

# Fine-tuning et RAG

EMSI - Université Côte d'Azur

Richard Grin

Version 1.32 - 24/10/25

1

## Plan du support

- Fine-tuning
- RAG
- Comparaison prompt engineering, fine-tuning, RAG
- Base de données vectorielle
- RAG avec LangChain4j
- Références

R. Grin

Fine-tuning et RAG

2

## Principaux problèmes LMs publics

- Ils peuvent halluciner
- Pas de connaissances 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

R. Grin

Fine-tuning et RAG

3

3

## Pour atténuer les problèmes

- Prompt engineering (déjà étudié)
- Fine-tuning qui modifie le LM (change ses paramètres)
- RAG qui ne modifie pas le LM mais qui, pour chaque question,
  - recherche dans des documents fiables des informations supplémentaires pertinentes pour y répondre
  - les ajoute au prompt pour que le LM puisse les utiliser

R. Grin

Fine-tuning et RAG

4

4

## Fine-tuning

R. Grin

Fine-tuning et RAG

5

5

## Fine-tuning (réglage fin)

- **Entraînement complémentaire** d'un modèle d'IA existant (OpenAi, Gemini,...), sur de nouvelles données liées à une tâche spécifique, à un nouveau domaine, ...
  - Le pré-entraînement du modèle existant a 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 acquiert des connaissances sur une tâche spécifique, dans un domaine particulier, ou apprend à converser avec un style ou un ton défini

R. Grin

Fine-tuning et RAG

6

6

## Entrainement pour fine-tuning

- On fournit à l'API des paires de texte (entrée, et sortie attendue) pour entrainer le modèle
- Chaque LM a son propre format pour les données d'entraînement
- Pour avoir un impact significatif sur le modèle de base pré-entraîné, il faut l'entraîner sur au moins plusieurs centaines (pour une tâche très spécifique) à plusieurs milliers d'exemples ou davantage

R. Grin

Fine-tuning et RAG

7

7

## Coûts

- L'entraînement du fine-tuning est bien plus coûteux que l'utilisation du modèle (inférence) ; il nécessite des machines puissantes et beaucoup de temps
- Les données utilisées pendant l'entraînement 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

R. Grin

Fine-tuning et RAG

8

8

## Options pour le fine-tuning

- Le fine-tuning étant long et très coûteux, des options ont vu le jour (soft-prompting, T-few, LORA, adapters, pré-entraînement continu, ...)
- Ces options offrent un compromis intéressant entre coût et personnalisation mais n'égale pas la performance d'un fine-tuning complet sur des tâches complexes

R. Grin

Fine-tuning et RAG

9

9

## Retrieval-Augmented Generation (RAG)

- Bases
- Techniques avancées

R. Grin

Fine-tuning et RAG

10

10

## Bases

R. Grin

Fine-tuning et RAG

11

11

## Présentation

- Le RAG est une technique qui permet de fournir à un LM des données externes fiables pour lui permettre de répondre aux questions
- Ces données peuvent être contenues dans des documents d'entreprise de haute expertise dans un domaine, ou des documents récents sur lesquels le LM n'a pas été entraîné
- L'exactitude et la pertinence des réponses du LM sont ainsi améliorées et le LM peut indiquer les sources d'information qu'il a utilisées pour répondre

R. Grin

Fine-tuning et RAG

12

12

## Utilisation des données

- Au contraire du fine-tuning, les données fournies au LM ne modifient pas les paramètres du LM
- Ces données sont automatiquement ajoutées au prompt envoyé au LM et ainsi le LM pourra les utiliser pour répondre aux questions

R. Grin

Fine-tuning et RAG

13

13

## Sources des données externes

- Articles
- Podcasts
- Vidéos
- Recherches sur Internet
- Fichiers de tous types (PDF, ...)
- BD structurées (relationnelles) ou semi-structurées (noSQL)
- Bases de connaissances
- API ou moteurs de recherche (API d'encyclopédies, Google, Bing)
- ...

R. Grin

Fine-tuning et RAG

14

14

## Avantages du RAG

- Réduction des hallucinations : le LM peut s'appuyer sur des sources fiables et adaptées au type de questions prévu pour l'application
- Accès à des informations privées de l'entreprise
- Accès à des informations à jour, plus récentes que la fin de l'entraînement du LM
- Plus souple et moins coûteux que le fine-tuning
- Meilleure explicabilité des réponses du LM car les sources utilisées peuvent être citées

R. Grin

Fine-tuning et RAG

15

15

- 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 LM avec, éventuellement, l'historique de la conversation

R. Grin

Fine-tuning et RAG

16

16

## Exemples d'utilisation du RAG

- Service client intelligent ; chatbot pour répondre aux clients de l'entreprise
- Interroger le LM sur des fichiers PDF qui contiennent les règles de fonctionnement d'une entreprise
- Résumer un document ou une page Web
- Interroger le modèle sur une vidéo YouTube
- 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

R. Grin

Fine-tuning et RAG

17

17

## 2 grandes phases pour le RAG

1. Ingestion des documents dans le système de RAG : nettoyage des documents, découpage en morceaux, ajout de métadonnées, enregistrement des morceaux
  2. Le système génère la réponse en retrouvant les documents les plus pertinents pour répondre à la question, pour les envoyer au LM avec la question
- Ces 2 phases sont **indépendantes** et se déroulent le plus souvent à des moments différents

R. Grin

Fine-tuning et RAG

18

18

## 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 répondre à la question, et les ajoute à la question
  - Génération : le tout (informations pertinentes + question + historique de la conversation éventuel) est envoyé au LM qui génère une réponse en utilisant ses capacités linguistiques et de compréhension

R. Grin

Fine-tuning et RAG

19

## Phases en images

### Phase 1 :



### Phase 2 :



R. Grin

Fine-tuning et RAG

(Source Oracle University)

20

## 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

R. Grin

Fine-tuning et RAG

21

## 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 <div> 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
- Supprimer les liens externes non pertinents

R. Grin

Fine-tuning et RAG

22

## 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
- ...

R. Grin

Fine-tuning et RAG

23

## Chunks (morceaux) (1/2)

- Les documents externes sont le plus 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 et des coûts
- De plus, les modèles d'embeddings ont une taille limite pour le texte à transformer en embedding

R. Grin

Fine-tuning et RAG

24

## 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
- Les morceaux peuvent se chevaucher légèrement pour éviter de couper au milieu d'une phrase, d'un paragraphe, ou d'une idée importante
- Essayer de tenir compte des phrases et des paragraphes (éviter de les couper au milieu) ; si le document est structuré (sections par exemple), essayer de tenir compte des structures

R. Grin

Fine-tuning et RAG

25

25

## Sélection des données

- Il n'est pas envisageable d'ajouter dans chaque prompt toutes les données ingérées Pourquoi ?
- Dans la phase 2, il faut donc un moyen de retrouver les données les plus pertinentes pour répondre à la question

R. Grin

Fine-tuning et RAG

26

26

## Pertinence des documents

- Plusieurs moyens de trouver les documents les plus pertinents pour répondre à une question :
  - Trouver les documents qui ont les embeddings les plus similaires à la question
  - Associer des mots-clés aux documents
  - Un mode hybride avec mots-clés et embedding
  - Une étape initiale (types précédents), suivie de techniques pour améliorer les résultats ou la rapidité des traitements (on verra, par exemple, le reranking, avec des modèles spécialisés pour évaluer la pertinence de documents)

R. Grin

Fine-tuning et RAG

27

27

## RAG avec embeddings

- La façon la plus classique de faire du RAG
- Les chunks sont transformés en embeddings qui sont enregistrés dans un entrepôt (magasin d'embeddings), avec les textes correspondants ou 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
- Les index de ce type de BDs permettent de faire des recherches de similarités très rapides

R. Grin

Fine-tuning et RAG

28

28

## Avantages 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 » et « thérapies contre les tumeurs » auront des embeddings proches, sans avoir aucun mot en commun
  - Pour répondre à une question sur les traitements du cancer, les documents traitant des thérapies contre les tumeurs seront ainsi retrouvés dans la phase de récupération

R. Grin

Fine-tuning et RAG

29

29

## Utilisation de mots-clés

- Des mots-clés peuvent permettre de retrouver les morceaux de document les plus pertinents
- Les mots-clés sont enregistrés dans les métadonnées des morceaux enregistrés
- Les mots-clés associés aux documents et aux questions peuvent être donnés explicitement mais ils peuvent être extraits automatiquement par des utilitaires spécialisés (KeyBERT ou keyword-spacy, par exemple)
- Pour la récupération on peut effectuer un premier filtrage « large » des morceaux en se basant uniquement sur les mots-clés, pour accélérer le traitement

R. Grin

Fine-tuning et RAG

30

30

## TF-IDF

- D'autres techniques comme TF-IDF (*Term Frequency-Inverse Document Frequency*) peuvent être utilisées pour accélérer la recherche des morceaux les plus pertinents
- Les mots d'un document ont un poids qui est 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

R. Grin

Fine-tuning et RAG

31

31

## 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, par exemple le titre du document qui contient le morceau, ou des mots-clés
3. Stockage des embeddings dans une BD vectorielle, avec le document correspondant et les éventuelles métadonnées

R. Grin

Fine-tuning et RAG

32

32

## Etapes récupération - génération

- 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 10) documents qui ont les embeddings les plus similaires à l'embedding de la question sont ajoutés au prompt, le plus souvent devant la question, les plus pertinents en premier
- Une étape optionnelle de « **reranking** » peut réordonner ces n documents plus finement dans le prompt
- Le tout est envoyé au LM qui répond à la question

R. Grin

Fine-tuning et RAG

33

33

## Température du LM

- Pour le RAG il est conseillé de réduire la température du LM 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

R. Grin

Fine-tuning et RAG

34

34

## Priorité informations récupérées

- Un LM considère le contenu du prompt comme fiable et donne souvent 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 » (si l'ajout est fait avant la question)
- D'autres techniques plus avancées peuvent aussi être utilisées

R. Grin

Fine-tuning et RAG

35

35

## 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

R. Grin

Fine-tuning et RAG

36

36

## 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 connaît les réponses
- Un autre LM peut aider à juger de la qualité des réponses fournies par le RAG

R. Grin

Fine-tuning et RAG

37

37

## Techniques avancées de RAG

R. Grin

Fine-tuning et RAG

38

38

## Indexation

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

R. Grin

Fine-tuning et RAG

39

39

## 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

R. Grin

Fine-tuning et RAG

40

40

## 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)

R. Grin

Fine-tuning et RAG

41

41

## 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 IA 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

R. Grin

Fine-tuning et RAG

42

42

## 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 LM

R. Grin

Fine-tuning et RAG

43

## 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



R. Grin

Fine-tuning et RAG

44

## 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 LM qui génère les tokens de la réponse
- RAG token model : souvent plus précis mais plus difficile à mettre en œuvre et plus coûteux ; chaque token est généré par le LM en tenant compte des documents les plus pertinents compte tenu des tokens déjà générés pour la réponse ; intéressant quand différentes parties de la réponse sont associées à des documents différents

R. Grin

Fine-tuning et RAG

45

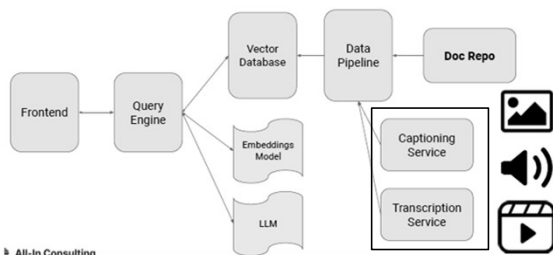
## RAG multi-modal

- 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 LMs 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

R. Grin

Fine-tuning et RAG

46



R. Grin

Fine-tuning et RAG

47

## 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 LM à utiliser les bonnes informations ; les 1<sup>ères</sup> étapes peuvent aussi influencer les étapes suivantes pour optimiser la recherche

R. Grin

Fine-tuning et RAG

48



## Exemple

- Une 1<sup>ère</sup> 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/>)

R. Grin

Fine-tuning et RAG

49

## Prompt engineering vs fine-tuning vs RAG

R. Grin

Fine-tuning et RAG

50

- Comparaison entre les 3 façons de prendre en compte des données sur lesquelles un LM n'a pas été entraîné

R. Grin

Fine-tuning et RAG

51

## Avantages et inconvénients (1/3)

- **Prompt engineering :**
  - simple, souple et rapide
  - **mais**
    - difficile de mettre à l'échelle (solution ad hoc pas généralisable ; peut dépendre des versions des LMs)
    - résultats aléatoires
    - le LM doit déjà connaître le domaine concerné par la question

R. Grin

Fine-tuning et RAG

52

## Avantages et inconvénients (2/3)

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

R. Grin

Fine-tuning et RAG

53

## 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 en réservant des informations sensibles à des utilisateurs privilégiés
  - permet de tester facilement plusieurs LMs
  - **mais**
    - traitement à répéter à chaque recherche (donc moins rapide que le fine tuning)
    - parfois difficile à mettre en œuvre
    - pas adapté pour certaines tâches, par exemple parler en imitant une personne ou répondre avec un certain style
    - lourd à gérer si documents très nombreux

R. Grin

Fine-tuning et RAG

54

## Que choisir ? (1/4)

- Le prompt engineering seul ne convient que si le LM connaît déjà le domaine de la question
- Sinon, il sera nécessaire d'ajouter des informations dans le prompt, et alors il faudra que ces informations ne soient ni trop volumineuses ni difficiles à trouver

R. Grin

Fine-tuning et RAG

55

55

## 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
  - reconnaître des patterns ou des sentiments ; par exemple, classer des emails envoyés par des clients : problèmes techniques, retours, satisfaction, ...

R. Grin

Fine-tuning et RAG

56

56

## Que choisir ? (3/4)

- Le RAG convient bien pour
  - des tâches nécessitant des informations actualisées en temps réel ; par exemple, assistant pour aider les clients à investir en bourse
  - obtenir des réponses basées sur des documents d'entreprise 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, ou recherche sur le Web ; par exemple recherche d'articles scientifiques à jour

R. Grin

Fine-tuning et RAG

57

57

## En résumé, que choisir ? (4/4)

- Le RAG permet d'**améliorer les réponses d'un modèle** grâce à des informations supplémentaires pertinentes et à jour
- Mais il **ne modifie pas fondamentalement le comportement ou le style linguistique du modèle** ; en ce cas, le fine-tuning sera meilleur
- Le Fine-tuning ne permet pas d'actualiser rapidement le modèle avec de nouvelles informations et il est coûteux en temps et en argent
- Le plus souvent, le RAG, ou une combinaison des 2 techniques, sera donc la meilleure solution

R. Grin

Fine-tuning et RAG

58

58

## Combinaison des techniques

- Le prompt engineering, 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, avec des techniques de prompt engineering (utiliser des templates)
  2. Ajouter du RAG s'il manque des informations pour bien répondre
  3. Faire du fine-tuning sur le LM de base du RAG pour le spécialiser dans un domaine ou/et si le ton, le style ou le format des réponses ne conviennent pas

R. Grin

Fine-tuning et RAG

59

59

## Et la sécurité ?

- Toutes ces techniques présentent le risque de fuites de données sensibles
- Il faut s'assurer que les données rendues accessibles au LM resteront privées :
  - ne seront pas utilisées par le LM ; lire attentivement les politiques d'utilisation des données du LM ; si nécessaire, signer un contrat avec le fournisseur du LM
  - n'apparaîtront pas en réponse à des questions posées par des utilisateurs malveillants
- Il faut anonymiser les données sensibles transmises au LM

R. Grin

Fine-tuning et RAG

60

60

## Bases de données vectorielles

R. Grin

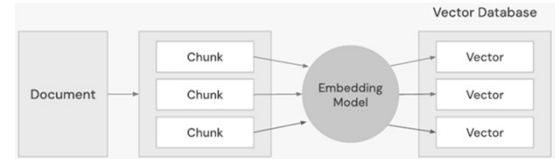
Fine-tuning et RAG

61

61

## 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 permettent accélérer les recherches



R. Grin

Fine-tuning et RAG

62

62

## 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
- Souvent intégration avec frameworks ML
- Support multimodal (texte, image, audio)

R. Grin

Fine-tuning et RAG

63

63

## Index

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

R. Grin

Fine-tuning et RAG

64

64

## 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)

R. Grin

Fine-tuning et RAG

65

65

## 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,...)

R. Grin

Fine-tuning et RAG

66

66

## 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 peuvent aussi faire des recherches hybrides qui combinent la précision et la rapidité des recherches par mots-clés et la compréhension du sens des recherches sémantiques

R. Grin

Fine-tuning et RAG

67

67

## 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

R. Grin

Fine-tuning et RAG

68

68



## BD vectorielle en mémoire

- Les TPs utilisent une BD en mémoire très simple InMemoryEmbeddingStore qui évite l'installation d'une « vraie » BD vectorielle
- A part la création de l'EmbeddingStore, tout le reste du code Java est identique pour toutes les BDs vectorielles, grâce à LangChain4j
- Dans une application d'entreprise il faut utiliser une des « vraies » BDs vectorielles supportées par LangChain4j pour ne pas perdre les embeddings entre 2 sessions de travail et profiter des index

R. Grin

Fine-tuning et RAG

69

69

## Produits supportés par LangChain4j

- Chroma
- Milvus
- Weaviate
- Pinecone
- Vespa (Yahoo!)
- Qdrant
- Elasticsearch (avec plugins spécialisés)
- AI Vector Search (Oracle)
- Redis
- Astra DB
- Cassandra



R. Grin

Fine-tuning et RAG

70

70

## Chroma

- BD vectorielle gratuite (<https://www.trychroma.com/home>) et open source (<https://github.com/chroma-core/chroma>)
- On peut l'installer en local ; le plus simple est d'utiliser une image Docker
- Fournit une API utilisable à distance mais le plus simple est de l'utiliser par l'intermédiaire de LangChain4j

R. Grin

Fine-tuning et RAG

71

71

## Chroma - Installation

- Le plus simple est de l'installer avec Docker en tapant cette commande dans un terminal :  
docker pull chromadb/chroma
- Une autre façon est d'aller dans Docker Desktop et de cliquer sur l'icône de Docker Hub pour chercher « chromadb » pour récupérer chromadb/chroma
- Dans Docker Desktop on peut alors lancer l'exécution de chromadb/chroma en cliquant sur le triangle « Run » ; dans la fenêtre qui s'ouvre, clic sur « Optional settings » et associer alors les ports 8000 de l'hôte et du container

R. Grin

Fine-tuning et RAG

72

72

## Chroma - Utilisation (1/2)

- Dans pom.xml :

```
<dependency>
  <groupId>dev.langchain4j</groupId>
  <artifactId>langchain4j-chroma</artifactId>
  <version>***</version>
</dependency>
```

R. Grin

Fine-tuning et RAG

73

## Chroma - Utilisation (2/2)

- Pour créer l'EmbeddingStore :

```
EmbeddingStore<TextSegment> embeddingStore =
  ChromaEmbeddingStore.builder()
    .baseUrl("http://localhost:8000/")
    .collectionName("ma-collection")
    .build();
```

- Tout le reste est identique à l'utilisation de InMemoryEmbeddingStore

R. Grin

Fine-tuning et RAG

74

## Code pour utiliser Chroma

```
EmbeddingStore<TextSegment> embeddingStore =
  ChromaEmbeddingStore.builder()
    .baseUrl("http://localhost:8000")
    .collectionName("ma-collection")
    .build();

// Tout le reste du code ne dépend pas de Chroma
EmbeddingModel embeddingModel =
  new AllMiniLmL6V2EmbeddingModel();

...
}
```

R. Grin

Fine-tuning et RAG

75

## RAG avec LangChain4j

- Généralités et RAG « facile »
- Classes et interfaces LangChain4j

R. Grin

Fine-tuning et RAG

76

## 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

R. Grin

Fine-tuning et RAG

77

## Phase 1 du RAG - Ingestion

- On indique
  - L'origine des données externes utilisées pour le RAG (base de données, documents d'entreprise,...)
  - Le magasin d'embeddings où seront entreposés les morceaux et leur embedding



R. Grin

Fine-tuning et RAG

78

## Exemple RAG facile (1/3)

```
List<Document> documents = FileSystemDocumentLoader.  
    loadDocuments("/home/langchain4j/documentation");  
EmbeddingStore<TextSegment> embeddingStore =  
    new InMemoryEmbeddingStore<>();  
EmbeddingStoreIngestor.ingest(documents, embeddingStore);
```

Pour que ce code fonctionne, il faut ajouter une dépendance pour langchain4j-easy-rag

R. Grin

Fine-tuning et RAG

79

## Phase 2 du RAG

- Création d'un assistant IA (avec AIServices) en précisant le magasin d'embeddings utilisé pour le RAG



R. Grin

Fine-tuning et RAG

80

## Exemple RAG facile (2/3)

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

Interface implémentée par l'assistant IA

R. Grin

Fine-tuning et RAG

81

## Exemple RAG facile (3/3)

```
ChatModel model = ...;  
Assistant assistant = AIServices.builder(Assistant.class)  
    .chatModel(model)  
    .chatMemory(MessageWindowChatMemory.withMaxMessages(10))  
    .contentRetriever(  
        EmbeddingStoreContentRetriever.from(embeddingStore))  
    .build();
```

```
String answer =  
    assistant.chat("Comment faire du RAG avec LangChain4j ?");
```

Il suffit de créer l'assistant et de le configurer en indiquant le modèle de chat et le magasin d'embeddings.  
Tout le reste est implémenté par LangChain4j, grâce à AIServices

R. Grin

Fine-tuning et RAG

82

## Types du code

- 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 (chunks) et métadonnées associées
- EmbeddingStore<TextSegment> : magasin pour embeddings de TextSegments
- EmbeddingStoreIngestor : Pipeline pour l'ingestion de documents dans un magasin d'embeddings
- EmbeddingStoreContentRetriever : retrouve dans un magasin d'embeddings les Contents (enveloppes de TextSegment) les plus similaires à la question

R. Grin

Fine-tuning et RAG

83

## Classes et interfaces pour RAG

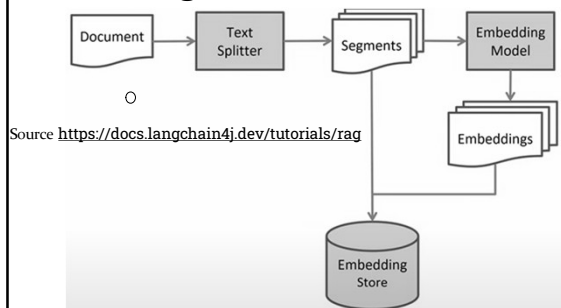
- Phase 1 : chargement (ingestion) des Documents
- Phase 2 : récupération (retrieval) des données pertinentes et augmentation du prompt

R. Grin

Fine-tuning et RAG

84

## RAG - Ingestion



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

R. Grin

Fine-tuning et RAG

85

## Exemple schématique simple

```

// Création d'un document avec un DocumentParser
DocumentParser documentParser = ... ;
Document document = UnDocumentLoader.loadDocument(
    cheminFichier, documentParser);
// Découper le document en morceaux (chunks)
DocumentSplitter splitter =
    DocumentSplitters.recursive(300, 20);
List<TextSegment> segments = splitter.split(document);
// Convertit les chunks en embeddings
EmbeddingModel embeddingModel = ... ;
List<Embedding> embeddings =
    embeddingModel.embedAll(segments).content();
// Ajoute embeddings et chunks dans magasin embeddings
EmbeddingStore<TextSegment> embeddingStore = ... ;
embeddingStore.addAll(embeddings, segments);
  
```

R. Grin

Fine-tuning et RAG

86

## Présentation des types (1/2)

- Document : un texte ; par exemple le contenu d'un fichier local ou d'une page Web
- DocumentParser : sait lire les données d'un certain type ; par exemple les données au format PDF ou les données au format Office Microsoft
- « *UnDocumentLoader* » : crée un Document à partir de données provenant d'une source (fichier local, page Web, ...) au format supporté par le DocumentParser
- DocumentSplitter : découpe un Document en une liste de chunks (*List<TextSegment>*)

R. Grin

Fine-tuning et RAG

87

## Présentation des types (2/2)

- EmbeddingModel : modèle d'embeddings qui peut convertir des mots, phrases, documents en Embeddings ; peut créer une *List<Embedding>* à partir d'une *List<TextSegment>*
- EmbeddingStore : magasin d'embeddings (BD vectorielle, ou autre type de magasin) qui contient des embeddings, leur segment et d'éventuelles métadonnées



R. Grin

Fine-tuning et RAG

88

- Création des documents

R. Grin

Fine-tuning et RAG

89

## Interface Document

- Package *dev.langchain4j.data.document*
- Représente un texte ; souvent le contenu d'un fichier local ou d'une page Web
- Le format du fichier 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

R. Grin

Fine-tuning et RAG

90

## Classe TextSegment

- Package `dev.langchain4j.data.segment`
- Représente un morceau de texte d'une entité plus large, avec ses éventuelles métadonnées
- Création avec méthodes `static` `from` auquel on passe le texte d'origine :
  - `from(String texte, Metadata metadata)` (`metadata` est un paramètre optionnel)
- Getters
  - `String text()` pour extraire le contenu
  - `Metadata metadata()`

R. Grin

Fine-tuning et RAG

91

91

## 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)

R. Grin

Fine-tuning et RAG

92

92

## Classe DocumentLoader

- Package `dev.langchain4j.data.document`
- Pour charger un document
- Une seule méthode `static` pour charger le document
 

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

R. Grin

Fine-tuning et RAG

93

93

## 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`, ...

R. Grin

Fine-tuning et RAG

94

94

## 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)

R. Grin

Fine-tuning et RAG

95

95

## Classes utilitaires

- Des classes permettent de charger des documents des types les plus utilisés, sans implémenter `DocumentSource` :
  - `FileSystemDocumentLoader`
  - `ClassPathDocumentLoader`
  - `UrlDocumentLoader`
  - `GitHubDocumentLoader`
  - `AmazonS3DocumentLoader`
  - ...


R. Grin

Fine-tuning et RAG

96

96





## Exemples création document

```
// 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 documentHtml =
    UrlDocumentLoader.load(url, new TextDocumentParser());
HtmlToTextDocumentTransformer transformer =
    new HtmlToTextDocumentTransformer(null, null, true);
Document documentText = transformer.transform(documentHtml);
```

R. Grin Fine-tuning et RAG 97

97

## FileSystemDocumentLoader

- Package `dev.langchain4j.data.document.loader`
- Classe utilitaire pour transformer en Documents des fichiers locaux dont on donne le chemin
- Méthodes static surchargées
  - `loadDocument` pour charger un document
  - `loadDocuments` pour charger les documents d'un répertoire
  - `loadDocumentsRecursively` 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`)

R. Grin Fine-tuning et RAG 98

98

## 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);
```

R. Grin Fine-tuning et RAG 99

99

## 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()`

R. Grin Fine-tuning et RAG 100

100

## UrlDocumentLoader

- 2 méthodes `load` static surchargées :
  - `Document load(String url, DocumentParser)`
  - `Document load(java.net.URL url, DocumentParser)`
- Le `DocumentParser` à utiliser est le plus souvent un `TextDocumentParser`
- Si on veut ensuite ne pas récupérer les balises HTML, il faut transformer le Document avec un `HtmlToTextDocumentTransformer` (voir [exemple déjà donné](#) sur `DocumentLoader`)

R. Grin Fine-tuning et RAG 101

101

## HtmlToTextDocumentTransformer

- Implémente `DocumentTransformer` (une seule méthode `Document transform(Document)`)
- Le constructeur peut prendre en paramètre
  - Un sélecteur CSS (type `String`), par exemple « `#page-content` » pour extraire le texte d'une partie de la page
  - Une `Map<String, String>` de sélecteurs CSS. Une clé peut être « `titre` » et sa valeur « `#page-title` » pour extraire le texte de l'élément d'id « `#page-title` » et l'enregistrer en métadonnée de clé « `titre` »
  - Un `boolean` qui indique si les liens de la page (pas les contenus des liens) doivent être inclus dans le texte extrait

R. Grin Fine-tuning et RAG 102

102

- Découper les Documents en morceaux

R. Grin

Fine-tuning et RAG

103

103

## Interface DocumentSplitter

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

R. Grin

Fine-tuning et RAG

104

104

## 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

R. Grin

Fine-tuning et RAG

105

105

## 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

R. Grin

Fine-tuning et RAG

106

106

## 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);
```

R. Grin

Fine-tuning et RAG

107

107



## Métadonnées

- Les splitters de `LangChain4j` ajoutent automatiquement aux segments générés une métadonnée « `index` » ; sa valeur est la position du segment dans le document original
- Il est possible d'ajouter d'autres métadonnées à tous les segments générés en les ajoutant au document, avant le découpage en morceaux
- Il est aussi possible d'ajouter des métadonnées à un segment particulier, après le découpage

R. Grin

Fine-tuning et RAG

108

108

## Exemple (1/2)

```
String path = "example.txt";

// Ajout métadonnées lors du chargement du document
Document document =
    FileSystemDocumentLoader.loadDocument(
        Paths.get(path),
        Metadata.from("file_name", "example.txt"));

// Splitter récursif (ajoute "index" automatiquement)
DocumentSplitter splitter =
    new RecursiveDocumentSplitter(300, 0);
List<TextSegment> segments = splitter.split(document);
```

R. Grin

Fine-tuning et RAG

109

109

## Exemple (2/2)

```
// Affichage des métadonnées de chaque segment
for (TextSegment segment : segments) {
    System.out.println("Segment : \n" + segment.text());
    System.out.println("Métadonnées :");
    segment.metadata().asMap().forEach(
        (key, value) ->
            System.out.println("  " + key + " = " + value));
    System.out.println("-----");
}
```

R. Grin

Fine-tuning et RAG

110



110

## Utilisation des métadonnées

- 2 exemples sont donnés dans la documentation de LangChain4j
- **Filtrage par métadonnées :**  
[https://github.com/langchain4j/langchain4j-examples/blob/main/rag-examples/src/main/java/\\_3\\_advanced/\\_05\\_Advanced\\_RAG\\_with\\_Metadata\\_Filtering\\_Examples.java](https://github.com/langchain4j/langchain4j-examples/blob/main/rag-examples/src/main/java/_3_advanced/_05_Advanced_RAG_with_Metadata_Filtering_Examples.java)
- **Ajouter le nom du fichier dans les métadonnées :**  
[https://github.com/langchain4j/langchain4j-examples/src/main/java/\\_3\\_advanced/\\_04\\_Advanced\\_RAG\\_with\\_Metadata\\_Example.java](https://github.com/langchain4j/langchain4j-examples/blob/main/rag-examples/src/main/java/_3_advanced/_04_Advanced_RAG_with_Metadata_Example.java)

R. Grin

Fine-tuning et RAG

111

111

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

R. Grin

Fine-tuning et RAG

112

112

## Interface EmbeddingModel

- Package dev.langchain4j.model.embedding
- Modèle d'embeddings 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)

R. Grin

Fine-tuning et RAG

113

113

## Exemples création EmbeddingModel

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

R. Grin

Fine-tuning et RAG

114

114

## Obtenir un embedding

```
TextSegment segment = TextSegment.from("un texte ...");
Response<Embedding> response =
    embeddingModel.embed(segment)
Embedding embedding = response.content();
```

Pourquoi Response ?  
Pourquoi ne pas obtenir directement un embedding ?

R. Grin

Fine-tuning et RAG

115

## Classe Response<T>

- Package dev.langchain4j.model.output
- Représente une réponse de plusieurs types de modèle (chat, embedding, image, modération, scoring)
- 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

R. Grin

Fine-tuning et RAG

116

## 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
- 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
- Implémenté par des classes adaptées aux BD vectorielles (Neo4j, Chroma, MongoDB,...) et par la classe InMemoryEmbeddingStore

R. Grin

Fine-tuning et RAG

117

## 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)
```



R. Grin

Fine-tuning et RAG

118

## 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 pour la recherche des embeddings similaires

R. Grin

Fine-tuning et RAG

119

## Création EmbeddingStore

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

R. Grin

Fine-tuning et RAG

120

## Exemple ajout embeddings

```

EmbeddingModel embeddingModel =
    new AllMiniLmL6V2EmbeddingModel();
Response<List<Embedding>> response =
    embeddingModel.embedAll(segments);
List<Embedding> listeEmbedding = response.content();

EmbeddingStore<TextSegment> embeddingStore =
    new InMemoryEmbeddingStore<>();
embeddingStore.addAll(listeEmbedding, segments);

```

R. Grin Fine-tuning et RAG 121

121

## Classe EmbeddingSearchRequest

- Représente une requête dans un EmbeddingStore
- Builder et constructeur avec les paramètres de type
  - Embedding : embedding dont on cherche les embeddings similaires
  - Integer : nombre maximum d'embeddings à retourner (3 par défaut)
  - Double : seulement les embeddings avec un score  $\geq$  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

R. Grin Fine-tuning et RAG 122

122

## 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>>

R. Grin Fine-tuning et RAG 123

123

## EmbeddingMatch<T>

- Classe qui représente un embedding retourné par une recherche, avec son score de pertinence (dérivé de la distance cosinusoidale) 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

R. Grin Fine-tuning et RAG 124

124

## 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());
});

```

R. Grin Fine-tuning et RAG 125

125

## Composants du RAG

- EmbeddingStoreIngestor (« ingesteur », collecteur de données), pipeline pour la phase 1 : découpe un document en morceaux, les enregistre dans un magasin d'embeddings
- RetrievalAugmentor (« augmenteur ») : recherche les informations pertinentes et les ajoute à la question ; utilise en particulier un ContentRetriever (récupérateur)
- Générateur : LM qui génère la réponse en utilisant ses capacités linguistiques et les informations pertinentes retrouvées

R. Grin Fine-tuning et RAG 126

126

## EmbeddingStoreIngestor (1/3)

- 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 et la transformation des segments après le découpage en segments
- Les informations nécessaires aux traitements sont indiquées à la création de l'ingestor

R. Grin

Fine-tuning et RAG

127

127

## EmbeddingStoreIngestor (2/3)

- La création de l'ingestor se fait par un builder ou un constructeur ; on indique le modèle et le magasin pour les embeddings, le splitter, des éventuels transformeurs pour le document ou/et les segments :
  - `DocumentTransformer` peut transformer le document (nettoyage, filtrage, enrichissement, extraire le texte du contenu HTML, métadonnées, ...) ; optionnel
  - `DocumentSplitter` ; optionnel
  - `TestSegmentTransformer` peut transformer les segments, par exemple pour ajouter des métadonnées
  - `EmbeddingModel`
  - `EmbeddingStore`

R. Grin

Fine-tuning et RAG

128

128

## EmbeddingStoreIngestor (3/3)

- L'ingestion se lance avec une des méthodes `ingest`
- La méthode `ingest` prend en paramètre un `Document` ou une liste de `Documents`
- 2 méthodes `ingest` sont static ; elles permettent d'indiquer un `EmbeddingStore<TextSegment>` s'il n'a pas été donné lors de la création de l'`EmbeddingStoreIngestor`
- La valeur retour des méthodes `ingest` est de type `IngestionResult`, ce qui permet de récupérer un `TokenUsage` (qui donne le nombre de tokens utilisés pendant la création des embeddings)

R. Grin

Fine-tuning et RAG

129

129

## Exemple ingestion avec EmbeddingStoreIngestor

```
DocumentSplitter splitter =
    DocumentSplitters.recursive(600, 0);
EmbeddingModel embeddingModel =
    new AllMiniLmL6V2EmbeddingModel();
EmbeddingStore<TextSegment> embeddingStore =
    new InMemoryEmbeddingStore<>();
EmbeddingStoreIngestor ingestor =
    EmbeddingStoreIngestor.builder()
        .documentSplitter(splitter)
        .embeddingModel(embeddingModel)
        .embeddingStore(embeddingStore)
        .build();
Document document = FileSystemDocumentLoader
    .loadDocument(path, new TextDocumentParser());
ingestor.ingest(document);
```

R. Grin

OpenAI

130

130

## Exemple avec transformations

```
EmbeddingStoreIngestor ingestor =
    EmbeddingStoreIngestor.builder()
        .embeddingModel(embeddingModel)
        .embeddingStore(embeddingStore)
        .documentSplitter(documentSplitter)
        .documentTransformer(document -> {
            document.metadata().put("user", "...");
            return document;
        })
        .textSegmentTransformer(segment -> TextSegment.from(
            segment.metadata().getString(nom_fichier) + "\n"
            + segment.text(), segment.metadata()))
        .build();

List<Document> docs = FileSystemDocumentLoader
    .loadDocumentsRecursively(path);
ingestor.ingest(docs);
```

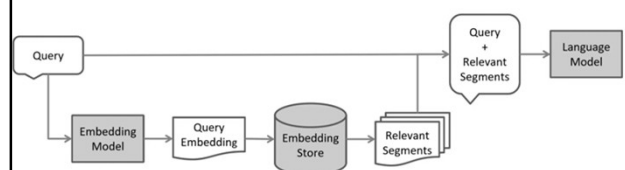
R. Grin

Fine-tuning et RAG

131

131

## RAG - récupération et augmentation

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

R. Grin

Fine-tuning et RAG

132

132

## Exemple schématique simple

```
ContentRetriever contentRetriever =
    EmbeddingStoreContentRetriever.builder()
        .embeddingStore(embeddingStore)
        .embeddingModel(embeddingModel)
        .maxResults(10) // Nombre de résultats à retourner
        .minScore(0.75) // Score minimal pour similarité
        .build();
ChatMemory chatMemory = ... ;
Assistant assistant = AIServices.builder(Assistant.class)
    .chatModel(model)
    .chatMemory(chatMemory)
    .contentRetriever(contentRetriever)
    .build();
System.out.println(assistant.chat("..."));
```

R. Grin

Fine-tuning et RAG

133

133

## Récupération

- Dans les cas les plus simples, la récupération des morceaux de texte les plus pertinents utilise un `ContentRetriever`
- Celui-ci utilise un `RetrievalAugmentor` par défaut
- Pour les cas les plus complexes, utiliser un `RetrievalAugmentor`
- Il faut choisir ; une erreur sera lancée si on indique à la fois un `ContentRetriever` et un `RetrievalAugmentor` pour l'assistant

R. Grin

Fine-tuning et RAG

134

134

## Interface ContentRetriever

- Retrouve des contenus (des `TextSegments`) d'une source de données, pertinents pour 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 les contenus retrouvés, triés par pertinence, les plus pertinents en premiers
 

```
List<Content> retrieve(Query)
```

R. Grin

Fine-tuning et RAG

135

135

## Implémentations 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) ➡

R. Grin

Fine-tuning et RAG

136

136

## 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()`

R. Grin

Fine-tuning et RAG

137

137

## Classe Metadata

- Package `dev.langchain4j.rag.query` ; ne pas confondre avec la classe du package `dev.langchain4j.data.document`
- Métadonnées utilisées pour retrouver des informations (utilisée par `Query`) ou pour augmenter une question posée par l'utilisateur (avant d'envoyer la requête au LM)
- Constructeur (ou méthode `static from`) avec les paramètres de type `UserMessage`, `Object` (`chatMemoryId` 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()`

R. Grin

Fine-tuning et RAG

138

138

## 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 (images, audio, ...)
- Constructeur avec un paramètre de type `String` ou `TextContent` ; une méthode `static from` avec les mêmes paramètres permet aussi de créer une instance
- Méthode `TextSegment textSegment()` pour récupérer le `TextSegment`

R. Grin

Fine-tuning et RAG

139

139

## EmbeddingStoreContentRetriever

- Classe du package `dev.langchain4j.rag.content.retriever`
- `EmbeddingStoreContentRetriever` : retrouve depuis un `EmbeddingStore` ; par défaut, retrouve les 3 contenus 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, ...)

R. Grin

Fine-tuning et RAG

140

140

## 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`

R. Grin

Fine-tuning et RAG

141

141

## Exemple de filtre dynamique

```
Function<Query, Filter> filterByUserId =
    (query) -> metadataKey("userId")
        .isEqualTo(query.metadata()
            .chatMemoryId().toString());

...
ContentRetriever contentRetriever =
    EmbeddingStoreContentRetriever.builder()
        .embeddingStore(embeddingStore)
        .embeddingModel(embeddingModel)
        // La recherche sera limitée aux segments qui
        // appartiennent à l'utilisateur en cours
        .dynamicFilter(filterByUserId)
        .build();
```

R. Grin

Fine-tuning et RAG

142

142

## WebSearchContentRetriever

- Package `dev.langchain4j.rag.content.retriever`
- Retrouve depuis le Web, en utilisant un `WebSearchEngine`
- Méthodes du builder pour le nombre maximum de résultats et pour le `WebSearchEngine`
- La seule méthode est celle de l'interface `ContentRetriever` : `List<Content> retrieve(Query query)`

R. Grin

Fine-tuning et RAG

143

143

## 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`

R. Grin

Fine-tuning et RAG

144

144



## 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])`

R. Grin

Fine-tuning et RAG

145

145

## Interface WebSearchResults

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

R. Grin

Fine-tuning et RAG

146

146



## Ce que peut faire un RAG avancé

- Chunks enregistrés dans plusieurs magasins d'embeddings
- Données recherchées sur le Web
- Recherche des documents dans toutes les sources ou bien privilégier certaines sources, suivant le contenu des questions
- Reranking pour réordonner plus finement les informations récupérées
- Utilisation d'un filtrage et classement différent, suivant le type de source utilisé
- ...

R. Grin

Fine-tuning et RAG

147

147

## RetrievalAugmentor

- **RetrievalAugmentor** est l'élément de haut niveau responsable de coordonner les étapes de la phase 2 du RAG avancé :
  1. Prend une requête utilisateur (Content)
  2. Utilise un `ContentRetriever` pour trouver les documents pertinents
  3. Utilise un `ContentInjector` (par exemple un `DefaultContentInjector`) pour construire le prompt enrichi
  4. Passe ce prompt à un modèle (ou à un agent)

R. Grin

Fine-tuning et RAG

148

148

## RetrievalAugmentor

- Schéma des différentes étapes (optionnelles) :
  - Récrire le prompt, par exemple avec des techniques de prompt engineering ou le découper en plusieurs `Query`s
  - Router vers les bons récupérateurs pour trouver les informations pertinentes
  - Agréger les informations retrouvées
  - Ajouter les informations au prompt
  - Passer le prompt au LLM

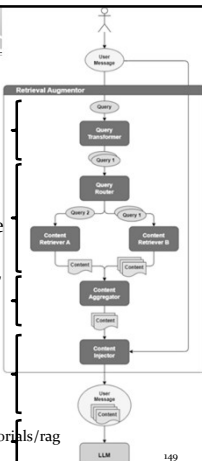
Source image <https://docs.langchain4j.dev/tutorials/rag>

R. Grin

Fine-tuning et RAG

149

149



## Utilisation RetrievalAugmentor

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

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

R. Grin

Fine-tuning et RAG

150

150

## 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

R. Grin

Fine-tuning et RAG

151

## Class AugmentationRequest

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

R. Grin

Fine-tuning et RAG

152

## 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

R. Grin

Fine-tuning et RAG

153

## 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`

R. Grin

Fine-tuning et RAG

154

## 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`

R. Grin

Fine-tuning et RAG

155

## 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/>

R. Grin

Fine-tuning et RAG

156

## 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 LM, 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` et `LanguageModelQueryRouter`

157

## 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

158

## LanguageModelQueryRouter

- Choix du ou des `ContentRetrievers` fait par un LM
- Si la connexion au LM ou si le LM 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`
- 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 LM
  - une stratégie de repli

159

## Exemple de routage

```
// 2 ContentRetrievers pour retrouver les informations
// à ajouter au prompt
// Descriptions associées à chacun des ContentRetriever
Map<ContentRetriever, String> descriptions =
    new HashMap<>();
descriptions.put(contentRetriever1,
    "Conditions pour départ en retraite");
descriptions.put(contentRetriever2,
    "Règles pour travail à distance");

QueryRouter queryRouter =
    new LanguageModelQueryRouter(chatModel,
        descriptions);
```

Les descriptions aideront le LM à choisir le bon `ContentRetriever` pour chaque question posée

160

## QueryRouter personnalisé

- Il faut implémenter l'interface `QueryRouter`
- Donc implémenter la méthode  
`Collection<ContentRetriever> route(Query query)`
- Le paramètre est la question envoyée au LM ; la méthode retourne le ou les `ContentRetriever` qui seront utilisés pendant la phase de récupération du RAG
- 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

161

## 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.chat(question);
        if (reponse.toLowerCase().contains("non")) {
            // Pas de RAG (aucun ContentRetriever utilisé)
            return Collections.emptyList();
        } else {
            return Collections.singletonList(contentRetriever);
        }
    }
}
```

162

## 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 `DefaultContentAgregator`,  
`ReRankingContentAgregator`

R. Grin

Fine-tuning et RAG

163

163

## DefaultContentAgregator

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

R. Grin

Fine-tuning et RAG

164

164

## ReRankingContentAgregator

- 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

R. Grin

Fine-tuning et RAG

165

165

## 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)`

R. Grin

Fine-tuning et RAG

166

166

## 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();
```

R. Grin

Fine-tuning et RAG

167

167

## Exemple pour reranking (2/2)

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

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

R. Grin

Fine-tuning et RAG

168

168

## Interface ContentInjector

- Package `dev.langchain4j.rag.content.injector`
- Injecte des Contents dans un `UserMessage`
- Permet d'ajouter au prompt les informations pertinentes récupérées dans les étapes précédentes
- Une seule méthode  
`ChatMessage inject(List<Content> contents, ChatMessage chatMessage)`

R. Grin

Fine-tuning et RAG

169

169

## Classe DefaultContentInjector

- Package `dev.langchain4j.rag.content.injector`
- Implémentation par défaut de `DefaultContentInjector`
- Ajoute les Contents à la fin du `UserMessage`, dans l'ordre de la liste (voir `DEFAULT_PROMPT_TEMPLATE`)
- Constructeurs et pattern builder pour créer une instance ; on peut indiquer un template (avec variables `{{userMessage}}` et `{{contents}}`) et une liste de clés pour métadonnées qui seront ajoutées à chaque segment (`Content.textSegment()`)

R. Grin

Fine-tuning et RAG

170

170

## Références

R. Grin

Fine-tuning et RAG

171

171

- RAG explained : embedding, sentence BERT, vector database par Umar Jamil : <https://www.youtube.com/watch?v=rhZgXNdhWDY>
- RAG et fine-tuning : <https://learn.microsoft.com/fr-fr/azure/developer/ai/augment-llm-rag-fine-tuning>
- RAG avancé : <https://learn.microsoft.com/fr-fr/azure/developer/ai/advanced-retrieval-augmented-generation>
- Quelques exemples d'utilisation du RAG : <https://www.cohesity.com/fr/glossary/retrieval-augmented-generation-rag/>

R. Grin

Fine-tuning et RAG

172

172

- Méthodes avancées de RAG : <https://fr.linkedin.com/pulse/m%C3%A9thodes-avanc%C3%A9es-de-g%C3%A9n%C3%A9ration-augment%C3%A9e-par-rag-ilan-bompuis-ynoof>

R. Grin

Fine-tuning et RAG

173

173