Praca z API: Generowanie tekstu

Widzieliśmy, że duże modele językowe (LLM) zapewniają potężny zestaw narzędzi uczenia maszynowego, zaprojektowanych specjalnie w celu poprawy rozumienia i generowania języka naturalnego. OpenAI oferuje dwa godne uwagi API: API uzupełniania i API uzupełniania czatu. Te API, unikalne ze względu na swoje dynamiczne i efektywne możliwości generowania tekstu, przypominają generowanie tekstu przez człowieka. Ponadto oferują programistom wyjątkowe możliwości tworzenia różnorodnych aplikacji, od chatbotów po asystentów pisania. OpenAI jako pierwszy wprowadził wzorzec API uzupełniania i API uzupełniania czatu, który obecnie ucieleśnia niemal wszystkie implementacje, zwłaszcza gdy firmy chcą tworzyć narzędzia i produkty oparte na generatywnej sztucznej inteligencji. API uzupełniania OpenAI to zaawansowane narzędzie, które generuje odpowiedni kontekstowo i spójny tekst, aby uzupełnić komunikaty użytkownika. Z kolei API uzupełniania czatu zostało zaprojektowane tak, aby emulować interakcję z modelem uczenia maszynowego, zachowując kontekst rozmowy w wielu miejscach, co czyni je odpowiednim do aplikacji interaktywnych. Ta część stanowi podstawę skalowania przedsiębiorstw. Te interfejsy API mogą znacznie przyspieszyć tworzenie inteligentnych aplikacji, skracając tym samym czas potrzebny do osiągnięcia wartości. Jako przykłady ilustracyjne będziemy używać głównie OpenAI i Azure OpenAI, często zamiennie. Modele kodu pozostają spójne, a interfejsy API są w dużej mierze podobne. Wiele przedsiębiorstw może skłaniać się ku Azure OpenAI ze względu na oferowaną przez niego kontrolę, podczas gdy inne mogą preferować OpenAI. Należy zauważyć, że zakładamy tutaj, że instancja Azure OpenAI została już wdrożona w ramach subskrypcji platformy Azure i będziemy się do niej odwoływać w kontekście naszych przykładów. W tym rozdziale opisano podstawy interfejsów API uzupełniania i uzupełniania czatów, w tym różnice między nimi i momenty ich użycia. Zobaczymy, jak je zaimplementować w aplikacji i jak sterować generowaniem modelu oraz jego losowością. Zobaczymy również, jak zarządzać tokenami, które są kluczowymi zagadnieniami operacyjnymi podczas wdrażania w środowisku produkcyjnym. Są to podstawowe aspekty niezbędne do tworzenia aplikacji o znaczeniu krytycznym. Najpierw jednak zacznijmy od zrozumienia różnych kategorii modeli i ich zalet.

Kategorie modeli

Generatywne modele sztucznej inteligencji (LA) można podzielić na różne kategorie w zależności od ich konkretnych zastosowań, a każda kategoria obejmuje różne typy modeli. Rozpoczniemy naszą dyskusję od zrozumienia różnych klasyfikacji modeli w ramach generatywnej AI. To zrozumienie pomoże nam zidentyfikować zakres dostępnych modeli i wybrać ten najbardziej odpowiedni do danej sytuacji. Dostępność różnych typów i modeli może się różnić w zależności od używanego interfejsu API. Na przykład Azure OpenAI i OpenAI oferują różne wersje modeli LLM. Niektóre wersje mogą być wycofywane, inne ograniczone, a jeszcze inne mogą być dostępne wyłącznie dla określonej organizacji. Różne modele mają unikalne cechy i możliwości, co bezpośrednio wpływa na ich koszty i wymagania obliczeniowe. Dlatego wybór odpowiedniego modelu do każdego przypadku użycia ma kluczowe znaczenie. W konwencjonalnej informatyce idea, że większy znaczy lepszy, często była stosowana do pamięci, pamięci masowej, procesorów lub przepustowości. Jednak w przypadku modeli LLM zasada ta nie zawsze ma zastosowanie. OpenAI oferuje wiele modeli skategoryzowanych, jak pokazano w tabeli 3.1. Należy pamiętać, że są one takie same dla OpenAI i Azure OpenAI, ponieważ modele bazowe są identyczne.

Kategoria modelu: Opis

GPT-4: Najnowsza i najpotężniejsza wersja to zestaw modeli multimodalnych. GPT-4 jest trenowany na większym zbiorze danych z większą liczbą parametrów, co czyni go jeszcze bardziej wydajnym. Może wykonywać zadania, które były poza zasięgiem poprzednich modeli. Rodzina GPT-4 obejmuje różne

modele — GPT-4.0, GPT-4 Turbo oraz najnowszy GPT-4o (omni), model multimodalny i najpotężniejszy w rodzinie w momencie publikacji.

GPT-3.5: Zestaw modeli, które są udoskonalone w stosunku do GPT-3 i potrafią rozumieć i generować język naturalny lub kod. W razie wątpliwości, powinny to być modele domyślne dla większości przedsiębiorstw.

DALL.E: Model, który może generować obrazy po otrzymaniu monitu. Szept. Model używany do zamiany mowy na tekst, konwertujący dźwięk na tekst. Osadzenia. Zestaw modeli do konwersji tekstu na postać numeryczną.

GPT-3 (starsza wersja): Zestaw modeli, które mogą generować i rozumieć język naturalny. To był oryginalny zestaw modeli, które obecnie uważa się za starsze. W większości przypadków chcielibyśmy zacząć od jednego z nowszych modeli, 3.5 lub 4.0, które wywodzą się z GPT-3.

Każda kategoria modeli zawiera warianty, które dodatkowo wyróżniają się pewnymi cechami, takimi jak rozmiar tokena. Jak omówiono w poprzednim rozdziale, rozmiar tokena określa okno kontekstowe modelu, które definiuje ilość danych wejściowych i wyjściowych, jakie model może przetworzyć. Na przykład, oryginalne modele GPT-3 miały maksymalny rozmiar tokena 2 KB. GPT-3.5 Turbo, podzbiór modeli w kategorii GPT-3.5, ma dwie wersje — jedną z rozmiarem tokena 4 KB i drugą z rozmiarem tokena 16 KB. Są one dwa i cztery razy większe od rozmiaru tokena oryginalnych modeli GPT-3. Tabela 3.2 przedstawia najpopularniejsze modele i ich możliwości.

Model: Możliwości

Ada (starszy): Prosta klasyfikacja, parsowanie i formatowanie tekstu. Ten model jest częścią dziedzictwa GPT-3.

Babbage (starszy): Semantyczne pozycjonowanie wyszukiwania, średnio złożona klasyfikacja. Ten model jest częścią dziedzictwa GPT-3.

Curie (starszy): Odpowiadanie na pytania, wysoce złożona klasyfikacja. Ten model jest częścią dziedzictwa GPT-3.

Davinci (starszy): Podsumowanie, generowanie kreatywnej treści. Ten model jest częścią dziedzictwa GPT-3.

Cushman-Codex (starszy): Potomek serii GPT-3, trenowany w języku naturalnym i miliardach linii kodu. Jest najbardziej zaawansowany w Pythonie i biegły w ponad tuzinie innych języków programowania.

Davinci-Codex: Bardziej zaawansowany model Cushman-Codex

GPT3.5-Turbo: Najbardziej wydajny model GPT-3.5 zoptymalizowany pod kątem zastosowań w czatach jest o 90% tańszy i bardziej efektywny niż GPT-3 Davinci.

GPT-4, GPT-4 Turbo: Większy możliwości niż jakikolwiek model GPT-3.5. Potrafi wykonywać bardziej złożone zadania i jest zoptymalizowany pod kątem modeli czatów.

GPT-4o: Najnowszy model GPT-4o jest bardziej wydajny niż GPT-4 i GPT-4 Turbo, ale jest również dwa razy szybszy i o 50% tańszy.

text-embedding-ada-002, text-embedding-ada-003: Ten nowy model osadzania zastępuje pięć oddzielnych modeli do wyszukiwania tekstu, podobieństwa i wyszukiwania kodu, przewyższając je w większości zadań; co więcej, jest o 99,8% tańszy.

Należy pamiętać, że wspomniane starsze modele są nadal dostępne i działają zgodnie z przeznaczeniem. Jednak nowsze modele są lepsze, cieszą się większym zainteresowaniem i są wspierane przez dłuższy czas. Większość powinna zacząć od modelu GPT-3.5 Turbo jako domyślnego i używać GPT-4 w zależności od przypadku. Czasami nawet mniejszy, starszy model, taki jak GPT-3 Curie, jest dobry. Zapewnia to odpowiednią równowagę między możliwościami modelu, kosztem i ogólną wydajnością. We wczesnych latach generatywnej sztucznej inteligencji wszystkie modele były dostępne tylko dla wybranych. Różnią się one w zależności od firmy, regionu, a w przypadku platformy Azure, między innymi, rodzaju subskrypcji. Musimy wymienić modele i ich możliwości, z których możemy korzystać. Zanim jednak przejdziemy do listy modeli, przyjrzyjmy się zależnościom wymaganym do uruchomienia.

Zależności

W tej sekcji omówimy zależności i konfiguracje środowiska wykonawczego niezbędne na wysokim poziomie. Aby wszystko działało, potrzebujemy co najmniej następujących elementów:

* Środowisko programistyczne (IDE) — w naszych przykładach używamy programu Visual Studio Code, ale możesz użyć dowolnego, z którym czujesz się komfortowo.

* Python - używamy wersji 3.11.3, ale możesz użyć dowolnej wersji, pod warunkiem, że jest to wersja 3.7.1 lub nowsza. Instrukcje instalacji są dostępne na stronie https://www.python.org/, jeśli musisz zainstalować Pythona.

* Biblioteki OpenAI Python — w większości kodu i demonstracji używamy bibliotek Python. Bibliotekę OpenAI Python można łatwo zainstalować w środowisku Conda, używając polecenia conda install -c conda-forge openai. Jeśli używasz narzędzia PIP, użyj polecenia pip install --upgrade openai. Dostępne są również zestawy SDK (Software Development Kit) dla określonych języków, jeśli wolisz z nich korzystać zamiast pakietów Pythona.

* Subskrypcja platformy Azure lub dostęp do interfejsu API OpenAI — używamy zamiennie punktu końcowego OpenAI i punktu końcowego Azure OpenAI (AOAI); w większości przypadków obie opcje będą działać. Biorąc pod uwagę nacisk, jaki w tej książce kładziemy na przedsiębiorstwa, skłaniamy się ku usłudze Azure OpenAI:

– Aby korzystać z biblioteki z punktami końcowymi platformy Azure, potrzebujemy klucza api_key.

– Musimy również ustawić właściwości api_type, api_base i api_version. Parametr api_type musi być ustawiony na azure, parametr api_base wskazuje na wdrażany punkt końcowy, a odpowiednia wersja interfejsu API jest określana za pomocą parametru api_version.

– Azure OpenAI używa parametru „engine” do określenia nazwy modelu. Podczas wdrażania modelu w subskrypcji platformy Azure, nazwa ta musi być ustawiona na wybraną nazwę. Na przykład, rysunek 3.1 przedstawia zrzut ekranu wdrożeń w ramach jednej subskrypcji. OpenAI używa jednak parametru „model” do określenia nazwy modelu. Te nazwy modeli są standardowe w momencie ich wydania.

Zakodowanie punktu końcowego i klucza na stałe nie jest zalecaną praktyką. Istnieje wiele metod wykonania tego zadania, z których jedna obejmuje użycie zmiennych środowiskowych. Zaprezentujemy tę metodę w poniższych krokach. Innymi alternatywami mogą być pobieranie ich z magazynów sekretów lub plików środowiskowych. Dla uproszczenia w tym przewodniku skupimy się na zmiennych środowiskowych. Zachęcamy jednak do przestrzegania najlepszych praktyk i zaleceń obowiązujących w firmie. Konfigurację zmiennych środowiskowych można wykonać za pomocą następujących poleceń.

W systemie Windows są to:

setx AOAI_KEY "klucz-otwiera-i"

setx AOAI_ENDPOINT "punkt-otwiera-i"

UWAGA: Aby odczytać nowe zmienne, może być konieczne ponowne uruchomienie terminala.

W systemach Linux/Mac mamy:

export AOAI_ENDPOINT=twój-punkt-końcowy-openai

export AOAI_KEY=twój-klucz-openai

Bash używa:

echo export AOAI_KEY="TWÓJ_KLUCZ" >> /etc/environment && source /etc/environment

echo export AOAI_ENDPOINT="TWÓJ_PUNKT_KOŃCOWY" >> /etc/environment && source /etc/environment

UWAGA: Będziemy używać conda, menedżera pakietów open source, do zarządzania naszymi konkretnymi wersjami środowiska uruchomieniowego i zależnościami. Technicznie rzecz biorąc, korzystanie z menedżera pakietów takiego jak conda nie jest obowiązkowe, ale jest niezwykle przydatne w izolowaniu i rozwiązywaniu problemów i jest zdecydowanie zalecane. Nie będziemy zagłębiać się w szczegóły instalacji conda w tym kontekście; szczegółowe informacje można znaleźć tutaj:

Szczegółowe instrukcje dotyczące instalacji znajdziesz w oficjalnej dokumentacji pod adresem https://docs.conda.io/.

Najpierw utwórzmy nowe środowisko Conda i zainstalujmy wymaganą bibliotekę OpenAI Python:

$ conda create -n openai python=3.11.3

(base) $ conda activate openai

(openai) $ conda install -c conda-forge openai

Teraz, gdy mamy już zainstalowane zależności, połączmy się z punktem końcowym Azure OpenAI i uzyskajmy szczegółowe informacje o dostępnych modelach.

Wyświetlanie modeli

Jak już wcześniej wspomnieliśmy, każda organizacja może korzystać z innych modeli. Zaczniemy od zrozumienia, do jakich modeli mamy dostęp; wykorzystamy interfejsy API, aby skonfigurować podstawowe środowisko i je uruchomić. Następnie pokażę, jak to zrobić za pomocą zestawu SDK języka Python platformy Azure OpenAI i omówię różnice w korzystaniu z interfejsu API OpenAI.

Jak pokazuje poniższy opis, łączymy się z punktem końcowym usługi Azure OpenAI, pobieramy listę wszystkich dostępnych modeli, iterujemy je i drukujemy szczegóły każdego modelu w konsoli.

Listing 3.1 Listing Azure OpenAI models available

```
import os
import json
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2023-05-15",
    api_key=os.getenv("AOAI_KEY")
    )

# Call the models API to retrieve a list of available models
models = client.models.list()

# save to file
with open('azure-oai-models.json', 'w') as file:
    models_dict = [model.__dict__ for model in models]
    json.dump(models_dict, file)

# Print out the names of all the available models, and their capabilities
for model in models:
    print("ID:", model.id)
    print("Current status:", model.lifecycle_status)
    print("Model capabilities:", model.capabilities)
    print("-------------------")
```

Required for Azure OpenAI endpoints

This is the environment variable pointing to the endpoint published via the Azure portal.

Choose the API version we want to use from the multiple options.

This is the environment variable with the API key.

Uruchomienie tego kodu wyświetli listę dostępnych modeli. Poniższa lista przedstawia przykład dostępnych modeli; dokładna lista może być inna dla Ciebie.

Listing 2 : Wykaz danych wyjściowych modeli Azure OpenAI

{

"id": "gpt-4-vision-preview",

"created": null,

"object": "model",

"owned_by": null

},

{

"id": "dall-e-3",

"created": null,

"object": "model",

"owned_by": null

},

{

"id": "gpt-35-turbo",

"created": null,

"object": "model",

"owned_by": null

},

…

Każdy model charakteryzuje się swoimi unikalnymi możliwościami, sugerującymi przypadki użycia, do których jest dostosowany – w szczególności do uzupełniania czatów, uzupełniania (czyli zwykłego uzupełniania tekstu), osadzania i dostrajania. Na przykład, model uzupełniania czatów byłby idealnym wyborem w sytuacji, gdy wymagane jest zaangażowanie konwersacyjne, na przykład interakcja oparta na czacie, która wymaga znaczącej wymiany dialogowej. Z kolei model uzupełniania byłby najbardziej odpowiedni do generowania tekstu. Modele bazowe OpenAI z Azure AI Studio można zobaczyć na rysunku.

Base models

Deploy | Create a custom model | Column options | Refresh

Model name	Model version	Status	Deployable
code-davinci-002	1	Succeeded	Yes
gpt-4	0613	Succeeded	Yes
gpt-4-32k	0613	Succeeded	Yes
gpt-35-turbo	0301	Succeeded	Yes
gpt-35-turbo	0613	Succeeded	Yes
gpt-35-turbo-16k	0613	Succeeded	Yes

Ta funkcja jest częścią Azure AI Studio, do którego można uzyskać dostęp po zalogowaniu się do subskrypcji platformy Azure i uzyskaniu dostępu do wdrożenia Azure OpenAI. Dostęp do niej można również uzyskać bezpośrednio za pośrednictwem portalu pod adresem https://oai.azure.com/portal. Teraz, gdy wiemy, którego modelu użyć, wygenerujmy tekst. Skorzystamy z interfejsu API uzupełniania i modelu obsługującego uzupełnianie.

Interfejs API uzupełniania

Interfejs API uzupełniania to zaawansowane narzędzie, które generuje tekst w celu uzupełnienia podpowiedzi dostarczonych przez użytkownika. Stanowi on podstawę interfejsu API OpenAI i oferuje proste, a jednocześnie solidne i elastyczne API. Został zaprojektowany w celu generowania tekstu, który jest spójny i kontekstowo dopasowany do danego podpowiedzi. Wiele przykładów generowania, które nie są konstrukcjami typu czat, korzysta z interfejsu API uzupełniania. Musimy użyć interfejsu API

uzupełniania, aby wygenerować tekst, który nie jest konwersacją w stylu czatu. Oto niektóre z zalet interfejsu API uzupełniania:

* Zrozumienie kontekstu — interfejs API uzupełniania potrafi zrozumieć kontekst podpowiedzi i wygenerować odpowiedni tekst.

* Wszechstronność — można go używać w różnych aplikacjach, od tworzenia treści po odpowiadanie na pytania, co czyni go cennym narzędziem dla wielu zastosowań.

* Rozumienie wielu języków — API uzupełniania potrafi rozumieć i generować treści w wielu językach, co czyni je globalnym zasobem.

* Łatwa implementacja — API uzupełniania jest proste, co czyni je dostępnym dla programistów o różnym poziomie umiejętności.

Struktura API jest dość prosta, jak pokazano w poniższym fragmencie kodu. Dane wejściowe (polecenie) i wyjściowe (uzupełnienie) są w formacie tekstowym. Odpowiedź API to obiekt JSON, z którego wygenerowany tekst można wyodrębnić za pomocą klucza tekstowego. Ta odpowiedź nazywa się uzupełnianiem tekstu. Uzupełnienie dąży do przestrzegania instrukcji i kontekstu podanych w poleceniu i jest jednym z potencjalnych wyników:

```
from openai import OpenAI

client = OpenAI()

response = client.completions.create(

model="gpt-3.5-turbo-instruct",

prompt="Write a few bullets on why pets are so awesome ",

max_tokens=100,

temperature=0.8

)

print(response.choices[0].text.strip())
```

Zaczynamy od instrukcji, czyli komunikatu, który określa, co chcemy wygenerować. W naszym przykładzie instrukcja prosi model o wygenerowanie kilku punktów wyjaśniających, dlaczego zwierzęta domowe są niesamowite. Interfejs API uzupełniania ma wiele parametrów, ale te najważniejsze zostały szczegółowo opisane w tabeli . Wiele innych parametrów omówiliśmy wcześniej w tym rozdziale i w książce (np. komunikat, tokeny i temperatury). Sekwencje zatrzymania to jednak nowa koncepcja. Możemy wykorzystać te sekwencje, aby model przestał generować tokeny w określonym momencie, na przykład na końcu zdania lub listy.

Parametr: Typ: Wartość domyślna: Opis

prompt: Ciąg lub tablica: <\|endoftext\|>: Ciąg lub tablica ciągów to zachęta używana do generowania tych uzupełnień.

max_tokens: Liczba całkowita: 16: Maksymalna liczba tokenów do wygenerowania w uzupełnieniu, wliczając zachętę. Wartość max_tokens nie może przekraczać długości kontekstu modelu.

temperature: Liczba (zmiennoprzecinkowa): 1: Zakres od 0 do 2. Wyższe wartości oznaczają, że model podejmuje większe ryzyko i staje się bardziej kreatywny.

stop: Ciąg lub tablica: Null: Może to być do czterech sekwencji, w których API przestaje generować kolejne tokeny. Zwrócony tekst nie będzie zawierał sekwencji zatrzymania.

n: Liczba całkowita: 1 (opcjonalnie): Definiuje liczbę uzupełnień do wygenerowania dla każdego monitu. Generuje to wiele uzupełnień i może szybko wyczerpać limit tokenów; Powinniśmy mieć rozsądne ustawienie dla max_tokens i przestać zarządzać kosztami.

stream: Boolean: False (opcjonalnie): Ta flaga kontroluje, czy przesyłać strumieniowo częściowy postęp podczas generowania tokenów. Jeśli jest ustawiona,

strumień jest przerywany komunikatem data[DONE].

best_of: Integer: 1 (opcjonalnie): Ta flaga generuje uzupełnienia best_of po stronie serwera i zwraca najlepsze uzupełnienie. Tego parametru

nie można używać z gpt-35-turbo.

top_p:Number: (float): 1 (opcjonalnie): Ta opcja kontroluje losowość za pomocą techniki zwanej próbkowaniem jądra, alternatywy dla

ustawienia temperatury z wartością z zakresu od 0 do 1.

logit_bias: Map: Null (opcjonalnie): Ta opcja definiuje prawdopodobieństwo pojawienia się określonych tokenów w uzupełnieniu. Wykorzystuje mapowanie

tokenów na wartość odchylenia (–100 w przypadku bana na 100 w przypadku wyboru wyłącznego).

user: String: Null (opcjonalnie): Ten parametr jest unikalnym identyfikatorem reprezentującym użytkownika końcowego; może pomóc w debugowaniu, monitorowaniu i wykrywaniu nadużyć.

logprobs: Integer: Null (opcjonalnie): Jest to opcjonalna tablica prawdopodobieństw logu reprezentująca alternatywne tokeny i prawdopodobieństwo ich

uznania za ukończone. Tego parametru nie można używać z gpt-35-turbo.

suffix: String: Null (opcjonalnie): Ten parametr może być ciągiem do 40 znaków dodanym jako sufiks do wygenerowanego tekstu.

echo : Boolean : False (opcjonalnie): Określa, czy monit jest uwzględniany w uzupełnieniu. Jest to przydatne w przypadkach użycia, w których konieczne jest przechwycenie monitów oraz do celów debugowania.

Nie można go używać z gpt-35-turbo.

present_penalty : Liczba (zmiennoprzecinkowa): 0 (opcjonalnie): Ten parametr steruje tendencją modelu i pomaga określić jego zachowanie w zakresie wprowadzania nowych tematów lub idei do generowanego tekstu. Jego zakres wynosi od 0,0 do 1,0.

frequency_penalty : Liczba (zmiennoprzecinkowa): 0 (opcjonalnie): To kolejny parametr, który pomaga sterować modelem i poprawiać wyniki generowania. Kontroluje poziom słów powszechnych lub nietypowych w generowanym tekście i może być ustawiony na wartość od 0,0 do 1,0.

function_call : Kontroluje sposób, w jaki model reaguje na funkcje, gdy wymagane jest ich wywołanie. Działa tylko z modelami OpenAI w wersji 0613 i nowszych.

functions : : Oto lista funkcji, z których może korzystać model.

Należy pamiętać, że tabela zawiera tylko najczęściej używane parametry. Pomaga nam to zrozumieć niektóre przepływy i koncepcje. Niektóre parametry, takie jak funkcje, mają bardziej zaawansowane zastosowania, które zostaną omówione w dalszych rozdziałach poświęconych inżynierii błyskawicznej. Kontynuujemy temat zwierząt domowych i korzystamy z modelu, aby zasugerować nazwy dla salonu pielęgnacji zwierząt. Prosimy o podanie trzech nazw, a instrukcje określają również niektóre ważne cechy, których należy użyć. Te aspekty instrukcji pomagają nam ukierunkować model na pożądane atrybuty. Pełną listę parametrów można znaleźć w dokumentacji API. Wywołajmy API uzupełniania i przejrzyjmy je.

Listing 3 : Wywołanie interfejsu API uzupełniania

```
import os
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2024-05-01-preview",
    api_key=os.getenv("AOAI_KEY"))

prompt_startphrase = "Suggest three names for a new pet salon business.
➥The generated name ideas should evoke positive emotions and the
➥following key features: Professional, friendly, Personalized Service."

response = client.completions.create(
    model="gpt35",
    prompt=prompt_startphrase,
```

Prompt

Specifies the model to use; note that this name will change based on what you set in the deployment

Completion API call for generating text

```
    temperature=0.7,
    max_tokens=100,
    suffix="\nThats all folks!",
    stop=None)

responsetext = response.choices[0].text

print("Prompt:" + prompt_startphrase + "\nResponse:" + responsetext)
```

Model configurations

Extracts the generated text from the response

Gratulacje! Użyliśmy API do naszego pierwszego generowania tekstu. Ze względu na niedeterministyczną naturę sztucznej inteligencji, zwłaszcza sztucznej inteligencji generatywnej, wynik, który zobaczysz po uruchomieniu, różni się od

$ python .\petsalon.py

Wynik jest następujący.

Zaproponuj trzy nazwy dla nowego salonu pielęgnacji zwierząt. Wygenerowane pomysły na nazwy powinny budzić pozytywne emocje i charakteryzować się następującymi kluczowymi cechami: profesjonalną, przyjazną i spersonalizowaną obsługą.

1 Pawsitively Professional Pet Salon

2 Fur & Feathers Friendly Pet Parlor

3 Happy Tails Personalized Pet Pampering

UWAGA: Modele LLM i większość innych generatywnych modeli sztucznej inteligencji są niedeterministyczne, co oznacza, że identyczne dane wejściowe mogą dawać różne dane wyjściowe. Zmiana ustawienia temperatury na zero może sprawić, że dane wyjściowe będą bardziej deterministyczne, ale może pozostać niewielka zmienność.

Rozszerzanie uzupełnień

Zobaczmy, jak wygląda kompletna odpowiedź z API i przejrzyjmy jej strukturę. Poniższy listing przedstawia pełną odpowiedź z API. Pole choices jest jednym z najciekawszych, ponieważ zawiera tekst uzupełnienia. Właściwość choices to tablica, w której każdy element ma indeks, powód zakończenia generacji (finish_reason) oraz wygenerowany tekst (za pośrednictwem właściwości text).

Listing 4: Odpowiedź API z interfejsu API uzupełnienia

```
{
  "choices": [        <- Array of completion data
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "\n\n1. Pawfect Professionals \n
              ➥2. Purrsonalized Pet Pampering\n
              ➥3. Friendly Fur Services",
              "content_filter_results"={...}


    }
  ],
  "created": 1689007663,          <- Response creation datetime stamp
  "id": "cmpl-7aoL1MaUEf2j3ZLfSvsUOR7EFyjqC",   <- Unique ID of the response
  "model": "gpt-35-turbo",        <- Model ID used to generate the response
  "object": "text_completion",
  "usage": {                      <- Count of tokens used in this request
    "completion_tokens": 26,
    "prompt_tokens": 32,
    "total_tokens": 58
  }
}
```

Tabela przedstawia pozostałe właściwości. Właściwość „use” określa użyte tokeny (total_tokens), w tym tokeny promptu i odpowiedzi. Ponieważ płacimy za token, ważne jest, aby prompt był

skonstruowany pod kątem aspektów – po pierwsze, aby zwracał tylko to, co jest potrzebne, minimalizując użycie tokenów, a po drugie, aby ograniczał liczbę generowanych tokenów.

Właściwość: Opis

choices: Tablica, która może zawierać jedną lub więcej danych uzupełnień

created: Znacznik daty i godziny utworzenia odpowiedzi w systemie UNIX

id: Unikalny identyfikator odpowiedzi jest przydatny, gdy musimy śledzić odpowiedzi

model: Reprezentuje model użyty do generowania

object: Określa typ danych odpowiedzi (np. w tym przypadku jest to text_completion, określający API uzupełnień)

use: Zlicza liczbę tokenów użytych przez to żądanie

Właściwość o nazwie logprobs określa liczbę logarytmicznych prawdopodobieństw, które mają zostać wygenerowane dla każdego tokena w odpowiedzi. Logarytmy prawdopodobieństw są przydatne do generowania bardziej zróżnicowanych i interesujących odpowiedzi. Zwraca logarytmiczne prawdopodobieństwa n najwyższych tokenów dla każdego tokena w odpowiedzi. Logarytmy prawdopodobieństw są zwracane jako tablica tablic, gdzie każda podtablica odpowiada tokenowi w odpowiedzi i zawiera logarytmiczne prawdopodobieństwa n najwyższych tokenów dla tego tokena.

Filtr bezpieczeństwa zawartości platformy Azure

Czasami interfejs API zwraca odpowiedź null, jak pokazano na listingu 3.5. W takim przypadku należy sprawdzić wartość pola finish_reason. Jeśli jego wartość jest ustawiona na content_filter, system filtrowania zawartości współpracujący z modelami został uruchomiony. Pole finish_reason wskazuje, dlaczego interfejs API zwrócił dane wyjściowe, a każda odpowiedź będzie zawierać to pole. Ten temat zostanie omówiony bardziej szczegółowo w dalszej części rozdziału. System filtrowania wykorzystuje określone kategorie do identyfikacji potencjalnie szkodliwych treści i reagowania na nie zarówno w ramach monitów wejściowych, jak i generowanych uzupełnień. Aplikacja korzystająca z tych interfejsów API musi obsłużyć tę sytuację i spróbować ponownie po odpowiednim okresie wycofywania.

Listing 5. Dane wyjściowe pokazujące odpowiedź zerową

```
$ python .\petsalon.py
Prompt:Suggest three names for a new pet salon business. The generated
       ➥name ideas should evoke positive emotions and the following key
       ➥features: Professional, friendly, Personalized Service.
Response:
{
  "choices": [
    {
      "finish_reason": "content_filter",
      "index": 0,
      "logprobs": null,
      "text": "",
      "content_filter_results"={...}
    }
  ],
  "created": 1689006467,
  "id": "cmpl-7ao1jIACW1v8mYH879EE1trbT9Ua6",
  "model": "gpt35",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 31,
    "prompt_tokens": 32,
    "total_tokens": 63
  }
}
```

null response

Content filter is the reason the response finished.

Wiele uzupełnień

Możemy chcieć wielu uzupełnień z kilku powodów. Czasami musimy wygenerować wiele opcji komunikatów dla tego samego monitu. Innym razem API jest ograniczone ze względu na przepustowość i możemy chcieć uzyskać więcej z jednego wywołania API, zamiast być ograniczonym przepustowością. API uzupełnień może zwracać wiele odpowiedzi; odbywa się to poprzez ustawienie parametru n na wartość większą niż domyślna 1. Na przykład, możemy dodać ten parametr do wywołania uzupełnienia:

response = client.completions.create(

model="gpt-35-turbo",

prompt=prompt_startphrase,

temperature=0.7,

max_tokens=100,

n=3,

stop=None)

loop through the response choices

for choice in response.choices:

print(choice.text)

Po uruchomieniu zaktualizowanego kodu otrzymujemy odpowiedź pokazaną na listingu 3.6. Wybrane właściwości to tablica, a my mamy trzy elementy, których indeks zaczyna się od podstawy zerowej. Każdy z nich zawiera wygenerowany tekst, którego możemy użyć. W zależności od przypadku użycia, jest to pomocne przy wyborze wielu uzupełnień.

Listing 6. Dane wyjściowe pokazujące wiele odpowiedzi

1. Pet Pampering Palace

2. Pet Grooming Haven

3. Perfect Pet Parlor

1. Pawsitive Pet Spa

2. Fur-Ever Friends Pet Salon

3. Purrfection Pet Care

1. Pampered Paws Professional Pet Care

2. Personalized Pet Pampering

3. Friendly Furrific Pet Care

Innym podobnym, ale bardziej wydajnym parametrem jest parametr best_of. Podobnie jak parametr n, generuje on wiele uzupełnień, umożliwiając wybór najlepszego. Best_of to uzupełnienie z najwyższym prawdopodobieństwem logarytmu na token. Nie możemy przesyłać strumieniowo wyników podczas korzystania z tej opcji. Można ją jednak połączyć z parametrami n, przy czym best_of wymaga wartości większej niż n. Jak pokazano na poniższym listingu, jeśli ustawimy n na 5, otrzymamy pięć uzupełnień, zgodnie z oczekiwaniami; dla zwięzłości nie pokazujemy tutaj wszystkich pięciu uzupełnień, ale należy zauważyć, że to wywołanie wykorzystuje 184 tokeny.

Listing 7. Dane wyjściowe pokazujące wiele odpowiedzi

{

"choices": [

{

…

],

"created": 1689097645,

"id": "cmpl-7bBkLk60mA8R9crAKXqTmTwzx2IEI",

"model": "gpt-35-turbo",

"object": "text_completion",

"usage": {

"completion_tokens": 152,

"prompt_tokens": 32,

"total_tokens": 184

}

}

Jeżeli uruchomimy podobne wywołanie, używając parametru best_of, nie określajmy parametru n:

response = client.completions.create(

model="gpt-35-turbo",

prompt=prompt_startphrase,

temperature=0.7,

max_tokens=100,

best_of=5,

stop=None)

Po uruchomieniu tego kodu otrzymujemy tylko jedno uzupełnienie, jak pokazano na listingu 3.8; jednak używamy podobnej liczby tokenów jak wcześniej (171 w porównaniu do 184). Dzieje się tak, ponieważ usługa generuje pięć uzupełnień po stronie serwera i zwraca najlepsze. Interfejs API wykorzystuje logarytm prawdopodobieństwa dla każdego tokena, aby wybrać najlepszą opcję. Im wyższe logarytm prawdopodobieństwa, tym model jest bardziej pewny swojej prognozy.

Listing 8 Generowanie wyników z najlepszymi z pięciu uzupełnień

"choices": [

{

"finish_reason": "stop",

"index": 0,

"logprobs": null,

"text": "\n\n1. Pawsitively Professional Pet Salon\n

➡2. Friendly Furr Friends Pet Salon\n

➡3. Personalized Pampered Pets Salon",

"content_filter_results"={...}

}

],

"created": 1689098048,

"id": "cmpl-7bBqqpfuoV5nrgHrahuWGVAiM50Aj",

"model": "gpt35",

"object": "text_completion",

"usage": {

"completion_tokens": 139,

"prompt_tokens": 32,

"total_tokens": 171

}

}

Jednym z parametrów wpływających na wiele odpowiedzi jest ustawienie temperatury. Zobaczmy, jak zmienia ono wynik.

Kontrola losowości

Jak omówiono wcześniej, ustawienie temperatury wpływa na losowość generowanego wyniku. Niższa temperatura generuje bardziej powtarzalne i deterministyczne odpowiedzi, podczas gdy wyższa temperatura generuje bardziej innowacyjne odpowiedzi. Zasadniczo nie ma jednego właściwego ustawienia — wszystko sprowadza się do przypadków użycia. W przypadku przedsiębiorstw bardziej kreatywnym wyjściem byłoby zainteresowanie zróżnicowanymi wynikami i tworzeniem tekstów do zastosowań, takich jak generowanie treści marketingowych, opowiadań, wierszy, tekstów piosenek, żartów itp. Są to zagadnienia, które zazwyczaj wymagają kreatywności. Przedsiębiorstwa potrzebują jednak bardziej wiarygodnych i precyzyjnych odpowiedzi w zastosowaniach, takich jak automatyzacja dokumentów do generowania faktur, ofert, generowania kodu itp. Ustawienia te są stosowane dla każdego wywołania API, więc możliwe jest łączenie różnych poziomów temperatury w tym samym przepływie pracy. Jak pokazano w poprzednich przykładach, zalecamy ustawienie temperatury na 0,8 dla kreatywnych odpowiedzi. Natomiast ustawienie 0,2 jest sugerowane dla bardziej przewidywalnych odpowiedzi. Na przykładzie przeanalizujmy, jak te ustawienia zmieniają wyjście i zaobserwujmy różnice między wieloma wywołaniami. Po ustawieniu temperatury na 0,8 otrzymaliśmy następujące odpowiedzi z trzech kolejnych wywołań. Wyjście zmienia się zgodnie z oczekiwaniami, oferując sugestie podobne do tych przedstawionych w tym rozdziale. Należy zauważyć, że nie musimy wykonywać trzech oddzielnych wywołań API. Możemy ustawić parametr n na 3 w jednym wywołaniu by wygenerować wiele odpowiedzi. Oto jak wygląda nasze wywołanie API:

```
response = client.completions.create(

model="gpt-35-turbo",

prompt=prompt_startphrase,

temperature=0.8,

max_tokens=100,

n=3,

stop=None)
```

Poniższa lista przedstawia generację kreatywną dla trzech odpowiedzi.

Listing 9 Uzupełnień z temperaturą równą 0,8

```
{
  "choices": [
    {
      "finish_reason": "content_filter",
      "index": 0,
      "logprobs": null,
      "text": "",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "stop",
      "index": 1,
      "logprobs": null,
      "text": "\n\n1. Pawsitively Professional Pet Styling\n
                ➥2. Fur-Ever Friendly Pet Groomers \n
                ➥3. Tailored TLC Pet Care",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "stop",
      "index": 2,


      "logprobs": null,
      "text": "\n\n1. Pawsitively Professional Pet Salon \n
                ➥2. Friendly Fur-ternity Pet Care \n
                ➥3. Personalized Pup Pampering Place",
      "content_filter_results"={...}
    }
  ],
  "created": 1689123394,
  "id": "cmpl-7bIRe6Ponn8y1198flJFfagq64r2E",
  "model": "gpt35",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 96,
    "prompt_tokens": 32,
    "total_tokens": 128
  }
}
```

First response: get blocked by the content filter

Second of three responses

Final response with very different generated text

Zmieńmy ustawienie, aby było bardziej deterministyczne i uruchommy je ponownie. Zwróć uwagę, że jedyną zmianą w wywołaniu API jest temperatura = 0,2. Dane wyjściowe są przewidywalne i deterministyczne, a tekst wygenerowany dla trzech odpowiedzi jest bardzo podobny.

Listing 10 Uzupełnień z temperaturą równą 0,2

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "\n\n1. Pawsitively Professional Pet Salon\n
                ➥2. Friendly Furr Salon\n
                ➥3. Personalized Pet Pampering",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "stop",
      "index": 1,
      "logprobs": null,
      "text": "\n\n1. Pawsitively Professional Pet Salon\n
                ➥2. Friendly Fur-Ever Pet Salon\n
                ➥3. Personalized Pet Pampering Salon",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "stop",
      "index": 2,
      "logprobs": null,
      "text": "\n\n1. Pampered Paws Pet Salon\n
                ➥2. Friendly Fur Salon\n
                ➥3. Professional Pet Pampering"
    }
  ],
  ...
}
```

One of three responses

Two of three responses; very similar generated text

The final response with very similar generated text

Wartość temperatury wzrasta do 2, ale nie zaleca się jej tak wysokiej, ponieważ model zaczyna mieć halucynacje i tworzy bezsensowny tekst. Jeśli zależy nam na większej kreatywności, zazwyczaj chcemy, aby wynosiła 0,8, a maksymalnie 1,2. Zobaczmy przykład, gdy temperatura zostanie zmieniona na 1,8. W tym przykładzie nie udało nam się nawet uzyskać trzeciej generacji, ponieważ osiągnęliśmy limit tokenów i przerwaliśmy generację.

Listing 11 Uzupełnień z temperaturą 1,8

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "\n\n1. ComfortGroom Pet Furnishing \n2. Pampered TreaBankant
     Carers \n3. Toptech Sunny Haven Promotion.",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "stop",
      "index": 1,
      "logprobs": null,
      "text": "\n\n1: Naturalistov ClearlywowGroomingz
                 ➥Pet Luxusia \n2: VipalMinderers Pet
                 ➥Starencatines grooming \n3: Brisasia
                 ➥Crownsnus Take Care Buddsroshesipalising",
      "content_filter_results"={...}
    },
    {
      "finish_reason": "length",
      "index": 2,
      "logprobs": null,
      "text": "\n\n1. TrustowStar Pet Salon\n
             ➥2. Hartipad TailTagz Grooming & Styles\n
             ➥3. LittleLoft Millonista Cosmania DipSavez
                ➥Hubopolis ShineBright Princessly
                ➥Prosnoiffarianistics Kensoph Cowlosophy
                ➥Expressionala Navixfordti Mundulante Effority
                ➥DivineSponn BordloveDV EnityzBFA Prestageinato
                ➥SuperGold Cloutoilyna Critinarillies
                ➥Prochromomumphance Toud",
                ➥"content_filter_results"={...}
    }
  ],
  ...
}
```

One of three responses with names that aren't very clear

Second and third of three responses, with nonsensical names

Kontrolowanie losowości za pomocą top_p

Alternatywą dla parametru temperatury do zarządzania losowością jest parametr top_p. Ma on taki sam wpływ na generowanie jak parametr temperatury, ale wykorzystuje inną technikę zwaną próbkowaniem jądra. Zasadniczo próbkowanie jądra pozwala na uwzględnienie w generowaniu tylko tokenów z prawdopodobieństwem równym lub mniejszym od wartości top_p. Próbkowanie jądra tworzy teksty poprzez wybieranie słów z małej grupy najbardziej prawdopodobnych słów o najwyższym skumulowanym prawdopodobieństwie. Wartość top_p decyduje o tym, jak mała jest ta grupa, na podstawie całkowitej szansy na pojawienie się w niej słów. Rozmiar grupy może się zmieniać w zależności od szansy na pojawienie się kolejnego słowa. Próbkowanie jądra może pomóc uniknąć powtórzeń i generować bardziej zróżnicowane i przejrzyste teksty niż inne metody. Na przykład, jeśli wartość top_p jest ustawiona na 0,9, do generowania tekstu zostaną pobrane tylko tokeny, które stanowią 90% rozkładu prawdopodobieństwa. Pozwala nam to uniknąć ostatnich 10%, które często są dość losowe i zróżnicowane i kończą się jako bezsensowne halucynacje. Niższa wartość top_p sprawia, że model jest bardziej spójny i mniej kreatywny, ponieważ wybiera mniej tokenów do generowania. Z drugiej strony, wyższa wartość sprawia, że generowanie jest bardziej kreatywne i zróżnicowane, ponieważ ma większy zestaw tokenów do obsługi. Wyższa wartość sprawia również, że jest bardziej

podatne na błędy i losowość. Dokładna wartość top_p zależy od przypadku użycia; w większości przypadków idealna wartość dla top_p mieści się w zakresie od 0,7 do 0,95. Powinniśmy zmienić albo atrybut temperatury, albo top_p, ale nie oba. Tabela 3.5 przedstawia relację między nimi.

Temperatura: top_p: Efekt

Niska: Niska: Generuje przewidywalny tekst, który ściśle odpowiada powszechnym wzorcom językowym

Niska: Wysoka: Generuje przewidywalny tekst, ale z okazjonalnymi mniej popularnymi słowami lub frazami

Wysoka: Niska: Generuje tekst, który jest często spójny, ale z kreatywnym i nieoczekiwanym użyciem słów

Wysoka: Wysoka: Generuje bardzo zróżnicowany i nieprzewidywalny tekst z różnymi doborami słów i pomysłami; ma bardzo kreatywny i zróżnicowany wynik, ale może zawierać wiele błędów

Przyjrzyjmy się niektórym zaawansowanym opcjom API dla konkretnych scenariuszy.

Zaawansowane opcje API uzupełniania

Teraz, gdy omówiliśmy podstawowe konstrukcje API uzupełniania i zrozumieliśmy, jak działają, musimy rozważyć bardziej zaawansowane aspekty API uzupełniania. Wiele z nich może nie wydawać się tak skomplikowanych, ale dodają one znacznie więcej obowiązków do architektury systemu, komplikując ogólną implementację.

Przesyłanie strumieniowe uzupełnień

API uzupełniania umożliwia przesyłanie strumieniowe odpowiedzi, oferując natychmiastowy dostęp do informacji, gdy tylko będą gotowe, zamiast oczekiwania na pełną odpowiedź. Dla przedsiębiorstw przesyłanie strumieniowe może być istotne w niektórych przypadkach, gdy kluczowe jest generowanie treści w czasie rzeczywistym z mniejszym opóźnieniem. Ta funkcja może poprawić doświadczenia użytkowników poprzez szybkie przetwarzanie odpowiedzi przychodzących. Aby włączyć przesyłanie strumieniowe z punktu widzenia API, zmień parametr strumienia na wartość true. Domyślnie ten opcjonalny parametr jest ustawiony na false. Przesyłanie strumieniowe wykorzystuje zdarzenia wysyłane przez serwer (SSE), które wymagają implementacji po stronie klienta. SSE to standardowy protokół umożliwiający serwerom kontynuowanie przesyłania danych do klientów po nawiązaniu początkowego połączenia. To długoterminowe, jednokierunkowe połączenie między serwerem a klientem. SSE oferuje takie zalety, jak niskie opóźnienie, mniejsze zużycie przepustowości i nieskomplikowana konfiguracja. Listing 3.12 pokazuje, jak nasz przykład można dostosować do obsługi strumieniowania. Chociaż modyfikacja API jest prosta, opis i żądane wielokrotne generacje zostały zmodyfikowane (za pomocą właściwości n). Pozwala to na sztuczne generowanie większej ilości tekstu, ułatwiając obserwację generowania strumieniowego.

Listing 12 Zakończenie przesyłania strumieniowego

```
import os
import sys
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2024-05-01-preview",
    api_key=os.getenv("AOAI_KEY"))

prompt_startphrase = "Suggest three names and a tagline
➥which is at least 3 sentences for a new pet salon business.
➥The generated name ideas should evoke positive emotions and the
➥followingkey features: Professional, friendly, Personalized Service."

for response in client.completions.create(
    model="gpt-35-turbo",
    prompt=prompt_startphrase,
    temperature=0.8,
    max_tokens=500,
    stream=True,
    stop=None):
    for choice in response.choices:
        sys.stdout.write(str(choice.text)+"\n")
        sys.stdout.flush()
```

Tweaked the prompt slightly to add descriptions

We need to handle the streaming response on the client side.

Enables streaming

We need to loop through the array and handle multiple generations.

Zarządzając wywołaniem strumieniowym, należy zwrócić szczególną uwagę na właściwość finish_reason. Podczas strumieniowania wiadomości, każda pojawia się jako standardowe uzupełnienie, a tekst reprezentuje nowo wygenerowany token. W takich przypadkach właściwość finish_reason pozostaje null. Jednak ostateczna wiadomość różni się; jej właściwość finish_reason może mieć wartość stop lub length, w zależności od tego, co ją wywołało.

Listing 13 Powód zakończenia przesyłania strumieniowego

...

{

"finish_reason": null,

"index": 0,

"logprobs": null,

"text": " Pet"

}

{

"finish_reason": null,

"index": 0,

"logprobs": null,

"text": " Pam"

}

{

"finish_reason": null,

"index": 0,

"logprobs": null,

"text": "pering"

}

{

"finish_reason": "stop",

"index": 0,

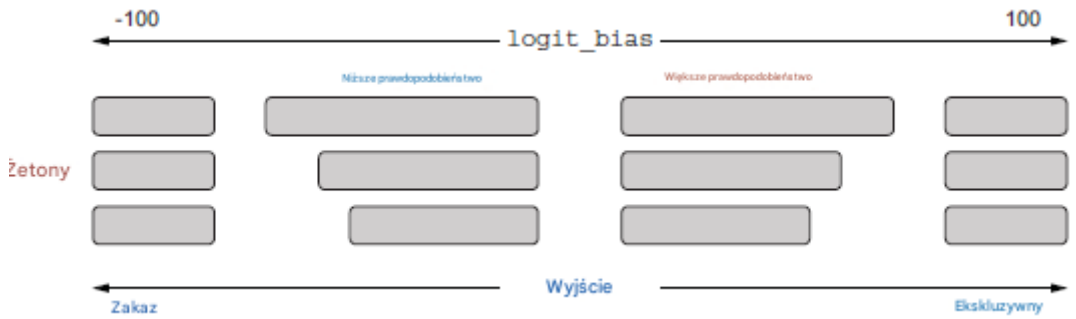
"logprobs": null,

"text": ""

}

Wpływanie na prawdopodobieństwo tokenów: logit_bias

Parametr logit_bias to jeden ze sposobów wpływania na ukończenie wyników. W API parametr ten pozwala nam manipulować prawdopodobieństwem określonych tokenów, które mogą być słowami lub frazami, generowanymi przez model w odpowiedziach. Nazywa się to logit_bias, ponieważ bezpośrednio wpływa na logarytm szans, czyli logity, które model oblicza dla każdego potencjalnego tokena podczas procesu generowania. Wartości odchyleń są dodawane do tych logarytmów szans przed przekształceniem ich w prawdopodobieństwa, zmieniając ostateczny rozkład tokenów, z którego model może wybierać. Znaczenie tej funkcji polega na jej zdolności do sterowania wynikami modelu. Załóżmy, że tworzymy chatbota i chcemy, aby unikał określonych słów lub fraz. Możemy użyć logit_bias, aby zmniejszyć prawdopodobieństwo wyboru tych tokenów przez model. Z kolei, jeśli chcemy, aby model faworyzował określone słowa lub frazy, możemy użyć logit_bias, aby zwiększyć ich prawdopodobieństwo. Zakres tego parametru wynosi od –100 do 100 i działa on na tokenach dla słowa. Ustawienie tokenu na –100 skutecznie wyklucza go z generacji, podczas gdy ustawienie na 100 powoduje jego wykluczenie. Aby użyć logit_bias, dostarczamy słownik, w którym klucze są tokenami, a wartości są odchyleniami, które należy zastosować do tych tokenów. Aby uzyskać token, używamy biblioteki tiktoken. Po uzyskaniu odpowiedniego tokena można przypisać dodatnie odchylenie, aby zwiększyć prawdopodobieństwo jego pojawienia się, lub ujemne odchylenie, aby zmniejszyć prawdopodobieństwo, jak pokazano na rysunku 2.



Bloki pokazują stopień prawdopodobieństwa, że różne tokeny mogą mieć różne prawdopodobieństwa wykluczenia lub wykluczenia generacji. Mniejsze zmiany wartości tokenów zwiększają lub zmniejszają prawdopodobieństwo tych tokenów w wygenerowanym wyjściu. Posłużmy się przykładem, aby zobaczyć, jak to działa. W nazwie naszego salonu dla zwierząt nie chcemy używać słów „mruczenie", „mruczy" ani „miau". Najpierw musimy utworzyć tokeny dla tych słów. Chcemy również dodać słowa z poprzedzającą je spacją i zapisać je wielką literą jako spacje. Wielkie litery to różne tokeny. Zatem „Miau" i „Miau" (ze spacją) oraz „miau" (ponownie ze spacją) mogą brzmieć dla nas tak samo, ale jeśli chodzi o tokeny, wszystkie te słowa są różne. Wynik pokazuje tokeny dla odpowiadającego słowa:

'Purr Purrs Meow Purr purr purrs meow:[30026, 81, 9330,

➥3808, 42114, 9330, 81, 1308, 81, 1308, 3808, 502, 322]'

Mając już tokeny, możemy dodać je do wywołania funkcji uzupełniania. Zwróć uwagę, że każdemu tokenowi przypisujemy odchylenie –100, co pozwala modelowi unikać tych słów.

Listing 14 Implementacja logit_bias

```
import os
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2024-05-01-preview",
    api_key=os.getenv("AOAI_KEY"))

GPT_MODEL = "gpt-35-turbo"

prompt_startphrase = "Suggest three names for a new pet salon
➥business. The generated name ideas should evoke positive
➥emotions and the following key features: Professional,
➥friendly, Personalized Service."

response = client.completions(
  model=GPT_MODEL,
```

```
    prompt=prompt_startphrase,
    temperature=0.8,
    max_tokens=100,
    logit_bias={
        30026:-100,
        81:-100,
        9330:-100,
        808:-100,
        42114:-100,
        1308:-100,
        3808:-100,
        502:-100,
        322:-100
    }
)

responsetext =response.choices[0].text

print("Prompt:" + prompt_startphrase + "\nResponse:" + responsetext)
```

Dictionary containing the tokens and the corresponding bias values to steer the model on these specific tokens

Nie chcemy pomijać żadnych słów podczas uruchamiania tego kodu.

Listing 15 Wynik generacji logit_bias

{

"choices": [

{

"finish_reason": "stop",

"index": 0,

"logprobs": null,

"text": "\n\n1. Paw Prints Pet Pampering\n2. Furry Friends Fussing\n3.

Posh Pet Pooches"

}

],

...

}

Możemy zrobić odwrotnie i również nadać tokenom dodatnie nastawienie. Załóżmy, że chcemy nadać modelowi zbyt duży nacisk i skierować go w stronę słowa „Furry”. Możemy skorzystać z biblioteki tiktoken, którą widzieliśmy wcześniej, i znaleźć tokeny dla słowa „Furry” w postaci [37, 16682]. Możemy zaktualizować poprzednie wywołanie API, dodając to i, w tym przypadku, dodatnie nastawienie na poziomie 5.

Listing 16 : logit_bias: Pozytywna implementacja

```
GPT_MODEL = "gpt-35-turbo"

response = client.completions.create(

model=GPT_MODEL,

prompt=prompt_startphrase,

temperature=0.8,

max_tokens=100,

logit_bias={

30026:-100,

81:-100,

9330:-100,

808:-100,

42114:-100,

1308:-100,

3808:-100,

502:-100,

322:-100,

37:5,

16682:5

}

)
```

Po uruchomieniu tego kodu otrzymujemy wynik pokazany na poniższym listingu. Jak widać, w naszym generowaniu znacznie większy nacisk kładziemy na „Furry”. Uzupełnianie trwa również dłużej, ponieważ model konkuruje z tendencyjnością podczas generowania niektórych tokenów.

Listing 17 : Wyjście logit_bias: Pozytywna implementacja


```
{

"choices": [

{

"finish_reason": "stop",

"index": 0,

"logprobs": null,

"text": "\n\n1.FurryFrendz Pet Salon\n2.FurryFurFection Pet

Pampering\n3.FurryFurFam Pet Spa"
```

}

],

…

}

Funkcji logit_bias należy używać ostrożnie; jest to potężne narzędzie do sterowania wynikami modelu. Jednak nadmierne lub niewłaściwe użycie może prowadzić do bezsensownych, nadmiernie powtarzalnych lub stronniczych wyników w nieoczekiwany sposób.

Kary za obecność i częstotliwość

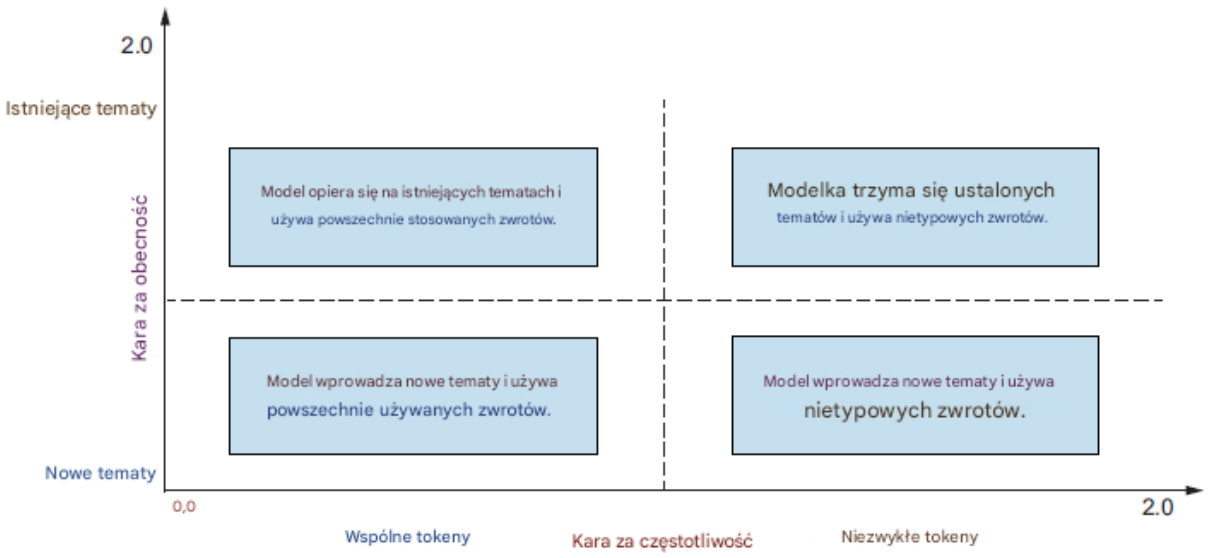
W API mamy dwa dodatkowe parametry, zwane karami za obecność i częstotliwości, które pomagają sterować wynikami modelu języka poprzez kontrolowanie powtarzalności generacji. Te dwa parametry wpływają na prawdopodobieństwo ponownego pojawienia się słów (technicznie sekwencji tokenów) w uzupełnieniu. Wyższa kara za obecność zachęca model do skupienia się na monicie i unikania używania tokenów, które już się tam znajdują. Natomiast wyższa kara za częstotliwość zniechęca model do powtarzania się. Przyjrzyjmy się obu tym parametrom nieco bardziej szczegółowo.

PARAMETR KARY ZA OBECNOŚĆ

Parametr kary za obecność wpływa na częstotliwość pojawiania się tego samego tokenu w wynikach. Można to osiągnąć, stosując karę za obecność jako wartość odejmowaną od prawdopodobieństwa tokena za każdym razem, gdy jest generowany. Oznacza to, że im częściej token jest używany, tym mniejsze jest prawdopodobieństwo jego ponownego użycia. Pomaga to modelowi wykorzystywać bardziej zróżnicowane tokeny podczas generowania i eksplorować nowe tematy. Wartość tego parametru może wynosić od 0 do 2. Wartość domyślna to 0, co oznacza, że model nie przejmuje się powtarzaniem tokena. Wysoka kara za obecność (1,0) zmniejsza prawdopodobieństwo ponownego użycia tego samego tokena przez model, a wyższa wartość powoduje, że model wprowadza nowe tematy w wynikach. Niska kara za obecność (0) sprawia, że model trzyma się istniejących tematów w tekście. Za każdym razem, gdy generowany jest token, wartość parametru jest odejmowana od logarytmu prawdopodobieństwa tego tokena. Możemy poprawić jakość generowania, zapobiegając wielokrotnemu powtarzaniu tego samego tekstu, pomagając kontrolować przepływ i czyniąc dane wyjściowe bardziej angażującymi. Przyjrzyjmy się teraz parametrowi kary za częstotliwość.

PARAMETR KARY CZĘSTOTLIWOŚCIOWEJ

Ten parametr kontroluje, w jakim stopniu model unika powtarzania się w wynikach. Im wyższa kara częstotliwości (1,0), tym częściej model próbuje używać różnych słów i fraz, co skutkuje bardziej zróżnicowanym generowaniem. Im niższa kara częstotliwości (0,0), tym częściej model może powtarzać te same słowa i frazy, a wynik jest bardziej przewidywalny. Różni się to od kary za obecność, która zachęca model do używania nowych słów i fraz. Kara częstotliwości zwiększa logarytm prawdopodobieństwa wystąpienia tokena za każdym razem, gdy pojawi się on w wynikach. Najlepsze wartości obu parametrów zależą od tego, co chcesz osiągnąć za pomocą wyników. Zazwyczaj najlepiej jest wybrać wartości z zakresu od 0,1 do 1,0, co zauważalnie wpływa na wynik. Aby uzyskać silniejszy efekt, możesz zwiększyć wartości do 2,0, ale może to obniżyć jakość wyników. Należy pamiętać, że dostrojenie tych parametrów wymaga prób i błędów, aby uzyskać pożądane rezultaty, ponieważ na wynik modelu wpływa również wiele innych czynników, w tym podany monit i inne parametry dostrajania. Rysunek 3.3. przedstawia korelację parametrów kary za obecność i częstotliwość.



Logowanie prawdopodobieństw

Gdy model LLM generuje token, przypisuje prawdopodobieństwo kolejnemu rozważanemu tokenowi i wykorzystuje różne techniki, aby wybrać token użyty w uzupełnieniu spośród tych opcji. Właściwość logprobs interfejsu API uzupełniania ujawnia logarytm naturalny dla tych prawdopodobieństw na każdym kroku. Jest to liczba całkowita (maksymalna wartość 5), która pokazuje alternatywne tokeny brane pod uwagę dla każdego tokenu uwzględnionego w uzupełnieniu. Jeśli ta wartość jest ustawiona na 3, interfejs API zwróci listę trzech najbardziej prawdopodobnych tokenów dla każdego wybranego tokenu w generacji. Należy pamiętać, że interfejs API zawsze zwraca logprobs próbkowanego tokenu, więc w odpowiedzi możemy otrzymać logprobs + 1 element w tablicy. Zasadniczo używamy tego podejścia, aby ułatwić debugowanie i ulepszyć monity. Jeśli model nie generuje tekstu, który nam odpowiada, możemy to wykorzystać, aby zobaczyć, jakie inne słowa (technicznie tokeny) model rozważył. Pozwala nam to dostroić inne ustawienia, aby sterować modelem. I odwrotnie, możemy użyć tego samego do kontrolowania losowości w generowaniu modelu i zwiększenia determinizmu wyników. Wreszcie, możemy również wykorzystać to do zrozumienia, jak pewny jest model. Jeśli prawdopodobieństwa są takie same dla kilku różnych słów, oznacza to, że model nie jest pewien, które słowo będzie następne. Załóżmy, że chcemy uzyskać imię białego psa; możemy wywołać API uzupełniania. W tym przykładzie otrzymujemy imię Cotton, co nie jest złe:

```
response = client.completions.create(

model=GPT_MODEL,

prompt="Suggest a one word name for a white miniature poodle.",

temperature=0.8,max_tokens=100,

stop=None)
```

Jeśli chcemy zobaczyć, jakie inne tokeny były brane pod uwagę dla tej nazwy, możemy dodać właściwości logprobs:

```
response = client.completions.create(

model=GPT_MODEL,

prompt="Suggest a one word name for a white miniature poodle.",
```

temperature=0.8,max_tokens=100,

logprobs=3,

stop=None)

Jak widać na wynikach ukończenia w poniższym zestawieniu, model uwzględniał następujące tokeny: Casper, Coco i Snow.

Listing 18 : Prawdopodobieństwa wyjściowe logarytmu

{

"id": "cmpl-7giPQGlKc6c7BaWmHgOLyZqabIruw",

"object": "text_completion",

"created": 1690414840,

"model": "gpt-35-turbo",

"choices": [

{

"text": "\n\nCotton",

"index": 0,

"finish_reason": "stop",

"logprobs": {

"tokens": [

"\n",

"\n",

"C",

"otton"

],

"token_logprobs": [

-0.0008873215,

-4.361666e-06,

-1.026479,

-0.56846446

],

"top_logprobs": [

{
"\n": -0.0008873215,
"\n\n": -7.660001,
" Angel": -10.180796
},
{
"\n": -4.361666e-06,
"\n\n": -12.970553,
"<|endoftext|>": -15.136529
},
{
"C": -1.026479,
"P": -2.255978,
"Snow": -2.1068947
},
{
"asper": -2.001854,
"oco": -1.957575,
"otton": -0.56846446
}
],
"text_offset": [
54,
55,
56,
57
]
}
}
],
"usage": {

"completion_tokens": 4,

"prompt_tokens": 12,

"total_tokens": 16

}

}

Przypominamy, że powinniśmy korzystać z tej właściwości rozważnie i tylko wtedy, gdy jest to konieczne. Nie tylko zwiększa ona liczbę generowanych tokenów, a tym samym koszt wywołania API, ale także zabiera czas i wydłuża wywołanie API, zwiększając tym samym ogólne opóźnienie. Teraz, gdy rozumiemy API uzupełniania dla generowania tekstu, zobaczmy, jak możemy wykorzystać API uzupełniania czatu.

API uzupełniania czatu

API uzupełniania czatu zostało zaprojektowane, aby ułatwić interaktywne i dynamiczne rozmowy. Jest to ewolucja API uzupełniania, zapewniająca użytkownikom bardziej konwersacyjne i angażujące doświadczenie. Dzięki temu API programiści mogą tworzyć aplikacje, które prowadzą dialog z użytkownikami, co czyni je idealnym rozwiązaniem do tworzenia chatbotów, asystentów pisania i innych. Kluczowe korzyści, jakie oferuje API uzupełniania czatu w porównaniu z API uzupełniania, to: „R Enhanced interactivityˇX – API uzupełniania czatu umożliwia bardziej dynamiczną i interaktywną rozmowę z użytkownikiem, dzięki czemu doświadczenie użytkownika jest bardziej angażujące i naturalne. „R Contextual understandingˇX – API utrzymuje kontekst rozmowy, zapewniając trafność i spójność odpowiedzi. „R Multiturn conversationˇX – W przeciwieństwie do API uzupełniania, które jest bardziej odpowiednie do zadań jednoobrotowych, API konwersacji wieloobrotowej pozwala programistom symulować rozmowy z wieloma giełdami. „R Cost-effectiveˇX – API uzupełniania czatu wykorzystuje modele GPT-3.5 Turbo lub GPT-4, które działają z podobną wydajnością jak text-davinci-003, ale przy 10% cenie za token, co czyni je bardziej ekonomicznym wyborem dla programistów. Ogólnie rzecz biorąc, korzystanie z API uzupełniania czatu jest podobne do korzystania z API uzupełniania. API przyjmuje serię wiadomości jako dane wejściowe, stanowiąc podstawę interakcji z modelem. Kolejność wiadomości jest ważna, ponieważ określa Interakcja krok po kroku. Każda wiadomość ma dwie właściwości: rolę i treść. Parametr roli ma trzy opcje: system, użytkownik lub asystent. Treść zawiera tekst wiadomości powiązany z daną rolą. Tabela 3.6 przedstawia szczegóły każdej roli i jej przeznaczenie.

Parametr roli: Opis

system: Rola systemowa jest zazwyczaj używana do określania zachowania asystenta i dostarczania modelowi instrukcji wysokiego poziomu, które kierują zachowaniem podczas całej rozmowy. W tym miejscu możemy opisać osobowość asystenta i wskazać mu, na co powinien, a czego nie powinien odpowiadać, a także jak formatować odpowiedzi. Chociaż nie ma limitu tokenów, jest on uwzględniany przy każdym wywołaniu API i stanowi część ogólnego limitu tokenów.

użytkownik: Reprezentuje dane wejściowe użytkownika w rozmowie; te wiadomości zawierają instrukcje lub zapytania użytkownika, na które odpowiada asystent.

asystent: Reprezentuje wcześniejsze wiadomości asystenta w rozmowie. Można to postrzegać jako pamięć bieżącą, która wspomaga model i zapewnia kontekst rozmowy w miarę jej przebiegu, krok po kroku.

Listing 19 przedstawia API dokończenia czatu. Jak wspomnieliśmy wcześniej, kolejność wiadomości w tablicy ma znaczenie, ponieważ odzwierciedla przebieg rozmowy. Zazwyczaj rozmowa rozpoczyna się od wiadomości systemowej, która określa zachowanie asystenta, a następnie następują naprzemienne wiadomości od użytkownika i asystenta w miarę postępu rozmowy. Odpowiedzi asystenta są generowane na podstawie historii rozmowy.

Listing 19 : API do uzupełniania czatów

```
import os
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2024-05-01-preview",
    api_key=os.getenv("AOAI_KEY"))

GPT_MODEL = "gpt-35-turbo"

response = client.chat.completions.create(
    model=GPT_MODEL,
    messages = [
      ➥{"role":"system","content":"You are an AI assistant
                ➥that helps people find information."},
      ➥{"role":"user","content":"Hello world"},
      ➥{"role":"assistant","content":"Hello! How can I assist you today?"},
       {"role":"user","content":"I want to know more
                ➥about pets and why dogs are good for humans?"}],
    temperature=0.8,
    max_tokens=800,
    user="amit",
    top_p=0.95,
    frequency_penalty=0,
    presence_penalty=0,
    stop=None
)

print(response.choices[0].message.content)
```

Chat complete API call

Different models needed (Turbo) compared to completion API

List of messages that form the heart of the API

These parameters are the same as for the completion API.

Musimy zaktualizować parametr silnika, aby użyć jednego z modeli zgodnych z czatem. Jak pokazano wcześniej w tym rozdziale, nie wszystkie modele obsługują styl czatu i musimy wybrać modele z funkcją chat_completion (GPT-3.5 Turbo, GPT-4, GPT-4 Turbo). Wszystkie pozostałe parametry są takie same jak w przypadku API uzupełniania, które omówiliśmy wcześniej w tym rozdziale i nie będziemy się do nich ponownie odwoływać.

UWAGA: Następujące parametry są niedostępne w nowych modelach GPT-35 Turbo i GPT-4: logprobs, best_of i echo. Próba ustawienia któregokolwiek z tych parametrów spowoduje zgłoszenie wyjątku. Wynik z poprzedniego przykładu przedstawiono na poniższym listingu. Użytkownik rozpoczął od „Witaj, świecie!”, a system odpowiedział, pytając, jak pomóc nam z komunikatem asystenta. Pytanie o szczegóły dotyczące psa to kolejna tura dialogowa.

Listing 20: Dane wyjściowe interfejsu API dotyczącego uzupełniania czatu

{

"id": "chatcmpl-7c9HOaRHmaut94oZacjdUqxOL96H3",

"object": "chat.completion",

"created": 1689326490,

"model": "gpt-35-turbo",

"choices": [

{

"index": 0,

"finish_reason": "stop",

"message": {

"role": "assistant",

"content": "Dogs are often considered man's best

➥friend for good reason. They provide

➥companionship, protection, and loyalty to

➥their owners. Here are some reasons why dogs are

➥good for humans:\n\n1. They reduce stress: Dogs

➥have been shown to reduce stress and anxiety in

➥their owners. Petting a dog can release endorphins,

➥which are feel-good hormones that can help to

➥reduce stress.\n\n2. They provide companionship..."

}

}

],

"usage": {

"completion_tokens": 238,

"prompt_tokens": 60,

"total_tokens": 298

}

}

Rola systemowa

Rola systemowa (niektórzy nazywają ją również komunikatem systemowym) znajduje się na początku tablicy komunikatów. Komunikat ten zawiera wstępne instrukcje dla modelu, a rola systemowa może zawierać różne informacje, w tym:

* Krótki opis asystenta

* Cechy osobowości asystenta

* Zasady i instrukcje, których asystent ma przestrzegać

* Dodatkowe informacje potrzebne dla modelu (np. istotne pytania z FAQ)

Dostosowujemy rolę systemową i dołączamy podstawowe instrukcje dla danego przypadku użycia. Z perspektywy API, mimo że rola systemowa jest opcjonalna, zdecydowanie zaleca się jej celowe wprowadzenie, aby uzyskać najlepsze rezultaty. Na przykład, jeśli rozszerzymy poprzedni przykład czatu dla zwierząt i salonów weterynaryjnych, możemy poinstruować model, aby odpowiadał tylko w formie rymów.

Listing 21: Przykład komunikatu systemu ukończenia czatu

```
GPT_MODEL = "gpt-35-turbo"

response = client.chat.completions.create(
  engine=GPT_MODEL,
  messages = [
    {"role": "system", "content": "You are a helpful AI
              ➥assistant that provides truthful information.
              ➥You answer all questions in rhyme."},
    {"role": "user", "content": "Hi"},
    {"role": "assistant", "content": "Hello there, how can
              ➥I assist?\nAsk me a question, don't resist!"},
    {"role": "user", "content": "who are the founders of Microsoft?"},
    {"role": "assistant", "content": "Bill Gates and Paul Allen,
              ➥it's true,\nAre the founders of Microsoft,
              ➥through and through."},
    {"role": "user", "content": "what is a good name for a pet salon?"},
    {"role": "assistant","content": "For a pet salon [
              ➥that's simply divine,\nHere's a name that's [
              ➥sure to shine:\n\"Paws and Pamper\" is what I
              ➥propose,\nA name that's catchy and easy to compose."}],
  temperature=0.7,
  max_tokens=800,
  top_p=0.95,
  frequency_penalty=0,
  presence_penalty=0,
  stop=None)
```

Instructs to answer in rhyme

W tym przykładzie możemy prowadzić konwersację zgodnie z oczekiwaniami, która może zmieniać tematy na zmianę, ale wszystkie odpowiedzi się rymują. Kiedy chcemy przekazać modelowi dodatkowe dane jako kontekst dla konwersacji, nazywa się to uziemieniem danych. Jeśli ilość danych jest niewielka, może to być częścią roli systemu, jak pokazano na poniższym listingu. Jeśli jednak ilość danych jest duża, powinniśmy użyć osadzenia i pobrać najistotniejsze informacje za pomocą wyszukiwania semantycznego (np. Azure Cognitive Search).

Listing 22: Przykład komunikatu systemu uziemienia

{"role": "system", "content": Assistant to inteligentny chatbot zaprojektowany, aby pomagać użytkownikom w udzielaniu odpowiedzi na pytania techniczne dotyczące usługi Azure OpenAI. Odpowiadaj na pytania wyłącznie w poniższym kontekście. Nie wymyślaj odpowiedzi. Jeśli nie jesteś pewien odpowiedzi, powiedz „Nie wiem”.

Kontekst:

- Usługa Azure OpenAI zapewnia dostęp przez interfejs API REST do zaawansowanych modeli językowych OpenAI, w tym serii modeli GPT-3, Codex i Embeddings.

- Usługa Azure OpenAI oferuje klientom zaawansowaną sztuczną inteligencję językową z modelami OpenAI GPT-3, Codex i DALL-E, zapewniając bezpieczeństwo i funkcjonalność korporacyjną platformy Azure.

..."

},

{"role": "user", "content": "Co to jest usługa Azure OpenAI?"}

Powód zakończenia

Każda odpowiedź API zakończenia czatu ma powód zakończenia zakodowany w polu finish_reason. Śledzenie jest w tym przypadku ważne, ponieważ pomaga nam zrozumieć, dlaczego API zwróciło daną odpowiedź. Może to być przydatne do debugowania i ulepszania aplikacji. Na przykład, jeśli otrzymasz niekompletną odpowiedź z powodu powodu zakończenia długości, możesz chcieć dostosować parametr max_tokens, aby generować więcej kompletnych odpowiedzi. Możliwe wartości parametru finish_reason to:

* stop — API zakończyło generowanie i zwróciło albo pełną wiadomość, albo wiadomość przerwaną przez jedną z sekwencji zatrzymania podanych za pomocą parametru stop.

* length — API zatrzymało dane wyjściowe modelu z powodu parametru max_tokens lub limitu tokenów.

* function_call — Model zdecydował się wywołać funkcję.

* content_filter — Część uzupełnień została odfiltrowana z powodu szkodliwej zawartości.

API zakończenia czatu dla scenariuszy bez czatu

Uzupełnianie czatu OpenAI może być używane w scenariuszach bez czatu. Interfejs API jest dość podobny i został zaprojektowany jako elastyczne narzędzie, które można dostosować do różnych przypadków użycia, nie tylko do rozmów. W większości przypadków zalecana ścieżka wykorzystuje interfejs API uzupełniania czatu tak, jakby był interfejsem API uzupełniania. Głównym powodem jest to, że nowsze modele (Chat 3.5-Turbo i GPT-4) są znacznie wydajniejsze, tańsze i bardziej wydajne niż wcześniejsze modele. Przypadki użycia uzupełniania, które widzieliśmy, takie jak analiza i generowanie tekstu oraz odpowiadanie na pytania z bazy wiedzy, nadal będą działać z interfejsem API uzupełniania czatu. Implementacja interfejsu API uzupełniania czatu w scenariuszach innych niż czat zazwyczaj polega na ustrukturyzowaniu rozmowy za pomocą serii wiadomości i komunikatu systemowego, który określa zachowanie asystenta. Na przykład, jak pokazano na poniższym listingu, komunikat systemowy określa rolę asystenta, a komunikat użytkownika określa zadanie.

Listing 23: Przykład uzupełniania czatu jako interfejsu API

```
GPT_MODEL = "gpt-35-turbo"

response = client.chat.completions.create(

model=GPT_MODEL,

messages=[
```

```
{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": "Translate the following

➥English text to Spanish: 'Hello, how are you?'"}

]

)
```

Możemy również użyć serii komunikatów użytkownika, aby zapewnić szerszy kontekst lub wykonać bardziej złożone zadania, jak pokazano na poniższym listingu. W tym przykładzie pierwszy komunikat użytkownika konfiguruje zadanie, a drugi zawiera bardziej szczegółowe informacje. Asystent generuje odpowiedź, która próbuje ukończyć zadanie w komunikatach użytkownika.

Listing 24 : Ukończenie czatu jako przykładu API uzupełniania

```
GPT_MODEL = "gpt-35-turbo"

response = client.chat.completions.create(

model=GPT_MODEL,

messages=[

{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": "I need to write a Python function."},

{"role": "user", "content": "This function should take two

➥numbers as input and return their sum."}

]

)
```

Zarządzanie konwersacją

Nasze przykłady są kontynuowane, ale konwersacja osiągnie limit tokenów modelu. Z każdą turą konwersacji (tj. zadanym pytaniem i otrzymaną odpowiedzią) lista wiadomości rośnie. Dla przypomnienia, limit tokenów dla GPT-35 Turbo wynosi 4 tys. tokenów, a dla GPT-4 i GPT-4 32 tys. to odpowiednio 8 tys. i 32 tys. tokenów; obejmuje to całkowitą liczbę tokenów z listy wysłanych wiadomości i odpowiedzi modelu. Otrzymujemy wyjątek, jeśli całkowita liczba przekroczy odpowiedni limit modelu. Żadna gotowa opcja nie jest w stanie śledzić tej liczby tokenów i upewnić się, że mieści się ona w limicie. W ramach projektowania aplikacji korporacyjnej musimy śledzić liczbę tokenów i wysyłać tylko te komunikaty, które mieszczą się w limicie. Wiele przedsiębiorstw jest w trakcie wdrażania wersji korporacyjnej ChatGPT z wykorzystaniem interfejsu API czatu. Oto kilka najlepszych praktyk, które mogą pomóc przedsiębiorstwom w zarządzaniu tymi konwersacjami. Pamiętaj, że najlepszym sposobem na uzyskanie pożądanego wyniku jest iteracyjne testowanie i dopracowywanie instrukcji:

* Ustalanie zachowania za pomocą komunikatu systemowego — komunikat systemowy należy wykorzystać na początku rozmowy, aby pokierować zachowaniem modelu i umożliwić przedsiębiorstwom dostosowanie go do swojej marki lub własności intelektualnej.

* Udzielanie jednoznacznych instrukcji — jeśli model nie generuje pożądanego wyniku, należy sprecyzować instrukcje. Pomyśl o tym tak, jakbyś mówił maluchowi, czego nie powinien robić.

Zarządzanie konwersacją

Nasze przykłady są kontynuowane, ale konwersacja osiągnie limit tokenów modelu. Z każdą turą konwersacji (tj. zadanym pytaniem i otrzymaną odpowiedzią) lista wiadomości rośnie. Dla przypomnienia, limit tokenów dla GPT-35 Turbo wynosi 4 tys. tokenów, a dla GPT-4 i GPT-4 32 tys. to odpowiednio 8 tys. i 32 tys. tokenów; obejmuje to całkowitą liczbę tokenów z listy wysłanych wiadomości i odpowiedzi modelu. Otrzymujemy wyjątek, jeśli całkowita liczba przekroczy odpowiedni limit modelu. Żadna gotowa opcja nie jest w stanie śledzić tej liczby tokenów i upewnić się, że mieści się ona w limicie. W ramach projektowania aplikacji korporacyjnej musimy śledzić liczbę tokenów i wysyłać tylko te komunikaty, które mieszczą się w limicie. Wiele przedsiębiorstw jest w trakcie wdrażania wersji korporacyjnej ChatGPT z wykorzystaniem interfejsu API czatu. Oto kilka najlepszych praktyk, które mogą pomóc przedsiębiorstwom w zarządzaniu tymi konwersacjami. Pamiętaj, że najlepszym sposobem na uzyskanie pożądanego wyniku jest iteracyjne testowanie i dopracowywanie instrukcji:

* Ustawianie zachowania za pomocą komunikatu systemowego — komunikat systemowy należy wykorzystać na początku konwersacji, aby pokierować zachowaniem modelu i umożliwić przedsiębiorstwom dostosowanie go do swojej marki lub własności intelektualnej.

* Udzielanie jednoznacznych instrukcji — jeśli model nie generuje pożądanego wyniku, instrukcje powinny być bardziej precyzyjne. Pomyśl o tym tak, jakbyś mówił małemu dziecku, czego nie powinno robić.

* Rozbijanie złożonych zadań na części — jeśli masz złożone zadanie, podziel je na kilka prostszych i wyślij je jako oddzielne wiadomości dla użytkownika. Często trzeba je pokazać, a nie wyjaśniać. Nazywa się to Łańcuchem Myśli (CoT) i zostanie omówione bardziej szczegółowo w rozdziale 6.

* Eksperymentowanie — możesz swobodnie eksperymentować z parametrami, aby uzyskać pożądany wynik. Wyższa wartość temperatury (np. 0,8) sprawia, że generowanie jest bardziej losowe, a niższa wartość (np. 0,2) sprawia, że jest ono bardziej deterministyczne. Możesz również użyć maksymalnej wartości tokena, aby ograniczyć długość odpowiedzi.

* Zarządzanie tokenami — pamiętaj o całkowitej liczbie tokenów w konwersacji, ponieważ tokeny wejściowe i wyjściowe wliczają się do całkowitej liczby tokenów. Jeśli konwersacja zawiera zbyt wiele tokenów, aby zmieścić się w maksymalnym limicie modelu, musisz skrócić tekst, pominąć go lub skrócić.

* Postępowanie z poufną treścią — jeśli masz do czynienia z potencjalnie niebezpieczną treścią, zapoznaj się z wytycznymi Azure OpenAI dotyczącymi odpowiedzialnej sztucznej inteligencji (https://mng.bz/pxVK). Jeśli jednak korzystasz z interfejsu API OpenAI, pomocny będzie przewodnik moderacji OpenAI (https://mng.bz/OmEw) dotyczący dodawania warstwy moderacji do danych wyjściowych interfejsu API czatu.

ŚLEDZENIE TOKENÓW

Jak wspomniano wcześniej, śledzenie tokenów podczas korzystania z API konwersacyjnego jest kluczowe. Nie tylko komfort użytkowania ucierpi, jeśli przekroczymy całkowity rozmiar tokena, ale całkowita liczba tokenów w API ma również bezpośredni wpływ na opóźnienie i czas trwania

wywołania. Wreszcie, im więcej tokenów używamy, tym więcej płacimy. Oto kilka sposobów zarządzania tokenami:

* Liczenie tokenów. Użyj biblioteki tiktoken, która pozwala nam zliczać liczbę tokenów w ciągu bez wywoływania API.

* Ogranicz długość odpowiedzi. Podczas wywoływania API użyj właściwości max_tokens, aby ograniczyć długość odpowiedzi modelu.

* Skróć długie konwersacje. Jeśli konwersacja ma zbyt wiele tokenów, aby zmieścić się w maksymalnym limicie modelu, musimy skrócić, pominąć lub pominąć tekst.

* Ogranicz liczbę tur. Ograniczenie liczby tur w konwersacji to dobry sposób na skrócenie tekstu. Pomaga to również lepiej sterować modelem, gdy konwersacja się wydłuża i ma tendencję do halucynacji.

* Sprawdź pole użycia w odpowiedzi API. Po wywołaniu API możemy sprawdzić pole użycia w odpowiedzi API, aby zobaczyć całkowitą liczbę użytych tokenów. Proces ten jest ciągły i obejmuje zarówno tokeny wejściowe, jak i wyjściowe. To dobry sposób na śledzenie tokenów i prezentowanie ich użytkownikowi za pośrednictwem UX.

* Zmniejszenie temperatury. Zmniejszenie parametru temperatury może sprawić, że dane wyjściowe modelu będą bardziej skupione i zwięzłe, co może pomóc w zmniejszeniu liczby tokenów użytych w odpowiedzi.

Załóżmy, że chcemy zbudować aplikację czatu dla naszego salonu weterynaryjnego i umożliwić klientom zadawanie pytań o zwierzęta, pielęgnację i ich potrzeby. Możemy zbudować aplikację czatu w konsoli, jak pokazano na listingu 3.25. Pokazuje nam ona również możliwy sposób śledzenia i zarządzania tokenami. W tym przykładzie mamy funkcję num_tokens_from_messages, która, jak sama nazwa wskazuje, służy do obliczania liczby tokenów w rozmowie. Wraz z rozwojem rozmowy, z każdą turą obliczamy liczbę wykorzystanych tokenów, a po osiągnięciu limitu modelu, stare wiadomości są usuwane z rozmowy. Zauważ, że zaczynamy od indeksu 1. Dzięki temu zawsze zachowujemy wiadomość systemową pod indeksem 0 i usuwamy tylko wiadomości od użytkownika/asystenta.

Listing 25 : ConsoleChatApp: Zarządzanie tokenami

```
import os
from openai import AzureOpenAI
import tiktoken

client = AzureOpenAI(
    azure_endpoint=os.getenv("AOAI_ENDPOINT"),
    api_version="2024-05-01-preview",
    api_key=os.getenv("AOAI_KEY"))

GPT_MODEL = "gpt-35-turbo"

system_message = {"role": "system", "content": "You are
    ➥a helpful assistant
max_response_tokens = 250
token_limit = 4096
conversation = []
conversation.append(system_message)

def num_tokens_from_messages(messages):
    encoding= tiktoken.get_encoding("cl100k_base")
    num_tokens = 0
    for message in messages:
        num_tokens += 4
        for key, value in message.items():
            num_tokens += len(encoding.encode(value))
            if key == "name":
                num_tokens += -1
    num_tokens += 2

print("I am a helpful assistant. I can talk about pets and salons.")

while True:
    user_input = input("")
    conversation.append({"role": "user", "content": user_input})
    conv_history_tokens = num_tokens_from_messages(conversation)

    while conv_history_tokens + max_response_tokens >= token_limit:
        del conversation[1]
        conv_history_tokens = num_tokens_from_messages(conversation)

    response = client.chat.completions.create(
        model=GPT_MODEL,
```

Sets up the OpenAI environment and configuration details

Sets up the system message for the chat

Function to count the total tokens from all the messages in the conversation

Uses the tiktoken library to count tokens

Loops through the messages

When the total tokens exceed the token limit, we remove the second token. The first token is the system token, which we always want.

Captures the user input

Chat completion API call

```
    messages=conversation,
    temperature=0.8,
    max_tokens=max_response_tokens)
```

```
conversation.append({"role": "assistant", "content":
➥response.choices[0].message.content})
print("\n" + response.choices[0].message.content)
print("(Tokens used: " + str(response.usage.total_tokens)  + ")")
```

UZUPEŁNIANIE CZATU A API UZUPEŁNIANIA

Zarówno API uzupełniania czatu, jak i API uzupełniania czatu zostały zaprojektowane do generowania tekstu zbliżonego do ludzkiego i są używane w różnych kontekstach. API uzupełniania jest przeznaczone do zadań jednoetapowych, zapewniając ukończenie komunikatu podanego przez użytkownika. Najlepiej sprawdza się w zadaniach, w których wymagana jest tylko jedna odpowiedź. Natomiast API uzupełniania czatu jest przeznaczone do rozmów wieloetapowych, utrzymując kontekst rozmowy w trakcie wielu wymian. Dzięki temu jest bardziej odpowiednie dla aplikacji interaktywnych, takich jak chatboty. API uzupełniania czatu to nowe, dedykowane API do interakcji z modelami GPT-35-Turbo i GPT-4 i jest preferowaną metodą. API uzupełniania czatu jest bardziej ukierunkowane na

chatboty, a korzystając z różnych ról (system, użytkownik i asystent), możemy uzyskać pamięć poprzednich wiadomości i organizować przykłady z krótkimi odpowiedziami.

Najlepsze praktyki zarządzania tokenami

W przypadku LLM tokeny stały się nową walutą. Ponieważ większość przedsiębiorstw wykracza poza testowanie i zajmuje się krytycznymi dla biznesu przypadkami użycia, zarządzanie tokenami staje się priorytetem pod względem obliczeń, kosztów i ogólnego doświadczenia. Z perspektywy aplikacji korporacyjnych, oto kilka kwestii dotyczących zarządzania tokenami:

* Zwięzłe komunikaty — tam, gdzie to możliwe, używanie zwięzłych komunikatów i ograniczanie maksymalnej liczby tokenów zmniejszy ich wykorzystanie, czyniąc je bardziej opłacalnymi.

* Sekwencje zatrzymania — używaj sekwencji zatrzymania, aby zatrzymać generowanie, unikając generowania niepotrzebnych tokenów.

* Zliczanie tokenów — możemy zliczać tokeny za pomocą biblioteki tiktoken, jak opisano wcześniej, i uniknąć wykonywania tych samych wywołań API.

* Mniejsze modele — ogólnie rzecz biorąc, w informatyce większy i nowszy sprzęt oraz oprogramowanie są uważane za szybsze, tańsze i lepsze; jednak niekoniecznie tak jest w przypadku modeli LLM. Tam, gdzie to możliwe, rozważ najpierw użycie mniejszych modeli, takich jak GPT-3.5 Turbo, a jeśli mogą one nie być odpowiednie, rozważ przejście na kolejny. Mniejsze modele wymagają mniejszej mocy obliczeniowej, a zatem są bardziej ekonomiczne.

* Użyj buforowania — w przypadku komunikatów, które są albo dość statyczne, albo często powtarzane, wdrożenie strategii buforowania pomogłoby zaoszczędzić tokeny i uniknąć wielokrotnych wywołań API. W przypadku bardziej złożonych scenariuszy rozważ buforowanie osadzenia za pomocą wyszukiwania i przechowywania wektorów, takiego jak Azure Cognitive Search, Pinecone itp.

RAG i komunikację z danymi.

Dodatkowi dostawcy LLM

Dodatkowi dostawcy oferują teraz również LLM do wykorzystania przez przedsiębiorstwa. Są one dostępne za pośrednictwem interfejsów API lub, w niektórych przypadkach, jako wagi modeli, które przedsiębiorstwa mogą samodzielnie hostować. Tabela 3.7 przedstawia niektóre z bardziej znanych rozwiązań dostępnych w momencie publikacji. Należy pamiętać, że obowiązują pewne ograniczenia z punktu widzenia licencji komercyjnych.

Modele: Opisy

Llama 2: Meta wydała Llama 2, model LLM o otwartym kodzie źródłowym, dostępny w trzech rozmiarach (7 miliardów, 13 miliardów i 70 miliardów parametrów) i jest darmowy do celów badawczych i komercyjnych. Firmy mogą uzyskać do niego dostęp za pośrednictwem rozwiązań chmurowych, takich jak katalog modeli Azure AI, Hugging Face lub AWS. Przedsiębiorstwa, które chcą hostować go na własnych komputerach i procesorach graficznych, mogą poprosić Meta o dostęp za pośrednictwem https://ai.meta.com/llama/.

PaLM: PaLM to model Google o 13 miliardach parametrów, który jest częścią ich generatywnej sztucznej inteligencji dla produktów deweloperskich. Model może wykonywać zadania podsumowania tekstu, generowania dialogów i wnioskowania w języku naturalnym. W momencie publikacji istniała lista oczekujących na klucz API.

BLOOM: Bloom to wielojęzyczny model open source o 223 miliardach parametrów, który potrafi rozumieć i generować tekst w ponad 100 językach dzięki współpracy z ponad 1000 badaczy z ponad 250 instytucji. Jest dostępny do wdrożenia za pośrednictwem Hugging Face.

Claude: Claude to 12-miliardowy model parametrów opracowany przez Anthropic. Jest dostępny za pośrednictwem interfejsu Playground i API w konsoli programistycznej wyłącznie w celach programistycznych i ewaluacyjnych. W przypadku publikacji, w celu wykorzystania w środowisku produkcyjnym, przedsiębiorstwa muszą skontaktować się z Claude w celu rozmów handlowych. .

Gemini: Google niedawno wydało nowy model LLM o nazwie Gemini, następcę PaLM 2, zoptymalizowany pod kątem różnych rozmiarów: ultra, pro i nano. Został zaprojektowany tak, aby był bardziej wydajny niż jego poprzednik i mógł być używany do generowania nowych treści. Google twierdzi, że jest to ich najpotężniejszy model sztucznej inteligencji.

Co ciekawe, wszyscy ci dostawcy stosują podobne podejście do koncepcji i interfejsów API opracowanych przez OpenAI. Na przykład, jak opisano w ich dokumentach, model PaLM z odpowiednika API Google w zakresie uzupełniania jest przedstawiony w poniższym zestawieniu.

Listing 26: Podpis API tekstowego wygenerowany przez PaLM

```
google.generativeai.generate_text(*,

model: Optional[model_types.ModelNameOptions] = 'models/text-bison-001',

prompt: str,

temperature: Optional[float] = None,

max_output_tokens: Optional[int] = None,

top_p: Optional[float] = None,

top_k: Optional[float] = None,

stop_sequences: Union[str, Iterable[str]] = None,

) -> text_types.Completion
```

Chociaż takie opcje istnieją, a niektóre pochodzą od renomowanych i wiodących firm technologicznych, dla większości przedsiębiorstw Azure OpenAI i OpenAI są najbardziej dojrzałe, oferując najwięcej potrzebnych mechanizmów kontroli i wsparcia. Następny rozdział będzie poświęcony obrazom i dowiemy się, jak przejść od tekstu do obrazów i generować dane w tym trybie.

Podsumowanie

* Modele GenAI są klasyfikowane do różnych kategorii, w zależności od typu. Każdy model ma dodatkowe możliwości i cechy. Wybór odpowiedniego modelu do danego przypadku użycia jest ważny. W przeciwieństwie do informatyki, w naszym przypadku, największy model niekoniecznie jest lepszy.

* Interfejs API uzupełniania to zaawansowane narzędzie generujące tekst, który może być używany do uzupełniania komunikatów użytkownika i stanowi podstawę paradygmatu generowania tekstu.

* Interfejs API uzupełniania jest stosunkowo łatwy w użyciu i wymaga tylko kilku kluczowych parametrów, takich jak komunikat, liczba tokenów do wygenerowania, parametr temperatury, który pomaga sterować modelem, oraz liczba uzupełnień do wygenerowania.

* Interfejs API udostępnia wiele zaawansowanych opcji sterowania modelami i kontrolowania losowości oraz generowanego tekstu, takich jak logit_bias, kara za obecność i kara za częstotliwość. Wszystkie te opcje działają w tandemie i pomagają generować lepsze wyniki.

* Podczas korzystania z Azure OpenAI filtr bezpieczeństwa treści może pomóc w filtrowaniu określonych kategorii, aby identyfikować i reagować na potencjalnie szkodliwe treści zarówno w ramach komunikatów wejściowych, jak i generowanych uzupełnień.

* Interfejs API uzupełniania czatu bazuje na interfejsie API uzupełniania, przechodząc od jednego zestawu instrukcji i interfejsów API do dialogu z użytkownikiem w interakcji krok po kroku. Uzupełnianie czatu obejmuje wiele ról systemowych, użytkownika i asystenta. Rozmowa rozpoczyna się od komunikatu systemowego, który określa zachowanie asystenta, a następnie następują naprzemienne komunikaty użytkownika i asystenta w miarę postępu rozmowy krok po kroku.

* Rola systemowa znajduje się na początku tablicy komunikatów. Zawiera ona początkowe instrukcje dla modelu, w tym cechy osobowości, instrukcje i reguły, których ma przestrzegać asystent, a także dodatkowe informacje, które chcemy zapewnić jako kontekst dla modelu; te dodatkowe informacje nazywane są uziemieniem danych.

* Każda odpowiedź interfejsu API uzupełniania i uzupełniania czatu ma przyczynę zakończenia, która pomaga nam zrozumieć, dlaczego API zwróciło daną odpowiedź. Może to być przydatne do debugowania i ulepszania aplikacji.

* Wszystkie modele nauki języków mają skończone okno kontekstowe i są dość drogie. Zarządzanie tokenami staje się dla nas ważne, aby móc uruchamiać je po rozsądnych kosztach i w ramach limitu API. Pomaga nam to również zarządzać tokenami w konwersacjach, co poprawia komfort użytkowania i obniża koszty.

* Oprócz Azure OpenAI i OpenAI istnieją inni dostawcy LLM, tacy jak Llama 2 firmy Meta, Gemini i PaLM firmy Google, Bloom firmy BigScience oraz Claude firmy Anthropic. Ich oferty są podobne i opierają się na paradygmacie uzupełniania i uzupełniania czatów, a także zawierają podobne interfejsy API.