) à plusieurs milliers d'exemples ou davantage
- Comme pendant le 1er apprentissage, on peut paramétrer avec des hyperparamètres : nombre d'itérations (d'époques), taux d'apprentissage, taille des batchs, etc.

7

Exemple de format

- Format JSONL (un objet JSON par ligne du fichier)

8

Stratégie pour l'entrainement

- On entraine avec les valeurs des hyperparamètres par défaut ; à la fin on regarde si le modèle généré est correct ; si ça n'est pas le cas, on recommence avec d'autres hyperparamètres
- Il faut juger le résultat sur des jeux de tests :
  - Combien de bons mots sont générés ?
  - Lorsque le bon mot n'est pas généré, est-ce que l'erreur est importante ?
  - Les outils ou services qui effectuent le fine-tuning fournissent des métriques pour aider à juger (suivi de la fonction de perte en particulier)

9

Coûts

- L'entrainement du fine-tuning est bien plus coûteux que l'utilisation du modèle ; il nécessite des machines puissantes et beaucoup de temps
- Les données utilisées pendant l'entrainement sont longues et complexes à créer, avec intervention humaine
- Le fine-tuning est de loin la solution la plus coûteuse pour modifier le comportement du modèle

10

Options pour le fine-tuning

- Le fine-tuning étant long et très coûteux, des options ont vu le jour
- Par exemple
  - Limiter les paramètres qui sont modifiés, et utiliser moins de données d'entrainement (T-few)
  - Seulement ajouter de nouveaux paramètres (LORA)
  - Injecter des embeddings (pas du texte) dans les entrées du modèle pour guider son comportement ; les embeddings injectés sont « appris » par entrainement (Soft-prompting)

11

Pré-entrainement continu

- Autre solution que le fine-tuning qui modifie tous les paramètres du modèle, mais sans fournir au modèle un ensemble de couples question-réponse qu'il doit apprendre, comme on le fait avec le fine-tuning
- Fournit au LLM des nouvelles données, comme dans le pré-entrainement (pas des couples question-réponse élaboré avec intervention humaine), évidemment orientées vers les nouvelles connaissances ou comportement à acquérir
- Moins coûteux en temps et argent mais souvent moins efficace que le fine-tuning

12

Retrieval-Augmented Generation (RAG)

- Bases
- Techniques avancées
- Comparaison prompt engineering, fine-tuning, RAG

13

Bases

14

Présentation du RAG

- Pour répondre à une question, combine
  - la génération de texte par un LLM
  - la recherche d'informations externes au LLM (dans bases de données, ensemble de documents, …)
- Les informations retrouvées améliorent l'exactitude et la pertinence des réponses du LLM ; elles permettent aussi d'indiquer les sources d'information utilisées
- Ces informations peuvent être contenues dans des documents d'entreprise, de haute expertise dans un domaine, ou des documents récents sur lesquels le LLM n'a pas été entrainé

15

Sources d'information externes

- Corpus de documents
- Bases de données (BD)
- Bases de connaissances
- Pages Web
- API ou moteurs de recherche (API d'encyclopédies, Google, Bing)

16

- Dans la suite on appellera
  - « document » une source d'information externe, qui pourra être un document entier ou, le plus souvent, un « chunk », un morceau de document, obtenu par découpage en morceaux d'un document
  - « question » une question posée à un LLM avec, éventuellement, l'historique de la conversation

17

Exemples d'utilisation du RAG

- Interroger le LLM sur des fichiers PDF qui contiennent les règles de fonctionnement d'une entreprise
- Résumer un document ou une page Web
- Interroger le LLM sur une vidéo YouTube
- Service client intelligent ; chatbot pour répondre aux clients de l'entreprise
- Assistance juridique en s'appuyant sur des lois, des règlements et la jurisprudence
- Aide à l'apprentissage ou à la recherche dans un domaine particulier
- Support technique avancé avec manuels techniques

18

Types de RAG

- Utilisation de mots-clés pour identifier les documents les plus pertinents pour répondre à une question
- Basé sur les embeddings, avec recherche de similarité entre la question et les documents
- Hybride ; combine les recherches par mots-clés et par similarité
- Avancé ; étape initiale d'un des types précédents, suivie de techniques pour améliorer les résultats ou la rapidité des traitements

19

2 grandes phases pour le RAG

1. Ingestion des documents dans le système de RAG : nettoyage des documents, découpage en morceaux, enregistrement
2. Le système génère la réponse en utilisant la question et les documents concernés par la question : ajout à la question des documents les plus pertinents, avant d'envoyer le tout au LLM

- Ces 2 phases sont indépendantes

20

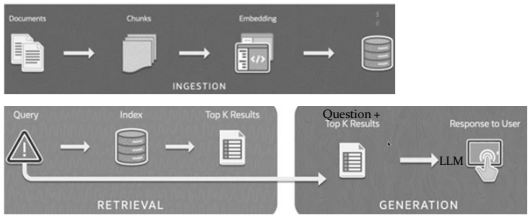
Phase 2 qui génère la réponse

- Elle peut elle-même être décomposée en 2 phases qui s'exécutent consécutivement :
  - Retrieval (récupération) / augmentation : parmi les données « ingérées » dans la phase 1, le système sélectionne les morceaux les plus pertinents pour la question posée et les ajoute à la question
  - Génération : le tout (informations pertinentes + question) est envoyé au LLM qui génère une réponse en utilisant ses capacités linguistiques, sa compréhension de la question, de l'historique de la conversation et des informations retrouvées

21

Phases en images

Pour le type de RAG avec embeddings



(Source Oracle University)

22

Principaux composants du RAG

- Ingestor (« ingesteur », collecteur, module d'ingestion) : pendant la phase 1, transforme les documents « bruts » en une forme exploitable dans la phase 2 ; enregistrement des morceaux, avec leur embedding, dans une BD vectorielle le plus souvent
- Augmentor (« augmenteur ») : recherche les informations pertinentes et les ajoute à la question ; utilise en particulier un Retriever (récupérateur)
- Générateur : LLM qui va générer la réponse en utilisant ses capacités linguistiques et les informations pertinentes retrouvées

23

Sources des données

- Articles
- Podcasts
- Vidéos
- Recherches sur Internet
- Fichiers de tous types (PDF, …)
- BD structurées (relationnelles, graphes de connaissance, BD NoSQL, …)
- …

24

Préparation des données

- Pendant la phase d'ingestion, la préparation des documents est très importante car elle permet d'améliorer grandement les résultats et d'économiser des ressources
- Le processus peut être complexe mais il ne s'effectue qu'une seule fois quand les documents sont ajoutés
- Il faut commencer par nettoyer les documents pour les uniformiser et enlever les parties non pertinentes
- Le plus souvent les documents sont ensuite découpés en morceaux (chunks)
- Des métadonnées peuvent être ajoutées

25

Nettoyage des documents

- Suppression des éléments non pertinents : en-têtes, pieds de page, signatures automatiques des emails, mentions légales, informations répétées à chaque page (nom de l'auteur, titre du document, du chapitre, date), publicités, …
- Supprimer les éléments de mise en forme ; par exemple dans pages HTML
- Corriger les fautes d'orthographe et de grammaire
- Supprimer les contenus obsolètes ou qui ne sont évidemment pas pertinents pour le type de question qui sera posé (pour RAG limité à un domaine bien précis)
- Supprimer les liens externes non pertinents

26

Autres préparations

- Uniformisation des textes : supprimer espaces ou saut de ligne superflus, tout mettre en minuscules ?, remplacer les caractères accentués ?
- Remplacer les sigles ou abréviations par leur signification ; on peut aussi garder l'abréviation mais ajouter la signification
- Les données spéciales (tables, images) peuvent nécessiter des traitements supplémentaires
- …

27

Chunks (morceaux) (1/2)

- Les documents externes sont souvent découpés en morceaux pendant la phase d'ingestion
- On peut ainsi mieux cerner les passages des documents qui sont pertinents pour la question posée
- D'autre part, on réduit ainsi le volume de données à traiter, ce qui donne des réponses plus rapides et une meilleure gestion des ressources
- De plus, pour le RAG avec embeddings, les modèles d'embeddings ont une taille limite pour le texte

28

Chunks (morceaux) (2/2)

- Les morceaux sont le plus souvent de taille fixe (200 à 300 mots est une taille courante) ; des tests sont souvent nécessaires pour trouver la meilleure taille
- Le plus souvent les morceaux se chevauchent légèrement pour éviter de couper au milieu d'une phrase ou d'un paragraphe important
- Essayer de tenir compte des phrases et des paragraphes ; si le document est structuré (sections par exemple), essayer de tenir compte des structures

29

RAG avec mots-clés

- La façon la plus basique de faire du RAG est d'utiliser des mots-clés pour retrouver les documents, ou morceaux de document, les plus pertinents
- Pendant l'ingestion, utilisation de techniques comme TF-IDF (Term Frequency-Inverse Document Frequency) ou BM25 (amélioration TF-IDF) pour calculer les poids des mots des documents
- Pendant la récupération, pour chaque mot de la question, le poids du mot dans chaque document est ajouté au score total du document
- Les documents les plus pertinents (meilleurs scores) sont utilisés pour la génération de la réponse

30

TF-IDF

- Calcule un poids pour chaque mot dans un document en fonction
  - de sa fréquence dans le document (TF, Term Frequency)
  - de la fréquence inverse dans l'ensemble des documents (IDF, Inverse Document Frequency)
- Les mots qui apparaissent fréquemment dans un document mais rarement dans d'autres sont considérés comme particulièrement significatifs

31

RAG avec embeddings

- Les morceaux de documents sont transformés en embeddings qui sont enregistrés dans un entrepôt (magasin d'embeddings), avec les textes correspondants ou bien avec une clé qui permet de retrouver rapidement ces textes
- Des modèles d'embedding sont utilisés pour cette transformation
- Les entrepôts sont le plus souvent des BDs vectorielles
- Ils permettent de faire des recherches de similarités

32

Pourquoi utiliser des embeddings ?

- Ils capturent les sens des mots et des textes, ce qui est bien plus souple que la correspondance exacte avec des mots-clés
- Exemple : « traitement du cancer » a un sens similaire à « thérapies contre les tumeurs », sans aucun mot en commun

33

Etapes ingestion

1. Chacun des documents est transformé en un embedding
2. Traitement optionnel des embeddings :
   - normalisation pour faciliter les calculs de similarité (norme des vecteurs égale à 1)
   - ajout de métadonnées
3. Stockage des embeddings dans une BD vectorielle, avec le document correspondant et d'éventuelles métadonnées

34

Etapes récupération

- La question est transformée en embedding avec le même modèle que les documents enregistrés
- Son embedding est comparé aux embeddings enregistrés
- Les n (par exemple n = 10) documents qui ont les embeddings les plus similaires à l'embedding de la question sont ajoutés au prompt, devant la question, les plus pertinents en premier ; le tout est envoyé au LLM
- Une étape optionnelle de « reranking » peut réordonner les embeddings plus finement
- Le LLM répond à la question en prenant en compte ce contexte enrichi par les documents

35

Template pour récupération

- Souvent l'augmentation utilise un template pour recevoir le contexte récupéré
- Par exemple,

```
En t'appuyant sur les informations suivantes, réponds à la question de l'utilisateur
Contexte :
{{contexte}}

Question de l'utilisateur :
{{question-utilisateur}}
```


36

Température du LLM

- Pour le RAG il est conseillé de réduire la température du LLM pour l'inciter à ne pas trop faire preuve de « créativité », à se limiter à choisir les mots les plus probables, à ne pas « divaguer »
- En effet, un des buts principaux du RAG est d'obtenir des résultats fiables, de limiter les hallucinations
- Température recommandée : entre 0 et 0,3

37

Priorité informations récupérées

- Un LLM considère le contenu du prompt comme fiable et donne donc la priorité aux informations récupérées qui sont ajoutées au prompt par rapport aux connaissances acquises lors de l'apprentissage
- Pour renforcer cette priorité, le prompt peut contenir ce type de phrase : « En utilisant uniquement les informations fournies dans le texte ci-dessus » (l'ajout est fait avant la question)
- D'autres techniques plus avancées peuvent aussi être utilisées

38

Paramètres

- Taille des morceaux
- Nombre d'embeddings récupérés
  - Un trop grand nombre peut nuire à la qualité car les informations importantes risquent d'être noyées dans des informations moins intéressantes
  - Un trop petit nombre risque de manquer des informations importantes
- Il faut tester pour choisir les valeurs de ces paramètres qui donnent les meilleurs résultats

39

Evaluation des performances

- Il est important de pouvoir juger de la qualité des résultats pour choisir les meilleures valeurs des paramètres en testant sur des questions dont on connait les réponses
- Un autre LLM peut aider à juger de la qualité des réponses fournies par le RAG

40

Techniques avancées de RAG

41

Indexation

- Les embeddings sont indexés pour retrouver plus rapidement les embeddings les plus pertinents
- Toutes les BDs vectorielles permettent l'indexation

42

Ajout de mémoire

- S'il peut y avoir une conversation entre l'utilisateur et le LLM, il faut ajouter un historique de la conversation au prompt

43

Métadonnées

- Peuvent être enregistrées avec les morceaux de texte et les embeddings pour accélérer la récupération, améliorer la pertinence des documents et la qualité de la réponse
- Peuvent contenir le titre du document, sa source (livre, site Web, …), la date de publication, les auteurs, …
- Au moment de la récupération, elles offrent la possibilité de filtrer des documents avant la recherche par mots-clés ou par similarité
- Les BD vectorielles permettent souvent d'associer des métadonnées aux embeddings

44

Exemples utilisation métadonnées

- Catégorie : Filtrer par sous-domaine du droit (par exemple, droit civil, droit commercial, droit du travail)
- Date : Filtrer par date de publication d'articles scientifiques pour ne garder que les articles récents ou ceux publiés dans une certaine période
- Confidentialité : Filtrer les documents selon leur niveau de confidentialité pour restreindre l'accès (public, interne, confidentiel)
- Ajouter à la réponse des informations sur les sources utilisées

45

Reranking

- Solution possible si une recherche sémantique ne donne pas un résultat satisfaisant
- Un reclassement (reranking) peut permettre d'obtenir un meilleur résultat : les items retrouvés sont réexaminés en appliquant un autre modèle ou des méthodes pour avoir un résultat plus pertinent
- On peut ainsi combiner plusieurs approches, ou appliquer une méthode plus précise, mais plus lourde et plus coûteuse, sur un nombre limité d'items

46

Routage

- Souvent les données privées des entreprises sont conservées dans des endroits et des formes diverses
- Plutôt que de parcourir toutes les sources de données à chaque prompt, un routage permet de ne consulter que certaines de ces sources de données
- Le routage peut s'appuyer sur
  - Des règles diverses (autorisations de l'utilisateur, service qui a émis la requête, …)
  - Des mots-clés
  - Des calculs de similarité
  - Un choix fait par le LLM

47

Sources de données diverses

- Lors de la récupération, des données peuvent provenir, par exemple, d'un moteur de recherche sur le Web, d'une base de données relationnelle ou d'un graphe de connaissance

48

Utilisation d'outils

- Des outils, par exemple pour effectuer des calculs complexes ou pour retrouver les cours de devises, peuvent être utilisés pendant la phase d'augmentation de la question

49

2 techniques pour RAG

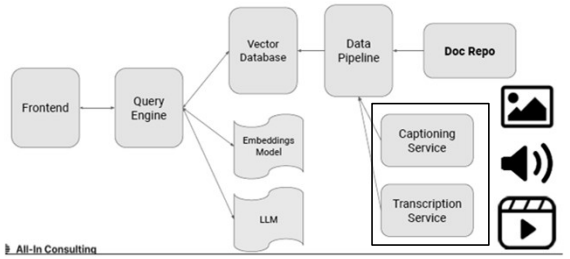
- RAG sequence model : technique la plus utilisée car la plus simple et suffisante dans la plupart des cas ; les documents les plus pertinents sont récupérés et ajoutés à la question et à l'historique de la conversation ; le tout est envoyé au LLM qui génère les tokens de la réponse
- RAG token model : souvent plus précise mais plus difficile à mettre en œuvre ; pour chaque token généré par le LLM on cherche les documents les plus pertinents ; les documents dépendent des tokens déjà générés pour la réponse ; intéressant quand différentes parties de la réponse sont associées à des documents différents

50

RAG multi-modal

- Prend en compte non seulement du texte mais aussi d'autres types de media comme l'audio, la vidéo, les images
- A l'architecture de base il faut ajouter des traitements pour prendre en compte ces médias car les LLMs sont centrés essentiellement sur le texte (et un peu les images)
- Par exemple, les vidéos sont représentées par des captures d'écran et par la transcription de la bande audio

51



52

RAG récursif

- Une requête peut être obtenue plus efficacement par plusieurs résultats intermédiaires qui sont agrégés pour obtenir une réponse
- Les étapes intermédiaires permettent de mieux cerner les besoins et d'aider le LLM à utiliser les bonnes informations ; les 1ères étapes peuvent aussi influencer les étapes suivantes pour optimiser la recherche

53

Exemple

- Une 1ère étape recherche dans une base qui contient des résumés d'articles, ce qui indique dans quels articles rechercher les détails de l'information cherchée
- Voir LlamaIndex (https://www.llamaindex.ai/)

54

Comparaison prompt engineering, fine-tuning, RAG

55

- Comparaison entre les 3 façons de prendre en compte des données sur lesquelles un LLM n'a pas été entrainé (prompt engineering, fine-tuning, RAG)

56

Avantages et inconvénients (1/3)

- Prompt engineering :
  - simple, souple et rapide
  - **mais**
    - difficile de mettre à l'échelle (solution ad hoc pas généralisable)
    - résultats aléatoires
    - le LLM doit déjà connaitre le domaine concerné par la question

57

Avantages et inconvénients (2/3)

- Fine tuning :
  - performant
  - pas limité par la taille maximale du contexte
  - parfait pour répondre avec un certain style
  - **mais**
    - demande une préparation lourde, coûteuse en temps et argent
    - à répéter quand les données changent
    - moins précis et moins souple que le RAG

58

Avantages et inconvénients (3/3)

- RAG (souvent la meilleure solution) :
  - permet de réduire les hallucinations en fournissant des informations fiables avec des sources vérifiables
  - peut fournir des informations différentes selon le type d'utilisateur, par exemple réservant des informations sensibles à des utilisateurs privilégiés
  - permet de tester plusieurs LLMs
  - **mais**
    - traitement à répéter à chaque recherche (pas comme le fine tuning)
    - parfois difficile à mettre en œuvre
    - pas adapté pour certaines tâches, par exemple parler en imitant une personne ou raconter des histoires avec un certain style
    - lourd à gérer si documents très nombreux

59

Que choisir ? (1/4)

- Choisir entre prompt engineering, fine-tuning et RAG dépend de la tâche à accomplir
- Le prompt engineering convient bien si le LLM connait déjà le domaine de la question et si les informations à ajouter au prompt ne sont pas trop volumineuses ou difficiles à trouver

60

Que choisir ? (2/4)

- Le fine tuning convient bien pour
  - répondre à des questions dans un domaine bien défini et spécifique avec des informations stables dans le temps ; par exemple assistance technique sur des produits de base, compréhension de termes médicaux
  - générer du contenu qui n'est pas basé sur des informations incluses dans des documents ; par exemple, imiter une voix, adopter un certain ton ou style
  - reconnaitre des patterns ou des sentiments ; par exemple, classer des emails envoyés par des clients : problèmes techniques, retours, satisfaction, …

61

Que choisir ? (3/4)

- Le RAG convient bien pour
  - des tâches nécessitant des informations actualisées ; par exemple, assistant pour aider les clients à investir en bourse
  - obtenir des réponses basées sur des documents d'entreprise très nombreux et qui peuvent être modifiés à tout moment ; par exemple, FAQ, manuels de procédures
  - outil de recherche dans des bases très volumineuses et qui sont continuellement mises à jour ; par exemple recherche d'articles scientifiques à jour

62

Que choisir ? (4/4)

- En résumé, le RAG
  - Permet d'**améliorer les réponses d'un modèle** grâce à des informations supplémentaires et pertinente
  - Mais il **ne modifie pas fondamentalement le comportement ou le style linguistique du modèle ;** les limitations ou bizarreries du modèle de base seront toujours présentes dans un système RAG
- Le Fine-tuning
  - Ne comprend pas de mécanisme direct pour actualiser rapidement le modèle avec de nouvelles informations
  - Pas aussi fiable que le RAG pour générer des réponses pertinentes et à jour

63

Combinaison des méthodes

- L'utilisation du prompt, le fine-tuning et le RAG peuvent être combinés sur un même projet pour en tirer le meilleur parti ; par exemple

1. Commencer par le plus simple (prompt), avec des techniques de prompt engineering
2. Ajouter du RAG s'il manque des informations pour bien répondre
3. Faire du fine-tuning sur le LLM de base du RAG si le ton, le style ou le format des réponses ne convient pas

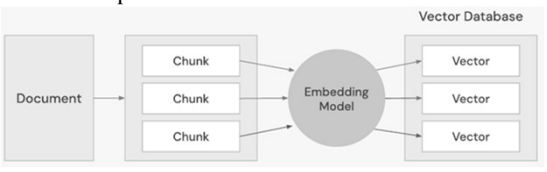
64

Base de données vectorielle

65

BDs vectorielles

- Fréquemment utilisées pour le RAG
- BD pour stocker et rechercher des vecteurs de données
- Ces BDs peuvent faire des recherches par similarités, plutôt que de trouver des correspondances exactes
- Des index peuvent accélérer les recherches



66

Principales caractéristiques

- Stockage et gestion d'un très grand nombre de vecteurs (jusqu'à des milliards)
- Recherche rapide des vecteurs similaires à un vecteur (recherche ANN, Approximate Nearest Neighbor) ; indexation adaptée à cette recherche
- Gestion de métadonnées associées aux vecteurs ; par exemple titre, date de parution, auteurs, emplacement du contenu d'articles
- Langage de requête utilisable par API REST
- Souvent intégration avec frameworks ML
- Forte utilisation des index

67

Index

- Les vecteurs sont indexés pour une permettre des recherches de similarité rapides
- Utilisent des techniques spéciales pour accélérer les recherches et réduire l'occupation de la mémoire
- Indispensables pour l'utilisation des BD vectorielles

68

Exemples d'utilisation

- Recherche sémantique (pour RAG en particulier)
- Recommandation de contenu (pour achats de produits similaires)
- Recherche d'images ou de vidéos (par caractéristiques visuelles)

69

Contenu de la BD vectorielle

- Une BD vectorielle peut stocker les textes associés aux embeddings ou ne stocker que les informations nécessaires pour retrouver ces textes ailleurs (par exemple, dans une base de données classique ou un système de stockage)
- Elle peut aussi contenir des métadonnées qui peuvent alors servir à filtrer les résultats des requêtes (date de création, mots-clés, catégories,…)

70

Types de recherche

- Les recherches dans les BD vectorielles sont le plus souvent des recherches sémantiques, appelées aussi recherches denses, qui utilisent des embeddings
- De nombreuses BD vectorielles peuvent aussi faire des recherches par mots-clés avec des filtres basés sur les métadonnées
- Ces BD vectorielles permettent aussi de faire des recherches hybrides qui combinent la précision et la rapidité des recherches par mots-clés et la compréhension du sens des recherches sémantiques

71

Exemples de recherche hybride

- Si les données sont très nombreuses, un système peut effectuer d'abord une recherche rapide par mots-clés pour réduire le nombre d'items et ensuite faire une recherche sémantique sur le résultat
- Utiliser une recherche par mots-clés pour obtenir un premier ensemble de résultats, puis réordonner ces résultats en fonction des similarités de vecteurs des documents
- Il est possible d'attribuer un poids à chacune des 2 recherches pour obtenir le résultat final
- Les résultats d'une recherche dense peuvent être filtrés pour éliminer les items qui n'ont pas certains mots-clés

72

Produits

- Milnius
- Weaviate
- Pinecone
- Chroma
- Vespa (Yahoo!)
- Qdrant
- Elasticsearch (avec plugins spécialisés)
- AI Vector Search (Oracle)

73

Outils pour BD vectorielle

- Faiss (Meta), Annoy (Spotify) : bibliothèques pour la recherche de similarité

74

RAG avec LangChain4j

75

LangChain4j

- LangChain4j fournit plusieurs types Java pour faciliter le RAG
- On verra d'abord un « RAG facile » pour faire du RAG sans configurations ni optimisations (https://docs.langchain4j.dev/tutorials/rag)
- Les classes qui permettent de construire des RAGs plus adaptés aux cas particuliers seront étudiées ensuite

76

Exemple RAG facile (1/2)

```
public interface Assistant {
  String chat(String userMessage);
}
```


77

Exemple RAG facile (2/2)

```
List<Document> documents = FileSystemDocumentLoader.
    loadDocuments("/home/langchain4j/documentation");
InMemoryEmbeddingStore<TextSegment> embeddingStore =
    new InMemoryEmbeddingStore<>();
EmbeddingStoreIngestor.ingest(documents, embeddingStore);
ChatLanguageModel model = …;
Assistant assistant = AiServices.builder(Assistant.class)
    .chatLanguageModel(model)
    .chatMemory(MessageWindowChatMemory.withMaxMessages(10))
    .contentRetriever(
        EmbeddingStoreContentRetriever.from(embeddingStore))
    .build();
String answer = assistant.chat("Comment peut-on faire du RAG
facile avec LangChain4j ?");
```


78

Types de données

- `Document` : représente un fichier local ou une page Web qui contient du texte
- `FileSystemDocumentLoader` : charge des fichiers pour obtenir des documents
- `TextSegment` : morceaux de texte
- `InMemoryEmbeddingStore<TextSegment>` : magasin en mémoire centrale pour embeddings de `TextSegment`
- `EmbeddingStoreIngestor` : Pipeline pour l'ingestion de documents dans un magasin d'embeddings
- `EmbeddingStoreContentRetriever` : retrouve les `Content` les plus similaires à la question, dans un magasin d'embedding

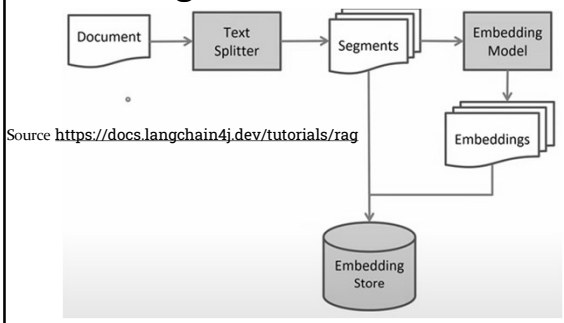
79

Classes et interfaces pour RAG

- Phase de chargement des `Documents`
- Phase de récupération des données pertinentes et augmentation du prompt

80

RAG - Ingestion



Source https://docs.langchain4j.dev/tutorials/rag

81

Présentation des types (1/3)

- Classe `DocumentLoader` met des données provenant de différentes sources (fichiers, URLs, inputStream, ...) dans une instance de `Document`
  - `FileSystemDocumentLoader` : depuis des fichiers
  - `UrlDocumentLoader` : depuis des URLs
- Une fois le document créé, on peut le découper en une `List<TextSegment>`, en utilisant un `DocumentSplitter`

82

Présentation des types (2/3)

- Interface `EmbeddingModel` est un modèle d'embeddings qui peut convertir des mots, phrases, documents en `Embeddings` ; plusieurs implémentations pour OpenAI, Ollama, HuggingFace, ...
  Un `EmbeddingModel` peut être utilisé pour créer une `List<Embedding>` à partir d'une `List<TextSegment>`

83

Présentation des types (3/3)

- Interface `EmbeddingStore` est un magasin d'embeddings (BD vectorielle, ou autre type de magasin) ; contient des Embedding et leur segment
- Plusieurs implémentations selon le magasin utilisé : en mémoire, Neo4j, Chroma, ...
- Classe `EmbeddingStoreIngestor` met des documents sous la forme d'embeddings dans un magasin d'embeddings
  Prend en compte les différentes étapes : découper le document en segments, traduire en embeddings, les ranger dans le magasin

84

- Création des documents

85

Classe `Document`

- Package dev.langchain4j.data.document
- Représente un fichier de texte local ou page Web
- Le format du fichier local peut être un simple fichier texte, un PDF, un docx, ...
- Des métadonnées peuvent être attachées au document (classe `Metadata` qui est une enveloppe pour une `Map`) ; par exemple, la source du document, sa date de création, son auteur
- Constructeurs avec paramètre `String` texte, `Metadata` (optionnel)
- Getters `String text()`, `MetaData metadata()`, `TextSegment toTextSegment()`

86

Classe `TextSegment`

- Package dev.langchain4j.data.segment
- Représente un morceau de texte d'une entité plus large, avec ses éventuelles métadonnées
- Getters
  - `String text()` pour extraire le contenu
  - `Metadata metadata()`

87

Classe `Metadata`

- Package dev.langchain4j.data.document
- Métadonnées d'un `Document` ou d'un `TextSegment`
- Pour un document, ça peut être la date de création, le propriétaire, ...
- Pour un segment, ça peut être un numéro de page, la position du segment dans le document, le chapitre, ...
- Les métadonnées sont enregistrées comme une `Map` avec les clés de type `String` et les valeurs de type `String`, `UUID`, `Long`, `Float`, `Double` (valeur `null` interdite)

88

Classe `DocumentLoader`

- package dev.langchain4j.data.document
- Classe utilitaire pour charger un document
- Une seule méthode `static` pour charger le document

```
Document load(DocumentSource source,
              DocumentParser parser)
```


89

Interface `DocumentSource`

- Package dev.langchain4j.data.document
- Source pour obtenir un `Document`
- 2 méthodes :
  - `InputStream inputStream() throws IOException` lit le contenu du document
  - `Metadata metadata()` retourne les métadonnées associées avec la source du document
- Nombreuses implémentations : `FileSystemSource`, `UrlSource`, `GitHubSource`, `AmazonS3Source`, ...

90

Interface `DocumentParser`

- Package dev.langchain4j.data.document
- Pour parser un `InputStream` en un `Document`
- Une seule méthode
  `Document parse(InputStream inputStream)`
- Plusieurs implémentations :
  `ApachePdfBoxDocumentParser` (fichiers PDF), `ApachePoiDocumentParser` (fichiers doc, docx, ppt, pptx, xls, xlsx), `ApacheTikaDocumentParser` (fichiers PDF, doc, ppt, xls), `TextDocumentParser` (fichiers texte)

91

Code avec `DocumentLoader`

```
// Charger un document local
Path path = Paths.get("/path/to/some/file.txt");
Document document =
    FileSystemDocumentLoader
    .loadDocument(path, new TextDocumentParser());

// Charger depuis le Web
URL url = new URL("https://...");
Document htmlDocument =
    UrlDocumentLoader.load(url, new TextDocumentParser());
HtmlToTextDocumentExtractor transformer =
    new HtmlTextExtractor(null, null, true);
Document document = transformer.transform(htmlDocument);
```


92

`FileSystemDocumentLoader`

- Package dev.langchain4j.data.document.loader
- Classe utilitaire pour transformer en `Documents` des fichiers locaux dont on connait le Path
- Méthodes `static`
  - `loadDocument` surchargées pour charger un document
  - `loadDocuments` surchargées pour charger les documents d'un répertoire
  - `loadDocumentsRecursively` surchargées pour charger les documents d'un répertoire (et des sous-répertoires)
- Chemin des fichiers ou du répertoire par String ou Path
- Paramètres optionnels pour le parser du document (`DocumentParser`) ou pour filtrer avec un pattern de nom de fichier (`PathMatcher`)

93

Exemples avec `PathMatcher`

- ```
  List<Document> documents = FileSystemDocumentLoader
      .loadDocuments("/home/langchain4j/documentation");
  ```
- ```
  PathMatcher pathMatcher = FileSystems.getDefault()
      .getPathMatcher("glob:*.pdf");
  List<Document> documents = FileSystemDocumentLoader
      .loadDocuments("/home/langchain4j/documentation",
                     pathMatcher);
  ```
- ```
  PathMatcher pathMatcher = FileSystems.getDefault()
      .getPathMatcher("glob:**.pdf");
  List<Document> documents = FileSystemDocumentLoader
      .loadDocumentsRecursively(
          "/home/langchain4j/documentation",
          pathMatcher);
  ```

94

Classe `FileSystemSource`

- Package dev.langchain4j.data.document.source
- Juste pour donner un exemple d'implémentation de `DocumentSource`
- Constructeur pour prend un `java.nio.file.Path` en paramètre ; on peut aussi utiliser les méthodes static from avec un de ces paramètres : `File`, `String`, `URI`, `Path`
- Méthodes :
  - `InputStream inputStream()`
  - `Metadata metadata()`

95

- Découper les `Documents` en morceaux

96
  ```

Interface DocumentSplitter

- Package dev.langchain4j.data.document
- Découpe un document en segments de texte
- Indispensable car les LLMs limitent la taille du contexte, et pour améliorer la précision des réponses
- Nombreuses classes d'implémentation parmi lesquelles `DocumentByParagraphSplitter`, `DocumentBySentenceSplitter`, `DocumentByWordSplitter`, `DocumentByRegexSplitter`, `HierarchicalDocumentSplitter`

97

Classe DocumentSplitters

- Package dev.langchain4j.data.document.splitter
- Classe utilitaire pour créer un `DocumentSplitter`
- Contient 2 méthodes `static recursive` qui retournent un `DocumentSplitter` recommandé pour découper un document en segments
- Ce splitter essaie de découper le document en paragraphes et met le plus de paragraphes possibles dans chaque `TextSegment`
- Si un paragraphe est trop long pour tenir dans un segment, il est récursivement découpé en lignes, puis en phrases, puis en mots et en caractères pour tenir dans un segment
- Paramètres des méthodes : taille maxi d'un segment (en caractères ou en tokens), taille maxi du chevauchement (seulement les phrases entières sont envisagées), tokenizer qui compte les tokens dans le texte

98

Interface Tokenizer

- Package dev.langchain4j.model
- Estime à l'avance le nombre de tokens d'un texte généré par un processus ; utilisé si la taille maximum d'un segment est donnée en tokens

99

Code pour splitter

```
URL fileUrl = TestRAG.class.getResource("/...");
Path path = Paths.get(fileUrl.toURI());
Document document = FileSystemDocumentLoader
    .loadDocument(path, new TextDocumentParser());

DocumentSplitter splitter =
    // découpe en morceaux de 600 caractères,
    // sans chevauchement
    DocumentSplitters.recursive(600, 0);

List<TextSegment> segments = splitter.split(document);
List<Embedding> embeddings =
    embeddingModel.embedAll(segments).content();

embeddingStore.addAll(embeddings, segments);
```


100

- Création des embeddings des morceaux et enregistrement dans un magasin d'embeddings

101

Interface EmbeddingModel

- Package dev.langchain4j.model.embedding
- Modèle d'embeddings IA qui crée des embeddings à partir de segments ; implémenté par de très nombreuses classes
- Méthode abstraite
  `Response<List<Embedding>> embedAll(List<TextSegment> textSegments)`
- Méthodes par défaut (`default`) :
  - `int dimension()` : renvoie dimension des embeddings
  - `Response<Embedding> embed(String text)` : renvoie embedding du texte
  - `Response<Embedding> embed(TextSegment segment)`

102

Création EmbeddingModel

```
EmbeddingModel embeddingModel =
    new AllMiniLmL6V2EmbeddingModel();
// ou
EmbeddingModel embeddingModel =
    OllamaEmbeddingModel.builder()
        .baseUrl("http://localhost:11434")
        .modelName("llama2")
        .build();
```


103

Création et enregistrement embedding

```
TextSegment segment = TextSegment.from("un texte …");
Embedding embedding =
    embeddingModel.embed(segment).content();
embeddingStore.add(embedding, segment);
```


104

Classe Response<T>

- Package dev.langchain4j.model.output
- Représente une réponse de plusieurs types de modèle (langage, embedding, modération)
- Méthodes :
  - `@NonNull T content()` : récupère le contenu
  - `TokenUsage tokenUsage()` : récupère les statistiques d'usage
  - `FinishReason finishReason()`
  - `@NonNull Map<String,Object> metadata()` : récupère les éventuelles métadonnées

105

EmbeddingStore<T> (1/2)

- Interface du package dev.langchain4j.store.embedding
- Pour un magasin/dépôt d'embeddings
- `T` représente la classe de ce qui va être transformé en embeddings, typiquement `TextSegment`
- Implémenté par classes pour BD vectorielles (Neo4j, Chroma, MongoDB,…) et par la classe `InMemoryEmbeddingStore`
- Méthodes `add` et `addAll` pour ajouter un ou plusieurs embeddings, avec ou sans leur texte d'origine
- Méthodes `remove` et `removeAll` pour supprimer des embeddings

106

EmbeddingStore<T> (2/2)

- Méthode pour chercher des embeddings similaires (pour la phase « retrieval » du RAG) ; ce que l'on cherche est défini par `EmbeddingSearchRequest`

```
default EmbeddingSearchResult<Embedded>
search(EmbeddingSearchRequest request)
```


107

Code avec EmbeddingStore

```
EmbeddingStore embeddingStore =
    new InMemoryEmbeddingStore();
// ou
EmbeddingStore embeddingStore =
    ChromaEmbeddingStore.builder()
        .baseUrl("http://localhost:8000")
        .collectionName("my-collection")
        .build();
```


108

InMemoryEmbeddingStore<T>

- Package dev.langchain4j.store.embedding.inmemory
- Entrepôt pour embeddings (représentés par T), en mémoire centrale ; pour tests rapides sans BD vectorielle
- Pas d'indexation des embeddings (ils sont parcourus du premier au dernier)

EmbeddingStoreIngestor (1/2)

- Package dev.langchain4j.store.embedding
- Pipeline responsable de l'ingestion des documents dans un magasin d'embeddings
- Il gère tout le processus : découpage en segments, génération des embeddings pour chaque segment, enregistrement des embeddings
- Il est possible d'ajouter à ce processus la transformation des documents (pour les nettoyer) et la transformation des segments après le découpage en segments
- Les informations nécessaires aux traitements sont transmises à la création de l'ingestor

EmbeddingStoreIngestor (2/2)

- Le constructeur reçoit en paramètre les informations nécessaires :
  - DocumentTransformer qui peut effectuer des tâches sur le document (filtrage, enrichissement, ...) ; optionnel
  - DocumentSplitter ; optionnel
  - TestSegmentTransformer (peut transformer les segments)
  - EmbeddingModel
  - EmbeddingStore

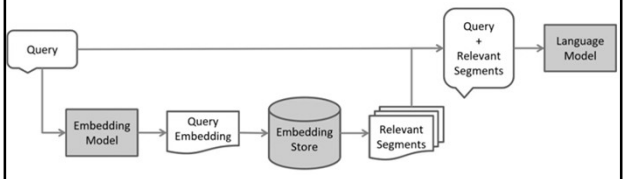
Code avec EmbeddingStoreIngestor

```
// Début comme code pour splitter
EmbeddingStoreIngestor ingestor =
    EmbeddingStoreIngestor
        .builder()
        .documentSplitter(splitter)
        .embeddingModel(embeddingModel)
        .embeddingStore(embeddingStore)
        .build();
ingestor.ingest(document);
```


Exemple d'ingestion

```
EmbeddingModel embeddingModel =
    new AllMiniLmL6V2EmbeddingModel();
EmbeddingStore<TextSegment> embeddingStore =
    new InMemoryEmbeddingStore<>();
EmbeddingStoreIngestor ingestor =
    EmbeddingStoreIngestor.builder()
    .documentSplitter(
        DocumentSplitters.recursive(300, 0))
        .embeddingModel(embeddingModel)
        .embeddingStore(embeddingStore)
        .build();
Document document = loadDocument(toPath("sxxx.txt"),
                                  new TextDocumentParser());
ingestor.ingest(document);
```


RAG - Retrieval (récupération)



Source https://docs.langchain4j.dev/tutorials/rag

Présentation des types

- `Query` : Représente une question posée par l'utilisateur pour retrouver des `Contents`
- `ContentRetriever` : pour retrouver les informations les plus pertinentes pour une question ; utilise un `RetrievalAugmentor` par défaut
- `RetrievalAugmentor` : à utiliser quand il faut configurer un comportement particulier pour améliorer les réponses du LLM (RAG « avancé »)

115

Interface `ContentRetriever`

- Retrouve des informations d'une source de données en utilisant un `Query`
- La source de données peut être
  - un magasin d'embeddings
  - un moteur de recherche dans du texte
  - un moteur de recherche sur le Web
  - un graphe de connaissance
  - une BD relationnelle
  - etc.
- Une seule méthode qui retourne le contenu retrouvé, trié par pertinence, les plus pertinents en premiers
  `List<Content> retrieve(Query)`

116

Classe `Query`

- Package dev.langchain4j.rag.query
- Représente une question posée par l'utilisateur pour retrouver des `Contents`
- 2 constructeurs avec ces paramètres (on peut aussi utiliser les méthodes `static from` avec les mêmes paramètres) :
  - `String`
  - `String` et `Metadata`
- 2 méthodes
  - `String txt()`
  - `Metadata metadata()`

117

Content

- Package dev.langchain4j.rag.content
- Représente un contenu pertinent pour un `Query`
- Uniquement pour `TextSegment` pour le moment, en attendant d'autres types de données
- Constructeur avec un paramètre de type `String` ou `TextContent` (on peut aussi utiliser une méthode `static from` avec les mêmes paramètres)
- Méthode `TextSegment textSegment()` pour récupérer le `TextSegment`

118

Implémentation de ContentRetriever

- `EmbeddingStoreContentRetriever` : retrouve depuis un `EmbeddingStore`
- `WebSearchContentRetriever` : retrouve depuis le Web, en utilisant un `WebSearchEngine`
- `SqlDatabaseContentRetriever` : génère des requêtes SQL qui correspondent à des requêtes exprimées en langage naturel (attention, danger !)
- `Neo4jContentRetriever` : utilise Neo4j, une BD orientée graphe
- `AzureAiSearchContentRetriever` : utilise le service de recherche de Azure (fournisseur de cloud)

119

`EmbeddingStoreContentRetriever`

- Classe du package dev.langchain4j.rag.content.retriever
- `EmbeddingStoreContentRetriever` : retrouve depuis un `EmbeddingStore` ; par défaut, retrouve les 3 `Contents` les plus pertinents, sans filtre ; créé avec un builder
- Méthodes du builder pour donner un nom, un nombre maximum de résultats, un score minimum de pertinence, un filtre (`Filter`) pour les métadonnées, un filtre dynamique (fonction qui peut dépendre du Query, de l'utilisateur, …)

120

Interface `Filter`

- Package dev.langchain4j.store.embedding.filter
- Pour filtrer les **métadonnées**
- Seule la méthode `boolean test(Object)` est abstraite et on peut donc utiliser une expression lambda ; elle teste si un objet satisfait le filtre
- Méthodes `static and`, `not`, `or` ; méthode default `and`, `or` qui correspondent aux méthodes `static`, avec `this` comme filtre gauche
- Implémentée par `And`, `IsEqualTo`, `IsGreaterThan`, `IsGreaterThanOrEqualTo`, `IsIn`, `IsLessThan`, `IsLessThanOrEqualTo`, `IsNotEqualTo`, `IsNotIn`, `IsTextMatch`, `IsTextMatchPhrase`, `Not`, `Or`

121

Exemple de filtre dynamique

```
Function<Query, Filter> filterByUserId =
   (query) -> metadataKey("userId")
              .isEqualTo(query.metadata()
              .chatMemoryId().toString());
...
ContentRetriever contentRetriever =
    EmbeddingStoreContentRetriever.builder()
    .embeddingStore(embeddingStore)
    .embeddingModel(embeddingModel)
    // by specifying the dynamic filter, we limit the
    // search to segments that belong only to the
    // current user
    .dynamicFilter(filterByUserId)
    .build();
```


122

`WebSearchContentRetriever`

- retrouve depuis le Web, en utilisant un `WebSearchEngine`
- Méthodes du builder pour le nombre maximum de résultats, le `WebSearchEngine`
- Seule méthode : celle de l'interface `List<Content> retrieve(Query query)`

123

Interface `WebSearchEngine`

- Package dev.langchain4j.web.search
- Représente un moteur de recherche sur le Web
- 2 méthodes
  - WebSearchResults search(WebSearchRequest webSearchRequest), la seule méthode abstraite
  - default WebSearchResults search(String query)
- Implémentée par les classes GoogleCustomWebSearchEngine, SearchApiWebSearchEngine, TavilyWebSearchEngine

124

Interface `WebSearchRequest`

- Package dev.langchain4j.web.search
- Représente une requête envoyée à un moteur de recherche sur le Web ; suit le standard OpenSearch
- Méthode principale pour créer une requête :
  - `from(String searchTerms[, Integer maxResults])`

125

Interface `WebSearchResults`

- Package dev.langchain4j.web.search
- Représente le résultat d'une requête de recherche sur le Web

126

Classe Metadata

- Package dev.langchain4j.rag.query
- Représente des métadonnées utilisées pour retrouver des informations (utilisée par `Query`) ou pour augmenter une question posée par l'utilisateur (avant d'envoyer la requête au LLM)
- Constructeur (ou méthode `static from`) avec les paramètres de type `UserMessage`, `Object` (chatMemoryId qui peut être utilisé pour distinguer les utilisateurs), `List<ChatMessage>` (chatMemory pour les messages précédents du chat, pour fournir du contexte)
- 3 « getters » `userMessage()`, `chatMemoryId()`, `chatMemory()`

127

Classe Content

- Package dev.langchain4j.rag.content
- Représente un contenu pertinent obtenu en réponse d'un `Query` ; pour le moment limité à `TextContent`
- Constructeur avec un paramètre de type `String` ou `TextContent` (on peut aussi utiliser une méthode `static from` avec les mêmes paramètres)
- Méthode `TextSegment textSegment()` pour récupérer le `TextSegment`

128

EmbeddingSearchRequest

- Classe qui représente une requête dans un `EmbeddingStore`
- Builder et constructeur avec les paramètres de type
  - `Embedding` : l'embedding dont on cherche les embeddings similaires
  - `Integer` : nombre maximum d'embeddings à retourner (3 par défaut)
  - `Double` : seulement les embeddings avec un score >= à ce nombre seront retournés (compris entre 0 et 1, bornes comprises ; 0 par défaut)
  - `Filter` : filtre à appliquer aux **métadonnées** ; seuls les segments qui correspondent à ce filtre seront retournés ; pas de filtre par défaut

129

EmbeddingSearchResult<T>

- Package dev.langchain4j.store.embedding
- Classe générique (paramètre de type nommé `T`, `TextSegment` le plus souvent) qui représente le résultat d'une recherche dans un magasin d'embeddings
- Constructeur qui prend en paramètre une `List<EmbeddingMatch<T>>`
- Instance retournée par la méthode `search` de `EmbeddingStore<T>`
- Méthode `matches()` qui retourne une `List<EmbeddingMatch<T>>`

130

EmbeddingMatch<T>

- Classe qui représente un embedding retourné par une recherche, avec son score de pertinence (dérivé de la distance cosinusoïdale) par rapport à l'embedding de la recherche, son ID et son contenu d'origine (`TextSegment` le plus souvent)
- Méthodes :
  - `T embedded()` retourne le contenu d'origine
  - `Embedding embedding()` retourne l'embedding
  - `String embeddingId()` retourne id embedding dans magasin
  - `Double score()` retourne le score de pertinence

131

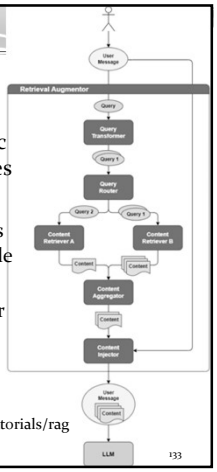
Exemple recherche plus proche

```
Embedding embeddingQuestion = embeddingModel.embed("Quel est
votre sport favori ?").content();
EmbeddingSearchRequest searchRequest =
    EmbeddingSearchRequest.builder()
        .queryEmbedding(embeddingQuestion)
        .maxResults(3)
        .build();
EmbeddingSearchResult<TextSegment> resultatRecherche =
    embeddingStore.search(searchRequest);
List<EmbeddingMatch<TextSegment>> pertinents =
    resultatRecherche.matches();
pertinents.forEach(pertinent -> {
    System.out.println(pertinent.score());
    System.out.println(pertinent.embedded().text());
});
```


132

RetrievalAugmentor

- Pour augmenter un `ChatMessage` avec les informations pertinentes retrouvées
- Il peut :
  - récrire le prompt, par exemple avec des techniques de prompt engineering ou le découper en plusieurs `Querys`
  - router vers les bons récupérateurs pour trouver les informations pertinentes
  - ajouter les informations au prompt



Source image https://docs.langchain4j.dev/tutorials/rag

133

Utilisation RetrievalAugmentor

- Le désigner lors de la création d'un AI service :

```
Assistant assistant =
  AiServices.builder(Assistant.class)
  ...
  .retrievalAugmentor(retrievalAugmentor)
  .build();
```


134

Interface RetrievalAugmentor

- Package dev.langchain4j.rag
- Une seule méthode :
  - `AugmentationResult augment(AugmentationRequest augmentationRequest)`
    `augmentationRequest` contient le `ChatMessage` à augmenter
- Implémentée par `DefaultRetrievalAugmentor` qui devrait convenir dans la majorité des cas

135

Class AugmentationRequest

- Package dev.langchain4j.rag
- Représente une requête pour une augmentation de `ChatMessage`
- Constructeur avec paramètres de type `ChatMessage` et `Metadata` (celui du package `dev.langchain4j.rag.query`)
- 2 « getters » `ChatMessage chatMessage()`, `MetaData metadata()`

136

Class AugmentationResult

- Package dev.langchain4j.rag
- Représente le résultat d'une augmentation
- 2 getters :
  - `ChatMessage chatMessage()` : message d'origine, éventuellement récrit
  - `List<Content> contents()` : les ajouts

137

DefaultRetrievalAugmentor (1/2)

- Package dev.langchain4j.rag
- Classe qui organise le flot entre ces composants :
  1. Un `QueryTransformer` pour transformer le `Query` (obtenu avec le `UserMessage`, le prompt de départ)
  2. Un `QueryRouter` va router le `Query` vers un ou plusieurs des `ContentRetrievers`
  3. Les `Contents` retrouvés sont alors transformés par un `ContentAggregator` en une liste finale de `Contents`
  4. Un `ContentInjector` ajoute la liste au `UserMessage`

138

DefaultRetrievalAugmentor (2/2)

- Chaque composant (à part `ContentRetriever`) est initialisé avec une classe d'implémentation par défaut, par exemple `DefaultQueryTransformer`
- D'autres classes avancées d'implémentation sont fournies par LangChain4j, par exemple `CompressingQueryTransformer` ou `ExpandingQueryTransformer`

139

QueryTransformer

- Interface du package dev.langchain4j.rag.query.transformer
- Transforme un `Query` en un ou plusieurs Query
- Pour plus de détails : https://blog.langchain.dev/query-transformations/

140

QueryRouter

- Interface du package dev.langchain4j.rag.query.router
- Une seule méthode : `Collection<ContentRetriever> route(Query query)` qui route vers un ou plusieurs `ContentRetriever`, selon la question (`Query`)
- L'aiguillage peut être fait par un LLM, un modèle d'embeddings, par des mots-clés, par l'utilisateur qui pose la question (`query.metadata().chatMemoryId()`) ou par les autorisations
- Implémentée par 2 classes `DefaultQueryRouter` `LanguageModelQueryRouter`

141

DefaultQueryRouter

- Package dev.langchain4j.rag.query.router
- Constructeur qui prend en paramètre un ou plusieurs `ContentRetriever`
- La méthode `route` renvoie tous les `ContentRetriever`
- Intéressant si on veut utiliser plusieurs types de `ContentRetriever` pour récupérer des documents pour le RAG

142

LanguageModelQueryRouter

- Choix du ou des `ContentRetrievers` fait par un LLM
- Si la connexion au LLM ou si le LLM donne une réponse non valable, on peut donner une stratégie de fallback (de repli) : pas de RAG, lancer une exception, router vers tous les `ContentRetrievers`
- Pattern builder pour créer une instance ; on peut passer
  - une `Map<ContentRetriever, String>` qui décrit chaque `ContentRetriever`
  - un `ChatLanguageModel`, celui qui décide
  - un `PromptTemplate` pour poser la question au LLM
  - une stratégie de repli

143

Exemple de routage

```
// 2 content retrievers qui utilisent chacun un fichier
// pour retrouver les informations à ajouter au prompt
Map<ContentRetriever, String> descriptions =
    new HashMap<>();
descriptions.put(contentRetriever1, "...");
descriptions.put(contentRetriever2, "...");
QueryRouter queryRouter =
    new LanguageModelQueryRouter(chatLanguageModel,
                                 descriptions);
```


144

QueryRouter personnalisé

- Il faut implémenter l'interface `QueryRouter`
- Donc implémenter la méthode `Collection<ContentRetriever> route(Query query)`
- Souvent la classe sera une classe interne à la méthode qui l'utilise ; classe anonyme ou pas
- L'exemple suivant indique qu'il ne faut pas de RAG si la question de l'utilisateur ne porte pas sur l'IA

145

Exemple classe interne

```
class QueryRouterPourEviterRag implements QueryRouter {
  @Override
  public List<ContentRetriever> route(Query query) {
    String question =
        "Est-ce que la requête '" + query.text()
        + "' porte sur l'IA ? Réponds seulement par"
        + " 'oui', 'non', ou 'peut-être'.";
    String reponse = model.generate(question);
    if (reponse.toLowerCase().contains("non")) {
      // Pas de RAG
      return Collections.emptyList();
    } else {
      return Collections.singletonList(contentRetriever);
    }
  }
}
```


146

Interface ContentAgregator

- Package dev.langchain4j.rag.content.aggregator
- Réunit tous les `Contents` retrouvés par tous les `ContentRetriever` afin de sélectionner les plus pertinents
- Par exemple pour faire du reranking
- Une méthode `List<Content> aggregate(Map<Query,Collection<List<Content>>> queryToContents)`
- Implémentée par `DefaultContentAggregator`, `ReRankingContentAggregator`

147

DefaultContentAgregator

- Implémentation de `ContentAgregator` qui convient dans la plupart des cas

148

ReRankingContentAggregator

- Package dev.langchain4j.rag.content.aggregator
- Effectue un reranking en utilisant un `ScoringModel`
- Peut-être configuré en donnant le score minimum d'un `Content` pour qu'il soit retenu

149

Interface ScoringModel

- Package dev.langchain4j.model.scoring
- Attribue un score de similarité par rapport à un texte sous la forme d'une `Response`
- Une méthode abstraite
  `Response<List<Double>> scoreAll( List<TextSegment> segments, String query)`
- 2 méthodes `default`
  - `Response<Double> score(String text, String query)`
  - `Response<Double> score(TextSegment segment, String query)`

150

Exemple pour reranking (1/2)

```
ScoringModel scoringModel = CohereScoringModel.builder()
                .apiKey(cohereKey)
                .modelName("rerank-multilingual-v3.0")
                .build();

ContentAggregator contentAggregator =
    ReRankingContentAggregator.builder()
                .scoringModel(scoringModel)
                .minScore(0.6)
                .build();
```


151

Exemple pour reranking (2/2)

```
RetrievalAugmentor retrievalAugmentor =
    DefaultRetrievalAugmentor.builder()
                .contentRetriever(contentRetriever)
                .contentAggregator(contentAggregator)
                .build();

AiServices.builder(Assistant.class)
                .chatLanguageModel(modele)
                .retrievalAugmentor(retrievalAugmentor)
                .chatMemory(memory)
                .build();
```


152

BDs vectorielles supportées par LangChain4j

- Chroma
- Elasticsearch
- Milvus
- Pinecone
- Vespa
- Weaviate
- Redis
- Astra DB
- Cassandra

153

Code pour utiliser Chroma

```
try(ChromaDBContainer chroma =
        new ChromaDBContainer("chromadb/chroma:0.4.22")) {
  chroma.start();
  EmbeddingStore<TextSegment> embeddingStore =
      ChromaEmbeddingStore.builder()
                          .baseUrl(chroma.getEndpoint())
                          .collectionName(randomUUID())
                          .build();
  // Tout le reste du code ne dépend pas de Chroma
  EmbeddingModel embeddingModel =
      new AllMiniLmL6V2EmbeddingModel();
  ...
}
```


154

Références

155

- RAG explained : embedding, sentence BERT, vector database par Umar Jamil : https://www.youtube.com/watch?v=rhZgXNdhWDY

156