Czatowanie z danymi

Wykorzystanie dużych modeli językowych (LLM) do implementacji czatu z danymi to obiecująca strategia, szczególnie przydatna dla przedsiębiorstw, które chcą wykorzystać potencjał generatywnej sztucznej inteligencji (AI) do swoich specyficznych potrzeb biznesowych. Synergizując możliwości LLM ze źródłami danych i narzędziami specyficznymi dla danego przedsiębiorstwa, firmy mogą tworzyć inteligentne i kontekstowe chatboty, które dostarczają cennych spostrzeżeń i rekomendacji swoim klientom i interesariuszom. Ogólnie rzecz biorąc, istnieją dwa sposoby komunikacji z danymi za pomocą platformy LLM — jeden polega na wykorzystaniu silnika wyszukiwania zaimplementowanego w oparciu o wzorzec generacji rozszerzonej wyszukiwania (RAG), a drugi na indywidualnym szkoleniu platformy LLM na danych. Drugi sposób jest bardziej złożony i niedostępny dla większości użytkowników. Niniejszy rozdział opiera się na wzorcu RAG z poprzedniego rozdziału, który służy do wzbogacania platform LLM o dane, zwłaszcza gdy przedsiębiorstwa chcą wdrożyć go na dużą skalę dla obciążeń produkcyjnych. Integrując swoje dane z platformami LLM za pomocą wzorca RAG, przedsiębiorstwa zyskują wiele korzyści, zwiększając funkcjonalność i przydatność tych systemów sztucznej inteligencji w swoich unikalnych kontekstach biznesowych. W rozdziale opisano, czym się one różnią i w wielu przypadkach są lepsze niż większe okna kontekstowe. Zacznijmy od zidentyfikowania korzyści, jakie mogą uzyskać przedsiębiorstwa, chcąc wprowadzić swoje dane.

Korzyści dla przedsiębiorstw korzystających z danych

W dynamicznym świecie technologii biznesowych integracja platform LLM z systemami danych przedsiębiorstw zwiastuje transformacyjną erę interaktywnych i intuicyjnych procesów. Jak już wcześniej omawialiśmy, te najnowocześniejsze narzędzia oparte na sztucznej inteligencji (AI) zmieniają sposób, w jaki firmy korzystają ze swoich danych, otwierając tym samym bezprecedensowe możliwości w zakresie wydajności i dostępności. LLM-y osiągnęły imponujące wyniki w różnych zadaniach przetwarzania języka naturalnego (NLP), takich jak odpowiadanie na pytania, streszczanie, tłumaczenie i dialog. Jednak LLM-y wiążą się z ograniczeniami i wyzwaniami, takimi jak jakość danych, problemy etyczne i skalowalność. Dlatego wiele przedsiębiorstw jest zainteresowanych wdrożeniem chatbotów z danymi za pomocą LLM-ów, co oferuje szereg korzyści dla ich celów biznesowych. Jedną z głównych zalet wykorzystania LLM-ów w tym celu jest to, że mogą one dostarczać inteligentne i kontekstowe chatboty, które mogą obsługiwać zapytania i wątpliwości klientów z biegłością zbliżoną do ludzkiej. LLM-y potrafią zrozumieć znaczenie i intencję użytkownika, generować trafne i spójne odpowiedzi, a nawet podejmować działania, wywołując API w razie potrzeby. To poprawia zadowolenie klienta i pozwala ludziom skupić się na bardziej złożonych zadaniach. Kolejną zaletą wykorzystania LLM do implementacji czatu z danymi jest możliwość ich dostosowania do danych specyficznych dla przedsiębiorstwa, co prowadzi do dokładniejszych i trafniejszych wniosków oraz rekomendacji generowanych przez sztuczną inteligencję. Wykorzystanie LLM do implementacji czatu z danymi może umożliwić bardziej efektywną i wydajną analizę danych. LLM może generować podsumowania lub wyjaśnienia wyników analizy danych w języku naturalnym, co pomaga użytkownikom zrozumieć kluczowe wnioski i implikacje. Ponadto LLM może generować interaktywne wykresy lub grafy, podkreślające wzorce lub trendy w danych. Funkcje te mogą usprawnić doświadczenie użytkownika i ułatwić podejmowanie decyzji w oparciu o dane w całej organizacji.

A co z dużymi oknami kontekstowymi?

Najnowsze modele OpenAI — na przykład GPT-4 Turbo z oknem kontekstowym o pojemności 128 KB i najnowszy Gemini Pro 1.5 firmy Google z 1,5 milionami okien kontekstowych — wzbudziły duży entuzjazm i zainteresowanie. Jednak samo większe okno kontekstowe nie wystarczy. Szkolenie LLM na

podstawie danych ma następujące zalety w porównaniu z samym korzystaniem z LLM z większym oknem kontekstowym:

* Bardziej precyzyjne i pouczające odpowiedzi — podczas rozmowy z danymi LLM może uzyskać dostęp do znacznie większej ilości informacji niż w przypadku samego większego okna kontekstowego. Dzięki temu LLM może udzielać bardziej precyzyjnych i pouczających odpowiedzi na pytania.

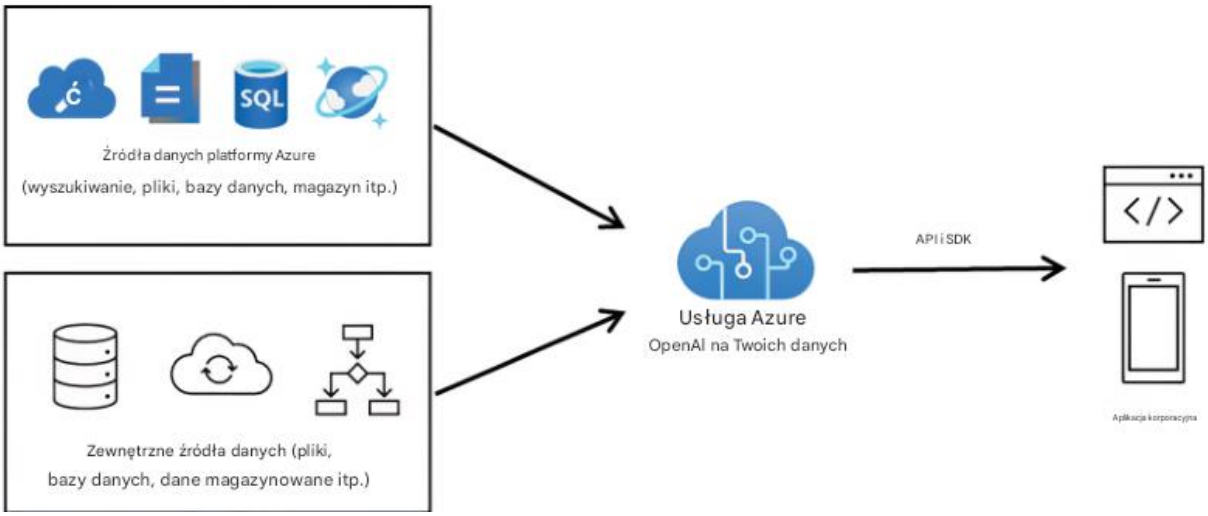
* Bardziej spersonalizowane odpowiedzi — LLM może również nauczyć się personalizować swoje odpowiedzi na podstawie danych. Na przykład, jeśli rozmawiasz z LLM, którego działania zostały dostrojone na podstawie danych klientów, może on nauczyć się udzielać bardziej trafnych odpowiedzi dostosowanych do konkretnych klientów i ich potrzeb. Na przykład możemy użyć wyszukiwarki do indeksowania danych klientów, a następnie połączyć ją z programem LLM. Pozwoliłoby to firmie komunikować się z klientami w bardziej spersonalizowany i bogaty w informacje sposób.

* Bardziej kreatywne odpowiedzi — program LLM może również wykorzystywać Twoje dane do generowania bardziej kreatywnych i interesujących odpowiedzi na Twoje pytania. Na przykład, jeśli rozmawiasz z programem LLM, który dopracował Twoje dane produktowe, program LLM może nauczyć się generować nowe pomysły na produkty lub kampanie marketingowe.

Oczywiście programy LLM z większym oknem kontekstowym mają swoje zalety, ale mogą być bronią obosieczną z pewnymi ograniczeniami. Większe okna kontekstowe pozwalają nam przekazać więcej informacji w jednym wywołaniu API i mniej martwić się o dzielenie aplikacji na mniejsze części. Na przykład niedawno zapowiedziany program GPT-4.5 Turbo ma okno kontekstowe o rozmiarze 128 KB, co pozwala na wyświetlenie około 300 stron tekstu w jednym komunikacie, w porównaniu z około 75 stronami we wcześniejszym modelu GPT-4 o rozmiarze 32 KB. Z drugiej strony, posiadanie większego okna kontekstowego wiąże się z pewnymi wyzwaniami. Na przykład, trenowanie i wdrażanie modeli LLM z większym oknem kontekstowym może być bardziej kosztowne obliczeniowo. Mogą one również być bardziej podatne na generowanie halucynacji lub niepoprawnych odpowiedzi, ponieważ duże okna kontekstowe zwiększają złożoność i niepewność wyników modelu. Modele LLM są trenowane na dużych, zróżnicowanych zbiorach danych, które mogą zawierać niekompletne, sprzeczne lub zaszumione informacje. Gdy model otrzymuje długie okno kontekstowe, musi przetworzyć więcej informacji i zdecydować, co wygenerować dalej, co może prowadzić do błędów, niespójności lub fałszerstw w wynikach, zwłaszcza jeśli model opiera się na heurystyce lub zapamiętywaniu, a nie na rozumowaniu lub rozumieniu. Z kolei interakcja z danymi może być bardziej wydajna i mniej podatna na błędy, głównie dlatego, że podczas interakcji z danymi bazujemy na tych danych i sterujemy modelem. Model LLM może uzyskać dostęp do szerszego zakresu informacji i nauczyć się personalizować odpowiedzi w oparciu o dane. Ostatecznie, najlepszy sposób wyboru między modelem LLM z większym oknem kontekstowym a interakcją z danymi będzie zależał od konkretnych potrzeb i zasobów.

Budowanie aplikacji czatu z wykorzystaniem naszych danych

Rozwijając implementację RAG z poprzedniego rozdziału, zbudujemy aplikację czatu, której będziemy mogli używać do czatowania z naszymi danymi. Jak widzieliśmy wcześniej, wektorowe bazy danych są kluczowe dla przedsiębiorstw, umożliwiając im zarządzanie, zabezpieczanie i skalowanie osadzenia w środowisku produkcyjnym. W wielu przedsiębiorstwach wektorowe bazy danych do zastosowań w wyszukiwaniu semantycznym rozwiązują problemy związane z wydajnością i bezpieczeństwem wymagane w systemach produkcyjnych. Rysunek przedstawia ogólne podejście do implementacji LLM w naszych danych.



Na przykład, wykorzystamy blog (https://blog.desigeek.com) jako zastrzeżone źródło danych. Zawiera on posty z 20 lat wstecz, dotyczące różnych tematów i technologii. Jeśli na każde pytanie zadane przez użytkownika wrócimy do bloga, wczytamy wszystkie posty, utworzymy osadzenia, przeszukamy je, a następnie użyjemy RAG, aby odpowiedzieć na pytanie, proces będzie bardzo czasochłonny i nieskalowalny. Dodatkowo, pojawią się dodatkowe koszty, ponieważ będziemy używać znacznie więcej tokenów w każdej turze konwersacji lub dla nowego zestawu konwersacji. Lepszym podejściem byłoby skonfigurowanie następujących czterech etapów, które przejdziemy:

* Odczyt i wstrzykiwanie informacji (tj. pobieranie)

* Tworzenie osadzeń i zapisywanie szczegółów w Redis

* Przeszukiwanie zapisanych szczegółów w celu implementacji pytań i odpowiedzi z wykorzystaniem postów na blogu (tj. rozszerzanie)

* Podłączenie tego do generowania LLM

Zacznijmy od skonfigurowania wektorowej bazy danych.

Korzystanie z wektorowej bazy danych

Jak widzieliśmy wcześniej, wektorowa baza danych została zaprojektowana do działania na wektorach osadzonych. W większości zastosowań korporacyjnych stanowią one doskonałe uzupełnienie implementacji RAG i pozwalają nam korzystać z naszych danych. Obecnie dostępnych jest wiele wektorowych baz danych, a wraz z rosnącą popularnością LLM i generatywnej sztucznej inteligencji, z każdym dniem rośnie wsparcie dla wyszukiwania semantycznego. Zobaczmy, jak możemy to wdrożyć. W naszym kontekście edukacyjnym potrzebujemy czegoś szybkiego i łatwego w konfiguracji i uruchomieniu, głównie po to, aby zrozumieć różne koncepcje i kroki wymagane do wdrożenia wektorowej bazy danych do osadzania oraz jak zintegrować ją z naszą implementacją RAG. W tym celu użyjemy Redis jako wektorowej bazy danych i uruchomimy ją lokalnie w kontenerze Dockera. Redis to otwarty, działający w pamięci, magazyn danych typu klucz-wartość, który może być używany jako baza danych, pamięć podręczna, broker wiadomości i wiele innych. Obsługuje struktury danych, takie jak ciągi znaków, listy, zbiory, hasze i strumienie. Redis jest szybki, skalowalny i niezawodny, co czyni go popularnym w wielu zastosowaniach wymagających niskich opóźnień i wysokiej przepustowości. Redis rozszerza swoje podstawowe możliwości, wykorzystując koncepcję modułów. Redis Search to moduł rozszerzający Redis o zaawansowane funkcje wyszukiwania tekstowego i indeksowania wtórnego. Umożliwia tworzenie indeksów dla danych Redis i przeszukiwanie ich za pomocą rozbudowanego

języka zapytań. Redis Search można również używać do wyszukiwania podobieństwa wektorów, co umożliwia wyszukiwanie semantyczne oparte na osadzeniu. Istnieje kilka sposobów wdrożenia Redis. W przypadku lokalnego tworzenia oprogramowania najszybszą metodą jest użycie kontenera Redis Stack Docker, którego będziemy używać. Redis Stack zawiera kilka modułów Redis, które w naszym przypadku mogą być używane razem, aby stworzyć szybki, wielomodelowy magazyn danych i silnik zapytań.

UWAGA: Głównym warunkiem wstępnym jest zainstalowanie i skonfigurowanie Dockera. Szczegóły instalacji Dockera wykraczają poza zakres książki, ponieważ istnieją książki poświęcone Dockerowi i jego zarządzaniu. Jeśli nie masz zainstalowanego Dockera, zapoznaj się z dokumentacją instalacji Docker Desktop, aby ułatwić sobie zarządzanie, lub przynajmniej z dokumentacją silnika Dockera.

Oprócz pakietów OpenAI, do uruchomienia Redisa wymagane są następujące wymagania wstępne:

* Docker musi być zainstalowany i uruchomiony lokalnie.

* W przypadku korzystania z Condy, pakiet redis-py można zainstalować poleceniem conda install -c conda-forge redis-py. Jeśli korzystamy z PIP, należy użyć polecenia pip install redis.

Użyjemy pliku docker-compose dla Dockera, jak pokazano na listingu 1. Nie zmieniliśmy domyślnych portów, ale możesz je skonfigurować według własnego uznania. W tym przykładzie pobieramy najnowszy obraz Redis-Stack z rejestru Dockera i udostępniamy dwa porty – 6379 i 8001. Konfigurujemy również wolumin danych, aby zachować informacje wprowadzone do bazy danych. Na koniec przeprowadzamy wstępne kontrole stanu usługi, aby sprawdzić podstawowe kwestie, takie jak to, czy usługa jest uruchomiona i dostępna na skonfigurowanych portach. Jeśli zmienisz porty, upewnij się, że zostaną one zaktualizowane w teście w ramach kontroli stanu usługi.

Listing 1: plik docker-compose dla redis-stack

```
version: '3.7'

services:

vector-db:

image: redis/redis-stack:latest

ports:

- 6379:6379

- 8001:8001

environment:

- REDISEARCH_ARGS=CONCURRENT_WRITE_MODE

volumes:

- vector-db:/var/lib/redis

- ./redis.conf:/usr/local/etc/redis/redis.conf

healthcheck:

test: ["CMD", "redis-cli", "-h", "localhost", "-p", "6379", "ping"]
```

interval: 2s

timeout: 1m30s

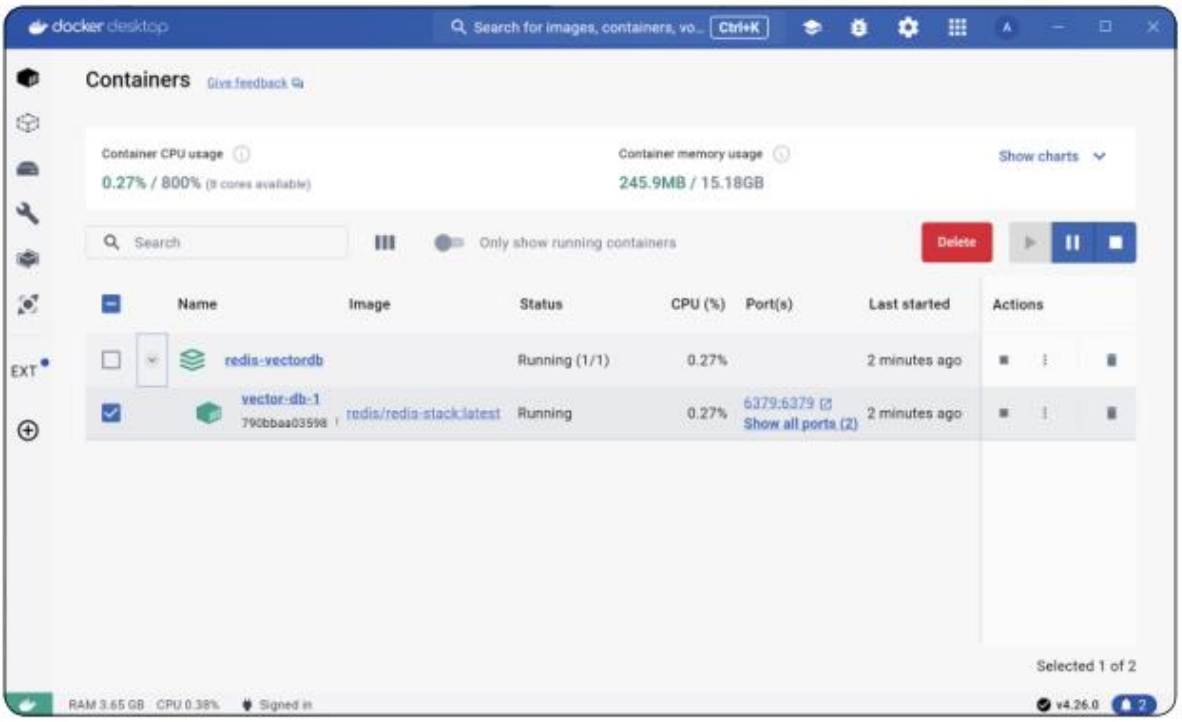
retries: 5

start_period: 5s

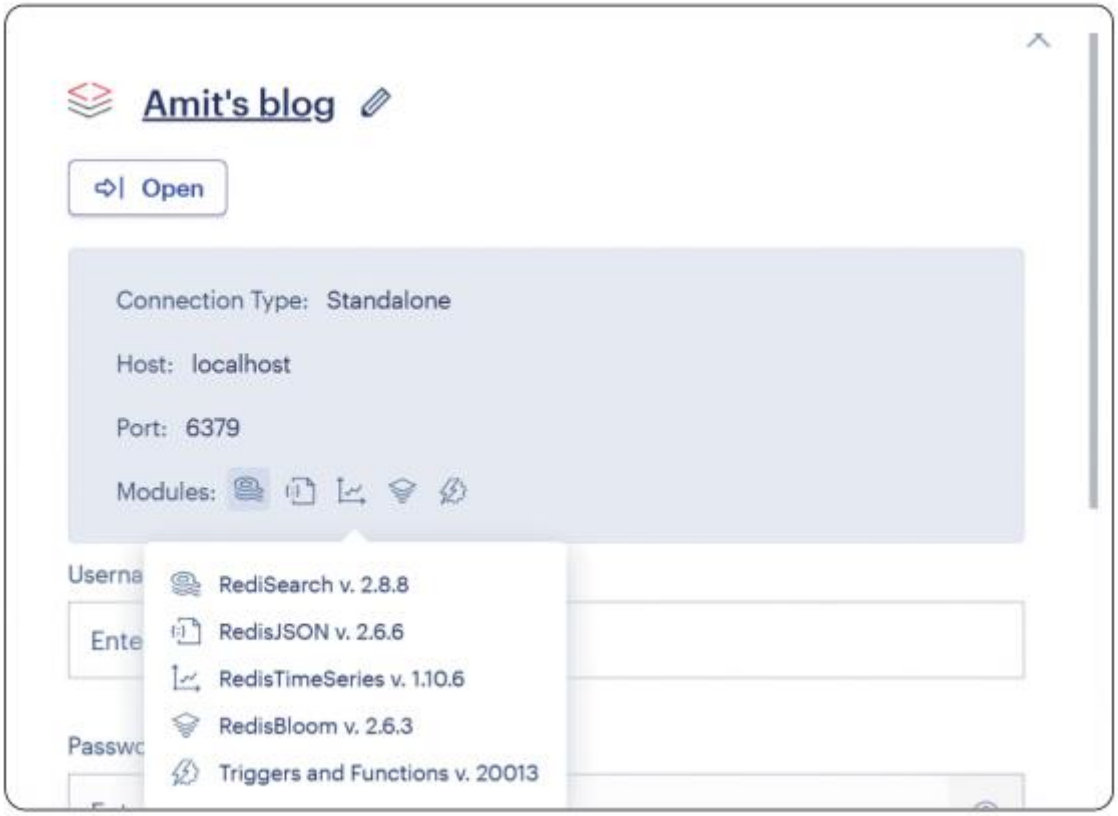
volumes:

vector-db:

Aby Docker działał, zgodnie z konwencją, musimy upewnić się, że plik ten jest zapisany jako docker-compose.yml. Można to uruchomić, wpisując następujące polecenia z tej samej lokalizacji, w której zapisano plik: docker-compose up -d. W naszym przykładzie kontener uruchamia się za pośrednictwem graficznego interfejsu użytkownika Docker Desktop, jak pokazano na rysunku 2.



Obejmuje to również Redis Insight, graficzny interfejs użytkownika do zarządzania naszą bazą danych Redis. Po uruchomieniu kontenera Dockera możemy uzyskać do niego dostęp lokalny pod adresem http://localhost:8001. Jeśli wszystko jest poprawnie skonfigurowane, możemy zobaczyć bazę danych i zainstalowane moduły (rysunek 3).



Skoro nasza baza danych wektorowych jest już uruchomiona, przejdźmy do kolejnego kroku, czyli pobierania informacji.

UWAGA: W tym rozdziale używamy Redis jako przykładu, ponieważ stosunkowo łatwo jest uruchomić go lokalnie w kontenerze, aby przedsiębiorstwa mogły wypróbować koncepcje i zapoznać się z powiązaną z tym złożonością. Działając lokalnie w kontenerze, pomaga to również uprościć początkowe problemy związane z przesyłaniem danych do chmury, co może stanowić problem, przynajmniej na wczesnym etapie rozwoju. Oprócz Redis, kilka innych wektorowych baz danych zyskuje na popularności. Do najpopularniejszych wektorowych baz danych należą Azure AI Search, Pinecone i Milvus.

Azure AI Search

Chociaż używamy Redis działającego lokalnie, przedsiębiorstwa, które muszą skalować się do większego korpusu danych, indeksów oraz obciążeń i dostępności na skalę produktu, stają się znacznie bardziej złożone. W takich scenariuszach Azure AI Search jest dobrym wyborem. Skoro nasza baza danych wektorowych jest już uruchomiona, przejdźmy do kolejnego kroku, czyli pobierania informacji.

UWAGA: Używamy Redis jako przykładu, ponieważ stosunkowo łatwo jest uruchomić go lokalnie w kontenerze, aby przedsiębiorstwa mogły wypróbować koncepcje i zapoznać się z powiązaną z tym złożonością. Działając lokalnie w kontenerze, pomaga to również uprościć początkowe problemy związane z przesyłaniem danych do chmury, co może stanowić problem, przynajmniej na wczesnym etapie rozwoju. Oprócz Redis, kilka innych wektorowych baz danych zyskuje na popularności. Do najpopularniejszych wektorowych baz danych należą Azure AI Search, Pinecone i Milvus.

Azure AI Search

Chociaż używamy Redis działającego lokalnie, przedsiębiorstwa, które muszą skalować się do większego korpusu danych, indeksów oraz obciążeń i dostępności na skalę produktu, stają się znacznie bardziej złożone. W takich scenariuszach Azure AI Search jest dobrym wyborem. Azure AI Search to usługa oparta na chmurze, która oferuje różne funkcje do tworzenia aplikacji wyszukiwawczych. Oprócz wyszukiwania wektorowego, które umożliwia znajdowanie i pobieranie obiektów danych semantycznie podobnych do danego zapytania na podstawie ich osadzeń wektorowych, usługa obsługuje również wyszukiwanie hybrydowe. Wyszukiwanie hybrydowe łączy zapytania pełnotekstowe i wektorowe, które są wykonywane na indeksie wyszukiwania zawierającym przeszukiwalną treść w postaci zwykłego tekstu i wygenerowane osadzenia. W pojedynczym żądaniu wyszukiwania zapytania hybrydowe mogą wykorzystywać istniejące funkcje, takie jak filtrowanie, fasetowanie, sortowanie, profile punktacji i ranking semantyczny. Odpowiedź na zapytanie dostarcza tylko jeden zestaw wyników, wykorzystujący wzajemną fuzję rang (RRF) do określania, które dopasowania zostaną uwzględnione. Azure AI Search oferuje szereg korzyści w porównaniu z Redis w przypadku wyszukiwania wektorowego z wykorzystaniem modeli LLM. Jest to w pełni zarządzana usługa wyszukiwania, która może indeksować i przeszukiwać dane ustrukturyzowane, częściowo ustrukturyzowane i nieustrukturyzowane. Azure AI Search jest wysoce skalowalny i z łatwością obsługuje duże ilości danych. Obsługuje bardziej zaawansowane funkcje zabezpieczeń wymagane przez przedsiębiorstwa, takie jak szyfrowanie REST i Transit, kontrola dostępu oparta na rolach (RBAC) i inne.

Planowanie pobierania informacji

Najpierw musimy zrozumieć, co próbujemy pobrać i zindeksować. To pomoże nam sformułować podejście i określić, które informacje są istotne, a które zbędne i można je pominąć. W ramach tego ćwiczenia musimy również uwzględnić aspekty techniczne, takie jak sposób połączenia z systemem źródłowym oraz wszelkie ograniczenia techniczne lub praktyczne. Musimy również zrozumieć format danych i wymagania inżynieryjne (w tym czyszczenie i konwersję danych). Zanim pobierzemy dane z bloga, zapoznaj się ze szczegółami przedstawionymi w tabeli .

Dane: Opis

URL: Adres URL do pojedynczego wpisu na blogu

Tytuł: Tytuł wpisu na blogu

Opis: Kilka zdań opisujących, czego dotyczy dany wpis na blogu

Data publikacji: Data publikacji wpisu

Treść: Faktyczna treść wpisu na blogu

Chociaż używamy wpisu na blogu jako systemu źródłowego, jest to holistyczny przykład reprezentujący większość aspektów RAG i pomagający nam zrozumieć najlepsze praktyki oraz sposób ich wykorzystania. Pobieramy informacje ze zdalnego systemu, aby odczytać wpisy na blogu. Jest to zasadniczo podobne do odczytywania informacji przez przedsiębiorstwa z różnych systemów biznesowych. W zależności od systemu źródłowego, odczytują je za pośrednictwem interfejsów API, wyeksportowanych plików lub łącząc się z różnymi bazami danych i źródłami danych. W naszym przykładzie odczytamy wszystkie wpisy za pomocą kanału RSS bloga. RSS to skrót od Really Simple Syndication, standardowej metody dystrybucji treści na stronie internetowej, często używanej do publikowania zmian. Blog można znaleźć pod adresem https://blog.desigeek.com/, a odpowiadający mu kanał RSS jest dostępny pod adresem https://blog.desigeek.com/index.xml. Najpierw zakładamy, że Redis działa lokalnie w kontenerze, jak pokazano wcześniej. Połączymy się z Redis i utworzymy nowy

indeks o nazwie „posts”. Schemat indeksu jest pokazany na kolejnym listingu i przedstawia strukturę naszych danych, którą widzieliśmy wcześniej. Oprócz głównej treści wpisu na blogu, przechwytujemy również powiązane metadane, które pomogą nam odpowiedzieć na pytania lub lepiej zrozumieć kontekst.

Listing 2: Schemat indeksu Redis

```
SCHEMA = [
TagField("url"),
TextField("title"),
TextField("description"),
TextField("publish_date"),
TextField("content"),
VectorField("embedding", "HNSW",
{"TYPE": "FLOAT32",
"DIM": 1536,
"DISTANCE_METRIC": "COSINE"}),
]
```

Ten schemat zawiera następujące typy pól:

* TagField — służy do przechowywania tagów, czyli krótkich, opisowych słów kluczowych, które można wykorzystać do kategoryzowania i organizowania danych. Tagi są zazwyczaj przechowywane jako lista ciągów znaków, a wyszukiwarka Redis obsługuje wyszukiwanie tagów za pomocą operatorów boolowskich, takich jak AND, OR i NOT.

* TextField — służy do przechowywania danych tekstowych, takich jak tytuł, opis i treść wpisu na blogu. Wyszukiwanie Redis obsługuje wyszukiwanie pełnotekstowe w polach TextField, co oznacza, że można wyszukiwać słowa i frazy w tekście.

* VectorField — przechowuje matematyczne reprezentacje danych wektorów, które można wykorzystać do wykonywania zadań uczenia maszynowego, takich jak klasyfikacja obrazów i przetwarzanie języka naturalnego. Wyszukiwanie Redis obsługuje wyszukiwanie podobieństwa wektorów, co oznacza, że można wyszukiwać wektory podobne do danego wektora.

Większość nazw pól nie wymaga wyjaśnień, z wyjątkiem pola o nazwie „embedding” typu VectorField, które służy do przechowywania wektorów wielowymiarowych. Redis obsługuje dwa algorytmy wyszukiwania podobieństw: FLAT i HNSW; w naszym przykładzie używamy HNSW. HSNW to skrót od hierarchicznego, nawigowalnego małego świata (ang. hierarchical navigable small world). Jest to algorytm używany do wyszukiwania najbliższego sąsiada w przestrzeniach wielowymiarowych i jest tutaj używany jako typ osadzania. Algorytm HNSW jest szczególnie przydatny w zadaniach takich jak wyszukiwanie podobieństwa lub klasteryzacja w przestrzeniach wielowymiarowych. Jest znany ze swojej wydajności i dokładności przy niższym narzucie obliczeniowym. HNSW organizuje wektory w strukturę grafu. FLAT to skrót od szybkiej liniowej transformacji przybliżonej (ang. fast linear approximation transformation). Jest to algorytm siłowy i proste podejście, w którym wszystkie wektory są indeksowane w pojedynczej strukturze drzewa lub listy. Znalezienie najbliższych sąsiadów punktu

zapytania jest zazwyczaj wyszukiwaniem siłowym realizowanym poprzez obliczenie odległości od punktu zapytania i innych indeksów. Dzięki temu jest znacznie dokładniejsze, ale wymaga dużych nakładów obliczeniowych i jest wolniejsze. Osadzenia są liczbami zmiennoprzecinkowymi, co oznacza FLOAT32. Ustawiamy wymiary tak, aby odpowiadały 1536 wymiarom modeli Azure OpenAI, które muszą być zgodne z architekturą LLM. Na koniec używamy metryki odległości COSINES do pomiaru podobieństwa. Redis obsługuje trzy typy metryk odległości (patrz tabela 2).

Metryka odległości HNSW: Opis

EUKLIDEJSKA: Odległość w linii prostej między dwoma punktami w przestrzeni euklidesowej. Jest to dobry wybór, gdy wszystkie wymiary są podobne (np. wszystkie odległości mierzone są w metrach).

ILOCZYN SKALARNY: Oblicza iloczyn skalarny między dwoma wektorami. Iloczyn skalarny jest sumą iloczynów odpowiednich elementów dwóch numerów sekwencji.

COSINUS: Oblicza cosinus kąta między dwoma wektorami. Niezależnie od ich modułu, mierzy podobieństwo wektorów. Jest to często używane w analizie tekstu, gdzie kierunek wektora (kąt) jest ważniejszy niż jego długość.

TagField kontra TextField

Pole URL to TagField, a nie TextField. Choć na pierwszy rzut oka może się to wydawać dziwne, istnieje ku temu dobry powód. W przypadku TagField cały adres URL jest traktowany jako pojedynczy tag. Ta właściwość jest przydatna, jeśli chcesz wyszukiwać dokumenty przy użyciu dokładnego adresu URL. Jednak wyszukiwanie dokumentów zawierających określone słowa w adresie URL byłoby bezużyteczne, ponieważ adres URL nie jest tokenizowany. Natomiast gdyby pole URL zostało zdefiniowane jako TextField, byłoby tokenizowane, a każda część adresu URL byłaby indeksowana osobno. Byłoby to przydatne, gdybyś wyszukiwał dokumenty zawierające określone słowa w adresie URL. Nie byłoby to jednak przydatne, gdybyś chciał wyszukiwać dokumenty według dokładnego adresu URL, ponieważ adres URL byłby tokenizowany. W tym przypadku, gdybyśmy uruchomili wyszukiwanie wymagające tokenizacji (tj. wyszukiwanie dokumentów zawierających określone słowo w adresie URL), wyszukiwanie nie zwróciłoby oczekiwanych wyników. Podobnie, jeśli zdefiniujesz adres URL jako pole tekstowe, a następnie spróbujesz wykonać wyszukiwanie wymagające dokładnego dopasowania (tj. wyszukasz dokumenty po dokładnym adresie URL), wyszukiwanie nie zwróci oczekiwanych wyników. Teraz, gdy rozumiemy potrzebne dane i powiązany schemat, utwórzmy indeks, aby rozpocząć pracę w Redis. Zaczynamy od połączenia z bazą danych Redis, która w naszym przypadku działa lokalnie na Dockerze i jest dostępna przez port 6379, jak pokazano na listingu 3. Potrzebujemy następujących zmiennych środowiskowych wskazujących odpowiednio na hosta serwera, port i hasło do ustawienia:

W systemie Windows użyj

setx REDIS_HOST "your-host-details"

setx REDIS_PORT "Port-number-exposed"

setx REDIS_PASSWORD "Password-required-to-connect"

UWAGA: Aby odczytać nowe zmienne, należy ponownie uruchomić terminal.

W systemie Linux/Mac użyj

export REDIS_HOST="your-host-details"

export REDIS_PORT="Port-number-exposed"

```
export REDIS_ PASSWORD="Password-required-to-connect"Bash:

echo export REDIS_HOST="your-host-details" >> /etc/environment && source /

etc/environment

echo export REDIS_PORT="Port-number-exposed" >> /etc/environment && source /

etc/environment

echo export REDIS_ PASSWORD="Password-required-to-connect" >> /etc/

environment && source /etc/environment
```

Najpierw musimy nawiązać połączenie z serwerem Redis, co jest dość proste

```
# Connect to the Redis server

conn = redis.Redis(host=redis_host,

port=redis_port,

password=redis_password,

encoding='utf-8',

decode_responses=True)
```

Ponieważ mamy już zdefiniowany schemat, jak pokazano na listingu 2, utworzenie indeksu wektora jest proste. Wywołujemy funkcję create_index i przekazujemy jej nazwę, schemat oraz opcjonalny prefiks. Obsługiwane są tylko dwa indeksy – HASH (domyślny) lub JSON – dla których potrzebujemy osobnego modułu. W naszym przypadku użyjemy domyślnego HASH:

```
conn.ft(index_name).create_index(

fields=schema,

definition=IndexDefinition(prefix=["post:"],

index_type=IndexType.HASH))
```

Oczywiście możemy usunąć indeks i wyświetlić jego szczegóły. Pełny kod tej funkcji pomocniczej znajduje się na poniższym listingu.

Listing 3: Operacje indeksowania wyszukiwania Redis

```
import redis
from redis.commands.search.field import VectorField, TextField
from redis.commands.search.query import Query
from redis.commands.search.indexDefinition import
     ➥IndexDefinition, IndexType
from redis.commands.search.field import TagField

redis_host = os.getenv('REDIS_HOST')
redis_port = os.getenv('REDIS_PORT')
redis_password = os.getenv('REDIS_PASSWORD')

conn = redis.Redis(host=redis_host,
                   port=redis_port,
                   password=redis_password,
                   encoding='utf-8',
                   decode_responses=True)

SCHEMA = [
    TagField("url"),
    TextField("title"),
    TextField("description"),
    TextField("publish_date"),
    TextField("content"),
    VectorField("embedding", "HNSW",
                {"TYPE": "FLOAT32",
                 "DIM": 1536,
                 "DISTANCE_METRIC": "COSINE"}),
]

def create_index(conn, schema, index_name="posts"):
    try:
        conn.ft(index_name).create_index(
            fields=schema,
            definition=IndexDefinition(prefix=["post:"],
                                       index_type=IndexType.HASH))
    except Exception as e:
        print("Index already exists")

def delete_index(conn, index_name="posts"):
    try:
        conn.execute_command('FT.DROPINDEX', index_name)
    except Exception as e:
        print("Failed to delete index: ", e)

def delete_all_keys_from_index(conn, index_name="posts"):
    try:
        # 1. Retrieve all document IDs from the index.
        # This assumes the total number of documents isn't large.
        # If it is, you might want to paginate the query.
        result = conn.execute_command('FT.SEARCH',
                                      index_name,
```

Redis connection details

Connects to the Redis server

Sets the dimensions to match the LLM design

Function to delete index

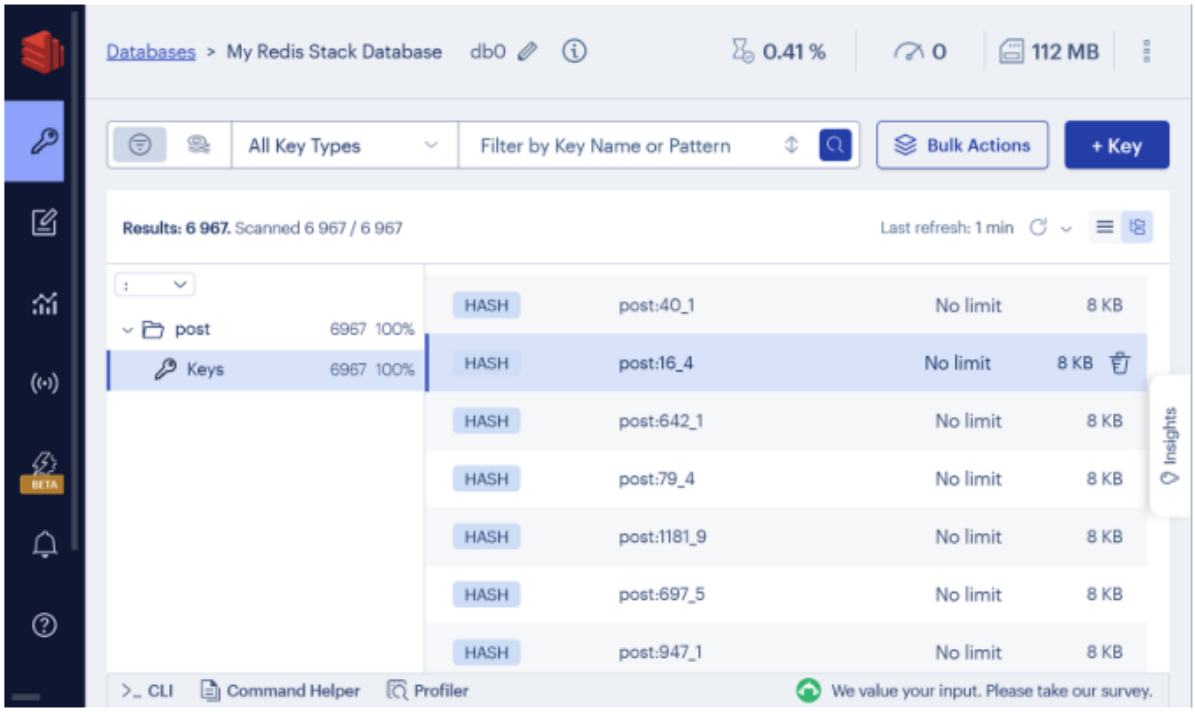
Function to delete the keys from the index

```
                                        '*',
                                        'NOCONTENT')

        # 2. Parse the result to get document IDs.
        # Skip the first element which is the total count.
        # Taking every second element starting from the first.
        doc_ids = result[1::2]

        # 3. Delete each document key.
        for doc_id in doc_ids:
            conn.delete(doc_id)

    except Exception as e:
        print("Failed to delete keys: ", e))

def view_index(conn, index_name="posts"):
    try:
        info = conn.execute_command('FT.INFO', index_name)
        for i in range(0, len(info), 2):
            print(f"{info[i]}: {info[i+1]}")
    except Exception as e:
        print("Failed to retrieve index details: ", e)

def main():
    while True:
        print("1. View index details")
        print("2. Create index")
        print("3. Delete index")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            # Call the function to view index
            view_index(conn)
            pass
        elif choice == '2':
            # Call the function to create index
            create_index(conn, SCHEMA)
        elif choice == '3':
            # Call the function to delete index
            delete_all_keys_from_index(conn)
            delete_index(conn)
        elif choice == '4':
            break
        else:
            print("Invalid choice. Please enter a valid option.")

if __name__ == "__main__":
    main()
```

Function to create an index

Function to run the main loop

Rysunek 4 przedstawia ten kod uruchomiony lokalnie jako przykład. Typ indeksu to HASH, a prefiks kluczy zaczyna się od „post”.



W naszym przypadku indeks jest już zapełniony i po uruchomieniu tej komendy w celu jego wyświetlenia otrzymujemy wynik podobny do poniższego. Należy pamiętać, że wynik został skrócony dla zwięzłości.

Listing 4: Szczegóły indeksu wyszukiwania Redis

index_name: posts

index_options: []

index_definition: ['key_type', 'HASH', 'prefixes', ['post:'],

'default_score', '1']

attributes: [['identifier', 'url', 'attribute', 'url', 'type',

➥'TAG', 'SEPARATOR', ','], ['identifier', 'title', 'attribute',

➥'title', 'type', 'TEXT', 'WEIGHT', '1'], ['identifier',

➥'description', 'attribute', 'description', 'type', 'TEXT',

➥'WEIGHT', '1'], ['identifier', 'publish_date', 'attribute',

➥'publish_date', 'type', 'TEXT', 'WEIGHT', '1'], ['identifier',

➥'embedding', 'attribute', 'embedding', 'type', 'VECTOR']]

num_docs: 1304

max_doc_id: 1304

num_terms: 3047

num_records: 14092

vector_index_sz_mb: 12.586814880371094

total_inverted_index_blocks: 4370

offset_vectors_sz_mb: 0.011086463928222656

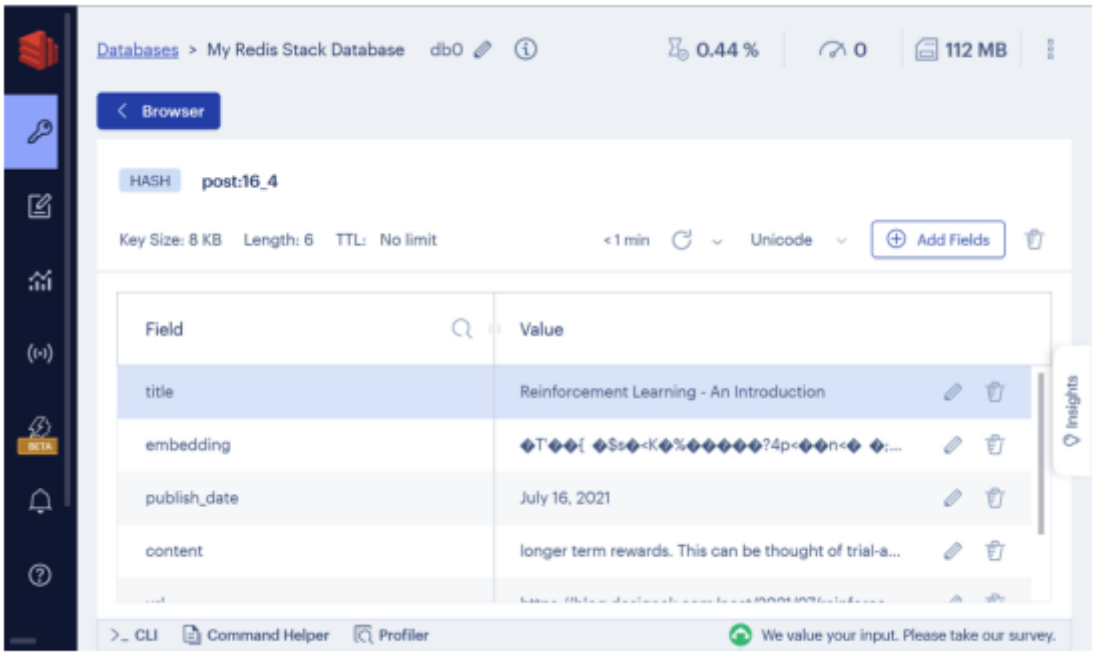
doc_table_size_mb: 0.09221076965332031

key_table_size_mb: 0.03916168212890625

total_indexing_time: 708.988

...

Rysunek 5 przedstawia szczegóły jednego z elementów indeksu za pomocą Redis Insight – graficznego interfejsu użytkownika, który umożliwia podstawowe zarządzanie bazą danych. Widać pola zidentyfikowane podczas konfigurowania indeksu. Osadzenia są binarne, więc wydają się bezsensowne.



Skoro mamy już ustawiony indeks, zobaczmy, jak możemy pobrać dane (tj. wpisy na blogu), podzielić je na fragmenty, zapełnić bazę danych wektorów i na koniec zaktualizować utworzony indeks.

Pobieranie danych

Na wysokim poziomie proces jest dość prosty. Zaczynamy od ładowania kanałów RSS za pomocą biblioteki feedparser; następnie pobieramy każdy znaleziony wpis na blogu, analizujemy go pod kątem interesującej nas treści, tworzymy odpowiednie osadzenie i zapisujemy wszystkie szczegóły w Redis.

Listing 5 ilustruje ten proces. Ponieważ każdy wpis na blogu jest stroną HTML, używamy BeautifulSoup, biblioteki Pythona, do analizy strony HTML, co pozwala nam wybrać potrzebną treść. Jak pokazano na listingu 5, musimy uporządkować pewne elementy i przeanalizować treść, dopasowując styl wpisu na blogu do wygenerowanego kodu HTML. Wyszukiwanie różnych atrybutów i klas (takich jak tytuł wpisu itp.) zależy od kształtu danych przychodzących i przypadku użycia, który próbujemy rozwiązać. W tym przykładzie kod musi zostać zaktualizowany, jeśli blog zmieni swój motyw lub renderowanie.

Listing 5: Wyodrębnianie zawartości z HTML

```
r = requests.get(post.link)
soup = BeautifulSoup(r.text, 'html.parser')
# Get the title
try:
article_title = soup.find('h1', {'class': 'post-title'}).text
article_title = article_title.replace("| Amit Bahree's
➡(useless?) insight!", "")
except AttributeError:
article_title = ""
print("\tTitle:" + article_title)
# get the post description
try:
article_desc = soup.find('div', {'class': 'post-description'}).text
except AttributeError as e:
#print("Error getting description: ", e)
article_desc = ""
# get the publish date
try:
temp = soup.find('div', {'class': 'post-meta'}, {'span',
'title'}).text
match = re.search(r"(\w+\s\d+,\s\d+)", temp)
if match:
publish_date = match.group(1)
except AttributeError:
publish_date = ""
# get the article body
try:
article_body = soup.find('div', {'class': 'post-content'}).text
except AttributeError:
article_body = ""
```

W rzeczywistych przypadkach użycia w przedsiębiorstwie, osoba pobierająca dane musi znać zawartość i strukturę systemu źródłowego, co może być dość złożone i zniechęcające. W większości przypadków wymagałoby to przetworzenia danych przez potok danych. Ten potok danych pomógłby rozwiązać wszelkie niezbędne aspekty inżynierii danych – wszystko w kontekście powiązanych przypadków użycia:

```
chunks = split_sentences_by_spacy(article, max_tokens=3000, overlap=10)

print(f"Number of chunks: {len(chunks)}")
```

Tworzymy nowy skrót indeksu, dodając szczegóły interesujących nas informacji jako osadzenia – adres URL, tytuł, datę publikacji i wpis na blogu. Porównujemy również różne fragmenty utworzone w tym samym kontekście. Zwróć uwagę, że w kolejnym listingu pokazujemy tylko kluczowy aspekt kodu, a ze względu na wagę problemu pomijamy funkcje pomocnicze, które widzieliśmy wcześniej.

Listing 6: Pobieranie wpisów blogowych i zapisywanie ich w Redis

```
# OpenAI API key

openai.api_key = os.getenv('OPENAI_API_BOOK_KEY')

# Redis connection details

redis_host = os.getenv('REDIS_HOST')

redis_port = os.getenv('REDIS_PORT')

redis_password = os.getenv('REDIS_PASSWORD')

def split_sentences_by_spacy(text, max_tokens,

overlap=0,

model="en_core_web_sm"):

...

# count tokens

def count_tokens(...)

...

def get_embedding(text):

...

# Connect to the Redis server

conn = redis.Redis(...)

SCHEMA = [ ... ]

# URL of the RSS feed to parse

url = https://blog.desigeek.com/index.xml

# Parse the RSS feed with feedparser

print("Parsing RSS feed...")
```

```
feed = feedparser.parse(url)
# get number of blog posts in feed
blog_posts = len(feed.entries)
print("Number of blog posts: ", blog_posts)
p = conn.pipeline(transaction=False)
for i, post in enumerate(feed.entries):
# report progress
print("Create embedding and save for entry #", i, " of ", blog_posts)
# Extract the content – using BeautifulSoup
r = requests.get(post.link)
soup = BeautifulSoup(r.text, 'html.parser')
# Get the title
...
# get the post description
...
# get the publish date
...
# get the article body
try:
article_body = soup.find('div', {'class': 'post-content'}).text
except AttributeError:
article_body = ""
# This should be chunked up
article = article_body
total_token_count = 0
chunks = []
# split the text into chunks by sentences
chunks = split_sentences_by_spacy(article, max_tokens=3000, overlap=10)
print(f"Number of chunks: {len(chunks)}")
for j, chunk in enumerate(tqdm(chunks))
vector = get_embedding(chunk)
```

```
# convert to numpy array

vector = np.array(vector).astype(np.float32).tobytes()

# Create a new hash with the URL and embedding

post_hash = {

"url": post.link,

"title": article_title,

"description": article_desc,

"publish_date": publish_date,

"content": chunk,

"embedding": vector

}

conn.hset(name=f"post:{i}_{j}", mapping=post_hash)

p.execute()

print("Vector upload complete.")
```

Po uzyskaniu treści wpisu na blogu musimy ją podzielić na fragmenty, jak omówiono w poprzednim rozdziale. W tym przykładzie używamy spaCy do podziału wpisu na fragmenty, a także do uwzględnienia pewnych nakładek między różnymi fragmentami.

Najlepsze praktyki dotyczące potoku pobierania

Wdrażając wzorzec RAG, kluczowe jest dogłębne zrozumienie treści i struktury systemu źródłowego. Sukces modelu RAG zależy od jego zdolności do dostępu do właściwych danych i ich interpretacji, co wymaga dobrze zaprojektowanego potoku danych. Ten potok to nie tylko kanał przepływu danych, ale także zaawansowana struktura, która zapewnia ekstrakcję, transformację, indeksowanie i przechowywanie danych zgodnie z wymaganiami modelu i zdefiniowanym przypadkiem użycia. Pierwszym krokiem do wdrożenia GPT i LLM w przedsiębiorstwach jest dogłębne zrozumienie systemu źródłowego. Obejmuje to dogłębną analizę struktury danych, w tym diagramów relacji encji, typów danych i dystrybucji danych. Narzędzia do profilowania danych mogą być pomocne w zrozumieniu charakteru treści.

UWAGA: Aby model RAG działał prawidłowo, ważne jest staranne zaplanowanie wstępnego przetwarzania w potoku wyszukiwania i nieużywanie wszystkiego bez zastanowienia się, czy jest to lepsze rozwiązanie. Brak dobrego planowania może prowadzić do problemów z wyszukiwaniem w ramach implementacji modelu RAG.

Następna faza definiuje przypadek użycia, co wiąże się ze stworzeniem szczegółowego dokumentu wymagań opisującego problem, potencjalne rozwiązania, oczekiwane rezultaty i wskaźniki sukcesu. Dokument ten powinien również szczegółowo opisywać potrzeby informacyjne użytkowników oraz scenariusze, w których model RAG będzie stosowany. Następnie nacisk kładzie się na ekstrakcję i transformację danych. Proces ten obejmuje użycie narzędzi ETL (ekstrakcja, transformacja, ładowanie) w celu ekstrakcji danych z systemu źródłowego i przekształcenia ich do formatu zrozumiałego dla modelu RAG. Może on obejmować techniki przetwarzania języka naturalnego (NLP), takie jak

tokenizacja, usuwanie słów ignorowanych (stop-word removal) i lematyzacja. Po transformacji dane muszą zostać zindeksowane w celu efektywnego pobierania. Azure AI Search, Elasticsearch, Solr i Lucene idealnie nadają się do tego celu, ponieważ oferują możliwości wyszukiwania pełnotekstowego i mogą efektywnie obsługiwać duże zbiory danych. Równolegle z indeksowaniem danych, ważny jest wybór odpowiedniego rozwiązania do przechowywania danych. W zależności od konkretnych potrzeb dotyczących rozmiaru, szybkości i typu danych, może to być tradycyjna baza danych SQL, baza danych NoSQL, taka jak Cosmos DB, lub rozproszony system plików, taki jak Hadoop HDFS. Jednym z najważniejszych etapów jest planowanie wstępnego przetwarzania. Obejmuje ono staranne zaplanowanie kroków wstępnego przetwarzania, które mogą obejmować techniki takie jak usuwanie szumów, normalizacja i redukcja wymiarowości. Celem jest zachowanie informacji istotnych dla przypadku użycia przy jednoczesnym zmniejszeniu złożoności modelu. Kolejnym etapem jest integracja modelu, która polega na wykorzystaniu interfejsów API lub zestawów SDK dostarczonych przez dostawcę modelu AI w celu zintegrowania modelu RAG z aplikacją. Narzędzie do pobierania musi być skonfigurowane z poprawnymi parametrami zapytania, a generator powinien mieć żądaną strukturę wyjściową. Dostrajanie i monitorowanie są kluczowe dla poprawy wydajności modelu i zapewnienia sprawności systemu. Wiąże się to z wykorzystaniem zestawu danych walidacyjnych do dostrajania oraz narzędzi do zarządzania wydajnością aplikacji (APM) do monitorowania. Aby zapewnić skalowalność i niezawodność, do skalowania systemu w razie potrzeby należy korzystać z platform chmurowych, takich jak AWS, Google Cloud lub Azure. Platformy konteneryzacyjne, takie jak Docker i Kubernetes, mogą pomóc w skalowaniu i zarządzaniu aplikacją. Strategie redundancji i przełączania awaryjnego mają kluczowe znaczenie dla zapewnienia niezawodności systemu. Ponadto nie można pominąć kwestii bezpieczeństwa i zgodności. Wdrożenie szyfrowania danych, uwierzytelniania użytkowników, kontroli dostępu i regularne audyty systemu mogą zapewnić bezpieczeństwo danych i zgodność z przepisami o ochronie danych, takimi jak RODO czy CCPA. Przed wdrożeniem konieczne są rygorystyczne testy i walidacja, aby upewnić się, że potok i model RAG spełniają oczekiwania określone w przypadku użycia. Po uruchomieniu systemu zespół powinien otrzymać kompleksową dokumentację i szkolenie techniczne w celu efektywnego zarządzania, konserwacji i rozwiązywania problemów. Wreszcie, kluczowe jest zapewnienie kontroli jakości korpusu wyszukiwania, wdrożenie środków bezpieczeństwa i prywatności informacji, regularna aktualizacja korpusu wyszukiwania oraz efektywna alokacja zasobów. Postępując zgodnie z tymi krokami, przedsiębiorstwa mogą skutecznie tworzyć i utrzymywać aplikacje oparte na sztucznej inteligencji.

Wyszukiwanie za pomocą Redis

Po pobraniu danych i przygotowaniu indeksu możemy je przeszukiwać. Tworzymy prostą aplikację konsolową, która akceptuje zapytanie użytkownika, wektoryzuje je i wyszukuje na podstawie trzech najbardziej podobnych wpisów, aby zwrócić je użytkownikowi. Jest to wyszukiwanie semantyczne. Poniższa lista pokazuje przykładowy wynik wygenerowany po zapytaniu o „Longhorn”.

Listing 7: Wyniki wyszukiwania

```
$ python .\search.py

Connected to Redis

Enter your query: Tell me about Longhorn

Vectorizing query...

Searching for similar posts...

Found 3 results:
```

You probably already heard this, but Chris Sells

➥has a new column on MSDN called Longhorn Foghorn

, that describes each of the â

Pillars of Longhorn

â - This is something that IMHO developers would understand and

➥appreciate. In the first article he explains the âPillarsâ and then

➥in the next two goes onto build Solitaire. You can download the sample

➥and play with it too.

From OSNews: Microsft has made hard statements about perfomance

➥improvements in Longhorn ...

UWAGA: Windows Longhorn był nazwą kodową systemu operacyjnego, który ostatecznie stał się Windows Vista.

Sprawdźmy kod implementujący wyszukiwanie za pomocą Redis. Najpierw bierzemy zapytanie użytkownika, takie jak „Opowiedz mi o Longhorn”, tworzymy wektor i używamy podobieństwa cosinusowego, aby uzyskać listę porównywalnych wyników.

Listing 8: Wyszukiwanie za pomocą Redis

```
def hybrid_search(query_vector, client, top_k=3, hybrid_fields="*"):
    base_query = f"{hybrid_fields}=>
                        [KNN {top_k}
                        @embedding $vector AS vector_score]"
    query = Query(base_query).return_fields(
        "url",
        "title",
        "publish_date",
        "description",
        "content",
        "vector_score").sort_by("vector_score").dialect(2)
    try:
        results = client.ft("posts").search(
            query, query_params={"vector": query_vector})
    except Exception as e:
        print("Error calling Redis search: ", e)
        return None

    if results.total == 0:
        print("No results found for the given query vector.")
        return None

    return results

# Connect to the Redis server
conn = redis.Redis(...)

query = input("Enter your query: ")

print("Vectorizing query...")
query_vector = get_embedding(query)

query_vector = np.array(query_vector).astype(

                        np.float32).tobytes()
print("Searching for similar posts...")
results = hybrid_search(query_vector, conn)

if results:
    print(f"Found {results.total} results:")
    for i, post in enumerate(results.docs):
        score = 1 - float(post.vector_score)
        print(post.content)
else:
    print("No results found")
```

A base query that prefilters fields and is implemented as a KNN search

Selects the different fields we are interested in searching

Sorts by cosine similarity in descending order

Executes the query

Captures the query from the user

Vectorizes the input

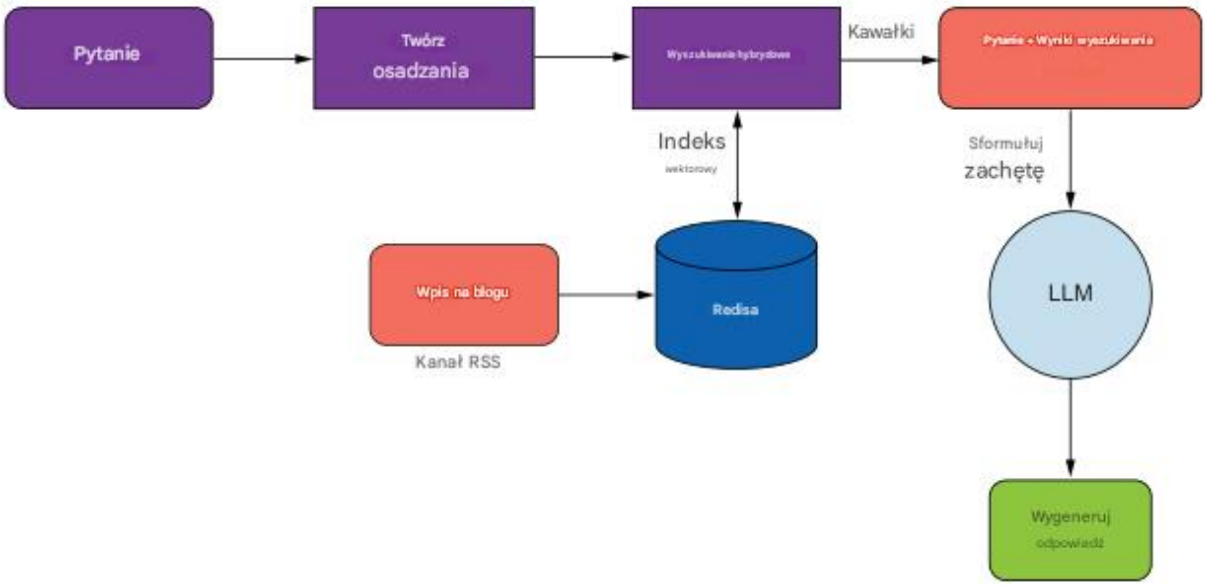
Converts the vector to a NumPy array

Performs the similarity search

Jak sama nazwa wskazuje, funkcja hybrid_search() wykonuje większość zadań związanych z uruchomieniem hybrydowego zapytania wyszukiwania. Hybrydowe zapytanie wyszukiwania łączy wiele typów wyszukiwań w jedno. Może to obejmować łączenie wyszukiwań tekstowych z innymi typami, takimi jak wyszukiwanie numeryczne, kategorialne, a nawet wektorowe. Należy pamiętać, że dokładny typ wyszukiwania zależy od informacji i wymagań. W naszym przykładzie łączymy wyszukiwanie metodą K-najbliższych sąsiadów (KNN) w wektorze osadzania z innymi polami wyszukiwania. Wyszukiwanie KNN znajduje elementy najbardziej powiązane z danym elementem, w tym przypadku najbardziej podobne wpisy do danego wektora zapytania. Wyniki zapytania są sortowane według wyniku wektora, co oznacza sortowanie od najwyższego do najniższego na podstawie podobieństwa cosinusowego. Innymi słowy, wyniki o najwyższym podobieństwie są wyświetlane jako pierwsze. Ograniczamy to również do trzech pierwszych elementów, jak pokazano za pomocą parametru top_k. Należy pamiętać, że dokładny charakter i typ wyszukiwania zależą również od wyszukiwarki i typu danych. Więcej informacji na temat typów wyszukiwania Redis i KNN można znaleźć w dokumentacji pod adresem https://mng.bz/o0Gp. Teraz, gdy poznaliśmy wyszukiwanie, połączmy wszystkie wymiary i zintegrujmy je w środowisku czatu za pomocą LLM.

Kompleksowa implementacja czatu oparta na RAG

W tym i poprzednim rozdziale omówiliśmy i przeanalizowaliśmy wszystkie elementy, aby pomóc nam zrozumieć niektóre z podstawowych koncepcji; teraz możemy połączyć to wszystko w całość i zbudować kompleksową aplikację czatu. W aplikacji możemy zadawać pytania, aby uzyskać szczegółowe informacje o naszych danych (tj. wpisach na blogu). Rysunek 6 przedstawia przepływ aplikacji.



Pytanie zadane przez użytkownika jest najpierw konwertowane na osadzenia, a następnie przeszukiwane w Redis za pomocą hybrydowego indeksu wyszukiwania w celu znalezienia podobnych fragmentów, które są zwracane jako wyniki wyszukiwania. Jak widzieliśmy wcześniej, wpisy na blogu zostały już wstrzyknięte do bazy danych Redis i zindeksowane. Po uzyskaniu wyników formułujemy monit LLM, łącząc oryginalne pytania i fragmenty pobrane w celu udzielenia odpowiedzi. Są one przekazywane do samego monitu, a następnie ostatecznie wywoływane przez LLM w celu wygenerowania odpowiedzi. W obszarze wyszukiwania wdrożyliśmy Redis działający lokalnie i utworzyliśmy indeks wektorowy. Przeczytaliśmy wszystkie wpisy na blogach sprzed prawie 20 lat. Stworzyliśmy odpowiednie fragmenty dla tych wpisów i odpowiadających im osadzeń, a następnie zapełniliśmy naszą bazę danych wektorowych. Zaimplementowaliśmy również wyszukiwanie wektorowe w tych osadzeniach. Pozostało nam tylko zintegrować to wszystko z naszą aplikacją i połączyć ją z LLM, aby ukończyć ostatni etap implementacji RAG. Listing 9 pokazuje dokładnie, jak to zrobić. Kilka funkcji pomocniczych, takich jak get_search_results(), przyjmuje zapytanie użytkownika, wywołuje inną funkcję pomocniczą w celu przeszukania Redis i zwraca znalezione wyniki. Faktyczne wywołanie API, które wywołuje GPT, znajduje się w funkcji ask_gpt() i jest to API ChatCompletion(), tak jak widzieliśmy wcześniej. Podobnie jak w poprzednich przykładach, pomijamy funkcje pomocnicze kodu i inne aspekty dla zwięzłości.

Listing 9: Czat typu end-to-end obsługiwany przez RAG

```
def hybrid_search(query_vector, client, top_k=5, hybrid_fields="*"):
...
    return results

def get_search_results(query:str, max_token=4096,
                       ➥debug_message=False) -> str:
    query_vector = get_embedding(query)

    query_vector = np.array(query_vector).astype(
        np.float32).tobytes()

    print("Searching for similar posts...")
    results = hybrid_search(query_vector, conn, top_k=5)

    token_budget = max_token - count_tokens(query)
    if debug_message:
        print(f"Token budget: {token_budget}")

    message = 'Use the blog post below to answer the subsequent
              ➥question. If the answer cannot be found in the
              ➥articles, write "Sorry, I could not find an answer in
              ➥the blog posts."'
    question = f"\n\nQuestion: {query}"

    if results:
        for i, post in enumerate(results.docs):
            next_post = f'\n\nBlog post:\n"""\n{post.content}\n"""'
            new_token_usage = count_tokens(message + question + next_post)
            if new_token_usage < token_budget:
                if debug_message:
                    print(f"Token usage: {new_token_usage}")
                message += next_post
            else:
                break
    else:
```

Vectorizes the query

Converts the vector to a numpy array

Performs the similarity search

Manages token budget

Loops through the results while still keeping within the token budget

```
        print("No results found")

    return message + question

def ask_gpt(query : str, max_token = 4096, debug_message = False) -> str:
    message = get_search_results(
        query,
        max_token,
        debug_message=debug_message)

    messages = [
        {"role":
         "system",
         "content": "You answer questions in summary from the [CA]
                     blog posts."},
        {"role":
          "user",
            "content": message},]

    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-16k",
        messages=messages,
        temperature=0.7,
        max_tokens=2000,
        top_p=0.95
    )
    response_message = response["choices"][0]["message"]["content"]
    return response_message

if __name__ == "__main__":
    # Enter a query
    while True:
        query = input("Please enter your query: ")
        print(ask_gpt(query, max_token=15000, debug_message=False))
        print("==="*20)
```

Runs a vector search to get embeddings

Sets up the chat completion calls

Calls the LLM

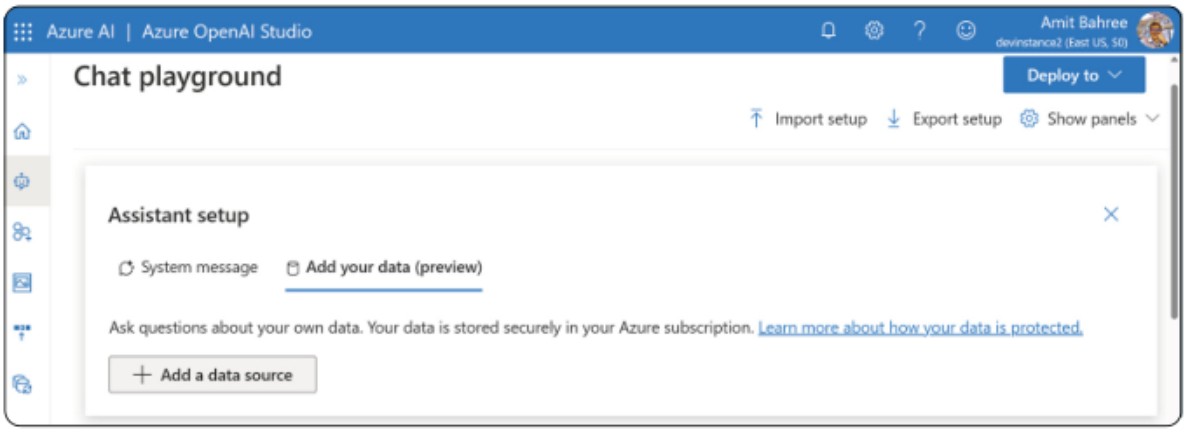
Wszystko to możemy zobaczyć, uruchamiając aplikację i rozmawiając z blogiem. Aplikacja rozumie zapytanie, tworzy osadzenia, wykorzystuje bazę danych wektorów i powiązane indeksy wektorów do pobrania pięciu najlepszych pasujących wyników, dodaje je do monitu i używa LLM do wygenerowania odpowiedzi (rysunek 7).

```
Command Prompt

Please enter your query: what is wcf?
Searching for similar posts...
WCF stands for Windows Communication Foundation.
============================================
Please enter your query: tell me about openai whisper
Searching for similar posts...
Whisper is a speech recognition model (ASR - automatic speech recognition) developed
by OpenAI. It is a multi-task model that not only performs speech recognition but als
o language identification and speech translation across multiple languages. Whisper i
s open-sourced and comes in five different sizes. Four of these sizes have an English
-only variant that performs better for English tasks. The model is robust to noise, a
ccents, background noise, and technical language. Whisper achieves near state-of-the-
art performance with zero-shot translation from multiple languages to English. It was
 trained on a large corpus of data using weak supervision and can transcribe and tran
slate across 98 languages. Whisper uses a seq-to-seq (transformer encoder-decoder) ar
chitecture and is best suited for transcribing audio to text use cases. It is not tra
ined for text-to-audio (TTS) or speech synthesis tasks and is not suitable for real-t
ime speech applications.
============================================
Please enter your query: what is longhorn
Searching for similar posts...
```

W przedstawionym do tej pory przykładzie jesteśmy odpowiedzialni za wszystko – od konfiguracji kontenerów Docker po wdrożenie Redis i pobieranie danych. To nie wystarczy, aby przedsiębiorstwa mogły przejść do produkcji. Wymagana jest dalsza inżynieria systemowa, taka jak konfiguracja różnych klastrów maszyn, skalowanie ich w górę lub w dół w razie potrzeby, zarządzanie Redis, wymaganiami bezpieczeństwa, ogólnymi operacjami itd. Wymaga to znacznej ilości czasu, wysiłku, kosztów i umiejętności, którymi nie każda organizacja może dysponować. Inną opcją jest użycie Azure OpenAI, które może wykonać wiele z tych zadań od razu po instalacji i pozwala organizacjom szybciej wprowadzić produkt na rynek, potencjalnie przy niższych kosztach. Zobaczmy, jak Azure OpenAI może osiągnąć ten sam rezultat, ale znacznie szybciej.

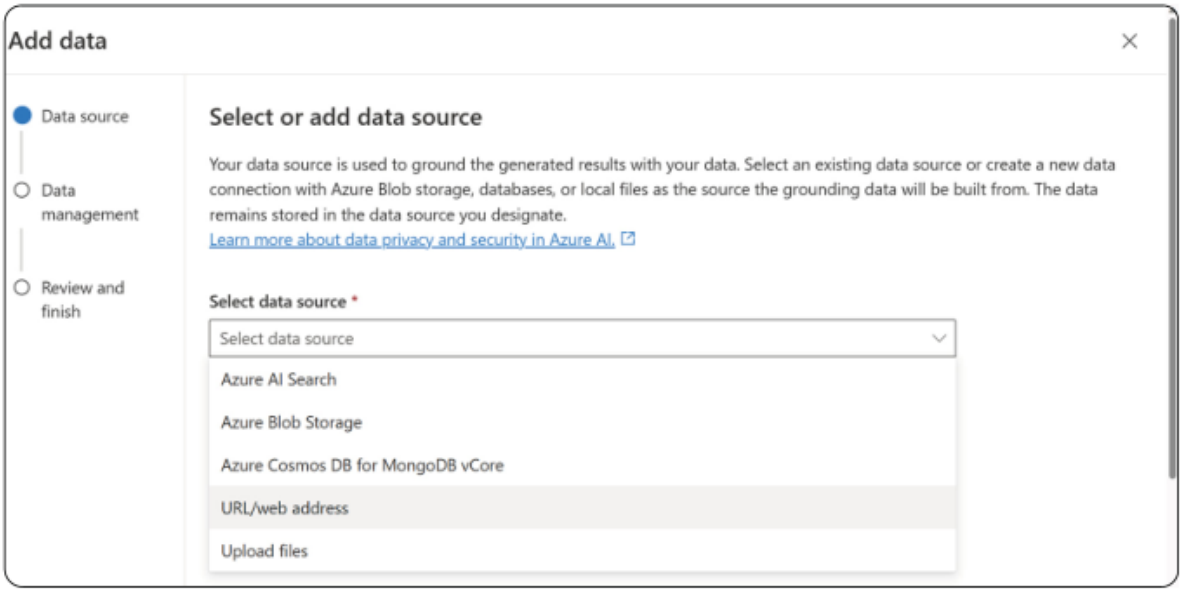
Korzystanie z Azure OpenAI w danych

Wiele przedsiębiorstw korzysta z platformy Azure, a włączenie Azure OpenAI do strategii zarządzania danymi stanowi kluczowy krok w wykorzystaniu potencjału generatywnej sztucznej inteligencji (AI) do transformacji biznesowej. Azure OpenAI zapewnia platformę klasy korporacyjnej do integracji zaawansowanych modeli AI, takich jak ChatGPT, z przepływami pracy danych. „Azure OpenAI w danych" to usługa, która umożliwia uruchamianie tych zaawansowanych modeli czatu w danych i korzystanie z gotowych funkcji, których przedsiębiorstwa potrzebują do obsługi obciążeń produkcyjnych: skalowalności, bezpieczeństwa, odświeżania i integracji z innymi. Źródło danych można połączyć za pomocą Azure OpenAI Studio (rysunek 8) lub interfejsu API REST.

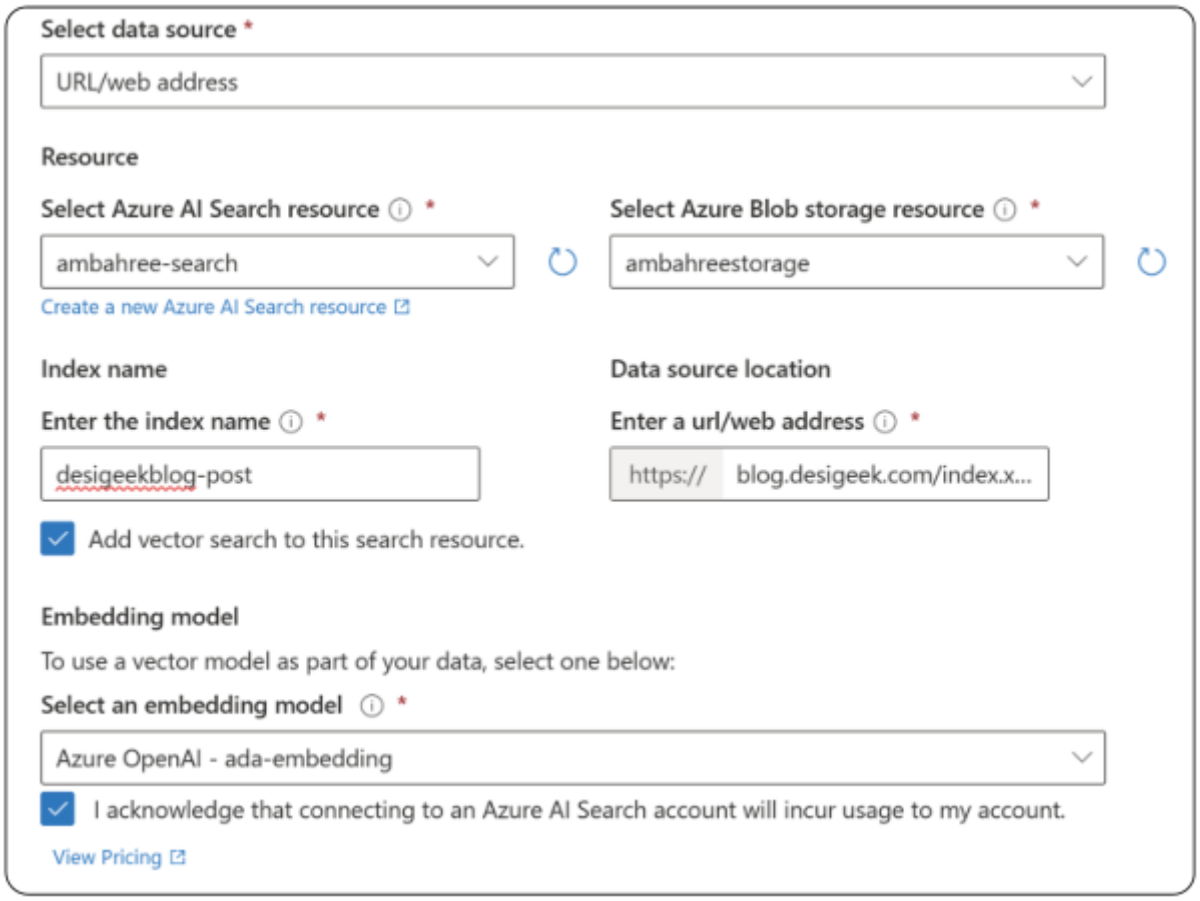


UWAGA: Azure AI Studio to platforma łącząca możliwości wielu usług Azure AI. Została zaprojektowana z myślą o deweloperach, którzy mogą tworzyć generatywne aplikacje AI na platformie klasy korporacyjnej. Można najpierw wejść w interakcję z kodem projektu za pośrednictwem zestawu Azure AI SDK i interfejsu wiersza polecenia Azure AI, a następnie płynnie eksplorować, kompilować, testować i wdrażać za pomocą najnowocześniejszych narzędzi AI i modeli uczenia maszynowego. Podstawą atrakcyjności Azure OpenAI jest płynna integracja z szerszym ekosystemem platformy Azure. Połączenie tych zaawansowanych modeli sztucznej inteligencji z repozytoriami danych otwiera możliwości bardziej zaawansowanej analizy danych, przetwarzania języka naturalnego i analiz predykcyjnych. Ta integracja jest szczególnie korzystna dla przedsiębiorstw z dużą liczbą użytkowników platformy Azure, umożliwiając im rozbudowę istniejącej infrastruktury przy minimalnych zakłóceniach.

Azure AI Studio obsługuje wiele opcji z istniejących indeksów Azure AI Search, magazynu obiektów blob, Cosmos DB i innych. Jedną z nich jest adres URL, którego użyjemy do pobierania wpisów z bloga (rysunek 9).



Możemy również zapisać kanał RSS lokalnie i przesłać go jako plik. Jedną z zalet korzystania z własnego indeksu Azure AI Search jest to, że przejmuje on trudną część zadań związanych z aktualizacją danych pobieranych z systemów źródłowych. Zastępuje on Redis i w razie potrzeby może być globalnie dystrybuowany do chmury. Tutaj możemy skonfigurować i skonfigurować większość rzeczy, w tym zasób pamięci masowej, w którym będą zapisywane te dane, zasób Azure AI Search, szczegóły indeksu, szczegóły osadzania itd. (rysunek 10). Wystarczy kilka kliknięć, aby wszystko to było gotowe do użycia.



W obszarze bezpieczeństwa informacji proces ten jest usprawniony dzięki solidnej strukturze zabezpieczeń i zgodności platformy Azure, która zapewnia ochronę danych w trakcie interakcji z modelami AI. Azure OpenAI obsługuje dwie kluczowe funkcje danych: kontrolę dostępu opartą na rolach i na poziomie dokumentów. Ta funkcja, współpracując z filtrami bezpieczeństwa Azure AI Search, może być używana do ograniczania dostępu tylko do tych użytkowników, którzy powinni mieć do niego dostęp na podstawie ich uprawnionych grup i członkostwa w LDAP, co jest kluczowym wymogiem dla wielu przedsiębiorstw, zwłaszcza w branżach regulowanych. Wreszcie, zdolność platformy Azure do przetwarzania i analizowania dużych wolumenów nieustrukturyzowanych danych w chmurze to kolejna istotna zaleta. Na przykład ChatGPT OpenAI wewnętrznie korzysta z Azure AI Search, a obciążenie to wynosi ponad 100 milionów użytkowników dziennie. Infrastruktura chmurowa platformy Azure umożliwia łatwe skalowanie możliwości AI w miarę wzrostu potrzeb w zakresie danych.

Korzyści z wykorzystania RAG w modelach LLM

Przedsiębiorstwa często mają trudności z wydobyciem istotnych wniosków z nieustrukturyzowanych źródeł danych, takich jak wiadomości e-mail, opinie klientów czy interakcje w mediach społecznościowych. Integrując swoje dane za pomocą RAG w modelach LLM, przedsiębiorstwa zyskują wiele korzyści, zwiększając funkcjonalność i przydatność tych systemów AI w swoich unikalnych kontekstach biznesowych. Ta funkcja oferuje wyraźne korzyści w porównaniu z samym rozszerzeniem okna kontekstowego tych modeli. Ten wzorzec zwiększa trafność i dokładność wyników LLM oraz zapewnia strategiczne korzyści, których samo szersze okno kontekstowe nie jest w stanie osiągnąć. Modele LLM mogą analizować te dane, interpretować je w sposób zbliżony do ludzkiego i dostarczać praktycznych wniosków, a wszystko to w ułamku czasu, jaki zajęłoby to tradycyjnym metodom.

Integracja RAG z danymi przedsiębiorstwa w czasie rzeczywistym gwarantuje, że informacje pobierane i uwzględniane w odpowiedziach są istotne i aktualne, co jest kluczowym czynnikiem w szybko rozwijających się branżach. Taka personalizacja prowadzi do bardziej precyzyjnych i przydatnych odpowiedzi, co jest szczególnie korzystne dla sektorów o specjalistycznej wiedzy, takich jak prawo, medycyna czy technika. Kluczową zaletą wykorzystania danych specyficznych dla przedsiębiorstwa w połączeniu z modelami RAG jest dostosowana dokładność i stosowalność odpowiedzi modelu. Modele LLM z szerszym oknem kontekstowym mogą przetwarzać więcej informacji na raz, ale nadal mogą im brakować dogłębnej wiedzy w wyspecjalizowanych dziedzinach. Gdy przedsiębiorstwa wprowadzają swoje dane, modele LLM mogą generować odpowiedzi ściśle powiązane ze specyfiką branży, żargonem i zawiłościami operacyjnymi organizacji. Ta specyfika jest kluczowa dla branż, w których specjalistyczna wiedza jest kluczowa i wykracza poza zakres tego, co może zapewnić szersze okno kontekstowe. Chociaż szersze okno kontekstowe pozwala na uwzględnienie szerszego zakresu istniejących wcześniej informacji w odpowiedziach modelu, niekoniecznie uwzględnia ono najbardziej aktualne lub specyficzne dla przedsiębiorstwa dane. Ponadto, im szersze okno kontekstowe, tym więcej danych musi przetworzyć model i tym wolniej działa. Co więcej, integracja zastrzeżonych danych usprawnia procesy decyzyjne skuteczniej niż samo rozszerzenie okna kontekstowego. Ta integracja umożliwia modelom LLM oferowanie spostrzeżeń i analiz głęboko zakorzenionych w historycznych danych przedsiębiorstwa i celach strategicznych. Z kolei szersze okno kontekstowe może dostarczać szerszych informacji, ale brakuje mu precyzji i bezpośredniego odniesienia do strategicznych pytań i wyzwań przedsiębiorstw. Jeśli chodzi o bezpieczeństwo i prywatność danych, objęcie danych zastrzeżonych kontrolą przedsiębiorstwa jest łatwiejsze w zarządzaniu niż poleganie na danych publicznych lub uogólnionych, do których dostęp może uzyskać szersze okno kontekstowe. Kontrolując dane wejściowe, przedsiębiorstwa mogą skuteczniej zapewnić zgodność z przepisami dotyczącymi prywatności danych. Wdrożenie RAG z danymi oferuje znaczące korzyści dla bezpieczeństwa sztucznej inteligencji w środowiskach przedsiębiorstw, przede wszystkim poprzez zwiększenie dokładności i wiarygodności informacji. To połączenie generatywnych możliwości modeli LLM z kompleksowym korpusem danych umożliwia modelowi dostęp do aktualnych, rzeczowych danych, kluczowych dla przedsiębiorstw zajmujących się informacjami wrażliwymi na czas i wymagającymi krytycznej dokładności. Co więcej, dzięki pobieraniu danych z różnych źródeł, RAG może łagodzić błędy wynikające z danych treningowych modeli LLM, co jest kluczową cechą podejmowania obiektywnych decyzji opartych na danych. Przedsiębiorstwa mogą dostosowywać korpus pobierania danych, zapewniając zgodność z przepisami branżowymi i wewnętrznymi zasadami. Co więcej, włączanie najnowszych informacji i udostępnianie źródeł generowanych treści zapewnia większą przejrzystość i wsparcie w podejmowaniu decyzji. O ile rozszerzenie okna kontekstowego LLM oferuje pewne korzyści, integracja zastrzeżonych danych z modelami RAG zapewnia szczegółowość, aktualną trafność, strategiczne dopasowanie, personalizację, bezpieczeństwo danych i potencjał innowacyjny, których nie zapewni samo rozszerzenie okna kontekstowego. Takie podejście umożliwia przedsiębiorstwom efektywniejsze wykorzystanie LLM w celu realizacji ich unikalnych potrzeb i celów biznesowych.

Podsumowanie

* Korzyści płynące z integracji danych zastrzeżonych z modelami RAG to zwiększenie specyficzności, trafności, strategicznego dopasowania, personalizacji, bezpieczeństwa danych i potencjału innowacyjnego systemów AI.

* Wykorzystanie danych zastrzeżonych zamiast samego rozszerzenia okna kontekstowego modeli LLM oferuje wiele korzyści, ponieważ pierwsze z nich zapewnia dokładniejsze, trafniejsze i bardziej spersonalizowane odpowiedzi.

* W środowisku produkcyjnym wykorzystanie wektorowej bazy danych i indeksu wektorów do zarządzania, zabezpieczania i skalowania osadzeń ma kluczowe znaczenie ze względu na wydajność i koszty.

* Proces pobierania danych zastrzeżonych, ich fragmentowania, tworzenia osadzeń i zapisywania szczegółów w wektorowej bazie danych zależy od kształtu dostępnych danych. Może wymagać znacznego nakładu pracy w zakresie planowania i inżynierii danych.

* Integracja wzorca RAG z systemem źródłowym jest złożona i wymaga planowania, solidnej inżynierii oraz zrozumienia szczegółów struktury danych.

* Kompleksowa aplikacja wykorzystująca RAG, szybką inżynierię, osadzenia i wyszukiwanie może być bardzo przydatna dla organizacji. Mimo to jest to również złożone rozwiązanie, które, jeśli nie zostanie odpowiednio zaprojektowane, spowolni proces wdrażania w środowisku produkcyjnym.

* W rozdziale opisano, jak przeprowadzać wyszukiwanie z wykorzystaniem wektorowej bazy danych, pobierając elementy najbardziej podobne do danego elementu na podstawie ich osadzeń wektorowych. Pokazano również, jak integracja wektorowych baz danych i RAG jest kluczowa dla wdrożenia kompleksowej aplikacji do czatu.

* „Azure OpenAI on your data” to usługa PaaS, która umożliwia przedsiębiorstwom uruchamianie modeli AI w oparciu o ich dane, oferując gotowe funkcje, takie jak skalowalność, bezpieczeństwo i integracja z innymi usługami platformy Azure.