atol Conseils & Développements

Atelier application des Large Language Models pour la création d'un chatbot

03/10/2024

GEVREY-CHAMBERTIN

DIJON
Maret / Carnot

BESANÇON

PARIS

LYON

Les intervenants de cet atelier

atol Conseils & Développements

Olivier Beltramo-Martin
Ingénieur IA @ Atol CD

Objectifs de cet atelier

atol Conseils & Développements

1. **Conceptualiser** et matérialiser le flux de données d'une application Retrieval-Augmented Generation (RAG)

2. Identifier les **briques technologiques** nécessaires et se familiariser avec quelques options

3. Implémenter une application **RAG bout-en-bout**

4. Comprendre les l**imites et les écueils** des LLMs

5. Prendre connaissance des concepts de **few-shot learning, fine-tuning** et autres

Déroulé de l'atelier

1. **Le cycle de vie d'un LLM**
   Quelques concepts pour aborder la pratique

2. **Intégration d'un LLM dans une application**
   Déploiement et sensibilité du LLM

3. **Illustration du few-shot prompting**
   Ajustement du modèle via le prompt

4. **Application RAG à la volée**
   Les étapes du RAG : concepts et implémentation

5. **Pour aller plus loin**
   Concepts de Query expansion, fine-tuning, LLM-as-a-judge, RLHF

Le cycle de vie d’un LLM

Concept du LLM : définitions

Un LLM est un **réseau de neurones profond** pour la modélisation du langage, comme gpt-4 qui est basé sur une architecture **transformers** (Google 2017).

Les LLMs permettent de délivrer une réponse (texte, image, vidéo, audio) à une requête transmise via un **prompt**

Les **modèles multimodaux** sont entrainés à interpréter une requête contenant des données **hétérogènes** comme du texte + image pour générer un contenu

Le modèle Llama3.1:405B possède 450 milliards de paramètres et a été entraîné sur 15 000 milliards de tokens issus de sources publiques avec une **fenêtre contextuelle** de 128k tokens
au prix de 30.84 millions d'heures GPU et 8930 tonnes de CO_2

Concept du LLM : apprentissage

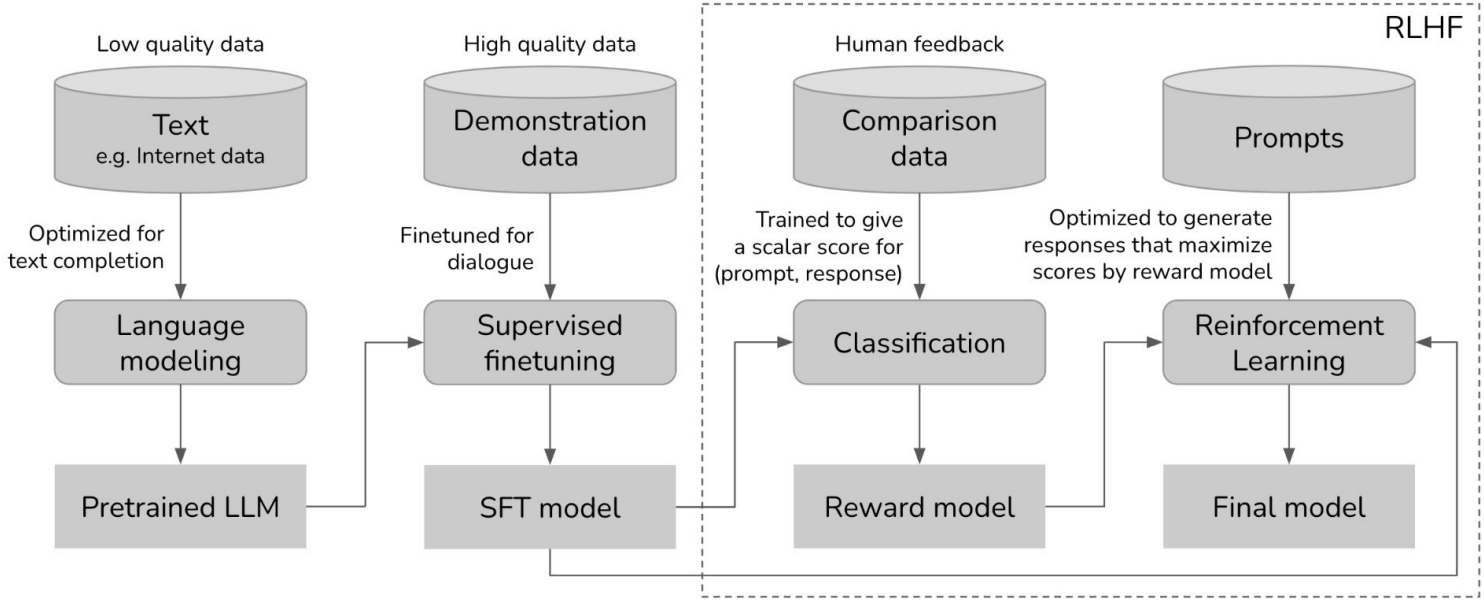
Les données d'entraînement (texte) :

- données web préalablement filtrées (langues, doublons, métadonnées, bias, cohérence, erreurs syntaxiques)
- tokenisation
- normalisation (lemmatisation ou racinisation)
- vectorisation (embedding)

3 phases d'entraînement :

- **Pré-entraînement** : apprentissage non-supervisé sur des données publiques pour la prédiction de séquence (GPT) ou de mots manquants (BERT)
- **Ajustement** : ajustement de l'apprentissage par supervision sur des cas d'usage ciblés
- **Renforcement** : amélioration continue du modèle sur la base du feedback humain

Concept du LLM : apprentissage



Scale May '23	**>1 trillion** tokens	**10K - 100K** (prompt, response)	**100K - 1M** comparisons (prompt, winning_response, losing_response)	**10K - 100K** prompts
Examples **Bolded**: open sourced	GPT-x, Gopher, **Falcon**, LLaMa, **Pythia**, **Bloom**, **StableLM**	**Dolly-v2, Falcon-Instruct**		InstructGPT, ChatGPT, Claude, **StableVicuna**

Intégration d’un LLM dans une application

Ollama

https://ollama.com/

Framework open-source pour le déploiement local de LLMs

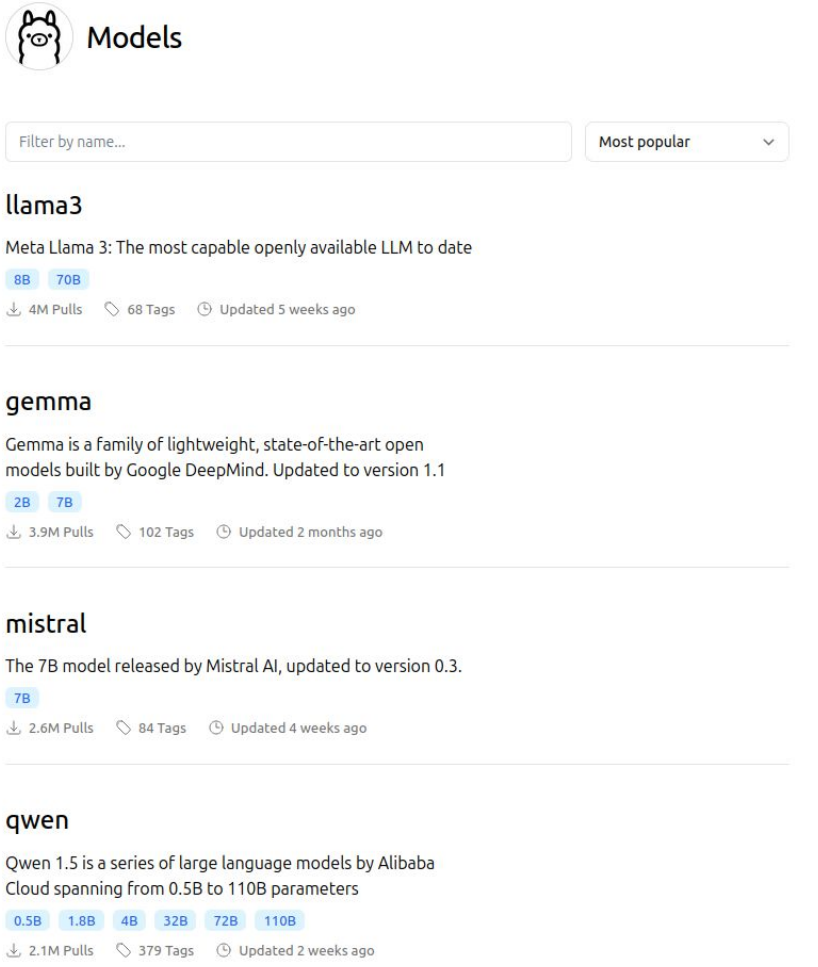
Permet d'interagir avec un LLM via le terminal

Disponible en tant que bibliothèque Python et Javascript

```
$ ollama pull llama3:8b
$ ollama list
$ ollama run llama3:8b
>>>
```

Pratique :

- tester différents prompts et modèles
- apprécier les différences de réponses et de vitesses d'exécution entre modèles
- tester la sensibilité du modèle au prompt



Mise en pratique

atol Conseils & Développements

- Lancer chat via Ollama

```
$ ollama run llama3:8b
```

Ollama

01-chat-ollama.py

- Bibliothèque python intégrant ollama
- Sélection du modèle et de ses entrées
- Introduction des **rôles**
  - “**system**” : les instructions supposément indélébiles
  - “**user**” : la requête de l’utilisateur
- **temperature** : variabilité de la réponse

Pratique :
- Créer un client ollama
- Évaluer l’impact des rôles
- Évaluer l’impact de la température du modèle

```
from ollama import Client


client.chat(
    model=model,
    messages=messages,
    options = {key1=val1, ...}
    stream=True
  ),
```

Mise en pratique Ollama

atol Conseils & Développements

- Création Client Ollama
- Construction message
- Appel model
- Extraction info dans la réponse Ollama
- Influence du system_prompt et température

```python
client = Client(host=args.ollama_url)


messages = [
  {"role": "system", "content": system_prompt},
  {"role": "user", "content": question},
]

return map(
  lambda x: x["message"]["content"],
  client.chat(
    model=model,
    messages=messages,
    options={"temperature": temperature},
    stream=True,
  ),
)
```

Illustration du few-shot prompting

Concept du few-shot prompting

02-few-shot-ollama.py

- Pre-trained LLM → modèle généraliste
- Instruct LLM → modèle plus spécifique sur une tâche

Comment améliorer la connaissance du modèle sans le réentrainer ?

=› lui donner des exemples et le laisser généraliser via sa connaissance propre

- différent du few-shot learning pour lequel le modèle est réentrainé pour conserver cette nouvelle connaissance

```
prompt="""
        Tu dois rédiger une description d'un produit donné par l'utilisateur en une phrase et en mettant en avant les qualités de ce produit. Si l'utilisateur émet une requête qui ne concerne pas un produit, réponds que tu es un Assistant et que tu ne peux pas répondre à sa question. Voici quelques exemples

Produit : xxxx
Description : xxxx

Produit : xxxx
Description : xxxx

Produit : xxxx
Description : xxxx

Répond par la description uniquement.
        """
```

Mise en pratique few-shot prompting

- Création Client Ollama
- Construction du system_prompt
- Appel model
- Extraction info dans la réponse Ollama

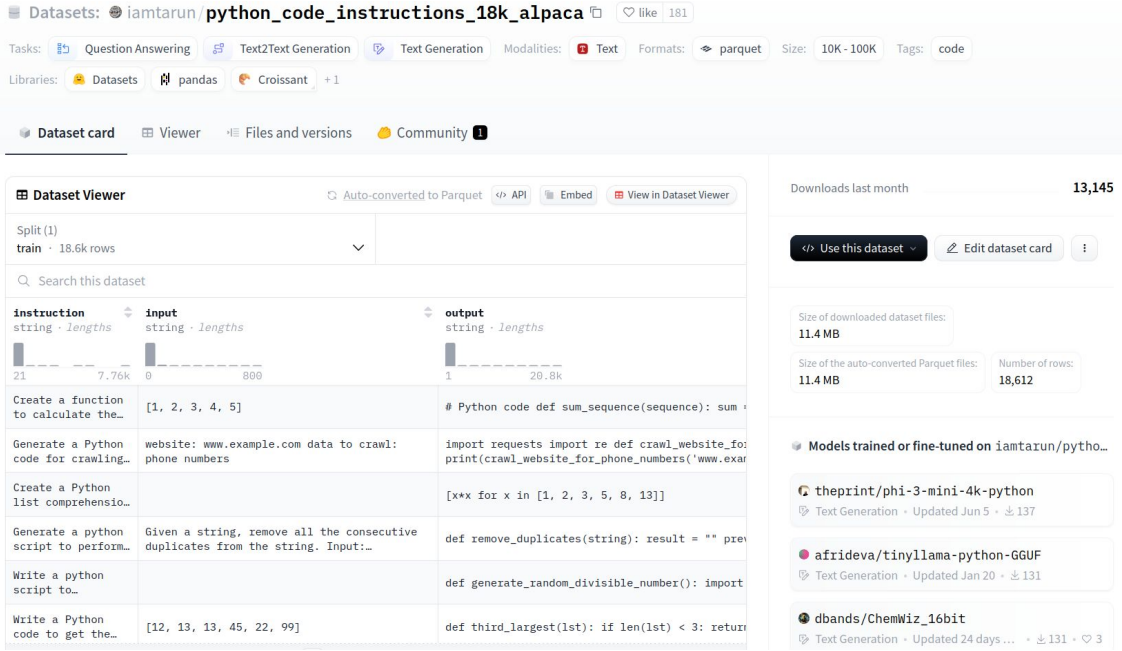
```
Tu es un assistant répondant à des questions sur du matériel informatique uniquement
vendus par notre société comme les casques audio, les souris ergonomiques, les écrans,
les ordinateurs portables, les claviers, ainsi que tous les éléments de connectiques et
les stations d'accueil pour ordinateurs portables.
Tu dois rédiger une description d'un produit donné par l'utilisateur en une phrase et
en mettant en avant les qualités de ce produit.  Voici quelques exemples :

Produit : Casque audio
Description : Notre casque audio offre un confort sonore sans précédent et une
performance acoustique de pointe
grâce à notre technologie UltraDMX.

Produit : Souris ergonomique
Description : Notre souris ergonomique s'adapte en temps-réel à votre posture pour vous
offrir un confort maximal
grâce à notre technologie MMXmorph.

Produit :Écran 512k
Description : Notre écran propose la plus grande résolution sur le marché pour une
immersion unique dans la réalité augmentée grâce à notre technologie AboveVision+
```

Introduction à HuggingFace

21-datasets-huggingface.py

huggingface.co/datasets

- Hugging Face est une bibliothèque open-source de modèles et de datasets
- Utilisation de la bibliothèque python datasets possible
- Inclusion des exemples dans le system prompt ou dans le user prompt



Pratique :

- Charger les données via la bibliothèque datasets
- Structurer et transférer les exemples dans prompt
- Évaluer la nouvelle connaissance du modèle et sa capacité à généraliser

Mise en pratique

atol Conseils & Développements

- Compléter user message avec contexte d'instructions à suivre
- Voir la capacité de généralisation

```
# load data using HuggingFace datasets API
ds = load_dataset("iamtarun/python_code_instructions_18k_alpaca",
split="train")[0:n_rows]

ds = ''.join(["{Question :\n"+ds["instruction"][x] + "\nAnswer
:\n{"+ds["output"][x]+"}\n\n" for x in range(n_rows)])


system_prompt = f"""
You are a Python instructor only that replies to questions and query about Python
programming.
Use the following examples within brackets to respond to the user's query
{ds}
"""


messages = [
 {"role": "system", "content": system_prompt},
 {"role": "user", "content": question},
]
```

Et si le modèle n'a pas l'information

Ajouter des données de contexte

03-context-ollama.py

- Enrichir la connaissance du LLM avec un contexte fourni à la volée
- Contexte = documents (pdf, html, ..) issus d'un nouveau corpus à transmettre

  → limitation : taille du contexte
  → réduire : quid de la pertinence de l'information fournie
- Intégrer le contexte dans le prompt

```
system_prompt = """
Tu es un chatbot qui répond à des questions en
utilisant uniquement les données de contexte fournies
et avec un ton formel"""


user_prompt = f"""
Context: {context_data}
Question: {question}
Réponse:
"""
```

Mise en pratique Ollama

atol Conseils & Développements

- Intégrer dans le user_prompt des informations que le modèle n'a pas
- Construire string par formatage / concaténation manuelle
- Voir les impacts : temps de réponse, informations "connues"

```
   system_prompt = """
   Tu es un chatbot qui répond à des questions en utilisant uniquement les données de contexte fournies et avec un ton formel
   Tu réponds toujours en français, quelque soit la langue dans laquelle la requête utilisateur est donnée.
   Lorsque le contexte ne fournit pas d'informations sur la question posée, réponds que tu n'as la réponse.
   Par exemple, si la question concerne une recette de cuisine, réponds que tu ne sais pas.
   """
   user_prompt = f"""
   Context: {_context_data}
   Question: {question}
   Réponse:
   """
   messages = [
     {"role": "system", "content": system_prompt},
     {"role": "user", "content": user_prompt},
   ]
```

Application RAG à la volée

Application RAG : le concept

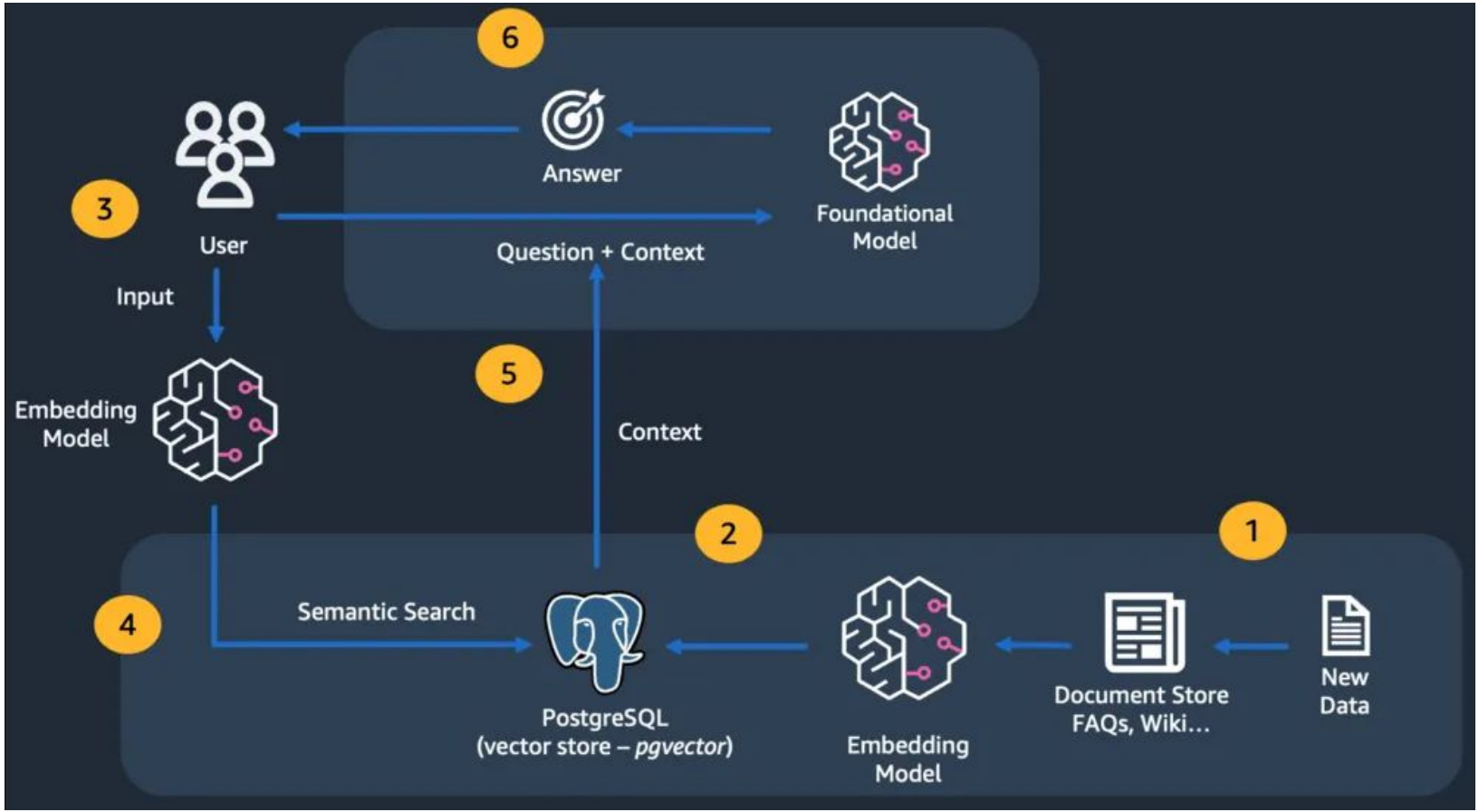
Comment utiliser un LLM pour lui faire générer du contenu sur un sujet qui n'est pas intégré dans ses données d'entrainements, typiquement un corpus documentaire privé ?

Principe de l'application RAG : Retrieval-Augmented Generation qui fonctionne en deux étapes :

- **Retrieval** : identifier les éléments d'un corpus documentaire (pdf, html, …) qui répondent à la requête de l'utilisateur
  - Formattage des données = > **chargement et traitement**
  - Identification et sélection des sources utiles à faire au préalable potentiellement très long si il y a beaucoup de sources volumineuses = > **vectorisation**
  - Gestion de la confidentialité = > **indexation et metadata avec BDD vectorielle**

- **Generation** : générer une réponse via les capacités internes du modèles et des éléments de contexte fournis
  - Réutilisation du principe de few-shot prompting
  - le contexte est donné dans le prompt, mais ne peut pas fonctionner pour le grands volumes (fenêtre contextuelle limitée)

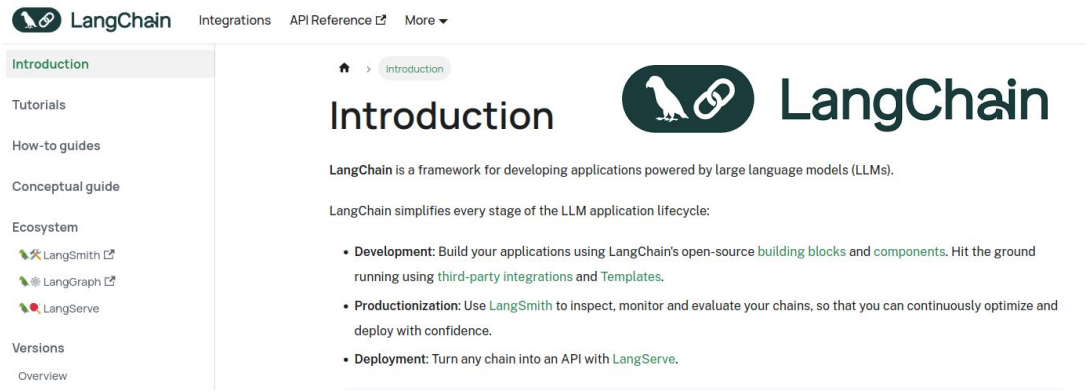
Application RAG : le concept



Application RAG : Langchain

04-chat-langchain.py

- LangChain est un framework open-source supporté sur Python et JS
- Plus haut niveau que Ollama
- Interfaçable avec Ollama et d'autres systèmes
- Prompte template
- Parser




```
custom_prompt = ChatPromptTemplate.from_messages(
            [SystemMessage(system_prompt),
            HumanMessagePromptTemplate.from_template(human_template),])

chain = custom_prompt  | llm | StrOutputParser()

chain.stream(query)
```

Pratique :
- Utilisation des prompt templates
- Construction d'une chaîne

Mise en pratique Langchain

atol Conseils & Développements

- Création Chat Langchain
- Construction de la chaine
- Appel de la chaine
- Parser
- Autre approche
- Templating
- Parser

```
human_template = "{question}"
custom_prompt = ChatPromptTemplate.from_messages(
  [
    SystemMessage(system_prompt),
    HumanMessagePromptTemplate.from_template(human_template),
  ]
)

return (
  custom_prompt
  | debug_runnable_fn("Prompt")
  | model
  | StrOutputParser()
)


model = ChatOllama(model=args.model, base_url=args.ollama_url)
chain = init_chain(model)
chain.stream(question)
```

Application RAG : gérer les données

05-loadnsplit-langchain.py

- Facilite le dev d'application LLM grâce à des modules clé en main
- Load des documents (pdf, html, sources web)
- Découpage des documents
- Le découpage n'est pas systématique : dépend de la volumétrie et du besoin d'indexation

Pratique :
- Tester les loaders
- Évaluer l'influence des paramètres

```
from langchain_community.document_loaders import PyPDFLoader,
BSHTMLLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# load a pdf file
docs_pdf = PyPDFLoader(file_path).load()
# load a html file
docs_html = BSHTMLLoader(file_path).load()

# split into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=50)
chunks = text_splitter.split_documents(docs_pdf)
```

Mise en pratique Langchain

atol Conseils & Développements

- Charger différentes sources de documents
- Découper les documents selon différentes stratégies
- Voir les stats des chunks

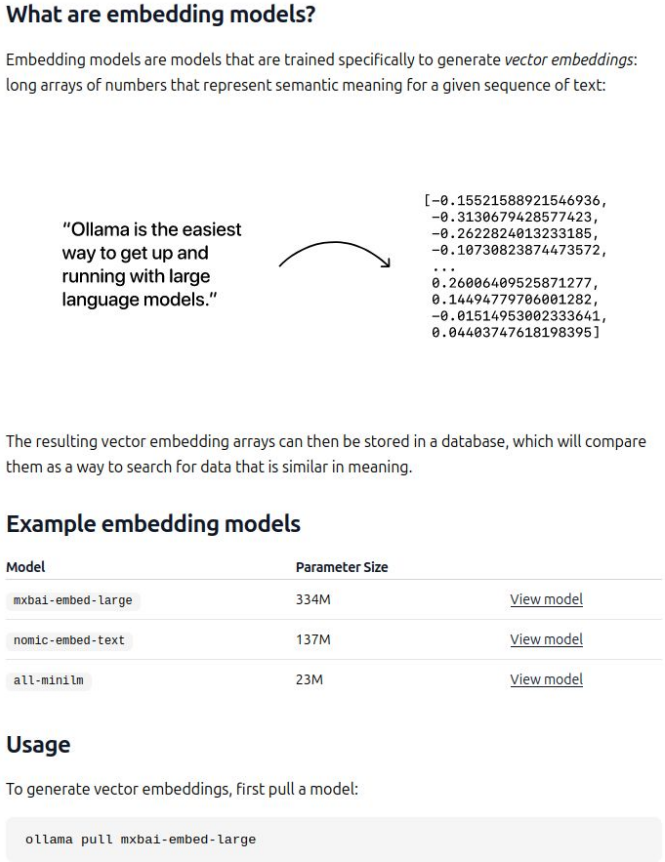
```
def log_chunks_stats(_chunks: List[Document]) -> None:
    # Evaluate chunks length and overlap
    def get_overlap(s1: str, s2: str) -> str:
        s = difflib.SequenceMatcher(None, s1, s2, False)
        pos_a, pos_b, size = s.find_longest_match(0, len(s1), 0, len(s2))

        # Getting overlaps at the start or end of chunks
        _overlap = s1[pos_a: pos_a + size]
        if (s1.startswith(_overlap) and s2.endswith(_overlap)) or (s1.endswith(_overlap)
and s2.startswith(_overlap)):
            return _overlap

        return ""

    n_chunks = len(_chunks)
    overlap = []
    for k in range(n_chunks - 1):
        tmp = get_overlap(_chunks[k].page_content, _chunks[k + 1].page_content)
        overlap.append(len(tmp))
```

Application RAG : la vectorisation

06-embeddings.py

- La vectorisation (n') est utile (que) pour la recherche par similarité
- La vectorisation est réalisée par un modèle d'embeddings (modèle NLP)
- La chaine de caractère est convertie en vecteur de taille dépendante du modèle (openAI → 1536 dimensions)
- Plusieurs modèles sur étagères disponibles depuis ollama, LangChain, SBERT, directement intégrable dans des BDD vectorielles

```
from langchain_community.embeddings.ollama import OllamaEmbeddings
from langchain_community.embeddings import FastEmbedEmbeddings
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("Alibaba-NLP/gte-base-en-v1.5", trust_remote_code=True)
embedding = model.encode(docs[0].page_content, normalize_embeddings=True)
```



What are embedding models?

Embedding models are models that are trained specifically to generate *vector embeddings*: long arrays of numbers that represent semantic meaning for a given sequence of text:

"Ollama is the easiest way to get up and running with large language models."

```
[-0.15521588921546936,
 -0.3130679428577423,
 -0.2622824013233185,
 -0.10730823874473572,
 ...
 0.26006409525871277,
 0.14494779706001282,
 -0.01514953002333641,
 0.04403747618198395]
```

The resulting vector embedding arrays can then be stored in a database, which will compare them as a way to search for data that is similar in meaning.

Example embedding models

Model	Parameter Size	
mxbai-embed-large	334M	View model
nomic-embed-text	137M	View model
all-minilm	23M	View model

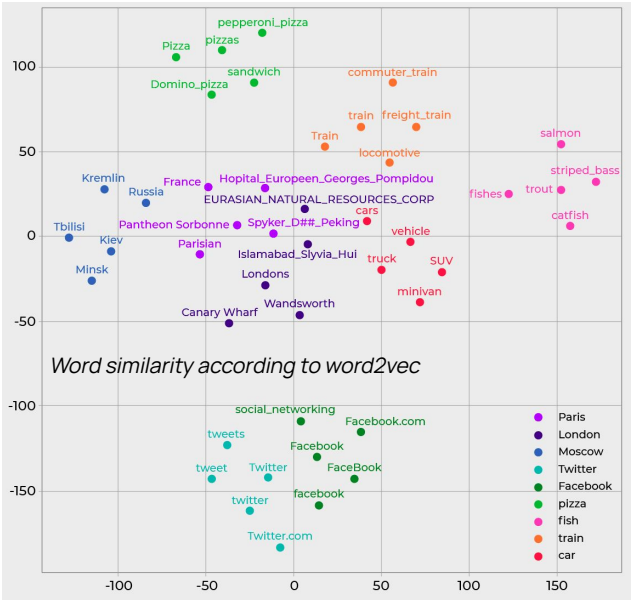
Usage

To generate vector embeddings, first pull a model:

```
ollama pull mxbai-embed-large
```

Application RAG : la vectorisation

- Similarité par calcul de distance sur [0, 1] via distance L2 ou cosine



Pratique :

- Tester plusieurs modèles d'embeddings
- Calculer les similarités entre contexte et requête

```python
from sentence_transformers import SentenceTransformer

# 1. Load a pretrained Sentence Transformer model
model = SentenceTransformer("all-MiniLM-L6-v2")

# The sentences to encode
sentences = [
  "The weather is lovely today.",
  "It's so sunny outside!",
  "He drove to the stadium.",
]

# 2. Calculate embeddings by calling model.encode()
embeddings = model.encode(sentences)
print(embeddings.shape)
# [3, 384]

# 3. Calculate the embedding similarities
similarities = model.similarity(embeddings, embeddings)
print(similarities)
# tensor([[1.0000, 0.5730, 0.2609],
#         [0.5730, 1.0000, 0.2828],
#         [0.2609, 0.2828, 1.0000]])
```

Mise en pratique Langchain

atol Conseils & Développements

- Utiliser les modèles d'embedding de ollama
- Calculer la similarité entre vecteurs
- Voir les différences entre les modèles d'embedding

```
# Evaluate the output dimension
n_sentences = len(list_sentences)
n_dim = len(ollama.embeddings(model=model, prompt="dummy")["embedding"])

# Get embeddings
t0 = time.time()
_vectors = np.empty(shape=[n_sentences, n_dim], dtype=float)

for i, sentence in enumerate(list_sentences):
    _vectors[i] = ollama.embeddings(model=model, prompt=sentence)["embedding"]


cos_matrix = np.empty(shape=[n_sentences, n_sentences], dtype=float)

for k in range(n_sentences):
    for j in range(k, n_sentences):
        cos_matrix[k, j] = get_similarity(_vectors[k], _vectors[j])
        cos_matrix[j, k] = cos_matrix[k, j]
```

Application RAG : bout-en-bout

- LangChain implémente une déclaration dédiée afin de créer une chaîne d'actions
- Utilisation d' une interface ***Runnable*** qui offre plusieurs méthodes :
  - *invoke*
  - *batch*
  - *stream*
  - *….*

- prompt | llm = llm.invoke(prompt)

Pratique :

- Interfacer tous les blocs nécessaires à l'application RAG
- tester l'application en fournissant un contexte inconnu au llm

```
from langchain_core.output_parsers import StrOutputParser

system_prompt = """Réponds en utilisant uniquement les données de contexte fournies entre triple backquotes. Lorsque le contexte ne fournit pas d'informations pour répondre à la question posée, réponds que tu n'as pas la réponse.
"""
 human_template = """
Contexte: {context_data}

Question: {question}

Réponse: """

 custom_prompt = ChatPromptTemplate.from_messages(
            [
            SystemMessage(system_prompt),
            HumanMessagePromptTemplate.from_template(human_template),
            ]
 )

return (
            custom_prompt
            | llm
            | StrOutputParser()
 )
```

Application RAG : la BDD vectorielle

07-rag-langchain.py

- Plusieurs BDD sur étagère existent et sont interfaçables avec LangChain : pgvector, ChromaDB, LanceDB, Neo4j, FAISS, …
- Chaque BDD permet de créer un objet retriever qui permet la recherche par similarité (attention aux métriques)
- Deux grands types d'algorithmes :
  - Minimisation de distance = > rapide, mais redondance
  - Maximum marginal relevance (MMR) = > ↗diversité et ↘redondance

Pratique :
- tester plusieurs BDD et comparer les performances
- tester plusieurs méthodes de recherche

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=_chunk_size,
                                               chunk_overlap=_chunk_overlap,
                                               add_start_index=True)

# Chunk the data
all_splits =
TextLoader(file_path="data/champ_euro_football_2024.txt").load_and_sp
lit(text_splitter)

# Store the chunks
store = store.from_documents(documents=all_splits,
embedding=embedding)
```

Mise en pratique Langchain

atol Conseils & Développements

- Création BDD
- Sauvegarde embeddings
- Retriever dans la chain

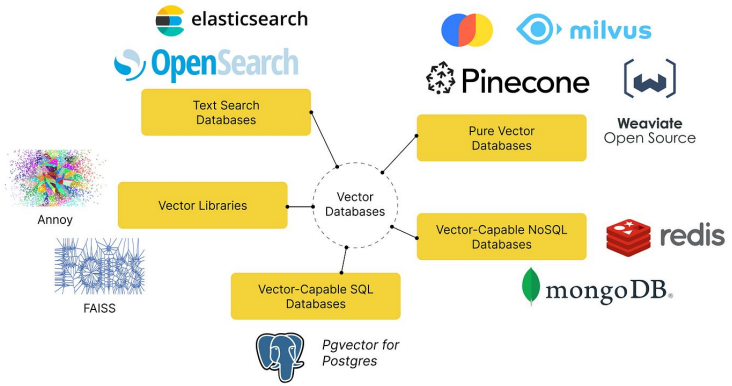
- Tester différentes recherches et différentes BDD

```
# Define the splitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=_chunk_size,
                                               chunk_overlap=_chunk_overlap,
                                               add_start_index=True)
# Chunk the data
all_splits = TextLoader(file_path=file_path).load_and_split(text_splitter)

# Store the chunks
store = store.from_documents(documents=all_splits, embedding=embedding)

chain =
       {"context_data": store.as_retriever() | format_docs, "question":
RunnablePassthrough()}
       | debug_runnable_fn("Données initiales")
       | custom_prompt
       | debug_runnable_fn("Prompt")
       | _model
       | StrOutputParser()
)
```

Application RAG : la BDD vectorielle

08-load-doc-in-db.py

- Certaines VDB offrent plus de fonctionnalités comme pgvector qui hérite du formalisme SQL de PostGreSQL, LanceDB qui inclut la recherche hybride (vecteurs + texte) ou Neo4j orientés graphes
- Dans certains cas, il est nécessaire de persister une base de données des embeddings
- Essayons PGVecto.rs qui hérite de PostGreSQL



Pratique :

- Démarrer une instance PGVecto.rs en utilisant docker compose et le fichier compose.yaml
- Remplir la VDB avec des chunks et vérifier son état

$ docker compose up

```
pg_db = PGVecto_rs.from_collection_name(
        embedding=embedding_model,
db_url=args.postgres_url,
collection_name="doc_embeddings",
)
```

Mise en pratique Langchain

atol Conseils & Développements

- Créer l'instance de la VDB
- intégrer les chunks
- Vérifier l'état de la VDB

```
pg_db = PGVecto_rs.from_collection_name(
    embedding=OllamaEmbeddings(model=args.embeddings),
    db_url=args.postgres_url,
    collection_name="doc_embeddings",
)

chunks = split_file(args)


pg_db.add_documents(chunks)
```

Application RAG : bout-en-bout

09-rag-existing-db-langchain.py

Pratique :

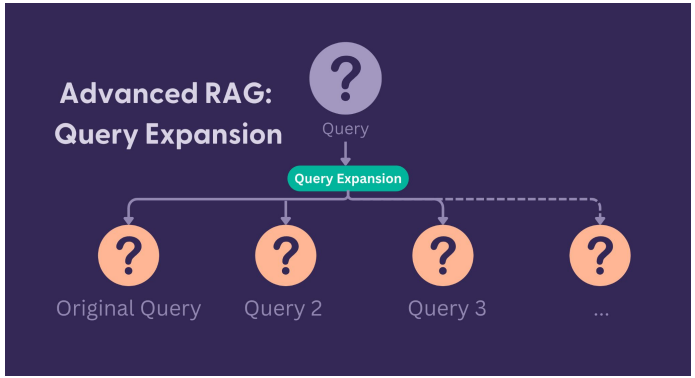
- Interfacer tous les blocs nécessaires à l'application RAG avec VDB externalisée
- tester l'application en fournissant un contexte inconnu au llm

```
db = PGVecto_rs.from_collection_name(
        embedding=OllamaEmbeddings(model=_embeddings),
        collection_name="doc_embeddings",
        db_url=_postgres_url,
        )

# Getting the retriever
retriever = db.as_retriever()

return (
        {"context_data": retriever | format_docs, "question":
RunnablePassthrough()}
        | llm
        | StrOutputParser()
 )
```

Pour aller plus loin

Query Expansion

- Bi-encoders : vectorisation de la query et des chunks puis similarité
- Cross-encoders : similarité sémantique jointe sur la query et les chunks (reranking)
    - TF-IDF : term frequency-inverse document frequency
    - BM25
- L'app RAG est très sensible à la formulation de la query = > la query expansion permet de créer des query supplémentaires pour palier aux imprécisions de la query originale



"""
You are part of an information system that processes users queries. You expand a given query into {{ number }} queries that are similar in meaning.
Structure:
Follow the structure shown below in examples to generate expanded queries.

Examples:

1. Example Query 1: "climate change effects"
Example Expanded Queries: ["impact of climate change", "consequences of global warming", "effects of environmental changes"]

2. Example Query 2: ""machine learning algorithms""
Example Expanded Queries: ["neural networks", "clustering", "supervised learning", "deep learning"]

Your Task:
Query: "{{query}}"
Example Expanded Queries:
""

Fine-tuning

Un exemple de fine-tuning avec HuggingFace (21-datasets-huggingface.py 91-fine-tuning-hf.py)

Pre-train model

```
# IMPORTING LIBRARIES
from transformers import, AutoTokenizer, pipeline, AutoModelForSequenceClassification
from transformers import Trainer, TrainingArguments
from datasets import load_dataset

def tokenize_function(examples):
        return tokenizer(examples["sentence"],
                         padding="max_length",
                         truncation=True)

# pretrained model
model_name = "distilbert/distilbert-base-uncased"
model = AutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# load data
dataset = load_dataset("stanfordnlp/sst2")
dataset = dataset.map(tokenize_function, batched=True)
```

Few-shot learning

```
# instantiate the training loop
training_args =
TrainingArguments(output_dir="./results")

trainer = Trainer(model=model,
        args=training_args,
        train_dataset=dataset['train'],
        eval_dataset=dataset['test'])

# train and save
trainer.train()
trainer.save_model(f"models/fine-tuned-{model_name}")

# test the model
classifier = pipeline(task="text-classification",
model=f"models/fine-tuned-{model_name}")
classifier(dataset["test"]["sentence"])
```

LLM-as-a-judge

- La réponse générée par un LLM peut (doit) être évaluée, soit par un humain (RLHF), voire par un autre LLM
- Plusieurs frameworks existent, comme G-eval qui est un évaluateur basé sur le prompt
- L'enjeu est de définir :
  - Des métriques d'évaluations claires
  - Des étapes de raisonnement (Chain-of-Thought)
  - Des exemples (few-shot prompting)
  - Une structure de rapport d'évaluation

additive_criteria = """

1. **Context**: Award 1 point if the answer uses only information provided in the context, without introducing external or fabricated details.

2. **Completeness**: Add 1 point if the answer addresses all key elements of the question based on the available context, without omissions.

3. **Conciseness**: Add a final point if the answer uses the fewest words possible to address the question and avoids redundancy."""

EVALUATION_PROMPT_TEMPLATE = """
You are an expert judge evaluating the Retrieval Augmented Generation applications. Your task is to evaluate a given answer based on a context and question using the criteria provided below.

Evaluation Criteria (Additive Score, 0-5):
{**additive_criteria**}

Evaluation Steps:
{**evaluation_steps**}

Output format:
{**json_schema**}

Examples:
{**examples**}

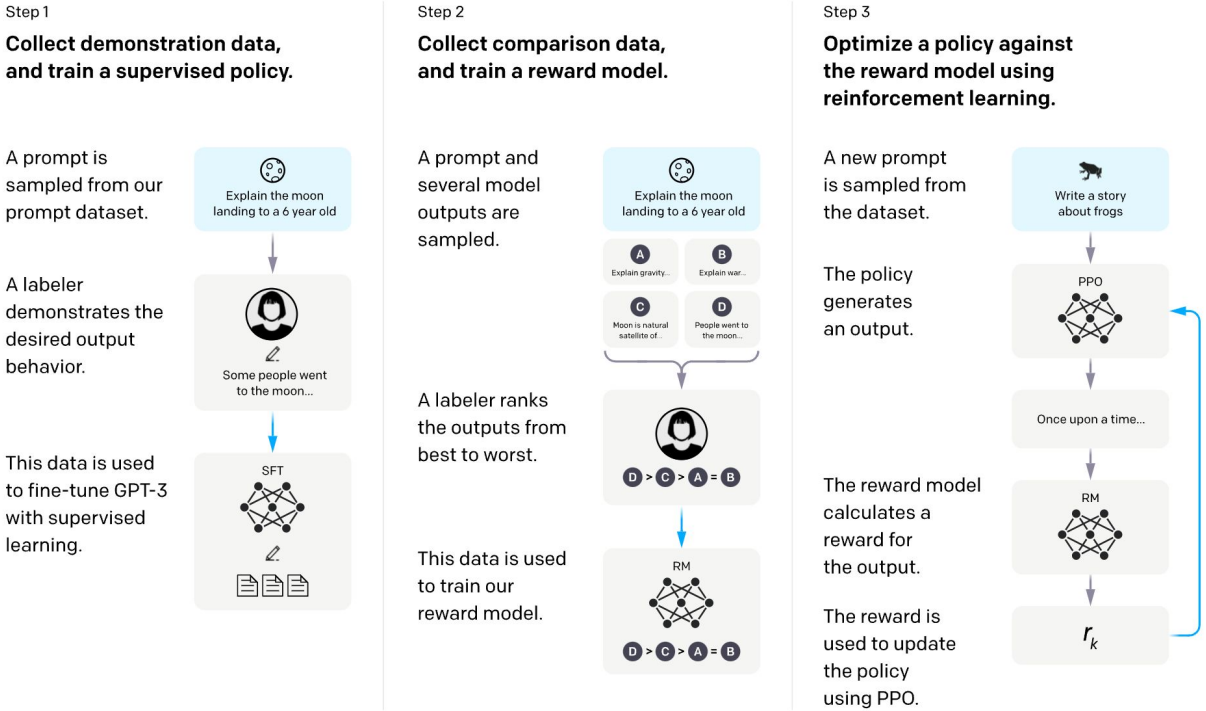
Now, please evaluate the following:
Question:
{**question**}

Context:
{**context**}

Answer:
{**answer**}
"""

Reinforcement Learning from Human Feedback

- Le RLHF est une méthodologie qui vise à biaiser les réponses générées par un LLM pour les adapter aux préférences humaines
- L'enjeu est de définir :
  - Un **modèle de récompense** qui associe un score à un couple requête - réponse (reward)
  - Une **politique d'optimisation** qui biaise les réponses du LLM avec le modèle de récompense



o.beltramo-martin@atolcd.com

x.calland@atolcd.com