7
Search and Recommendation Engines with LLMs

In the previous chapter, we covered the core steps involved in building conversational applications. We started with a plain vanilla chatbot, then added more complex components, such as memory, non-parametric knowledge, and external tools. All of this was made straightforward with the pre-built components of LangChain, as well as Streamlit for UI rendering. Even though conversational applications are often seen as the "comfort zone" of generative AI and LLMs, those models do embrace a wider spectrum of applications.

In this chapter, we are going to cover how LLMs can enhance recommendation systems, using both embeddings and generative models. We will learn how to create our own recommendation system application leveraging state-of-the-art LLMs using LangChain as the framework.

Throughout this chapter, we will cover the following topics:

- Definition and evolutions of recommendation systems
- How LLMs are impacting this field of research
- Building recommendation systems with LangChain

Technical requirements

To complete the tasks in this book, you will need the following:

- Hugging Face account and a user access token.
- OpenAI account and a user access token.
- Python version 3.7.1 or later.
- Make sure to have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `streamlit`, `lancedb`, `openai`, and `tiktoken`. These can be easily installed via `pip install` in your terminal.

You'll find the code for this chapter in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications.

Introduction to recommendation systems

A recommendation system is a computer program that recommends items for users of digital platforms such as e-commerce websites and social networks. It uses large datasets to develop models of users' likes and interests, and then recommends similar items to individual users.

There are different types of recommendation systems, depending on the methods and data they use. Some of the common types are:

- **Collaborative filtering**: This type of recommendation system uses the ratings or feedback of other users who have similar preferences to the target user. It assumes that users who liked certain items in the past will like similar items in the future. For example, if user A and user B both liked movies X and Y, then the algorithm may recommend movie Z to user A if user B also liked it.

Collaborative filtering can be further divided into two subtypes: user-based and item-based:

- **User-based collaborative filtering** finds similar users to the target user and recommends items that they liked.
- **Item-based collaborative filtering** finds similar items to the ones that the target user liked and recommends them.

- **Content-based filtering**: This type of recommendation system uses the features or attributes of the items themselves to recommend items that are similar to the ones that the target user has liked or interacted with before. It assumes that users who liked certain features of an item will like other items with similar features. The main difference with item-based collaborative filtering is that, while this latter item-based uses patterns of user behavior to make recommendations, content-based filtering uses information about the items themselves. For example, if user A liked movie X, which is a comedy with actor Y, then the algorithm may recommend movie Z, which is also a comedy with actor Y.
- **Hybrid filtering**: This type of recommendation system combines both collaborative and content-based filtering methods to overcome some of their limitations and provide more accurate and diverse recommendations. For example, YouTube uses hybrid filtering to recommend videos based on both the ratings and views of other users who have watched similar videos, and the features and categories of the videos themselves.
- **Knowledge-based filtering**: This type of recommendation system uses explicit knowledge or rules about the domain and the user's needs or preferences to recommend items that satisfy certain criteria or constraints. It does not rely on ratings or feedback from other users, but rather on the user's input or query. For example, if user A wants to buy a laptop with certain specifications and budget, then the algorithm may recommend a laptop that satisfies those criteria. Knowledge-based recommender systems work well when there is no or little rating history available, or when the items are complex and customizable.

Within the above frameworks, there are then various machine learning techniques that can be used, which we will cover in the next section.

Existing recommendation systems

Modern recommendation systems use **machine learning** (**ML**) techniques to make better predictions about users' preferences, based on the available data such as the following:

- **User behavior data**: Insights about user interaction with a product. This data can be acquired from factors like user ratings, clicks, and purchase records.

- **User demographic data**: This refers to personal information about users, including details like age, educational background, income level, and geographical location.
- **Product attribute data**: This involves information about the characteristics of a product, such as genres of books, casts of movies, or specific cuisines in the context of food.

As of today, some of the most popular ML techniques are K-nearest neighbors, dimensionality reduction, and neural networks. Let's look at these methods in detail.

K-nearest neighbors

K-nearest neighbors (KNN) is an ML algorithm that can be used for both classification and regression problems. It works by finding the *k* closest data points (where *k* refers to the number of nearest data point you want to find, and is set by the user before initializing the algorithm) to a new data point and using their labels or values to make a prediction. KNN is based on the assumption that similar data points are likely to have similar labels or values.

KNN can be applied to recommendation systems in the context of collaborative filtering, both user-based and item-based:

- User-based KNN is a type of collaborative filtering, which uses the ratings or feedback of other users who have similar tastes or preferences to the target user.

For example, let's say we have three users: Alice, Bob, and Charlie. They all buy books online and rate them. Alice and Bob both liked (rated highly) the series, *Harry Potter*, and the book, *The Hobbit*. The system sees this pattern and considers Alice and Bob to be similar.

Now, if Bob also liked the book *A Game of Thrones*, which Alice hasn't read yet, the system will recommend *A Game of Thrones* to Alice. This is because it assumes that since Alice and Bob have similar tastes, Alice might also like *A Game of Thrones*.

- Item-based KNN is another type of collaborative filtering, which uses the attributes or features of the items to recommend similar items to the target user.

For example, let's consider the same users and their ratings for the books. The system notices that the *Harry Potter* series and the book, *The Hobbit* are both liked by Alice and Bob. So, it considers these two books to be similar.

Now, if Charlie reads and likes *Harry Potter*, the system will recommend *The Hobbit* to Charlie. This is because it assumes that since *Harry Potter* and *The Hobbit* are similar (both liked by the same users), Charlie might also like *The Hobbit*.

KNN is a popular technique in recommendation systems, but it has some pitfalls:

- **Scalability**: KNN can become computationally expensive and slow when dealing with large datasets, as it requires calculating distances

between all pairs of items or users.
- **Cold-start problem**: KNN struggles with new items or users that have limited or no interaction history, as it relies on finding neighbors based on historical data.
- **Data sparsity**: KNN performance can degrade in sparse datasets where there are many missing values, making it challenging to find meaningful neighbors.
- **Feature relevance**: KNN treats all features equally and assumes that all features contribute equally to similarity calculations. This may not hold true in scenarios where some features are more relevant than others.
- **Choice of K**: Selecting the appropriate value of K (number of neighbors) can be subjective and impact the quality of recommendations. A small K may result in noise, while a large K may lead to overly broad recommendations.

Generally speaking, KNN is recommended in scenarios with small datasets with minimal noise (so that outliers, missing values and other noises do not impact the distance metric) and dynamic data (KNN is an instance-based method that doesn't require retraining and can adapt to changes quickly).

Additionally, further techniques are widely used in the file of recommendation systems, such as matrix factorization.

Matrix factorization

Matrix factorization is a technique used in recommendation systems to analyze and predict user preferences or behaviors based on historical data. It involves decomposing a large matrix into two or more smaller matrices to uncover latent features that contribute to the observed data patterns and address the so-called "curse of dimensionality."

Definition

The curse of dimensionality refers to challenges that arise when dealing with high-dimensional data. It leads to increased complexity, sparse data, and difficulties in analysis and modeling due to the exponential growth of data requirements and potential overfitting.

In the context of recommendation systems, this technique is employed to predict missing values in the user-item interaction matrix, which represents users' interactions with various items (such as movies, products, or books).

Let's consider the following example. Imagine you have a matrix where rows represent users, columns represent movies, and the cells contain ratings (from 1 as lowest to 5 as highest). However, not all users have rated all movies, resulting in a matrix with many missing entries:

	Movie 1	Movie 2	Movie 3	Movie 4
User 1	4	-	5	-

User 2	-	3	-	2
User 3	5	4	-	3

Table 7.1: Example of a dataset with missing data

Matrix factorization aims to break down this matrix into two matrices: one for users and another for movies, with a reduced number of dimensions (latent factors). These latent factors could represent attributes like genre preferences or specific movie characteristics. By multiplying these matrices, you can predict the missing ratings and recommend movies that the users might enjoy.

There are different algorithms for matrix factorization, including the following:

- **Singular value decomposition** (**SVD**) decomposes a matrix into three separate matrices, where the middle matrix contains singular values that represent the importance of different components in the data. It's widely used in data compression, dimensionality reduction, and collaborative filtering in recommendation systems.
- **Principal component analysis** (**PCA**) is a technique to reduce the dimensionality of data by transforming it into a new coordinate system aligned with the principal components. These components capture the most significant variability in the data, allowing efficient analysis and visualization.
- **Non-negative matrix factorization** (**NMF**) decomposes a matrix into two matrices with non-negative values. It's often used for topic modeling, image processing, and feature extraction, where the components represent non-negative attributes.

In the context of recommendation systems, probably the most popular technique is SVD (thanks to its interpretability, flexibility, and ability to handle missing values and performance), so let's use this one to go on with our example. We will use the Python `numpy` module to apply SVD as follows:

```
import numpy as np
# Your user-movie rating matrix (replace with your actual data)
user_movie_matrix = np.array([
    [4, 0, 5, 0],
    [0, 3, 0, 2],
    [5, 4, 0, 3]
])
# Apply SVD
U, s, V = np.linalg.svd(user_movie_matrix, full_matrices=False)
# Number of latent factors (you can choose this based on your preference)
num_latent_factors = 2
# Reconstruct the original matrix using the selected latent factors
reconstructed_matrix = U[:, :num_latent_factors] @ np.diag(s[:num_latent_factors]) @ V[:nu
# Replace negative values with 0
reconstructed_matrix = np.maximum(reconstructed_matrix, 0)
print("Reconstructed Matrix:")
print(reconstructed_matrix)
```

The following is the output:

```
Reconstructed Matrix:
[[4.2972542  0.         4.71897811 0.        ]
 [1.08572801 2.27604748 0.         1.64449028]
 [4.44777253 4.36821972 0.52207171 3.18082082]]
```

In this example, the `U` matrix contains user-related information, the `s` matrix contains singular values, and the `V` matrix contains movie-related information. By selecting a certain number of latent factors (`num_latent_factors`), you can reconstruct the original matrix with reduced dimensions, while setting the `full_matrices=False` parameter in the `np.linalg.svd` function ensures that the decomposed matrices are truncated to have dimensions consistent with the selected number of latent factors.

These predicted ratings can then be used to recommend movies with higher predicted ratings to users. Matrix factorization enables recommendation systems to uncover hidden patterns in user preferences and make personalized recommendations based on those patterns.

Matrix factorization has been a widely used technique in recommendation systems, especially when dealing with large datasets containing a substantial number of users and items, since it efficiently captures latent factors even in such scenarios; or when you want personalized recommendations based on latent factors, since it learns unique latent representations for each user and item. However, it has some pitfalls (some similar to the KNN's technique):

- **Cold-start problem**: Similar to KNN, matrix factorization struggles with new items or users that have limited or no interaction history. Since it relies on historical data, it can't effectively provide recommendations for new items or users.
- **Data sparsity**: As the number of users and items grows, the user-item interaction matrix becomes increasingly sparse, leading to challenges in accurately predicting missing values.
- **Scalability**: For large datasets, performing matrix factorization can be computationally expensive and time-consuming.
- **Limited context**: Matrix factorization typically only considers user-item interactions, ignoring contextual information like time, location, or additional user attributes.

Hence, **neural networks (NNs)** have been explored as an alternative to mitigate these pitfalls in recent years.

Neural networks

NNs are used in recommendation systems to improve the accuracy and personalization of recommendations by learning intricate patterns from data. Here's how neural networks are commonly applied in this context:

- **Collaborative filtering with neural networks**: Neural networks can model user-item interactions by embedding users and items into continuous vector spaces. These embeddings capture latent features that represent user preferences and item characteristics. Neural collaborative filtering models combine these embeddings with neural network architectures to predict ratings or interactions between users and items.

- **Content-based recommendations**: In content-based recommendation systems, neural networks can learn representations of item content, such as text, images, or audio. These representations capture item characteristics and user preferences. Neural networks like **convolutional neural networks (CNNs)** and **recurrent neural networks (RNNs)** are used to process and learn from item content, enabling personalized content-based recommendations.
- **Sequential models**: In scenarios where user interactions have a temporal sequence, such as clickstreams or browsing history, RNNs or variants such as **long short-term memory (LSTM)** networks can capture temporal dependencies in the user behavior and make sequential recommendations.
- **Autoencoders and variational autoencoders (VAEs)** can be used to learn low-dimensional representations of users and items.

Definition

Autoencoders are a type of neural network architecture used for unsupervised learning and dimensionality reduction. They consist of an encoder and a decoder. The encoder maps the input data into a lower-dimensional latent space representation, while the decoder attempts to reconstruct the original input data from the encoded representation.

VAEs are an extension of traditional autoencoders that introduce probabilistic elements. VAEs not only learn to encode the input data into a latent space but also model the distribution of this latent space using probabilistic methods. This allows for the generation of new data samples from the learned latent space. VAEs are used for generative tasks like image synthesis, anomaly detection, and data imputation.

In both autoencoders and VAEs, the idea is to learn a compressed and meaningful representation of the input data in the latent space, which can be useful for various tasks including feature extraction, data generation, and dimensionality reduction.

These representations can then be used to make recommendations by identifying similar users and items in the latent space. In fact, the unique architecture that features NNs allows for the following techniques:

- **Side information integration**: NNs can incorporate additional user and item attributes, such as demographic information, location, or social connections, to improve recommendations by learning from diverse data sources.
- **Deep reinforcement learning**: In certain scenarios, deep reinforcement learning can be used to optimize recommendations over time, learning from user feedback to suggest actions that maximize long-term rewards.

NNs offer flexibility and the ability to capture complex patterns in data, making them well suited for recommendation systems. However, they also require careful design, training, and tuning to achieve optimal performance. NNs also bring their own challenges, including the following:

- **Increased complexity**: NNs, especially **deep neural networks (DNNs)**, can become incredibly complex due to their layered architecture. As we add more hidden layers and neurons, the model's capacity to learn intricate patterns increases.
- **Training requirements**: NNs are heavy models whose training requires special hardware requirements including GPUs, which might be very expensive.
- **Potential overfitting**: Overfitting occurs when an ANN learns to perform exceptionally well on the training data but fails to generalize to unseen data

Selecting appropriate architectures, handling large datasets, and tuning hyperparameters are essential to effectively use NNs in recommendation systems.

Even though relevant advancements have been made in recent years, the aforementioned techniques still suffer from some pitfalls, primarily their being task-specific. For example, a rating-prediction recommendation system will not be able to tackle a task where we need to recommend the top *k* items that likely match the user's taste. Actually, if we extend this limitation to other "pre-LLMs" AI solutions, we might see some similarities: it is indeed the task-specific situation that LLMs and, more generally, Large Foundation Models are revolutionizing, being highly generalized and adaptable to various tasks, depending on user's prompts and instructions. Henceforth, extensive research in the field of recommendation systems is being done into what extent LLMs can enhance the current models. In the following sections, we will cover the theory behind these new approaches referring to recent papers and blogs about this emerging domain.

How LLMs are changing recommendation systems

We saw in previous chapters how LLMs can be customized in three main ways: pre-training, fine-tuning, and prompting. According to the paper *Recommender systems in the Era of Large Language Models (LLMs)* from Wenqi Fan et al., these techniques can also be used to tailor an LLM to be a recommender system:

- **Pre-training**: Pre-training LLMs for recommender systems is an important step to enable LLMs to acquire extensive world knowledge and user preferences, and to adapt to different recommendation tasks with zero or few shots.

Note

An example of a recommendation system LLM is P5, introduced by Shijie Gang et al. in their paper *Recommendation as Language Processing (RLP): A Unified Pretrain, Personalized Prompt & Predict Paradigm (P5)*.

P5 is a unified text-to-text paradigm for building recommender systems using **large language models (LLMs)**. It consists of three steps:

- Pretrain: A foundation language model based on T5 architecture is pretrained on a large-scale web corpus and fine-tuned on recommendation tasks.

- Personalized prompt: A personalized prompt is generated for each user based on their behavior data and contextual features.
- Predict: The personalized prompt is fed into the pretrained language model to generate recommendations.

P5 is based on the idea that LLMs can encode extensive world knowledge and user preferences and can be adapted to different recommendation tasks with zero or few shots.

- **Fine-tuning**: Training an LLM from scratch is a highly computational-intensive activity. An alternative and less intrusive approach to customize an LLM for recommendation systems might be fine-tuning.

More specifically, the authors of the paper review two main strategies for fine-tuning LLMs:

- **Full-model fine-tuning** involves changing the entire model's weights based on task-specific recommendation datasets.
- **Parameter-efficient fine-tuning** aims to change only a small part of weights or develop trainable adapters to fit specific tasks.

- **Prompting**: The third and "lightest" way of tailoring LLMs to be recommender systems is prompting. According to the authors, there are three main techniques for prompting LLMs:
  - **Conventional prompting** aims to unify downstream tasks into language generation tasks by designing text templates or providing a few input-output examples.
  - **In-context learning** enables LLMs to learn new tasks based on contextual information without fine-tuning.
  - **Chain-of-thought** enhances the reasoning abilities of LLMs by providing multiple demonstrations to describe the chain of thought as examples within the prompt. The authors also discuss the advantages and challenges of each technique and provide some examples of existing methods that adopt them.

Regardless of the typology, prompting is the fastest way to test whether a general-purpose LLM can tackle recommendation systems' tasks.

The application of LLMs within the recommendation system domain is raising interest in the research field, and there is already some interesting evidence of the results as seen above.

In the next section, we are going to implement our own recommendation application using the prompting approach and leveraging the capabilities of LangChain as an AI orchestrator.

Implementing an LLM-powered recommendation system

Now that we have covered some theory about recommendation systems and emerging research on how LLMs can enhance them, let's start building our recommendation app, which will be a movie recommender system called MovieHarbor. The goal will be to make it as general as possible, meaning that we want our app to be able to address various recommendations tasks with a conversational interface. The scenario we are going to simulate will be that of the so-called "cold start," concerning the

first interaction of a user with the recommendation system where we do not have the user's preference history. We will leverage a movie database with textual descriptions.

For this purpose, we will use the *Movie recommendation data* dataset, available on Kaggle at https://www.kaggle.com/datasets/rohan4050/movie-recommendation-data.

The reason for using a dataset with a textual description of each movie (alongside information such as ratings and movie titles) is so that we can get the embeddings of the text. So let's start building our MovieHarbor application.

Data preprocessing

In order to apply LLMs to our dataset, we first need to preprocess the data. The initial dataset included several columns; however, the ones we are interested in are the following:

- **Genres**: A list of applicable genres for the movie.
- **Title**: The movie's title.
- **Overview**: Textual description of the plot.
- **Vote_average**: A rating from 1 to 10 for a given movie
- **Vote_count**: The number of votes for a given movie.

I won't report here the whole code (you can find it in the GitHub repo of this book at https://github.com/PacktPublishing/Building-LLM-Powered-Applications), however, I will share the main steps of data preprocessing:

1. First, we format the `genres` column into a `numpy` array, which is easier to handle than the original dictionary format in the dataset:

```
import pandas as pd
import ast
# Convert string representation of dictionaries to actual dictionaries
md['genres'] = md['genres'].apply(ast.literal_eval)
# Transforming the 'genres' column
md['genres'] = md['genres'].apply(lambda x: [genre['name'] for genre in x])
```

2. Next, we merge the `vote_average` and `vote_count` columns into a single column, which is the weighted ratings with respect to the number of votes. I've also limited the rows to the 95th percentile of the number of votes, so that we can get rid of minimum vote counts to prevent skewed results:

```
# Calculate weighted rate (IMDb formula)
def calculate_weighted_rate(vote_average, vote_count, min_vote_count=10):
    return (vote_count / (vote_count + min_vote_count)) * vote_average + (min_vote_coun
# Minimum vote count to prevent skewed results
vote_counts = md[md['vote_count'].notnull()]['vote_count'].astype('int')
min_vote_count = vote_counts.quantile(0.95)
# Create a new column 'weighted_rate'
md['weighted_rate'] = md.apply(lambda row: calculate_weighted_rate(row['vote_average'],
```

3. Next, we create a new column called `combined_info` where we are going to merge all the elements that will be provided as context to the LLMs. Those elements are the movie title, overview, genres, and ratings:

```
md_final['combined_info'] = md_final.apply(lambda row: f"Title: {row['title']}. Overvie
```

4. We tokenize the movie `combined_info` so that we will get better results while embedding:

```
import pandas as pd
import tiktoken
import os
import openai
openai.api_key = os.environ["OPENAI_API_KEY"]
from openai.embeddings_utils import get_embedding
embedding_encoding = "cl100k_base"  # this the encoding for text-embedding-ada-002
max_tokens = 8000  # the maximum for text-embedding-ada-002 is 8191
encoding = tiktoken.get_encoding(embedding_encoding)
# omit reviews that are too long to embed
md_final["n_tokens"] = md_final.combined_info.apply(lambda x: len(encoding.encode(x)))
md_final = md_final[md_final.n_tokens <= max_tokens]
```

Definition

`cl100k_base` is the name of a tokenizer used by OpenAI's embeddings API. A tokenizer is a tool that splits a text string into units called tokens, which can then be processed by a neural network. Different tokenizers have different rules and vocabularies for how to split the text and what tokens to use.

The `cl100k_base` tokenizer is based on the **byte pair encoding** (**BPE**) algorithm, which learns a vocabulary of subword units from a large corpus of text. The `cl100k_base` tokenizer has a vocabulary of 100,000 tokens, which are mostly common words and word pieces, but also include some special tokens for punctuation, formatting, and control. It can handle texts in multiple languages and domains, and can encode up to 8,191 tokens per input.

5. We embed the text with `text-embedding-ada-002`:

```
md_final["embedding"] = md_final.overview.apply(lambda x: get_embedding(x, engine=embed
```

After changing some columns' names and dropping unnecessary columns, the final dataset looks as follows:

	genres	title	overview	weighted_rate	combined_info	n_tokens	embedding
0	[Adventure, Action, Thriller]	GoldenEye	James Bond must unmask the mysterious head of ...	6.173464	Title: GoldenEye. Overview: James Bond must un...	59	[-0.023236559703946114, -0.015966948121786118,...
1	[Comedy]	Friday	Craig and Smokey are two guys in Los Angeles h...	6.083421	Title: Friday. Overview: Craig and Smokey are ...	52	[0.0015918031567707658, -0.010778157971799374,...
2	[Horror, Action, Thriller, Crime]	From Dusk Till Dawn	Seth Gecko and his younger brother Richard are...	6.503176	Title: From Dusk Till Dawn. Overview: Seth Gec...	105	[-0.008583318442106247, -0.004688787739723921,...

Figure 7.1: Sample of the final movies dataset

Let's have a look at a random row of text:

```
md['text'][0]
```

The following output is obtained:

```
'Title: GoldenEye. Overview: James Bond must unmask the mysterious head of the Janus Syndi
```

The last change we will make is modifying some naming conventions and data types as follows:

```
md_final.rename(columns = {'embedding': 'vector'}, inplace = True)
md_final.rename(columns = {'combined_info': 'text'}, inplace = True)
md_final.to_pickle('movies.pkl')
```

6. Now that we have our final dataset, we need to store it in a VectorDB. For this purpose, we are going to leverage **LanceDB**, an open-source database for vector-search built with persistent storage, which greatly simplifies the retrieval, filtering, and management of embeddings and also offers a native integration with LangChain. You can easily install LanceDB via `pip install lancedb`:

   ```
   import lancedb
   uri = "data/sample-lancedb"
   db = lancedb.connect(uri)
   table = db.create_table("movies", md)
   ```

Now that we have all our ingredients, we can start working with those embeddings and start building our recommendation system. We will start with a simple task in a cold-start scenario, adding progressive layers of complexity with LangChain components. Afterwards, we will also try a content-based scenario to challenge our LLMs with diverse tasks.

Building a QA recommendation chatbot in a cold-start scenario

In previous sections, we saw how the cold-start scenario – that means interacting with a user for the first time without their backstory – is a problem often encountered by recommendation systems. The less information we have about a user, the harder it is to match the recommendations to their preferences.

In this section, we are going to simulate a cold-start scenario with LangChain and OpenAI's LLMs with the following high-level architecture:

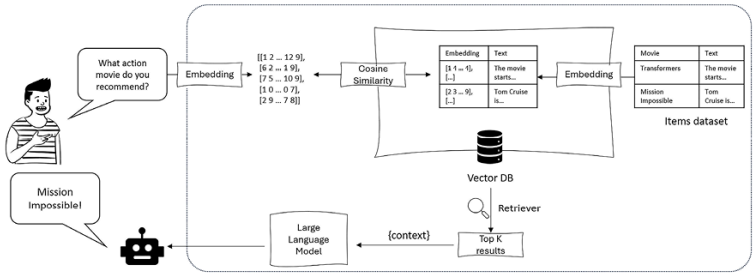

Figure 7.2: High-level architecture of recommendation system in a cold-start scenario

In the previous section, we've already saved our embeddings in LanceDB. Now, we are going to build a LangChain RetrievalQA retriever, a chain component designed for question-answering against an index. In our case, we will use the vector store as our index retriever. The idea is that the chain returns the top *k* most similar movies upon the user's query, using cosine similarity as the distance metric (which is the default).

So, let's start building the chain:

1. We are using only the movie overview as information input:

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import LanceDB
os.environ["OPENAI_API_KEY"]
embeddings = OpenAIEmbeddings()
docsearch = LanceDB(connection = table, embedding = embeddings)
query = "I'm looking for an animated action movie. What could you suggest to me?"
docs = docsearch.similarity_search(query)
docs
```

The following is the corresponding output (I will display a truncated version of the output, showing only the first out of four document sources):

```
[Document(page_content='Title: Hitman: Agent 47. Overview: An assassin teams up with a wom
```

As you can see, alongside each `Document`, all variables are reported as metadata, plus the distance is also reported as a score. The lower the distance, the greater the proximity between the user's query and the movie's text embedding.

2. Once we have gathered the most similar documents, we want a conversational response. For this goal, in addition to the embedding models, we will also use OpenAI's completion model GPT-3 and combine it in RetrievalQA:

```
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff", retriever=docsearch.
query = "I'm looking for an animated action movie. What could you suggest to me?"
result = qa({"query": query})
result['result']
```

Let's look at the output:

```
' I would suggest Transformers. It is an animated action movie with genres of Adventure, S
```

3. Since we set the `return_source_documents=True` parameter, we can also retrieve the document sources:

```
result['source_documents'][0]
```

The following is the output:

```
Document(page_content='Title: Hitman: Agent 47. Overview: An assassin teams up with a woma
       -0.01303058, -0.00709073], dtype=float32), '_distance': 0.42414575815200806})
```

Note that the first document reported is not the one the model suggested. This occurred probably because of the rating, which is lower than Transformers (which was only the third result). This is a great example of how the LLM was able to consider multiple factors, on top of similarity, to suggest a movie to the user.

4. The model was able to generate a conversational answer, however, it is still using only a part of the available information – the textual overview. What if we want our MovieHarbor system to also leverage the other variables? We can approach the task in two ways:
   1. **The "filter" way**: This approach consists of adding some filters as **kwargs** to our retriever, which might be required by the application before responding to the user. Those questions might be, for example, about the genre of a movie.

   For example, let's say we want to provide results featuring only those movies for which the genre is tagged as comedy. You can achieve this with the following code:

```
df_filtered = md[md['genres'].apply(lambda x: 'Comedy' in x)]
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff",
    retriever=docsearch.as_retriever(search_kwargs={'data': df_filtered}), return_sourc
query = "I'm looking for a movie with animals and an adventurous plot."
result = qa({"query": query})
```

   The filter can also operate at the metadata level, as shown in the following example, where we want to filter only results with a rating above 7:

```
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff",
    retriever=docsearch.as_retriever(search_kwargs={'filter': {weighted_rate__gt:7}}),
```

   1. **The "agentic" way**: This is probably the most innovative way to approach the problem. Making our chain agentic means converting the retriever to a tool that the agent can leverage if needed, including the additional variables. By doing so, it would be sufficient for the user to provide their preferences in natural language so that the agent can retrieve the most promising recommendation if needed.

Let's see how to implement this with code, asking specifically for an action movie (thus filtering on the `genre` variable):

```
from langchain.agents.agent_toolkits import create_retriever_tool
from langchain.agents.agent_toolkits import create_conversational_retrieval_agent
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature = 0)
retriever = docsearch.as_retriever(return_source_documents = True)
tool = create_retriever_tool(
    retriever,
    "movies",
    "Searches and returns recommendations about movies."
)
tools = [tool]
agent_executor = create_conversational_retrieval_agent(llm, tools, verbose=True)
result = agent_executor({"input": "suggest me some action movies"})
```

Let's see a glimpse of the chain of thoughts and the output produced (always based on the four most similar movies according to cosine similarity):

```
> Entering new AgentExecutor chain...
Invoking: `movies` with `{'genre': 'action'}`
[Document(page_content='The action continues from [REC], […]
Here are some action movies that you might enjoy:
1. [REC]² - The action continues from [REC], with a medical officer and a SWAT team sent i
2. The Boondock Saints - Twin brothers Conner and Murphy take swift retribution into their
3. The Gamers - Four clueless players are sent on a quest to rescue a princess and must na
4. Atlas Shrugged Part III: Who is John Galt? - In a collapsing economy, one man has the a
Please note that these recommendations are based on the genre "action" and may vary in ter
> Finished chain.
```

5. Finally, we might also want to make our application more tailored toward its goal of being a recommender system. To do so, we need to do some prompt engineering.

Note

One of the advantages of using LangChain's pre-built components, such as the RetrievalQA chain, is that they come with a pre-configured, well-curated prompt template. Before overriding the existing prompt, it's a good practice to inspect it, so that you can also see which variables (within `{}`) are already expected from the component.

To explore the existing prompt, you can run the following code:

```
print(qa.combine_documents_chain.llm_chain.prompt.template)
```

Here is the output:

```
Use the following pieces of context to answer the question at the end. If you don't know t
{context}
Question: {question}
Helpful Answer:
```

Let's say, for example, that we want our system to return three suggestions for each user's request, with a short description of the plot and the

reason why the user might like it. The following is a sample prompt that could match this goal:

```
from langchain.prompts import PromptTemplate
template = """You are a movie recommender system that help users to find movies that match
Use the following pieces of context to answer the question at the end.
For each question, suggest three movies, with a short description of the plot and the reas
If you don't know the answer, just say that you don't know, don't try to make up an answer
{context}
Question: {question}
Your response:"""

PROMPT = PromptTemplate(
    template=template, input_variables=["context", "question"])
```

6. Now we need to pass it into our chain:

```
PROMPT = PromptTemplate(
    template=template, input_variables=["context", "question"])
chain_type_kwargs = {"prompt": PROMPT}
qa = RetrievalQA.from_chain_type(llm=OpenAI(),
    chain_type="stuff",
    retriever=docsearch.as_retriever(),
    return_source_documents=True,
    chain_type_kwargs=chain_type_kwargs)
query = "I'm looking for a funny action movie, any suggestion?"
result = qa({'query':query})
print(result['result'])
```

The following output is obtained:

```
1. A Good Day to Die Hard: An action-packed comedy directed by John Moore, this movie foll
2. The Hidden: An alien is on the run in America and uses the bodies of anyone in its way
3. District B13: Set in the ghettos of Paris in 2010, this action-packed science fiction m
```

7. Another thing that we might want to implement in our prompt is the information gathered with the conversational preliminary questions that we might want to set as a welcome page. For example, before letting the user input their natural language question, we might want to ask their age, gender, and favorite movie genre. To do so, we can insert in our prompt a section where we can format the input variables with those shared by the user, and then combine this prompt chunk in the final prompt we are going to pass to the chain. Below you can find an example (for simplicity, we are going to set the variables without asking the user):

```
from langchain.prompts import PromptTemplate
template_prefix = """You are a movie recommender system that help users to find movies
Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an ansu
{context}"""
user_info = """This is what we know about the user, and you can use this information to
Age: {age}
Gender: {gender}"""
template_suffix= """Question: {question}
Your response:"""
user_info = user_info.format(age = 18, gender = 'female')
```

```
COMBINED_PROMPT = template_prefix +'\n'+ user_info +'\n'+ template_suffix
print(COMBINED_PROMPT)
```

Here is the output:

```
You are a movie recommender system that help users to find movies that match their prefere
Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer
{context}
This is what we know about the user, and you can use this information to better tune your
Age: 18
Gender: female
Question: {question}
Your response:
```

8. Now let's format the prompt and pass it into our chain:

```
PROMPT = PromptTemplate(
    template=COMBINED_PROMPT, input_variables=["context", "question"])
chain_type_kwargs = {"prompt": PROMPT}
qa = RetrievalQA.from_chain_type(llm=OpenAI(),
    chain_type="stuff",
    retriever=docsearch.as_retriever(),
    return_source_documents=True,
    chain_type_kwargs=chain_type_kwargs)
result = qa({'query':query})
result['result']
```

We receive the following output:

```
' Sure, I can suggest some action movies for you. Here are a few examples: A Good Day to D
```

As you can see, the system considered the user's information provided. When we build the front-end of MovieHarbor, we will make this information dynamic as preliminary questions proposed to the user.

Building a content-based system

In the previous section, we covered the cold-start scenario where the system knew nothing about the user. Sometimes, recommender systems already have some backstory about users, and it is extremely useful to embed this knowledge in our application. Let's imagine, for example, that we have a users database where the system has stored all the registered user's information (such as age, gender, country, etc.) as well as the movies the user has already watched alongside their rating.

To do so, we will need to set a custom prompt that is able to retrieve this information from a source. For simplicity, we will create a sample dataset with users' information with just two records, corresponding to two users. Each user will exhibit the following variables: username, age, gender, and a dictionary containing movies already watched alongside with the rating they gave to them.

The high-level architecture is represented by the following diagram:

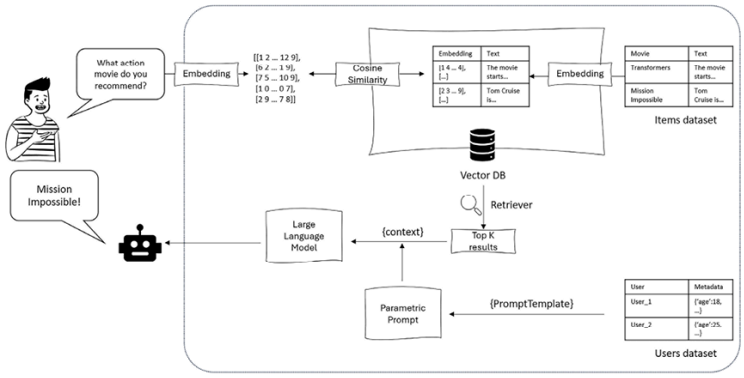

Figure 7.3: High-level architecture of a content-based recommendation system

Let's break down this architecture and examine each step to build the final chat for this content-based system, starting from the available users' data:

1. As discussed earlier, we now have a bit of information about our users' preferences. More specifically, imagine we have a dataset containing users' attributes (name, age, gender) along with their reviews (a score from 1 to 10) of some movies. The following is the code used to create the dataset:

```
import pandas as pd
data = {
    "username": ["Alice", "Bob"],
    "age": [25, 32],
    "gender": ["F", "M"],
    "movies": [
        [("Transformers: The Last Knight", 7), ("Pokémon: Spell of the Unknown", 5)],
        [("Bon Cop Bad Cop 2", 8), ("Goon: Last of the Enforcers", 9)]
    ]
}
# Convert the "movies" column into dictionaries
for i, row_movies in enumerate(data["movies"]):
    movie_dict = {}
    for movie, rating in row_movies:
        movie_dict[movie] = rating
    data["movies"][i] = movie_dict
# Create a pandas DataFrame
df = pd.DataFrame(data)
df.head()
```

The following output is obtained:

	username	age	gender	movies
0	Alice	25	F	{'Transformers: The Last Knight': 7, 'Pokémon:...
1	Bob	32	M	{'Bon Cop Bad Cop 2': 8, 'Goon: Last of the En...

Figure 7.4: Sample users dataset

2. What we want to do now is apply the same logic of the prompt of the cold start with the formatting with variables. The difference here is that, rather than asking the user to provide the values for those vari-

ables, we will directly collect them from our user dataset. So, we first define our prompt chunks:

```
template_prefix = """You are a movie recommender system that help users to find movies
Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an ans
{context}"""
user_info = """This is what we know about the user, and you can use this information to
Age: {age}
Gender: {gender}
Movies already seen alongside with rating: {movies}"""
template_suffix= """Question: {question}
Your response:"""
```

3. We then format the `user_info` chunk as follows (assuming that the user interacting with the system is `Alice`):

```
age = df.loc[df['username']=='Alice']['age'][0]
gender = df.loc[df['username']=='Alice']['gender'][0]
movies = ''
# Iterate over the dictionary and output movie name and rating
for movie, rating in df['movies'][0].items():
    output_string = f"Movie: {movie}, Rating: {rating}" + "\n"
    movies+=output_string
    #print(output_string)
user_info = user_info.format(age = age, gender = gender, movies = movies)
COMBINED_PROMPT = template_prefix +'\n'+ user_info +'\n'+ template_suffix
print(COMBINED_PROMPT)
```

Here is the output:

```
You are a movie recommender system that help users to find movies that match their prefere
Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer
{context}
This is what we know about the user, and you can use this information to better tune your
Age: 25
Gender: F
Movies already seen alongside with rating: Movie: Transformers: The Last Knight, Rating: 7
Movie: Pokémon: Spell of the Unknown, Rating: 5
Question: {question}
Your response:
```

4. Let's now use this prompt within our chain:

```
PROMPT = PromptTemplate(
    template=COMBINED_PROMPT, input_variables=["context", "question"])
chain_type_kwargs = {"prompt": PROMPT}
qa = RetrievalQA.from_chain_type(llm=OpenAI(),
    chain_type="stuff",
    retriever=docsearch.as_retriever(),
    return_source_documents=True,
    chain_type_kwargs=chain_type_kwargs)
query = "Can you suggest me some action movie based on my background?"
result = qa({'query':query})
result['result']
```

We then obtain the following output:

```
" Based on your age, gender, and the movies you've already seen, I would suggest the follo
'
```

As you can see, the model is now able to recommend a list of movies to Alice based on the user's information about past preferences, retrieved as context within the model's metaprompt.

Note that, in this scenario, we used as dataset a simple pandas dataframe. In production scenarios, a best practice for storing variables related to a task to be addressed (such as a recommendation task) is that of using a feature store. Feature stores are data systems that are designed to support machine learning workflows. They allow data teams to store, manage, and access features that are used for training and deploying machine learning models.

Furthermore, LangChain offers native integrations towards some of the most popular features stores:

- **Feast:** This is an open-source feature store for machine learning. It allows teams to define, manage, discover, and serve features. Feast supports batch and streaming data sources and integrates with various data processing and storage systems. Feast uses BigQuery for offline features and BigTable or Redis for online features.
- **Tecton:** This is a managed feature platform that provides a complete solution for building, deploying, and using features for machine learning. Tecton allows users to define features in code, version control them, and deploy them to production with best practices. Furthermore, it integrates with existing data infrastructure and ML platforms like SageMaker and Kubeflow, and it uses Spark for feature transformations and DynamoDB for online feature serving.
- **Featureform:** This is a virtual feature store that transforms existing data infrastructure into a feature store. Featureform allows users to create, store, and access features using standard feature definitions and a Python SDK. It orchestrates and manages the data pipelines required for feature engineering and materialization, and it is compatible with a wide range of data systems, such as Snowflake, Redis, Spark, and Cassandra.
- **AzureML Managed Feature Store:** This is a new type of workspace that lets users discover, create, and operationalize features. This service integrates with existing data stores, feature pipelines, and ML platforms like Azure Databricks and Kubeflow. Plus, it uses SQL, PySpark, SnowPark, or Python for feature transformations and Parquet/S3 or Cosmos DB for feature storage.

You can read more about LangChain's integration with features at https://blog.langchain.dev/feature-stores-and-llms/.

Developing the front-end with Streamlit

Now that we have seen the logic behind an LLM-powered recommendation system, it is time to give a GUI to our MovieHarbor. To do so, we will once again leverage Streamlit, and we will assume the cold-start scenario. As always, you can find the whole Python code in the GitHub book reposi-

tory at **https://github.com/PacktPublishing/Building-LLM-Powered-Applications**.

As per the Globebotter application in *Chapter 6*, in this case also you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `movieharbor.py`.

Let's now summarize the key steps to build the app with the front-end:

1. Configure the application webpage:

```
import streamlit as st
st.set_page_config(page_title="GlobeBotter", page_icon="📊")
st.header('📊 Welcome to MovieHarbor, your favourite movie recommender')
```

2. Import the credentials and establish the connection to LanceDB:

```
load_dotenv()
#os.environ["HUGGINGFACEHUB_API_TOKEN"]
openai_api_key = os.environ['OPENAI_API_KEY']
embeddings = OpenAIEmbeddings()
uri = "data/sample-lancedb"
db = lancedb.connect(uri)
table = db.open_table('movies')
docsearch = LanceDB(connection = table, embedding = embeddings)
# Import the movie dataset
md = pd.read_pickle('movies.pkl')
```

3. Create some widgets for the user to define their features and movies preferences:

```
# Create a sidebar for user input
st.sidebar.title("Movie Recommendation System")
st.sidebar.markdown("Please enter your details and preferences below:")
# Ask the user for age, gender and favourite movie genre
age = st.sidebar.slider("What is your age?", 1, 100, 25)
gender = st.sidebar.radio("What is your gender?", ("Male", "Female", "Other"))
genre = st.sidebar.selectbox("What is your favourite movie genre?", md.explode('genres'
# Filter the movies based on the user input
df_filtered = md[md['genres'].apply(lambda x: genre in x)]
```

4. Define the parametrized prompt chunks:

```
template_prefix = """You are a movie recommender system that help users to find movies
Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an ansı
{context}"""
user_info = """This is what we know about the user, and you can use this information to
Age: {age}
Gender: {gender}"""
template_suffix= """Question: {question}
Your response:"""
user_info = user_info.format(age = age, gender = gender)
COMBINED_PROMPT = template_prefix +'\n'+ user_info +'\n'+ template_suffix
print(COMBINED_PROMPT)
```

5. Set up the `RetrievalQA` chain:

```
#setting up the chain
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff",
    retriever=docsearch.as_retriever(search_kwargs={'data': df_filtered}), return_sourc
```

6. Insert the search bar for the user:

```
query = st.text_input('Enter your question:', placeholder = 'What action movies do you
if query:
    result = qa({"query": query})
    st.write(result['result'])
```

And that's it! You can run the final result in your terminal with `streamlit run movieharbor.py`. It looks like the following:

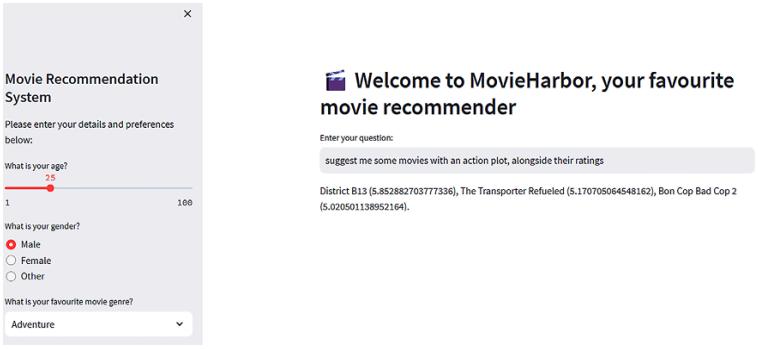

Figure 7.5: Sample front-end for Movieharbor with Streamlit

So, you can see, in just few lines of code we were able to set up a webapp for our MovieHarbor. Starting from this template, you can customize your layout with Streamlit's components, as well as tailor it to content-based scenarios. Plus, you can customize your prompts in such a way that the recommender acts as you prefer.

Summary

In this chapter, we explored how LLMs could change the way we approach a recommendation system task. We started from the analysis of the current strategies and algorithms for building recommendation applications, differentiating between various scenarios (collaborative filtering, content-based, cold start, etc.) as well as different techniques (KNN, matrix factorization, and NNs).

We then moved to the new, emerging field of research into how to apply the power of LLMs to this field, and explored the various experiments that have been done in recent months.

Leveraging this knowledge, we built a movie recommender application powered by LLMs, using LangChain as the AI orchestrator and Streamlit as the front-end, showing how LLMs can revolutionize this field thanks to their reasoning capabilities as well as their generalization. This was just one example of how LLMs not only can open new frontiers, but can also enhance existing fields of research.

In the next chapter, we will see what these powerful models can do when working with structured data.

References

- **Recommendation as Language Processing (RLP)**: A Unified **Pretrain, Personalized Prompt & Predict Paradigm (P5)**. https://arxiv.org/abs/2203.13366
- LangChain's blog about featurestores. https://blog.langchain.dev/feature-stores-and-llms/
- Feast. https://docs.feast.dev/
- Tecton. https://www.tecton.ai/
- FeatureForm. https://www.featureform.com/
- Azure Machine Learning feature store. https://learn.microsoft.com/en-us/azure/machine-learning/concept-what-is-managed-feature-store?view=azureml-api-2