Helion

<packt>

Tworzenie aplikacji AI z LlamaIndex

Praktyczny przewodnik po RAG i LLM

ANDREI GHEORGHIU

Tytuł oryginału: Building Data-Driven Applications with LlamaIndex: A practical guide to retrieval-augmented generation (RAG) to enhance LLM applications

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-2305-8

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/twapai
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 2. Rozpoczęcie pracy nad pierwszym projektem z użyciem frameworka LlamaIndex

CZĘŚĆ 3. Przeszukiwanie i praca ze zindeksowanymi danymi

CZĘŚĆ 4. Dostosowywanie, inżynieria promptów i końcowe uwagi

Rozpoczęcie pracy z LlamaIndexem

Rozdział 3

Czas zgłębić temat i zdobyć bardziej techniczną wiedzę, jak LlamaIndex działa „pod maską". W tym rozdziale zbadamy kluczowe koncepcje i elementy składowe, które tworzą architekturę LlamaIndexu. Poznasz podstawowe komponenty wykorzystywane przez framework do pobierania, strukturyzowania i przeszukiwania danych. Zrozumienie tych podstaw zapewni solidną bazę, zanim zaczniemy stosować je w praktyce. Wyjaśnię teoretyczne aspekty każdej koncepcji, a następnie połączymy teorię z praktycznym zastosowaniem.

W tym rozdziale omawiam następujące tematy:

- Podstawowe elementy LlamaIndexu: **dokumenty**, **węzły** i **indeksy**.
- Budowanie pierwszej interaktywnej aplikacji z użyciem **dużego modelu językowego**.
- Rozpoczęcie pracy nad projektem **PITS** — ćwiczenie praktyczne.

Wymagania techniczne

Aby móc uruchomić przykłady zawarte w tym rozdziale, musisz zainstalować w swoim środowisku następujące biblioteki Pythona:

- PYYAML (*https://pyyaml.org/wiki/PyYAMLDocumentation*),
- Wikipedia (*https://wikipedia.readthedocs.io/en/latest/*).

Będą również potrzebne dwa pakiety integracyjne LlamaIndexu:

- czytnik danych z Wikipedii (*https://pypi.org/project/llama-index-readers-wikipedia/*),
- modele LLM OpenAI (*https://pypi.org/project/llama-index-llms-openai/*).

Wszystkie spolonizowane przykładowe fragmenty kodu zaprezentowane w tym rozdziale znajdują się w materiałach do pobrania w podfolderze *r03* pod adresem *https://ftp.helion.pl/przyklady/twapai.zip*. Natomiast anglojęzyczną wersję kodu można znaleźć w podfolderze *ch3* w repozytorium na GitHubie pod adresem *https://github.com/PacktPublishing/Building-Data-Driven-Applications-with-LlamaIndex*.

Podstawowe elementy LlamaIndexu: dokumenty, węzły i indeksy

Rozpoczynając pracę z LlamaIndexem, należy zrozumieć kluczowe koncepcje i komponenty, które tworzą jego architekturę. Możesz traktować ten rozdział jako szybkie wprowadzenie do typowej architektury **generowania wspomaganego wyszukiwaniem** (**RAG**) z LlamaIndexem oraz przegląd najważniejszych narzędzi dostarczanych przez ten framework. Dzięki temu zrozumiesz, jak zbudować prostą aplikację RAG. W kolejnych rozdziałach będę krok po kroku omawiać szczegółowo każdy z prezentowanych tutaj komponentów.

Na wysokim poziomie LlamaIndex pomaga łączyć zewnętrzne źródła danych z dużymi modelami językowymi. Aby robić to skutecznie i wydajnie, musi pobierać, strukturyzować i organizować dane w sposób umożliwiający szybkie wyszukiwanie i wysyłanie zapytań. W tej pierwszej części rozdziału zbadamy podstawowe elementy, które umożliwiają LlamaIndexowi wzbogacenie modeli LLM: dokumenty, węzły i indeksy.

Dokumenty

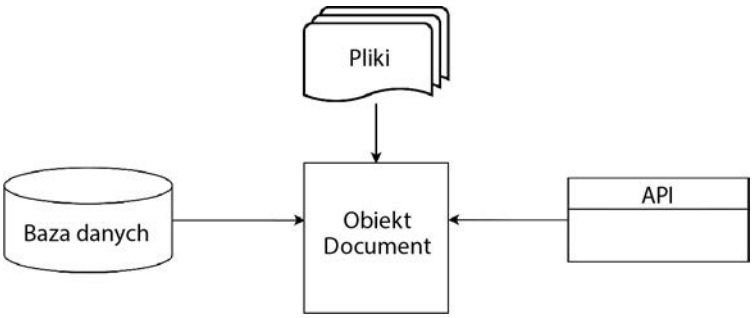
Wszystko zaczyna się od danych. Bezpośrednie zarządzanie nieprzekształconymi danymi może być równie trudne jak utrzymanie wody w dłoniach. Często dane są wszędzie i nie mają określonej struktury. Właśnie tutaj musimy wkroczyć i nadać im formę. W LlamaIndexie robimy to za pomocą czegoś, co nazywamy *Dokuments* (dokumenty). Dokument to sposób, w jaki przechowujemy dowolny rodzaj danych, niezależnie od tego, czy wprowadzasz je ręcznie, czy ładujesz z zewnętrznego źródła. To tak, jakby umieszczać dane w ładnym pudełku, aby łatwiej było nimi zarządzać.

Wyobraź sobie, że masz mnóstwo procedur swojej firmy zapisanych w plikach PDF i chcesz je zrozumieć za pomocą potężnego modelu językowego, takiego jak GPT-4. W LlamaIndexie każda z tych procedur zostałaby przekształcona w obiekt `Document` — i nie chodzi tylko o pliki. Załóżmy, że masz dane w bazie danych lub pochodzące z API — one również mogą być obiektami `Document`. Na rysunku 3.1 zostało to przedstawione w sposób graficzny.

Klasę `Document` można porównać do kontenera. Zawiera ona nie tylko nieprzekształcony tekst lub dane z miejsca, z którego pochodzą, ale także dodatkowe fragmenty informacji, które zdecydujesz się dołączyć. Ta dodatkowa informacja, zwana **metadanymi**, jest kluczowa, gdy zaczynasz przeszukiwać swoje dane, ponieważ pozwala wysyłać bardzo precyzyjne zapytania.

Oto podstawowy przykład, jak można ręcznie utworzyć obiekt `Document`:

```
from llama_index.core import Document
text = "Szybki brązowy lis przeskakuje nad leniwym psem."
doc = Document(
    text=text,
    metadata={'autor': 'Jan Kowalski','kategoria': 'inni'},
```



Rysunek 3.1. Dane mogą pochodzić z wielu źródeł, a następnie zostać przekształcone w obiekt Document

```
    id_='1'
)
print(doc)
```

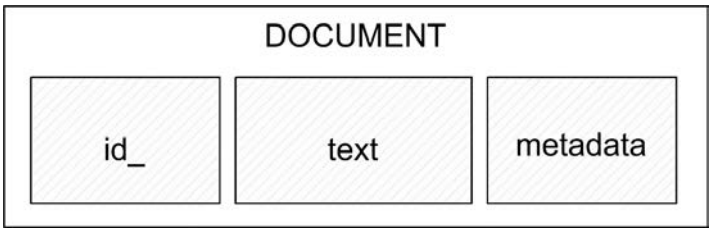
W tym przykładzie, po zaimportowaniu klasy `Document`, tworzymy obiekt `Document` o nazwie `doc`. Obiekt zawiera rzeczywisty tekst, identyfikator dokumentu oraz dodatkowe metadane według naszego wyboru, które są dostarczane jako słownik.

Oto niektóre z najważniejszych atrybutów obiektu dokumentu:

- `text` — przechowuje zawartość tekstową danych.
- `metadata` — to słownik, którego można użyć do uwzględnienia dodatkowych informacji o dokumencie, takich jak nazwa pliku czy kategorie. Klucze w słowniku metadanych muszą być łańcuchami znaków, a wartości mogą być łańcuchami, liczbami zmiennoprzecinkowymi lub całkowitymi.
- `id_` — unikalny identyfikator dla każdego obiektu `Document`. Możesz go ustawić ręcznie, jeśli chcesz, a jeśli nie określisz identyfikatora, LlamaIndex zrobi to automatycznie.

Istnieją również inne atrybuty, które można znaleźć w repozytorium GitHub LlamaIndex. Jednak aby nie komplikować, na tym etapie skupimy się tylko na tych trzech. Te atrybuty oferują różne sposoby dostosowywania i ulepszania funkcjonalności klasy `Document` w LlamaIndexie.

Rysunek 3.2 przedstawia podstawową strukturę obiektu `Document` w LlamaIndexie.



Rysunek 3.2. Podstawowa struktura obiektu Document

Obiekty `Document` zawierają dane w ich nieprzetworzonej, surowej formie. Chociaż podany przykład ilustruje, jak można ręcznie utworzyć jeden obiekt `Document`, zazwyczaj w praktycznych zastosowaniach obiekty te są generowane masowo poprzez pobieranie ich z różnych źródeł danych. To masowe pobieranie danych wykorzystuje wspomniane wcześniej **ładowarki danych** (ang. *data loaders*), nazywane też **konektorami** (ang. *connectors*) lub po prostu **czytnikami** (ang. *readers*), które pochodzą z obszernej biblioteki znanej jako LlamaHub (*https://llamahub.ai/*).

Uwaga

Gotowe do użycia pakiety, opracowane głównie przez społeczność LlamaIndexu, rozszerzają funkcjonalność kluczowych komponentów frameworka. Oferują różne modele LLM, narzędzia agentów, modele osadzania, magazyny wektorowe i ładowarki danych. Narzędzia do pobierania danych obsługują szeroki wachlarz formatów plików danych, baz danych i punktów końcowych API. Na LlamaHubie znajduje się już ponad 130 różnych czytników danych, a lista ta ciągle rośnie. Temat LlamaHuba omówię znacznie bardziej szczegółowo w następnym rozdziale. Na razie skupimy się na ładowarkach danych.

Oto podstawowy przykład automatycznego pobierania danych za pomocą jednej z predefiniowanych ładowarek danych z LlamaHuba. Przed uruchomieniem przykładu upewnij się, że zainstalowałeś biblioteki wymienione w punkcie „Wymagania techniczne" na początku tego rozdziału i przeprowadziłeś wszystkie niezbędne przygotowania środowiska, o których mowa w rozdziale 2. Bibliotekę *wikipedia* i czytnik danych z Wikipedii możesz zainstalować za pomocą tych poleceń:

```
pip install wikipedia
pip install llama-index-readers-wikipedia
```

Pierwsza biblioteka umożliwia łatwy dostęp do danych z Wikipedii i ich parsowanie, druga natomiast to integracja LlamaIndexu dla ładowarki danych z Wikipedii. Po zainstalowaniu obu bibliotek możesz uruchomić następujący przykład:

```
from llama_index.readers.wikipedia import WikipediaReader
loader = WikipediaReader()
documents = loader.load_data(
    pages=['Pythagorean theorem','General relativity']
)
print(f"Wczytano {len(documents)} dokumentów")
```

Ładowarka danych `WikipediaReader` wyodrębnia tekst z artykułów Wikipedii za pomocą pakietu `Python Wikipedia`. Oprócz `WikipediaReader` na LlamaHubie dostępnych jest wiele innych specjalistycznych konektorów danych.

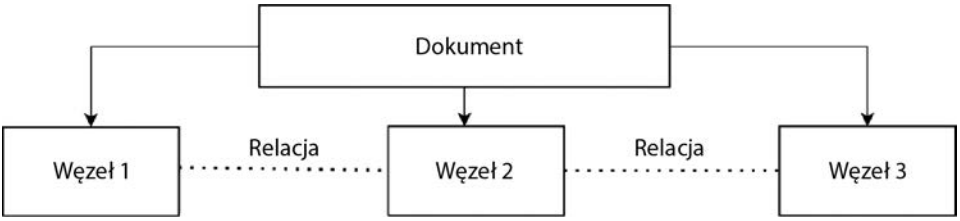
Tworzenie obiektów `Document` jest bardzo łatwym procesem. Ale jak surowe obiekty `Document` zostają przekształcone w format, który modele LLM mogą efektywnie przetwarzać i analizować? W tym miejscu do akcji wkraczają węzły (ang. *nodes*).

Węzły

Podczas gdy obiekty `Document` reprezentują nieprzekształcone dane i mogą być używane jako takie, węzły są mniejszymi fragmentami treści wyodrębnionymi z obiektów `Document`. Ma to na celu rozbicie danych zapisanych w obiektach `Document` na mniejsze, bardziej zrozumiałe fragmenty tekstu. Węzeł ma następujące zadania:

- **Pozwala dopasować naszą wewnętrzną wiedzę do limitów zapytań modelu**. Wyobraź sobie, że mamy wewnętrzną procedurę, która ma 50 stron. Na pewno napotkalibyśmy problemy z limitem rozmiaru, próbując umieścić to w kontekście naszego zapytania. Jednak w praktyce najprawdopodobniej nie musielibyśmy dodawać całej procedury w jednym zapytaniu. Dlatego wybieranie tylko odpowiednich węzłów może rozwiązać ten problem.
- **Tworzy semantyczne jednostki danych skupione wokół określonych informacji**. To może ułatwić pracę z danymi i ich analizę, ponieważ są one podzielone na mniejsze, bardziej skoncentrowane jednostki.
- **Umożliwia tworzenie relacji między węzłami**. Oznacza to, że węzły mogą być powiązane ze sobą na podstawie ich relacji, tworząc sieć połączonych danych. Może to być przydatne do zrozumienia połączeń i zależności między różnymi fragmentami informacji zawartymi w obiektach `Document`.

Na rysunku 3.3 pokazałem to w formie graficznej.



Rysunek 3.3. Relacje między węzłami wyciągniętymi z dokumentu

Ponadto w LlamaIndexie węzły mogą przechowywać obrazy, ale w niniejszej książce nie będziemy skupiać się na tej funkcjonalności. Od teraz naszym głównym bohaterem będzie klasa `TextNode`.

Oto lista niektórych ważnych atrybutów klasy `TextNode`:

- `text` — fragment tekstu pochodzący z oryginalnego obiektu `Document`.
- `start_char_idx` i `end_char_idx` — opcjonalne wartości całkowite, które mogą przechowywać pozycje początkowego i końcowego znaku tekstu w obiekcie `Document`. Może to być pomocne, gdy tekst jest częścią większego dokumentu i potrzeba dokładnie określić jego położenie.
- `text_template` i `metadata_template` — pola szablonów, które definiują, jak formatowane są tekst i metadane. Pomagają w uzyskaniu bardziej strukturalnej i czytelnej reprezentacji obiektu `TextNode`.

- `metadata_seperator` — pole tekstowe, które definiuje separator między polami metadanych. Gdy uwzględnionych jest wiele elementów metadanych, ten separator służy do utrzymania czytelności i struktury.
- `metadata` — wszelkie przydatne metadane, takie jak identyfikator nadrzędnego obiektu `Document`, relacje z innymi węzłami oraz opcjonalne tagi. Te metadane mogą posłużyć do przechowywania dodatkowego kontekstu, gdy zachodzi taka potrzeba. Omówię to bardziej szczegółowo w rozdziale 4. „Wprowadzanie danych do przepływu pracy RAG".

Podobnie jak w przypadku klasy `Document`, pełną listę atrybutów `TextNode` znajdziesz w repozytorium GitHub pod adresem *https://github.com/run-llama/llama_index/blob/main/llama-index-core/llama_index/core/schema.py*.

Należy wiedzieć, że węzły automatycznie odziedziczą wszelkie metadane obecne na poziomie dokumentu, ale metadane węzłów mogą być również indywidualnie dostosowywane. Węzły w LlamaIndexie można tworzyć na kilka sposobów, które omówię w kolejnych punktach. Zacznijmy od ręcznego tworzenia węzłów.

Ręczne tworzenie obiektu węzła

Oto przykład, jak można ręcznie utworzyć obiekty `Node`:

```
from llama_index.core import Document
from llama_index.core.schema import TextNode
doc = Document(text="To jest przykładowy tekst dokumentu")
n1 = TextNode(text=doc.text[0:19], doc_id=doc.id_)
n2 = TextNode(text=doc.text[20:35], doc_id=doc.id_)
print(n1)
print(n2)
```

W tym przykładzie wykorzystujemy możliwości dzielenia tekstu w Pythonie do ręcznego wyodrębniania tekstu dla dwóch węzłów. To podejście może być bardzo przydatne, gdy chcesz mieć pełną kontrolę zarówno nad tekstem węzłów, jak i nad towarzyszącymi metadanymi.

Aby zrozumieć, co dzieje się w tle, spójrzmy na wynik tego kodu:

```
Node ID: e1d6e6cb-3d46-4ced-b66a-deed90216162
Text: To jest przykładowy
Node ID: 32f1562f-5d89-4219-8408-1b1e8a28783d
Text: tekst dokumentu
```

Uwaga

Jak widzisz, oba węzły zawierają losowo wygenerowany identyfikator oraz fragmenty tekstu, które wycięliśmy z oryginalnego obiektu dokumentu. Konstruktor `TextNode` automatycznie wygenerował identyfikator dla każdego węzła, korzystając z modułu UUID w Pythonie. Jeśli chcemy zastosować inny schemat identyfikacji, możemy zmienić ten identyfikator po utworzeniu węzłów.

Automatyczne wyodrębnianie węzłów z dokumentów za pomocą separatorów

Ponieważ **dzielenie dokumentów** jest bardzo ważne w przepływie pracy RAG, do tego celu LlamaIndex ma wbudowane narzędzia. Jednym z takich narzędzi jest `TokenText ↪Splitter`.

W ramach przykładu pokazującego, jak automatycznie generować węzły, `TokenText ↪Splitter` próbuje podzielić tekst dokumentu na fragmenty zawierające całe zdania. Każdy fragment będzie zawierał jedno lub więcej zdań. Ponadto stosowane jest domyślne nakładanie się fragmentów, aby zachować więcej kontekstu.

Pod spodem istnieje wiele parametrów, które możemy dostosować w `TokenTextSplitter`, takich jak `chunk_size` i `chunk_overlap`, ale te parametry oraz sposób działania tego dzielnika tekstu omówię szerzej w następnym rozdziale. Na razie spójrzmy na prosty przykład, jak użyć narzędzia `TokenTextSplitter` z jego domyślnymi ustawieniami na obiekcie `Document`:

```
from llama_index.core import Document
from llama_index.core.node_parser import TokenTextSplitter
doc = Document(
    text=(
    "To jest zdanie 1. To jest zdanie 2. "
    "Tutaj zdanie 3."
    ),
    metadata={"autor": "Jan Kowalski"}
)
splitter = TokenTextSplitter(
    chunk_size=16,
    chunk_overlap=0,
    separator=" "
)
nodes = splitter.get_nodes_from_documents([doc])
for node in nodes:
    print(node.text)
    print(node.metadata)
```

Ten kod zwrócił następujący rezultat:

```
Metadata length (9) is close to chunk size (16). Resulting chunks are less
↪than 50 tokens. Consider increasing the chunk size or decreasing the size
↪of your metadata to avoid this.
To jest zdanie 1.
{'autor': 'Jan Kowalski'}
To jest zdanie 2.
{'autor': 'Jan Kowalski'}
Tutaj zdanie 3.
{'autor': 'Jan Kowalski'}
```

Uwaga

Biorąc pod uwagę, że rozmiar fragmentu określa, ile treści może być przetworzone jednocześnie, jeśli metadane są zbyt duże, zajmą większość miejsca w każdym fragmencie, pozostawiając mniej miejsca na faktyczną treść. Może to powodować, że fragmenty będą się składać głównie z metadanych i będą zawierać bardzo mało faktycznej treści. W naszym przykładzie widzimy ostrzeżenie („Długość metadanych (9) jest zbliżona do rozmiaru fragmentu (12). Powstałe fragmenty mają mniej niż 50 tokenów. Rozważ zwiększenie rozmiaru fragmentu lub zmniejszenie rozmiaru metadanych, aby temu zapobiec."), ponieważ nasz rozmiar (rozmiar fragmentu minus miejsce zajmowane przez metadane) przełoży się na fragmenty, które mają mniej niż 50 tokenów. Jest to zbyt mało dla wydajnego działania.

To był tylko podstawowy przykład mający na celu zilustrowanie automatycznej metody dzielenia danych na oddzielne węzły. Jeśli spojrzysz na metadane każdego węzła, zauważysz również, że zostały one automatycznie odziedziczone z oryginalnego dokumentu.

Czy istnieją inne sposoby tworzenia węzłów?

Tak, istnieje kilka innych metod. W następnym rozdziale dokładniej przybliżę techniki dzielenia tekstu i analizy węzłów dostępne w LlamaIndexie. Będziesz mieć również okazję zrozumieć sposób działania tych technik i zobaczysz jakie, opcje dostosowywania oferują.

Ale to nie koniec. Jest jeszcze kilka rzeczy, które musisz wiedzieć o węzłach.

Węzły nie lubią być same — pragną relacji

Skoro omówiłem już podstawowe przykłady tworzenia prostych węzłów, co powiesz na dodanie między nimi relacji?

Oto przykład, który ręcznie tworzy prostą relację między dwoma węzłami:

```
from llama_index.core import Document
from llama_index.core.schema import (
    TextNode,
    NodeRelationship,
    RelatedNodeInfo
)
doc = Document(text="Pierwsze zdanie. Drugie zdanie.")
n1 = TextNode(text="Pierwsze zdanie.", node_id=doc.doc_id)
n2 = TextNode(text="Drugie zdanie.", node_id=doc.doc_id)
n1.relationships[NodeRelationship.NEXT] = n2.node_id
n2.relationships[NodeRelationship.PREVIOUS] = n1.node_id
print(n1.relationships)
print(n2.relationships)
```

W tym przykładzie ręcznie utworzyliśmy dwa węzły i zdefiniowaliśmy relację PREVIOUS (poprzedni) lub NEXT (następny) między nimi. Relacja ta śledzi kolejność węzłów w oryginalnym dokumencie. Ten kod informuje LlamaIndex, że dwa węzły należą do pierwotnego obiektu Document i występują w określonej kolejności.

Rysunek 3.4 pokazuje, co LlamaIndex rozumie po uruchomieniu wyżej pokazanego kodu.

Rysunek 3.4. Relacja typu poprzedni lub następny między dwoma węzłami

Uwaga

Wiedz, że LlamaIndex zawiera niezbędne narzędzia do *automatycznego* tworzenia relacji między węzłami. Na przykład podczas korzystania ze wspomnianych wcześniej automatycznych analizatorów węzłów w ich domyślnej konfiguracji LlamaIndex automatycznie utworzy relacje „poprzedni" lub „następny" między generowanymi węzłami.

Istnieją inne typy relacji, które możemy zdefiniować. Oprócz prostych relacji, takich jak *poprzedni* lub *następny*, węzły mogą być połączone za pomocą następujących relacji:

- SOURCE — **relacja źródła** reprezentuje oryginalny źródłowy obiekt Document, z którego węzeł został wyodrębniony lub przetworzony. Kiedy przetwarzasz dokument na wiele węzłów, używając relacji źródła, możesz śledzić, z którego obiektu pochodzi każdy węzeł.
- PARENT — **relacja nadrzędna** wskazuje na strukturę hierarchiczną, w której węzeł z tą relacją jest o jeden poziom wyżej niż powiązany węzeł. W strukturze drzewa węzeł nadrzędny miałby jeden lub więcej węzłów podrzędnych (tzw. dzieci). Ta relacja jest używana do poruszania się po zagnieżdżonych strukturach danych lub do zarządzania nimi. Główny węzeł i podrzędne węzły reprezentują sekcje, paragrafy lub inne podziały głównego węzła.
- CHILD — jest to przeciwieństwo relacji PARENT. Węzeł z **relacją podrzędną** jest podporządkowany innemu węzłowi — rodzicowi. Węzły potomne (podrzędne) można postrzegać jako liście lub gałęzie w strukturze drzewa wychodzące od węzła nadrzędnego.

Ale dlaczego relacje są ważne? Zobaczmy, dlaczego relacje są tak użyteczne.

Dlaczego relacje są ważne?

Tworzenie relacji między węzłami w LlamaIndexie może być użyteczne z kilku powodów:

- **Umożliwia bardziej kontekstowe zapytania**. Poprzez łączenie węzłów możesz wykorzystać ich relacje podczas zapytań, aby pobrać dodatkowy, istotny kontekst. Na przykład podczas wyszukiwania węzła możesz również zwrócić poprzednie lub następne węzły, aby dostarczyć więcej kontekstu.
- **Pozwala śledzić pochodzenie**. Relacje zapisują, skąd pochodzą źródłowe węzły i jak są połączone. Jest to przydatne na przykład wtedy, gdy musisz znaleźć oryginalne źródło węzła.
- **Umożliwia poruszanie się po węzłach**. Poruszanie się po węzłach według ich relacji umożliwia nowe typy zapytań, na przykład znalezienie następnego węzła zawierającego jakieś słowo kluczowe. Nawigacja wzdłuż relacji zapewnia dodatkowy wymiar do wyszukiwania.
- **Wspiera budowę grafów wiedzy**. Węzły i relacje są budulcem grafów wiedzy. Łączenie węzłów w strukturę grafu pozwala budować grafy wiedzy z tekstu przy użyciu LlamaIndexu. Więcej o grafach wiedzy opowiem w rozdziale 5. „Indeksowanie z LlamaIndexem".
- **Poprawia strukturę indeksu**. Niektóre indeksy LlamaIndexu, takie jak drzewa i grafy, wykorzystują relacje węzłów do budowy swojej wewnętrznej struktury. Relacje pozwalają budować bardziej złożone i ekspresywne topologie indeksów. Omówię to szerzej w rozdziale 5. „Indeksowanie z LlamaIndexem".

Podsumowując: relacje wzbogacają węzły o dodatkowe połączenia kontekstowe, co przekłada się na bardziej ekspresywne zapytania, budowę grafów wiedzy śledzących źródła oraz złożone struktury indeksów.

Po przetworzeniu surowych danych jako dokumentów i poukładaniu ich w węzły, do których można wysyłać zapytania, ostatnim krokiem jest zorganizowanie węzłów w wydajne indeksy.

Indeksy

Trzecia ważna koncepcja — indeks — odnosi się do szczególnej struktury danych używanej do organizowania zbioru węzłów w celu optymalizacji przechowywania i wyszukiwania.

Właśnie dlatego indeksowanie danych jest tak ważne podczas ich przygotowywania pod kątem rozszerzania modeli LLM. Bez indeksowania dane to nieuporządkowany stos faktów i plików, co jest jak przeszukiwanie pękającej w szwach walizki, by znaleźć skarpetki do pary.

Właściwe indeksowanie porządkuje informacje w logiczne kategorie. Na przykład rekordy sprzedaży znajdują się w jednym indeksie, a prośby o pomoc w innym. To jak pakowanie powiązanych przedmiotów razem. Przekształca to chaotyczne dane

Uproszczona analogia

Przygotowanie danych do RAG jest trochę jak pakowanie ubrań na daleką podróż. Wszystko musi być odpowiednio poukładane i łatwo dostępne! Powiedzmy, że pakujesz się na ważną podróż służbową. Możesz po prostu wrzucić wszystko do walizki, ale wówczas koszule, skarpetki, spodnie i inne rzeczy będą pomieszane! Problem polega na tym, że gdy chcesz szybko sięgnąć po to, czego potrzebujesz, możesz wyciągnąć niewłaściwy ciuch i ubrać się nieodpowiednio do okazji.

w starannie zorganizowaną wiedzę, z której sztuczna inteligencja może skorzystać. Zamiast losowo przeszukiwać walizkę, sięgasz do niestandardowych kieszeni dokładnie po to, czego potrzebujesz.

Pamiętaj więc, że aby zaoszczędzić sobie nerwów i nie marnować czasu w przyszłości, warto już na początku pochylić się nad indeksowaniem i strukturyzacją danych. To znacznie ułatwi Ci pracę.

LlamaIndex wspiera różne typy indeksów, a z każdym z nich wiąże się „coś za coś". Oto lista niektórych dostępnych typów indeksów:

- `SummaryIndex` — bardzo podobny do segregatora na przepisy: utrzymuje węzły w porządku, dzięki czemu możesz uzyskać do nich dostęp jeden po drugim. Przyjmuje zestaw dokumentów, dzieli je na węzły, a następnie łączy w listę. Jest świetny do przeglądania dużego dokumentu.
- `DocumentSummaryIndex` — tworzy zwięzłe podsumowanie dla każdego dokumentu, mapując te podsumowania do ich odpowiednich węzłów. Ułatwia wydajne wyszukiwanie informacji poprzez używanie tych podsumowań do szybkiego znajdowania odpowiednich dokumentów.
- `VectorStoreIndex` — jeden z bardziej zaawansowanych typów indeksów i prawdopodobnie najczęściej wykorzystywany w większości aplikacji RAG. Konwertuje tekst na osadzenia wektorowe i używa matematyki do grupowania podobnych węzłów, pomagając zlokalizować podobne węzły.
- `TreeIndex` — idealne rozwiązanie dla tych, którzy kochają porządek. Ten indeks działa podobnie do umieszczania mniejszych pudełek w większych, organizując węzły według poziomów w strukturze drzewa. W środku każdy węzeł nadrzędny przechowuje podsumowania węzłów podrzędnych. Te są generowane przez duże modele językowe z użyciem ogólnego polecenia podsumowania. Ten indeks może być bardzo przydatny do podsumowania.
- `KeywordTableIndex` — wyobraź sobie, że musisz znaleźć danie na podstawie składników, które posiadasz. Indeks słów kluczowych łączy ważne słowa z węzłami, w których się znajdują. Ułatwia to znalezienie dowolnego węzła poprzez wyszukiwanie słów kluczowych.
- `KnowledgeGraphIndex` — przydatny, gdy trzeba powiązać fakty w dużej sieci danych przechowywanej jako graf wiedzy. Jest dobry do odpowiadania na trudne pytania dotyczące wielu powiązanych informacji.

- `ComposableGraph` — umożliwia tworzenie złożonych struktur indeksów, w których indeksy na poziomie dokumentów są indeksowane w kolekcjach wyższego poziomu. Można nawet zbudować indeks indeksów, jeśli chcesz uzyskać dostęp do danych z wielu dokumentów w większej kolekcji dokumentów.

Omówię dokładniej działanie tych indeksów i innych wariantów w rozdziale 5. „Indeksowanie z LlamaIndexem".

Wszystkie typy indeksów w LlamaIndexie mają wspólne podstawowe cechy:

- **Budowanie indeksu**. Każdy typ indeksu można zbudować poprzez przekazanie zestawu węzłów podczas inicjalizacji indeksów. Tworzy to podstawową strukturę indeksu.
- **Wstawianie nowych węzłów**. Po zbudowaniu indeksu można ręcznie wstawiać nowe węzły, które są uwzględniane w istniejącej strukturze indeksu.
- **Zapytania do indeksu**. Po zbudowaniu indeksy udostępniają interfejs zapytania do pobierania odpowiednich węzłów w oparciu o konkretne zapytanie. Logika pobierania różni się w zależności od typu indeksu.

Szczegóły struktury indeksu i zapytań różnią się w zależności od typów indeksów. Jednak schemat budowania, wstawiania i wysyłania zapytań jest spójny. Zrozumienie szczególnych cech każdego typu indeksu jest naprawdę ważne, jeśli chcesz w pełni wykorzystać ich potencjał. W rozdziale 5. „Indeksowanie z LlamaIndexem" omówię ten temat znacznie bardziej szczegółowo i podam konkretne przykłady dla każdego typu indeksu.

Na razie rozważmy prosty przykład ilustrujący tworzenie `SummaryIndex`:

```
from llama_index.core import SummaryIndex, Document
from llama_index.core.schema import TextNode
nodes = [
  TextNode(
      text="Lionel Messi to piłkarz z Argentyny."
  ),
  TextNode(
      text="Wygrał Złotą Piłkę 7 razy."
  ),
  TextNode(text="Rodzinnym miastem Lionela Messiego jest Rosario."),
  TextNode(text="Urodził się 27 czerwca 1987 roku.")
]
index = SummaryIndex(nodes)
```

To jest bardzo proste do zrozumienia. Najpierw zdefiniowaliśmy zestaw węzłów zawierających dane, a następnie utworzyliśmy `SummaryIndex` na podstawie tych węzłów. Ten indeks to prosta struktura danych oparta na liście.

Wyobraź sobie, że `SummaryIndex` to mały notatnik, w którym zapisujesz punkty z wielu opowieści. Podczas tworzenia indeksu wiele opowieści jest rozbijanych na mniejsze fragmenty i układanych w listę. A co jest w tym najlepsze? To, że LlamaIndex nawet nie musi używać dużego modelu językowego, kiedy buduje ten typ indeksu.

Czy to już wszystko?

Prawie. Indeksy są świetne do organizowania danych, ale jak uzyskujemy z nich odpowiedzi? Tutaj wchodzą do gry **mechanizmy wyszukiwania** (ang. *retrievers*) i **syntezatory odpowiedzi** (ang. *response synthesizers*)!

Jako przykładu użyjmy indeksu Lionela Messiego, który właśnie stworzyliśmy. Powiedzmy, że chcesz zapytać: „Jakie jest rodzinne miasto Messiego?". W kodzie robimy to tak:

```
query_engine = index.as_query_engine()
response = query_engine.query("Jak nazywa się rodzinne miasto Lionela
↪Messiego?")
print(response)
```

A oto rezultat:

```
Rosario.
```

Indeks podsumowujący układa po kolei wszystkie węzły na liście.

Przy zapytaniu wyszukuje wszystkie węzły, co umożliwia syntezę odpowiedzi z pełnym kontekstem.

Jak to właściwie działa?

Obiekt `QueryEngine` (silnik zapytań) zawiera mechanizm wyszukiwania, który jest odpowiedzialny za wyszukanie odpowiednich węzłów z indeksu dla zapytania. Mechanizm wyszukiwania przeszukuje indeks, aby pobrać i uszeregować odpowiednie węzły dla tego zapytania. Wyszukuje węzły z indeksu, które prawdopodobnie zawierają informacje o mieście rodzinnym Messiego.

Ale samo uzyskanie listy węzłów nie jest zbyt użyteczne. W tym momencie do akcji wkracza inna część obiektu `QueryEngine`, zwana **postprocesorem węzłów**. Umożliwia ona przekształcenie, ponowne uszeregowanie lub przefiltrowanie węzłów po ich pobraniu i przed stworzeniem ostatecznej odpowiedzi. Istnieje wiele typów postprocesorów, a każdy z nich można skonfigurować i dostosować w zależności od przypadku użycia.

Obiekt `QueryEngine` zawiera również syntezator odpowiedzi, który bierze pobrane węzły i tworzy ostateczną odpowiedź przy użyciu dużego modelu językowego poprzez wykonanie następujących kroków:

1. Syntezator odpowiedzi bierze węzły wybrane przez mechanizm wyszukiwania i przetworzone przez postprocesor węzłów, a następnie formatuje je w prompt dla modelu LLM.
2. Prompt zawiera zapytanie wraz z kontekstem z węzłów.
3. Ten prompt jest przekazywany do modelu LLM w celu wygenerowania odpowiedzi.

4. Wszelkie niezbędne procesy końcowe są wykonywane na nieprzekształconej odpowiedzi przy użyciu modelu LLM, aby zwrócić ostateczną odpowiedź w języku naturalnym.

Zatem `index.as_query_engine()` tworzy dla nas pełny silnik zapytań, zawierający domyślną wersję trzech elementów: mechanizmu wyszukiwania, postprocesora węzłów i syntezatora odpowiedzi.

W rozdziałach 6. i 7. omówię te trzy elementy bardziej szczegółowo.

Ostatecznym wynikiem działania tego silnika będzie odpowiedź w języku naturalnym, na przykład: „Miasto rodzinne Messiego to Rosario".

Pamiętaj

To tylko podstawowy przykład z użyciem wybranego typu indeksu, w tym przypadku `SummaryIndex`. Każdy typ indeksu działa inaczej, co omówię w rozdziale 5. Na przykład `TreeIndex` układa węzły w hierarchię, co pozwala na podsumowanie, a `Key`
`↪wordIndex` mapuje słowa kluczowe dla szybkiego wyszukiwania. Struktura indeksu wpływa na wydajność i określa jego najlepsze zastosowania. Sama struktura indeksu definiuje logikę zarządzania danymi. Jak widzieliśmy, indeks musi być połączony z mechanizmem wyszukiwania, postprocesorem i syntezatorem odpowiedzi, aby utworzyć pełną ścieżkę zapytań, co pozwala aplikacjom wykorzystywać indeksowane dane.

Więcej dowiesz się w kolejnych rozdziałach. Na tym etapie powinieneś mieć ogólne pojęcie o indeksach i ich roli.

Spójrzmy na rysunek 3.5, na którym pokazałem cały proces.

Jak widać na rysunku 3.5, proces obejmuje następujące kroki:

1. Wczytywanie danych jako dokumentów.
2. Przekształcanie dokumentów w spójne węzły.
3. Budowanie zoptymalizowanego indeksu z węzłów.
4. Wysyłanie zapytań do indeksu w celu pobrania odpowiednich węzłów.
5. Syntezowanie ostatecznej odpowiedzi.

Za dużo do zapamiętania? Podsumujmy elementy składowe LlamaIndexu.

Krótki przegląd kluczowych koncepcji

Oto szybkie podsumowanie tego, co omówiliśmy do tej pory:

- **Dokumenty** — nieprzetworzone dane.
- **Węzły** — logiczne fragmenty wyodrębnione z dokumentów.

Rysunek 3.5. Pełny przepływ pracy RAG w LlamaIndexie

- **Indeksy** — struktury danych organizujące węzły w zależności od przypadku użycia.
- **Obiekt QueryEngine** — zawiera mechanizm wyszukiwania, postprocesor węzłów i syntezator odpowiedzi.

Zrozumienie tych elementów składowych jest niezbędne podczas pracy z LlamaIndexem. Dzięki temu będziesz skutecznie strukturyzować i łączyć zewnętrzne dane z dużymi modelami językowymi.

Znasz już podstawy. Teraz utrwalimy tę wiedzę, przyglądając się uproszczonemu modelowi przepływu pracy i budując aplikację.

Budowanie pierwszej interaktywnej aplikacji z użyciem dużego modelu językowego

Czas połączyć kropki i wykorzystać tę wiedzę w praktyce. Pokazany dotychczas kod możemy połączyć w naszą pierwszą aplikację zbudowaną za pomocą LlamaIndexu.

Zanim przejdziesz dalej, upewnij się, że spełniłeś wymagania techniczne wspomniane na początku rozdziału. Do poniższego przykładu kodu będziemy potrzebować biblioteki

`Wikipedia`, aby móc przetworzyć artykuł z Wikipedii i wyciągnąć z niego przykładowe dane.

Po pomyślnym zainstalowaniu biblioteki `Wikipedia` przykładowa aplikacja powinna działać bez problemów. Oto kod:

```
from llama_index.core import Document, SummaryIndex
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.readers.wikipedia import WikipediaReader
loader = WikipediaReader()
documents = loader.load_data(pages=["Messi Lionel"])
parser = SimpleNodeParser.from_defaults()
nodes = parser.get_nodes_from_documents(documents)
index = SummaryIndex(nodes)
query_engine = index.as_query_engine()
print("Zapytaj mnie o cokolwiek, co dotyczy Lionela Messiego!")
while True:
    question = input("Twoje pytanie: ")
    if question.lower() == "koniec":
        break
    response = query_engine.query(question)
    print(response)
```

Należy zauważyć, że aplikacja nie działa jak prawdziwy czat, ponieważ nie zachowuje kontekstu rozmowy. Tę aplikację można określić jako prosty system pytań i odpowiedzi.

Oto krótkie omówienie kodu:

1. Zaczynamy od wczytania strony Wikipedii o Lionelu Messim jako dokumentu za pomocą ładowarki danych `WikipediaReader`. Powoduje to przetworzenie surowych danych tekstowych.
2. Dzielimy dokument na mniejsze fragmenty przy użyciu `SimpleNodeParser`. Następuje podział tekstu na logiczne segmenty.
3. Budujemy indeks `SummaryIndex` z węzłów. To organizuje węzły sekwencyjnie w celu wyszukania pełnego kontekstu.
4. Definiujemy obiekt `QueryEngine`, tworząc kompletną ścieżkę zapytań.
5. Tworzymy pętlę, która przesyła zapytanie do indeksu, przekazując nasze pytanie do obiektu `QueryEngine`. Następuje pobranie odpowiednich węzłów, utworzenie promptu dla modelu LLM i zwrócenie końcowej odpowiedzi.

Spójrz ponownie na rysunek 3.5, aby zobaczyć cały proces: przetwarzanie danych, dzielenie ich na węzły, budowanie indeksu i zapytania w celu pobrania i zsyntezowania ostatecznej odpowiedzi.

Ale co, jeśli chcemy wiedzieć dokładnie, co dzieje się w tle?

Wykorzystanie funkcji rejestru w LlamaIndexie do zrozumienia logiki i debugowania aplikacji

Kiedy uruchamiasz kod tak jak w naszym poprzednim przykładzie, możesz mieć wrażenie, że coś magicznego dzieje się za kulisami. Przekazujesz tekst, wywołujesz prostą metodę indeksowania i nagle możesz zacząć wysyłać zapytania do asystenta sztucznej inteligencji zasilanego Twoimi danymi.

Ale gdy Twoje aplikacje staną się bardziej skomplikowane, zechcesz zrozumieć dokładnie, jak LlamaIndex działa „pod maską". Wtedy **zapisywanie informacji z działania aplikacji w rejestrze** (ang. *logging*) staje się ważne. LlamaIndex dostarcza wiele przydatnych informacji, które pokazują krok po kroku, co dzieje się podczas indeksowania i obsługi zapytań. To jak mieć małego narratora debugowania, który opisuje każdą akcję.

Włączenie podstawowego zapisu w rejestrze jest tak proste, jak dodanie tego kodu:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Po włączeniu tej funkcji zobaczysz, jak LlamaIndex wykonuje następujące czynności:

- Przekształca Twoje dokumenty w węzły.
- Decyduje, jakiej użyć struktury indeksowania.
- Formatuje prompty dla dużego modelu językowego.
- Wyszukuje odpowiednie węzły na podstawie Twoich zapytań.
- Syntezuje odpowiedź z węzłów.

Jak zobaczysz w następnych rozdziałach, logi ujawniają także przydatne dane, takie jak:

- liczba tokenów użytych do wywołań API,
- informacje o opóźnieniach,
- wszelkie ostrzeżenia lub błędy.

Uwaga

Gdy coś nie działa zgodnie z oczekiwaniami, nie panikuj! Po prostu sprawdź rejestr. Dostarcza on kluczowych wskazówek do identyfikacji problemów. Na razie użycie podstawowej funkcji logowania powinno być wystarczające. Gdy będziesz mieć włączoną tę funkcję, większość „zakulisowych" działań będzie wyświetlana w czasie działania, więc będziesz mógł monitorować działanie swojej aplikacji krok po kroku. Więcej o zaawansowanym debugowaniu powiem w rozdziale 9. „Dostosowywanie i wdrażanie projektu stworzonego za pomocą LlamaIndexu".

A co powiesz na mały tuning?

Dostosowywanie modelu LLM używanego przez LlamaIndex

Załóżmy, że chcielibyśmy skonfigurować framework, aby używał innego modelu LLM. Domyślnie LlamaIndex korzysta z API OpenAI z modelem **GPT-3.5-Turbo**. Oto przegląd kluczowych cech GPT-3.5-Turbo:

- Jest szybszy i tańszy w użyciu niż **GPT-4**.
- Choć nie jest tak zaawansowany jak inne modele, takie jak GPT-4, jest bardzo zdolnym modelem generatywnym i konwersacyjnym.
- Może działać bardzo dobrze w różnych zadaniach **przetwarzania języka naturalnego**, takich jak klasyfikacja, podsumowanie czy tłumaczenie.

Teraz wiadomo, dlaczego twórcy LlamaIndexu wybrali ten model. Biorąc wszystko pod uwagę, dla większości przypadków użycia zapewnia on dobrą równowagę między wydajnością a kosztami. W przypadku znakomitej większości aplikacji jest prawdopodobnie wystarczający. Jak już miałeś okazję się przekonać, jeśli przetestowałeś naszą pierwszą aplikację, radzi sobie całkiem dobrze z pytaniami o Lionelu Messim.

Ale co, jeśli musimy dostosować aplikację do bardziej szczególnego przypadku? Załóżmy, że potrzebujemy najlepszej możliwej wydajności modelu GPT-4 lub większego kontekstu dostarczanego przez model **Claude-2** albo chcemy użyć sztucznej inteligencji open source.

Łatwe jak 1, 2, 3

Musimy dodać tylko trzy linie kodu na początku naszej aplikacji:

```
from llama_index.llms.openai import OpenAI
from llama_index.core.settings import Settings
Settings.llm = OpenAI(temperature=0.8, model="gpt-4")
```

Linię `Settings.llm` należy dodać bezpośrednio po importach, aby miała zastosowanie do wszystkich innych operacji. Oto wyjaśnienie poszczególnych kroków:

1. Pierwsza linia importuje klasę OpenAI z `llama_index.llms.openai`, abyśmy mogli użyć jej do zainicjalizowania modelu LLM OpenAI.
2. Drugi import odpowiada za klasę `Settings`, którą wykorzystamy do dostosowania modelu LLM.
3. Następnie konfigurujemy `Settings` z instancją LLM OpenAI, podając model GPT-4 i temperaturę ustawioną na `0.8`, co nadpisuje domyślne ustawienia dla modelu LLM.

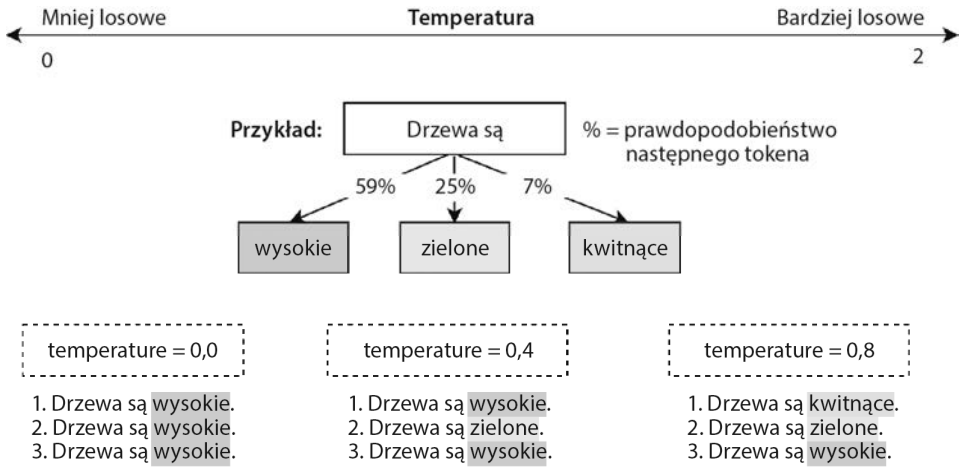
Właśnie skonfigurowaliśmy LlamaIndex, aby używał GPT-4 do wszystkich operacji zamiast domyślnego modelu GPT-3.5-Turbo. Następna część kodu zbuduje indeks i uruchomi proste zapytanie, używając nowo skonfigurowanego modelu LLM:

```
from llama_index.core.schema import TextNode
from llama_index.core import SummaryIndex
nodes = [
    TextNode(text="Rodzinnym miastem Lionela Messiego jest Rosario."),
    TextNode(text="Urodził się 27 czerwca 1987 roku.")
]
index = SummaryIndex(nodes)
query_engine = index.as_query_engine()
response = query_engine.query(
    "Jak nazywa się rodzinne miasto Lionela Messiego?"
)
print(response)
```

Teraz musimy omówić parametr `temperature`.

Parametr temperatury

W modelach OpenAI, takich jak GPT-3.5 i GPT-4, parametr temperatury (`temperature`) kontroluje losowość i kreatywność odpowiedzi sztucznej inteligencji. Spójrz na rysunek 3.6, który pokazuje, jak działa ten parametr.



Rysunek 3.6. Wpływ parametru temperatury na różnorodność odpowiedzi

Wartości temperatury dla modeli OpenAI wahają się od 0 do 2. Wyższe wartości generują bardziej losowe i kreatywne odpowiedzi. Niższe wartości produkują bardziej skoncentrowane i deterministyczne odpowiedzi.

Wartość parametru `temperature` równa 0 spowoduje, że dla tego samego promptu model za każdym razem będzie zwracał prawie identyczne wyniki. Zauważ, że użyłem słowa „prawie". To dlatego, że nawet przy ustawieniu tego parametru na 0 większość modeli prawdopodobnie nadal będzie produkować niewielkie wariacje odpowiedzi dla tego samego promptu. Jest to spowodowane wewnętrzną losowością w inicjalizacji modelu

lub subtelnymi różnicami w wewnętrznym stanie modelu, które mogą wystąpić z powodu takich czynników jak ograniczenia precyzji arytmetyki zmiennoprzecinkowej lub stochastyczna natura niektórych operacji w ramach sieci neuronowej. Nawet przy wartości temperatury równej 0, która ma na celu zminimalizowanie losowości, te małe różnice mogą prowadzić do nieznacznie różnych wyników dla identycznych danych wejściowych.

Ustawienie odpowiedniej temperatury zależy od przypadku użycia — czy chcesz, aby odpowiedzi były mocno oparte na faktach, czy bardziej kreatywne. Do generowania kodu lub zadań analizy danych odpowiednia byłaby wartość temperatury `0.2`, natomiast dla zadań wymagających większej kreatywności, takich jak pisanie lub odpowiedzi czatbotów, lepszy byłby parametr `temperature` ustawiony na co najmniej `0.5`.

Uwaga

Jeśli przypadek użycia wymaga naprawdę spójnych odpowiedzi dla wielu iteracji z wykorzystaniem tego samego promptu, to dam Ci radę. Otóż podczas swoich eksperymentów najbardziej spójne wyniki uzyskałem, używając modelu GPT-3.5--Turbo-1106 z wartością temperatury równą 0.

Oprócz temperatury istnieje kilka innych parametrów, które można dostosować, przekazując je jako słownik do argumentu `additional_kwargs`. Jeśli w swoim przepływie pracy RAG planujesz używać modeli OpenAI, radzę zapoznać się z tymi ustawieniami modeli LLM, ponieważ mogą być bardzo ważne w scenariuszu z generowaniem wspomaganym wyszukiwaniem. Oprócz `temperature` szczególnie przydatne są parametry `top_p` i `seed`, ponieważ można je wykorzystać do sterowania losowością wyników. Szczegółową listę modeli znajdziesz w oficjalnej dokumentacji OpenAI pod adresem *https://platform.openai.com/docs/models*.

Oto prosty kod, którego możesz użyć do eksperymentowania z różnymi ustawieniami modelu LLM:

```
from llama_index.llms.openai import OpenAI
llm = OpenAI(
    model="gpt-3.5-turbo-1106",
    temperature=0.2,
    max_tokens=50,
    additional_kwargs={
        "seed": 12345678,
        "top_p": 0.5
    }
)
response = llm.complete(
    "Wyjaśnij grawitację w jednym zdaniu."
)
print(response)
```

Za pomocą tego kodu możesz eksperymentować z różnymi ustawieniami i analizować wyniki, co pozwala znaleźć najlepszą konfigurację dla swojego konkretnego przypadku użycia.

Jeśli zastanawiasz się, co różne obecnie dostępne modele LLM mogą zrobić, jeśli chodzi o RAG, to spójrz na porównanie pochodzące z dokumentacji LlamaIndexu. Ta lista została stworzona przez społeczność LlamaIndexu poprzez testowanie różnych modeli LLM: *https://docs.llamaindex.ai/en/stable/module_guides/models/llms.html*.

Jak używać Settings do dostosowywania modeli?

Prawdopodobnie zauważyłeś, że w poprzedniej sekcji użyłem czegoś o nazwie `Settings`, aby dostosować model AI. Pora na krótkie wyjaśnienie.

`Settings` to kluczowy komponent w LlamaIndexie. Umożliwia on dostosowanie i skonfigurowanie *elementów* używanych podczas indeksowania i zapytań. Zawiera wspólne obiekty potrzebne w całym LlamaIndexie, takie jak:

- **LLM** — pozwala zastąpić domyślny model LLM modelem niestandardowym, co widzieliśmy w poprzednim przykładzie
- **Model osadzania** — jest używany do generowania wektorów dla tekstu, aby umożliwić wyszukiwanie semantyczne. Takie wektory nazywane są **osadzeniami** i omówię je bardziej szczegółowo w rozdziale 5. „Indeksowanie z LlamaIndexem”.
- **Obiekt `NodeParser`** — służy do ustawiania domyślnego parsera węzłów.
- **Obiekt `CallbackManager`** — obsługuje wywołania zwrotne dla zdarzeń wewnątrz LlamaIndexu. Jak zobaczymy później, jest używany do debugowania i śledzenia aplikacji.

Istnieją również inne parametry, które można dostosować w komponencie `Settings`. Zagłębimy się znacznie bardziej w różne opcje dostosowywania w rozdziale 9. „Dostosowywanie i wdrażanie projektu stworzonego za pomocą LlamaIndexu”. Bez względu na to, co chcesz zmienić, dostosowanie będzie przeprowadzane tak jak w poprzednim przykładzie. Po zdefiniowaniu własnych ustawień wszystkie kolejne operacje będą korzystać z tej konfiguracji.

Omówiłem wystarczająco dużo koncepcji jak na jeden rozdział. Co powiesz na trochę kodowania?

Rozpoczęcie pracy nad projektem PITS — ćwiczenie praktyczne

Czy jesteś gotowy na trochę praktyki? Czas zacząć budować nasz projekt PITS. Masz już wystarczająco dużo teoretycznych podstaw i w tym rozdziale rozpoczniemy przygotowania do bardziej zaawansowanych zadań, które czekają nas w następnych rozdziałach.

Starałem się zbudować projekt o strukturze modułowej. Wierzę, że to bardzo poprawia przejrzystość kodu i pozwoli Ci poznać po kolei niektóre z ważnych koncepcji LlamaIndexu. Jak wspomniałem w poprzednim rozdziale, możesz albo pisać kod równolegle z czytaniem książki, albo pobrać go i przestudiować w całości, korzystając z repozytorium GitHub (aplikacja w wersji angielskojęzycznej) lub materiałów do pobrania dla tej książki (kod w wersji polskojęzycznej).

Wyjaśnienie

Istnieje wiele aspektów, które można ulepszyć w istniejącej bazie kodu, a także całkiem sporo funkcji brakuje, aby PITS można było uznać za aplikację gotową do wdrożenia na produkcji. Na przykład w mojej implementacji aplikacja nie ma uwierzytelniania i jest przeznaczona dla jednego użytkownika. Ponadto, aby kod był krótki, nie zajmowałem się zbytnio obsługą błędów. Ale oczywiście to nie są błędy, lecz funkcje. Później możesz samodzielnie rozwijać PITS, dodając brakujące elementy i przekształcając ją w aplikację klasy komercyjnej. Dlaczego nie?

Zanim zaczniemy, chciałbym krótko wyjaśnić strukturę kodu, która będzie stanowić fundament naszej aplikacji. Oto lista plików źródłowych Pythona używanych przez aplikację PITS wraz z krótkimi opisami każdego z nich:

- *app.py* — główny punkt wejściowy dla aplikacji Streamlit. Obsługuje inicjalizację aplikacji i zarządza nawigacją między różnymi ekranami w oparciu o logikę aplikacji.
- *document_uploader.py* — interfejs dla LlamaIndexu do pobierania i indeksowania przesłanych dokumentów.
- *training_material_builder.py* — tworzy materiały do nauki (slajdy i tekst) na podstawie aktualnej wiedzy użytkownika. Wykorzystuje przesłane i zindeksowane materiały do generowania treści edukacyjnych.
- *training_interface.py* — to tutaj odbywa się właściwe nauczanie. Wyświetla slajdy i narrację nauczyciela wraz z panelem bocznym do rozmowy, by użytkownik mógł wchodzić w interakcję z aplikacją.
- *quiz_builder.py* — generuje quizy na podstawie przetworzonych materiałów i aktualnej wiedzy użytkownika.
- *quiz_interface.py* — zarządza quizami i ocenia poziom wiedzy użytkownika w zależności od wyników (coś, czego nie lubiliśmy w szkole).

- *conversation_engine.py* — zarządza panelem bocznym rozmowy, odpowiada na pytania użytkowników i udziela wyjaśnień. Śledzi również kontekst rozmów z nauczycielem, aby uniknąć powtórzeń i zapewnić trafną pomoc. Pobiera także podsumowania poprzednich dyskusji i dba o to, aby nauczyciel kontynuował tam, gdzie skończył.
- *storage_manager.py* — obsługuje wszystkie operacje na plikach, takie jak zapisywanie i wczytywanie stanów sesji oraz przesyłanych przez użytkowników plików. Zarządza lokalnym przechowywaniem plików i później może być dostosowany do rozwiązań chmurowych.
- *session_functions.py* — zajmuje się przechowywaniem i odzyskiwaniem informacji o sesji, na początku lokalnie, a docelowo w chmurze.
- *logging_functions.py* — obsługuje logowanie wszystkich interakcji użytkownika z aplikacją. Zapisuje opisowe wpisy logów ze znacznikami czasu, aby śledzić działania użytkownika w całej aplikacji. Przechowuje i odzyskuje logi aplikacji lokalnie i ostatecznie w chmurze.
- *global_settings.py* — zawiera ustawienia aplikacji, konfiguracje, a także dane potrzebne do wdrożenia Streamlita. Zbiera parametry w jednym miejscu dla łatwego zarządzania i aktualizacji.
- *user_onboarding.py* — zajmuje się krokami wprowadzającymi użytkownika.
- *index_builder.py* — buduje indeksy używane w całej aplikacji.

Pamiętaj, że obecnie aplikacja jest zaprojektowana do uruchamiania lokalnie. W rozdziale 9. „Dostosowywanie i wdrażanie projektu stworzonego za pomocą LlamaIndexu" omówię bardziej szczegółowo dostępne opcje wdrażania aplikacji Streamlit. Zanim przejdziesz dalej, upewnij się, że zainstalowałeś drugi pakiet, wspomniany na początku tego rozdziału — YAML dla Pythona.

Aplikacja PITS będzie potrzebować tego pakietu dla modułu *session_functions.py*. Wyjaśnię to za chwilę.

Aby zainstalować pakiet YAML, użyj następującego kodu:

```
pip install pyyaml
```

Na razie skupimy się na tych trzech modułach PITS:

- *global_settings.py*,
- *session_functions.py*,
- *logging_functions.py*.

Kod źródłowy

Zaczniemy od ustawień globalnych w pliku *global_settings.py*:

```
LOG_FILE = "session_data/user_actions.log"
SESSION_FILE = "session_data/user_session_state.yaml"
CACHE_FILE = "cache/pipeline_cache.json"
```

```
CONVERSATION_FILE = "cache/chat_history.json"
QUIZ_FILE = "cache/quiz.csv"
SLIDES_FILE = "cache/slides.json"
STORAGE_PATH = "ingestion_storage/"
INDEX_STORAGE = "index_storage"

QUIZ_SIZE = 5
ITEMS_ON_SLIDE = 4
```

Tutaj będziemy przechowywać globalne konfiguracje. Wykorzystamy różne parametry, aby dostosować działanie PITS i niektóre z jego wewnętrznych ustawień.

Na razie chciałbym zwrócić Twoją uwagę na dwa parametry: `LOG_FILE` i `SESSION_FILE`. Są one używane do zdefiniowania lokalizacji przechowywania pliku rejestru oraz danych związanych z sesją. Plik z logami posłuży do zapisywania wszystkich interakcji użytkownika i zachowywania kontekstu rozmowy. Plik sesji pozwoli wznawiać istniejące sesje z zachowaniem ich stanu.

Teraz przejdźmy do pliku *session_functions.py*.

Moduł *session_functions.py* zawiera funkcje, które obsługują zapisywanie, wczytywanie i usuwanie stanu sesji użytkownika:

```
from global_settings import SESSION_FILE, STORAGE_PATH
import yaml
import os
def save_session(state):
    state_to_save = {key: value for key, value in state.items()}
    with open(SESSION_FILE, 'w') as file:
        yaml.dump(state_to_save, file)
```

Funkcja `save_session` przyjmuje obecny stan jako argument, który zawiera wszystkie niezbędne informacje o sesji użytkownika i zapisuje je do pliku *SESSION_FILE*. Stan jest konwertowany na format YAML przed zapisaniem, dzięki czemu może być łatwo wczytany później:

```
def load_session(state):
    if os.path.exists(SESSION_FILE):
        with open(SESSION_FILE, 'r') as file:
            try:
                loaded_state = yaml.safe_load(file) or {}
                for key, value in loaded_state.items():
                    state[key] = value
                return True
            except yaml.YAMLError as e:
                return False
    return False
```

Ta funkcja próbuje odczytać plik *SESSION_FILE*, a jeśli istnieje, wczytuje zapisane dane sesji do przekazanego obiektu stanu. Jeśli plik zostanie pomyślnie odczytany, a zawartość YAML poprawnie sparsowana, funkcja zwraca `True`, co oznacza, że stan sesji został przywrócony. W przeciwnym razie zwraca `False`.

```
def delete_session(state):
    if os.path.exists(SESSION_FILE):
        os.remove(SESSION_FILE)
    for key in list(state.keys()):
        del state[key]
```

Kiedy sesja musi zostać wyczyszczona, funkcja pokazana powyżej usuwa plik *SESSION_FILE* i wszystkie klucze z przekazanego obiektu stanu, skutecznie resetując sesję.

Dlaczego YAML?

Użyłem YAML jako formatu serializacji zamiast formatu zachowywania stanu używanego w Streamlicie, ponieważ YAML jest czytelny dla człowieka i niezależny platformowo. YAML dobrze współpracuje z hierarchicznymi strukturami danych, co sprawia, że jest łatwy do odczytania i edytowania poza aplikacją, jeśli to konieczne. Pozwala zapisać stan sesji w ustrukturyzowanym, standardowym formacie, który można łatwo przenieść lub zmodyfikować według potrzeb. YAML często jest używany do plików konfiguracyjnych, ale jest również odpowiedni do przechowywania prostych struktur danych, takich jak w naszym przypadku stan sesji.

Musimy również utworzyć plik *logging_functions.py*. Oto kod:

```
from datetime import datetime
from global_settings import LOG_FILE
import os
def log_action(action, action_type):
    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    log_entry = f"{timestamp}: {action_type} : {action}\n"
    with open(LOG_FILE, 'a') as file:
        file.write(log_entry)
def reset_log():
    with open(LOG_FILE, 'w') as file:
        file.truncate(0)
```

Moduł *logging_functions.py* jest odpowiedzialny za rejestrowanie zdarzeń, działań użytkownika i innych istotnych zdarzeń podczas działania aplikacji w pliku logów. Został zaprojektowany w celu śledzenia działań użytkownika i zdarzeń systemowych, głównie po to, aby dostarczyć kontekst agentowi PITS podczas jego interakcji z użytkownikiem, ale także do monitorowania i debugowania.

Oto co robią funkcje w tym module:

- `log_action(action, action_type)` — rejestruje działanie lub zdarzenie. Przyjmuje dwa argumenty: *action*, czyli ciąg opisujący, co się wydarzyło, oraz *action_type*, który kategoryzuje działanie. Funkcja pobiera bieżący znacznik czasu, formatuje go z działaniem i typem, a następnie dodaje wpis do *LOG_FILE*. Dzięki temu mamy chronologiczny zapis działań i zdarzeń.
- `reset_log()` — w bieżącej implementacji, gdy użytkownicy wracają do istniejącej sesji, mają możliwość rozpoczęcia nowej. W takim przypadku

czyścimy plik logów, aby zapobiec gromadzeniu zbyt dużej ilości danych. Ta funkcja otwiera *LOG_FILE* i czyści jego zawartość (a dokładniej zmniejsza jego rozmiar do 0 bajtów), skutecznie usuwając wszystkie zanotowane wpisy. Zwykle nie jest to powszechnie stosowane w środowiskach produkcyjnych, ponieważ logi są cenne dla analizy danych historycznych, ale w naszym przypadku upraszcza to przepływ.

Wiem, że obiecałem, iż będziemy się dobrze bawić, pisząc kod dla PITS, i doskonale zdaję sobie sprawę, że rejestrowanie zdarzeń wydaje się mniej *zabawne*, a bardziej *monotonne*, ale uwierz mi, koszmarem jest brak możliwości debugowania aplikacji. Stworzyliśmy tutaj fundamenty dla naszej aplikacji, a resztą modułów zajmiemy się w następnych rozdziałach.

Podsumowanie

W tym rozdziale omówiłem podstawowe pojęcia, takie jak dokumenty, węzły i indeksy, czyli podstawowe elementy konstrukcyjne LlamaIndexu. Przedstawiłem prosty przepływ pracy do ładowania danych jako obiektów dokumentów, przetwarzania ich w spójne węzły za pomocą parserów, budowania zoptymalizowanego indeksu z węzłów, a następnie wysyłania zapytań do indeksu w celu uzyskania odpowiednich węzłów i zsyntezowanej odpowiedzi.

Funkcje rejestrowania LlamaIndexu zostały przedstawione jako ważne narzędzie umożliwiające zrozumienie logiki działania i debugowanie aplikacji. Logi ujawniają, jak LlamaIndex parsuje, indeksuje, wspomaga modele LLM, wyszukuje węzły i syntezuje odpowiedzi. Pokazałem również, jak za pomocą klasy `Settings` można dostosowywać modele LLM i inne usługi używane przez LlamaIndex.

Zaczęliśmy budować aplikację edukacyjną PITS. Napisaliśmy podstawowe funkcje zarządzania sesjami i rejestrowania. Ta modularna struktura umożliwi stopniowe odkrywanie możliwości LlamaIndexu w miarę rozwoju aplikacji.

Gdy już znasz te podstawy, możesz przejść do bardziej zaawansowanych funkcji LlamaIndexu. Przygoda trwa!

Pokochaj LlamaIndex — i twórz inteligentne aplikacje!

Chociaż sztuczna inteligencja (AI), która generuje treści, wciąż się rozwija, to nadal boryka się z pewnymi ograniczeniami. Mogą to być trudności w odróżnianiu prawdy od fałszu, problem z utrzymaniem kontekstu w długich dokumentach czy występowanie nieprzewidywalnych błędów w rozumowaniu i zapamiętywaniu faktów. Generowanie wspomagane wyszukiwaniem (RAG) ułatwia rozwiązanie wielu z tych problemów, a narzędziem, które do tego służy, jest framework LlamaIndex.

Dzięki tej książce poradzisz sobie z zastosowaniem ekosystemu LlamaIndex i nauczysz się wdrażać własne projekty. Na praktycznych przykładach zapoznasz się z procesem personalizacji i uruchamiania projektów LlamaIndex. Dowiesz się, jak przezwyciężać ograniczenia dużych modeli językowych, zbudujesz aplikacje dla użytkowników końcowych i zdobędziesz umiejętności w zakresie pozyskiwania danych, indeksowania, obsługi zapytań i łączenia dynamicznych baz wiedzy, obejmujących generatywną sztuczną inteligencję i duże modele językowe. Pod koniec lektury zagłębisz się w tworzenie niestandardowych rozwiązań, co pozwoli Ci dobrze zrozumieć możliwości i zastosowania LlamaIndex.

Ciekawsze zagadnienia:

- ekosystem LlamaIndex i typowe przypadki użycia
- wprowadzanie i analizowanie w LlamaIndex danych z różnych źródeł
- tworzenie zoptymalizowanych indeksów
- wysyłanie zapytań do LlamaIndex i interpretacja odpowiedzi
- koszty i kwestie prywatności
- wdrażanie aplikacji LlamaIndex

Andrei Gheorghiu jest doświadczonym inżynierem, wykładowcą i konsultantem z ponad 20-letnim stażem w branży. Specjalizuje się w zarządzaniu usługami IT, cyberbezpieczeństwie, audycie i projektach technologicznych. Posiada liczne certyfikaty, w tym ITIL Master, CISA, CISSP i Lead Auditor ISO 27001. Przeszkolił tysiące specjalistów, którym pomagał rozwijać kompetencje w obszarze IT.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2305-8
9 788328 923058
89,00 zł
packt