

Agenta zatrudnię, od zaraz

W przeciągu ostatnich kilku miesięcy na świecie zaroilo się od agentów AI do tego stopnia, że popularne powiedzenie o strachu przed otwarciem lodówki stało się już archaicznym przeżytkiem [1]. Co zrobić, teraz trzeba zacząć uważać z otwieraniem konserw. Albo... wsadzić do nich własnych agentów.

I LICENCJA NA PRZETWARZANIE

Pojęcie „agent”, wbrew obiegowym opiniom, istnieje w świecie badań nad sztuczną inteligencją już od lat 80., kiedy naukowcy zajmowali się koncepcją tak zwanych systemów wieloagentowych. W kontekście tych badań mianem „agenta” określano jednostkę, która działa w jakimś środowisku i autonomicznie podejmuje decyzje. O samej koncepcji możemy przeczytać na przykład w artykule o nazwie „Intelligence without representation” autorstwa Rodney’a A. Brooksa z MIT Artificial Intelligence Laboratory (1987), w którym pisze on:

I wish to build completely autonomous mobile agents that co-exist in the world with humans, and are seen by those humans as intelligent beings in their own right. I will call such agents Creatures.

Z kolei w książce „Artificial Intelligence: A Modern Approach” autorstwa Stuarta Russella oraz Petera Norviga (1995, nazywanej też skrótowo AIMA) pojęcie agenta jest już dobrze zdefiniowane: ma to być sztuczny twór, który działa autonomicznie, doświadcza i wchodzi w interakcję z otoczeniem, działa przez dłuższy czas, adaptuje się do zmian, a także tworzy i próbuje osiągnąć cele.

Pojawienie się dużych modeli językowych stworzyło wymarzone warunki do implementowania agentów, ponieważ sieci pracujące na tych modelach spełniają większość wymagań, które stawiane są agentom. Będąc jednak (efektywnie) funkcjami przekształcającymi tekst w tekst, nie są w stanie zrealizować wymagań analizy i interakcji z otoczeniem.

Dostarczenie LLM-om możliwości wpływania na otoczenie jest dużym wyzwaniem – i to z kilku powodów. Ogólnie sieci te nie są jako takie specjalizowane pod konkretne zadania (uczone są na dużych zbiorach informacji z wielu różnych dziedzin), więc trudno jest odpowiedzieć na pytanie, jaką konkretnie interakcję chcielibyśmy umożliwić. Poza tym probabilistyczna natura tych sieci, a także szereg znanych problemów, które w nich występują – z halucynacjami na czele – natychmiast przywodzi na myśl całą plejadę potencjalnych zagrożeń, mogących pojawić się, gdy pozwolimy sieciom na zbyt wiele. Nie trzeba szukać daleko – powszechnie znana jest sytuacja, w której agent omyłkowo usunął całą zawartość partycji na komputerze developera pracującego w środowisku Google Antigravity. Do licha, sam doświadczyłem tego, choć w mniejszej skali: agent powiedział w pewnym momencie: „usuwa wszystkie pliki, ponieważ problem jest zbyt trudny do usunięcia” – i faktycznie usunął kilka plików z kodem źródłowym.

Mimo wszystko agenty AI otwierają przed użytkownikami ogromne możliwości – zwłaszcza w obszarze zadań związanych z przetwarzaniem tekstu w jego różnych formach. Co więcej, tworzenie własnych agentów jest dziś stosunkowo proste, ponieważ modele językowe moż-

na integrować za pomocą publicznych interfejsów API, rozszerzając ich możliwości o dodatkowe narzędzia umożliwiające kontrolowaną pracę na lokalnych danych. Warto też podkreślić, że interfejsy API udostępniane przez Microsoft i OpenAI dla środowiska .NET są na tyle przystępne w użyciu, iż nawet mniej doświadczeni programiści powinni bez większych trudności wdrożyć je w swoich projektach.

I MICROSOFT.AGENTS.AI

Pierwotnie Microsoft opublikował bibliotekę o nazwie Microsoft SemanticKernel, której zadaniem było umożliwienie łatwej integracji systemów typu LLM do różnego rodzaju aplikacji. Z czasem wyewoluowała ona jednak w znacznie bardziej rozbudowany mechanizm o nazwie Microsoft Agents Framework, który służy do pisania agentów oraz systemów multi-agentowych.

Ponieważ najłatwiej jest opisać podstawy tej biblioteki na przykładzie, postawmy sobie problem, który będziemy próbowali rozwiązać.

W poprzednim numerze „Programisty” ukazał się mój artykuł poświęcony automatycznemu generowaniu kodu źródłowego z wykorzystaniem silnika T4 oraz mechanizmów kompilatora C#. Zanim jednak trafił do publikacji, gorączkowo przeszukiwałem archiwalne numery czasopisma, ponieważ byłem niemal przekonany, że temat ten opisywałem już wcześniej. Na szczęście redakcja „Programisty” szybko wyprowadziła mnie z błędu, dzięki czemu artykuł mógł ujrzeć światło dzienne. Ta sytuacja uświadomiła mi jednak, jak przydatny byłby mechanizm umożliwiający katalogowanie wszystkich napisanych przeze mnie tekstów oraz ich semantyczne przeszukiwanie – a więc według tematyki, a nie wyłącznie słów kluczowych. Właśnie taki program spróbujemy stworzyć w niniejszym artykule.

Za chwilę przejdziemy do pisania kodu, ale zanim to się stanie, musimy poznać specyficzne pojęcie ze świata LLM-ów, a mianowicie „embedding”.

I EMBEDDING

Programy komputerowe – w tym również i sieci neuronowe – najlepiej radzą sobie, pracując z liczbami. Siłą rzeczy więc musi istnieć sposób na to, aby słowa – na których operują przecież LLM-y – przekształcić w jakiś sposób na liczby. Oczywiście nie jest to problemem, bo słowa możemy rozbić na litery, a te zapisać na przykład w Unicode. W ten sposób jednak zatracamy istotny aspekt słów, który jest absolutnie kluczowy podczas przetwarzania danych językowych, czyli ich znaczenie.

Embedding stanowi sposób odwzorowania słowa w wektor, kodujący w swoich współrzędnych różne aspekty znaczenia słowa. Jest on skonstruowany w taki sposób, że jeżeli weźmiemy na przykład

embedding słowa „król”, odejmiemy od niego wektorowo embedding słowa „mężczyzna”, a potem dodamy embedding słowa „kobieta”, to otrzymamy wektor bardzo zbliżony do embeddingu słowa „królowa”. Lub na przykład jeżeli odejmiemy embeddingi słów „mąż” i „żona” oraz słów „brat” i „siostra”, to otrzymamy podobne wyniki.

Oczywiście trudno byłoby zakodować znaczenia słów w wektorach trójwymiarowych, więc zwykle stosuje się znacznie większe struktury danych. Popularnym wymiarem, obecnym w świecie LLM-ów, są wektory 1536-wymiarowe. Liczba ta może przyprawiać o zawrót głowy – a szczególnie gdyby ktoś próbował sobie wyobrazić przestrzeń 1536-wymiarową, ale matematycy opracowali pewną metodę, która ułatwia poruszanie się w nawet tak bardzo złożonych przestrzeniach. Powstał na ten temat nawet branżowy dowcip:

Matematyk i fizyk newtonowski poszli na wykład fizyka teoretycznego. Ten przez cały wykład operował na przestrzeniach 15-, 27- i 45-wymiarowych. Fizyk newtonowski nic nie rozumiał, a matematyk był cały podekscytowany. Po skończonym wykładzie fizyk newtonowski zapytał matematyka:

– Jak ty to robisz, że jesteś w stanie poradzić sobie z przestrzeniami 27-wymiarowymi?

– To proste – odpowiedział matematyk – wyobrażam sobie przestrzeń n -wymiarową, a potem przyjmuję, że n jest równe 27.

Sztuczka polega na obserwacji, że wektory w różnych przestrzeniach, niezależnie od liczby wymiarów ich przestrzeni, zachowują się w bardzo podobny sposób. Jeżeli więc jako przykład weźmiemy iloczyn skalarny dwóch wektorów (stanowiący sumę iloczynów odpowiadających sobie współrzędnych, czyli $x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2 + \dots$, to niezależnie od tego, czy operujemy na przestrzeni dwu-, trój- czy dwudziestosiędmowymiarowej, w każdym przypadku otrzymamy wartość reprezentującą iloczyn długości obu wektorów i kosinusa kąta pomiędzy nimi. Analogicznie, obliczenie długości wektora w każdej z przestrzeni odbywa się w taki sam sposób: wyciągamy pierwiastek z sumy kwadratów jego współrzędnych.

Embeddingi możemy więc wyobrazić sobie jako wektory wskazujące na punkty jednostkowej (choć 1536-wymiarowej) sfery. Jeżeli wskazują one (mniej więcej) ten sam kierunek, to oznacza, że znaczenia reprezentowanych przez nie słów są (mniej więcej) takie same. I analogicznie, jeżeli wskazują przeciwne kierunki, to możemy domniemywać, że ich znaczenia są również przeciwstawne. Mówiąc innymi słowy, znaczenie słowa jest mapowane na pewien kierunek w wielowymiarowej przestrzeni.

Opisana powyżej własność pozwala skonstruować banalny, matematyczny konstrukt, pozwalający ocenić, w jakim stopniu dwa embeddingi są podobne, a jest nim po prostu iloczyn skalarny.

Specjalną cechą (opisanego chwilę wcześniej) iloczynu skalarnego jest to, że jeśli obliczamy go na wektorach jednostkowych (czyli takich, których długość jest równa 1), to jego wynikiem jest po prostu kosinus kąta pomiędzy nimi. Kąt pomiędzy dwoma wektorami w dowolnej przestrzeni musi zawsze mieścić się w zakresie od 0 (gdy wektory leżą na jednej prostej) do 180 stopni (gdy wektory leżą na jednej prostej, ale wskazują w przeciwnych kierunkach). Z kolei kosinus ma tę własność, że gdy kąt zbiega do 0, jego kosinus zbiega do 1, zaś kiedy kąt zbiega do 180 stopni, jego kosinus zbiega do -1.

Skoro więc wiemy, że dwa embeddingi są tym bardziej do siebie podobne, im bardziej kąt pomiędzy nimi zbiega do 0, oznacza to automatycznie, że są one do siebie tym bardziej podobne, im bardziej kosinus

kąta pomiędzy nimi będzie zbiegał do 1. W ten sposób prosty iloczyn skalarny – którego obliczenie sprowadza się przecież do banalnej sumy iloczynów – staje się narzędziem, które pozwala nam w niesamowicie wygodny sposób ocenić, jak podobne do siebie są dwa embeddingi, czyli dwie matematyczne reprezentacje jakichś słów lub zdań.

I BUDOWANIE BAZY WIEDZY

Aby skonstruować agenta, który pomoże wyszukiwać w plikach wiedzę semantycznie, a nie syntaktycznie (czyli pod kątem znaczeń, a nie słów), będziemy najpierw musieli skonstruować bazę danych, która takie przeszukiwanie umożliwi.

Tu rodzi się pewien problem. Technicznie każdy tekst można przekształcić w embedding, choć oczywiście im krótszy tekst, tym lepszy, bo wtedy zbudowany wektor bardziej bezpośrednio odpowiadał będzie jego treści. Jednak z drugiej strony przekształcenie tekstu w embedding może skończyć się tym, że baza danych (stanowiącą formę indeksu), którą zbudujemy, będzie zajmować wielokrotnie więcej pamięci od samego tekstu, czego oczywiście chcielibyśmy uniknąć. Dlatego przed wygenerowaniem tekstu warto jest go (semantycznie) skompresować, aby w embeddingu znalazły się tylko najważniejsze informacje na jego temat. A taką semantyczną formą kompresji tekstu jest po prostu jego streszczenie.

Mamy już wystarczającą wiedzę, aby zarysować szkielet programu, który będziemy budować. Nasze narzędzie będzie składało się z dwóch głównych części.

1. Generator indeksu (bazy danych embeddingów fragmentów plików), który powinien realizować kroki:
 - 1.1. Wczytanie pliku.
 - 1.2. Przetworzenie go fragmentami (na przykład po 10 linijek).
 - 1.3. Streszczenie zawartości każdego fragmentu.
 - 1.4. Wykonanie embeddingów streszczeń i zapisaniu ich w bazie wraz z nazwami plików i położeniami tekstów (zakresami linii). Opcjonalnie można zapisać tam również samo streszczenie, ale nie jest ono konieczne do realizacji zadania.
2. Semantyczne wyszukiwanie dokumentu, realizowane poprzez:
 - 2.1. Zadanie pytania agentowi AI przez użytkownika.
 - 2.2. Przeszukanie przez agenta indeksu dokumentów, w poszukiwaniu tych, których embedding będzie możliwie najbardziej zgodny z zapytaniem użytkownika.
 - 2.3. Wczytanie fragmentu tekstu wskazywanego przez odnaleziony embedding (lub kilka embeddingów), aby stwierdzić, czy faktycznie odpowiadają one zapytaniu użytkownika.
 - 2.4. Zwrócenie użytkownikowi nazwy pliku, lokalizacji oraz fragmentów stanowiących odpowiedź na zapytanie użytkownika.

Spróbujmy teraz zaimplementować powyższy mechanizm za pomocą frameworka Microsoft Agents Framework.

I PODSTAWY

Zacniemy od stworzenia aplikacji konsolowej dla .NET 10 oraz dodania kilka pakietów, z których będziemy później korzystać:

- » OpenAI (dostarczy możliwości wykorzystania modeli językowych typu gpt-5).
- » Microsoft.Agents.AI.OpenAI (zapewni warstwę proxy do OpenAI).

Oraz dodatkowo, opcjonalnie:

- » Autofac (lub inny kontener DI).
- » protobuf-net (upraszczający wydajne serializowanie danych binarnych).
- » Serilog (silnik logowania).

Drugim krokiem będzie pozyskanie klucza API. W tym celu przejdziemy na stronę <https://platform.openai.com> i zakładamy tam konto, a następnie generujemy klucz API. Później – tu zła wiadomość – musimy zasilić swoje konto jakimiś środkami, bo korzystanie z OpenAI przez API jest płatne. Dobra wiadomość jest jednak taka, że zapytania do modeli nie są specjalnie drogie i nawet za 5\$ (około 20 PLN) można z powodzeniem napisać i przetestować aplikację zawierającą agenta AI. Dla zobrazowania skali, przetworzenie plików i wykonanie zapytań, o których zaraz będzie mowa, kosztowało mnie raptem 45 centów.

I to wszystko! Mając klucz API, możemy zacząć już implementować infrastrukturę aplikacji.

Tu mała uwaga. Napisana przeze mnie aplikacja zawiera trochę takiego kodu infrastrukturalnego (konfiguracja DI, loggera itp.), który nie jest istotny dla omawianego tematu, więc go pominę i skupię się tylko na tych aspektach implementacji, które dotyczą implementowanego agenta AI. Pełny kod całego programu można ściągnąć z mojego repozytorium [2].

I INFRASTRUKTURA

Na początku przygotujemy grunt pod dalszą implementację. Zaczniemy od najprostszej części, czyli bazy danych.

„Baza danych” to duże słowo, w praktyce wystarczy nam zwyczajna lista – szczególnie że nasz indeks będziemy przeszukiwali metodą *brute-force*. To nie lenistwo: nawet mechanizmy przeszukujące bazy wektorowe obecne na Azure również korzystają w niektórych okolicznościach z tej metody. Wynika to z faktu, że obliczenia konieczne do porównania semantycznego dwóch embeddingów są bardzo szybkie (same w sobie, a jeszcze szybsze, jeśli implementacja wykorzystuje odpowiednie rozkazy procesora), więc nie ma żadnego rozsądnego powodu tego procesu jeszcze bardziej optymalizować.

Nasza baza – czy indeks – będzie składać się więc z listy wpisów następującej postaci:

Listing 1. Wpis indeksu

```
using ProtoBuf;
using System;
using System.Collections.Generic;
using System.Text;

namespace AgentDemo.Models
{
    [ProtoContract]
    public class VectorEntry
    {
        [ProtoMember(1)]
        public Guid Id { get; set; } = new();

        [ProtoMember(2)]
        public string Content { get; set; } =
            string.Empty;
        [ProtoMember(3)]
```

```
public float[] Vector { get; set; } = [];

        [ProtoMember(4)]
        public string Filename { get; set; } =
            string.Empty;

        [ProtoMember(5)]
        public int StartLine { get; set; }

        [ProtoMember(6)]
        public int EndLine { get; set; }
    }

    [ProtoContract]
    public class VectorDatabase
    {
        [ProtoMember(1)]
        public List<VectorEntry> Entries { get; set; } = [];
    }
}
```

Każdy wpis zawiera identyfikator (dla porządku), zawartość (w której, dla celów diagnostycznych, umieścimy streszczenie fragmentu artykułu), jej embedding w postaci wektora, a także nazwę plików i zakres linii reprezentowanych przez rzeczony embedding. Sama „baza” będzie natomiast stanowić po prostu listę takich wpisów.

Przygotujemy również serwis, który na życzenie będzie dostarczał nam instancję klienta OpenAPI.

Listing 2. Serwis OpenAIProvider

```
using AgentDemo.Services.Configuration;
using OpenAI;

namespace AgentDemo.Services.OpenAI
{
    public class OpenAIProvider : IOpenAIProvider
    {
        private readonly IConfig config;

        public OpenAIProvider(IConfig config)
        {
            this.config = config;
        }

        public OpenAIClient GetOpenAIClient()
        {
            return new(config.ApiKey);
        }
    }
}
```

Oprócz powyższych klas stworzymy również dodatkowe serwisy – PathService, który będzie dostarczał nam przydatne ścieżki (w tym ścieżkę do katalogu z dokumentami), ResourceService, który ułatwi wydobywanie z osadzonych w zestawie zasobów tekstowych (w których zapiszemy między innymi instrukcje dla agenta), oraz Config, który pobierze z pliku *appsettings.json* odpowiednie dane konfiguracyjne i dostarczy nam je, gdy będą potrzebne. Ich implementacja jest bardzo prosta, więc zostawię ją jako ćwiczenie dla czytelnika (lub do pobrania z repozytorium).

Kluczowym aspektem budowanego przez nas rozwiązania jest oczywiście przekształcanie tekstu na embeddingi. Nie jest to zadanie proste, ale na szczęście usługę taką oferuje nam biblioteka OpenAI. Aby wygenerować embedding, konieczne jest podanie modelu, za pomocą którego ów proces zostanie zrealizowany. Tu ważna infor-

Index: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

3/2026 (124)

Cena 32.90 zł (w tym VAT 8%)

Już w Empiku

RUST LECI NA KSIĘŻYC

Randka z LOLA

Sprzedaż licencji
z Paddle

Agenta zatrudnię,
od zaraz

STM32 w środowisku
VS Code

Z archiwum CVE

Strefa CTF


mydevil.net

DOSKONAŁY HOSTING
DLA PROGRAMISTÓW

www.mydevil.net

ISSN 2084-9400

03



9 772084 940602