ChatGPT per API einbinden

Ein Programmbeispiel zeigt, wie man ohne Overhead und Umwege die Modelle hinter ChatGPT per OpenAI-API in eigenen C#-Anwendungen nutzt.

Von Daniel Basler

-tract

- Mit der OpenAI-API lässt sich das ChatGPT-Modell gpt-3.5-turbo in eigene Programme einbinden.
- Anstelle von Token aus unstrukturiertem Text verwendet ChatGPT eine Folge von Nachrichten zusammen mit Metadaten. OpenAI nennt das Format Chat Markup Language (ChatML).
- Mit einer in C# implementierten WPF-App können Interessierte erste Erfahrungen mit dem Einbinden der Sprach-KI in eigene Anwendungen sammeln.
- Durch Zugriff auf verschiedene Modellparameter bietet die API die Möglichkeit, den Umgang mit Sprachmodellen zu erlernen und zu verbessern.

Mit ChatGPT (OpenAI) und Bard beziehungsweise LaMDA von Google ist die künstliche Intelligenz (KI) massenkompatibel geworden. ChatGPT hat sich auf Grundlage von Transformertechnologie und dem Attention-Mechanismus mit Millionen von Textseiten vollgesogen und daraus ein großes Modell der menschlichen Sprache gebaut: ein Large Language Model (LLM).

Dadurch kann ChatGPT eine Vielzahl von Aufgaben der natürlichen Sprachverarbeitung ausführen, darunter das Generieren, Zusammenfassen und Klassifizieren von Text und Frage-Antwort-Systeme. Des Weiteren erstellt ChatGPT Programme und hilft beim Debuggen von Code. All diese Aufgaben beherrschen auch andere Modelle aus der GPT-Reihe von OpenAI in unterschiedlichem Ausmaß. ChatGPT selbst ist für den Dialog

mit Nutzern optimiert, basiert aber ansonsten auf der Generation GTP 3.5 und damit den leistungsfähigsten und neuesten Modellen, die OpenAI derzeit im Angebot hat. OpenAI macht seine GPT-Modelle über eine einheitliche API (Application Programming Interface) verfügbar, über die sich mit diesen Modellen kommunizieren lässt. Damit ist es also möglich, eigene Programme mit den Fähigkeiten von ChatGPT auszustatten, seit Anfang März ist auch der Zugriff auf das ChatGPT-Modell über die API möglich. Unser Beispiel zeigt, wie sich die API und die Modelle von OpenAI mit C# ansprechen lassen. Über die API lassen sich die Modelle auch über unterschiedliche Parameter für verschiedene Zwecke optimieren.

Softwareautomatisierung mit generativer KI

Bei ChatGPT und den anderen GPT-Modellen von OpenAI handelt es sich um generative KI-Stacks. Hierbei kommen Deep- und Machine-Learning-Algorithmen sowie Generative Adversarial Networks (GAN) zum Einsatz, die aus vorhandenen Texten, Audiodateien oder Bildern Inhalte erstellen. Die Modelle Erlernen Muster in der Eingabe und verwenden sie, um ähnliche Inhalte zu erzeugen.

Transformermodelle wie GPT oder LaMDA simulieren kognitive Prozesse und versuchen die Bedeutung der Eingabedaten auf unterschiedliche Weise zu messen. Man trainiert die Modelle darauf, die Sprache oder das Bild im Kontext zu verstehen, Klassifizierungen zu erlernen und Texte oder Bilder aus großen Datensätzen zu generieren.

Das verwendete Sprachmodell Generative Pre-trained Transformer, kurz GPT, kann Befehle in natürlicher Sprache auch in Programmcode übersetzen. GPT ist also in der Lage, Websites oder Codebeispiele über automatisches Codegenerieren zu erzeugen. Das aus GPT entstandene Modell Codex ist auf Codegenerierung optimiert. GitHubs Tool Copilot liegt das

Codex-Modell zugrunde. Copilot analysiert Kommentare aus dem Code und schlägt einzelne Codezeilen oder Funktionen während des Entwickelns vor. Das am 1. März für die API veröffentlichte gpt-3.5-turbo-Modell, auf dem ChatGPT basiert, ist für die Ein- und Ausgabe von Konversationschats optimiert. Des Weiteren ist die Turbo-Modellfamilie auch die erste, die regelmäßige Modellupdates erhält.

Das Projekt

Das Programmierbeispiel zeigt, wie man mit der OpenAI-API auf GPT-3-Modelle zugreift. Dort hat man Zugriff auf gpt-3.5-turbo. Das Projekt Example-ChatGPTApplication ist überschaubar und greift per HTTP und JSON auf den API-Endpunkt zu. Zwar ist ein Zugriff auch über die Bibliothek .NET SDK for OpenAI GPT-3 möglich, die Library kommt jedoch mit einem viel größeren Funktionsumfang, den man für das Beispiel hier nicht benötigt. Das Programm kommt als Windows-Presentation-Foundation-App (WPF) mit einer einfachen Benutzeroberfläche für Anfragen, Modellauswahl und Einstellungen für Parameter daher. Man erstellt die Anwendung mit der Visual Studio Community Edition 2022 und dem .NET Framework 7.0. Um schlank zu bleiben, verwendet das Beispiel Code aus einer Code-Behind-Datei der Klasse MainWindow.xaml.cs.

Zum Verwenden der API benötigt man einen aktiven Account bei OpenAI, über den sich ein Authentifikationsschlüssel für die Verwendung der API erstellen lässt. OpenAI berechnet zurzeit 0,02 US-Dollar für 1000 Eingabetoken bei den bisherigen GPT-Modellen. Bei ChatGPT belaufen sich die Kosten auf 0,002 US-Dollar pro 1000 Token. Token stellen Teile von Wörtern dar, wobei 1000 Token etwa 750 Wörtern entsprechen. Beim Anmelden erhält man in der Regel ein Startguthaben von etwa 18 US-Dollar, das sich in den ersten drei Monaten frei verwenden lässt. Ist es aufgebraucht, benötigt man einen neuen API-Key, den OpenAI in Rechnung stellt.

Die Oberfläche

Das Programm verwendet die Projektvorlage WPF Application mit der Projektbezeichnung ExampleChatGPTApplication und dem .NET Framework 7.0. Visual Studio erstellt aus der Vorlage automatisch eine bereits lauffähige WPF-Applikation. Listing 1 zeigt die Erweiterung der Oberfläche in der XAML-Datei MainWindows.xaml. Das GUI lässt sich nach eigenem Geschmack erstellen oder später anpassen.

Listing 1: Aufbau der Oberfläche in der MailWindow.xaml-Datei

```
<Window x:Class="ExampleChatGPTApplication.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExampleChatGPTApplication"
        mc:Ignorable="d"
        WindowStartupLocation="CenterScreen"
           Title="MainWindow" Height="500" Width="720"
Loaded="Window_Loaded">
    <Grid Margin="0,0,10,0">
          <TextBox Height="396" HorizontalAlignment="Left"
Margin="11,10,0,0"  Name="chatBox"  VerticalAlignment="Top"
Width="535"
                                    IsReadOnly="True"
TextWrapping="WrapWithOverflow"
VerticalScrollBarVisibility="Visible"/>
              <Label  Content="Message:"  Height="28"
HorizontalAlignment="Left" Margin="11,411,0,0" Name="label5"
VerticalAlignment="Top" />
          <TextBox Height="22" HorizontalAlignment="Left"
Margin="86,414,0,0" Name="messageText" VerticalAlignment="Top"
Width="460" />
                <Button  Content="Send"  Height="23"
HorizontalAlignment="Left"      Margin="564,413,0,0"
Name="sendMessageButton" VerticalAlignment="Top" Width="117"
```

```
Click="sendMessageButton_Click"/>
        <ComboBox x:Name="comboBox" HorizontalAlignment="Left"
Margin="564,36,0,0"  VerticalAlignment="Top"  Width="120"
SelectionChanged="comboBox_SelectionChanged">
            <ComboBoxItem>text-davinci-002</ComboBoxItem>
                <ComboBoxItem IsSelected="True">text-
davinci-003</ComboBoxItem>
            <ComboBoxItem>code-davinci-002</ComboBoxItem>
            <ComboBoxItem>gpt-3.5-turbo</ComboBoxItem>
        </ComboBox>
              <Label  x:Name="label"  Content="Model:"
HorizontalAlignment="Left"       Margin="564,10,0,0"
VerticalAlignment="Top" Width="70"/>
          <Label x:Name="label1" Content="Temperature"
HorizontalAlignment="Left"       Margin="564,63,0,0"
VerticalAlignment="Top"/>
         <TextBox x:Name="textBox" HorizontalAlignment="Left"
Margin="564,94,0,0"   TextWrapping="Wrap"   Text="0.5"
VerticalAlignment="Top" Width="77"/>
          <Label x:Name="label2" Content="Presence Penalty"
HorizontalAlignment="Left"       Margin="564,122,0,0"
VerticalAlignment="Top"/>
        <TextBox x:Name="textBox1" HorizontalAlignment="Left"
Margin="564,153,0,0"   TextWrapping="Wrap"   Text="0"
VerticalAlignment="Top" Width="77"/>
         <Label x:Name="label3" Content="Frequency Penalty"
HorizontalAlignment="Left"       Margin="564,186,0,0"
VerticalAlignment="Top"/>
        <TextBox x:Name="textBox2" HorizontalAlignment="Left"
Margin="564,214,0,0"   TextWrapping="Wrap"   Text="0.5"
VerticalAlignment="Top" Width="77"/>
              <Label  x:Name="label4"  Content="TopP"
HorizontalAlignment="Left"       Margin="566,239,0,0"
VerticalAlignment="Top"/>
        <TextBox x:Name="textBox3" HorizontalAlignment="Left"
Margin="564,265,0,0"   TextWrapping="Wrap"   Text="0.3"
VerticalAlignment="Top" Width="77"/>
    </Grid>
</Window>
```

Die ComboBox verfügt in der XAML-Datei über kein spezielles ItemTemplate oder DataTemplate. Die Auswahl des Modells

erfolgt später im Code-Behind der Datei MainWindow.xaml. Über die Event-Handler-Methoden Loaded im Window und Click beim Send-Button lassen sich die Erweiterungen im Code-Behind vornehmen.

Programmlogik

Nach dem Vorbereiten der Oberfläche beginnt das Implementieren der Programmlogik. Der Code besteht aus fünf C#-Klassen:

- RequestChatGPT.cs für Anfragen an das Chat-Modell,
- ResponseChatGPT.cs für Antworten des Chat-Modells,
- RequestGPT.cs für Anfragen,
- ResponseGPT.cs für Antworten,
- ChatCompletionMessage.cs für Nachrichten im Chat-Modell.

In Visual Studio lassen sich die benötigten Klassen erstellen. Listing 2 und Listing 3 zeigen den Code für das Erstellen der RequestGPT– und der ResponseGPT-Klasse. Für ChatGPT müssen Entwickler die Request- und Response-Klasse für die Message-Methode (Nachrichten) erweitern. Listing 4 und 5 zeigen das Implementieren der Klassen RequestChatGPT und ResponseChatGPT. Die Klasse ChatCompletionMessage führt die Messages (Nachrichten) für die Chatvervollständigung ein, die nur gpt-3.5-turbo benötigt (siehe Listing 6).

Listing 2: Implementieren der Klasse RequestGPT

```
using System.Text.Json.Serialization;

namespace ExampleChatGPTApplication
{
    public class RequestGPT
    {
        public RequestGPT() { }

        [JsonPropertyName("model")]
        public string? Model { get; set; }
```

```
        [JsonPropertyName("prompt")]
        public string? Prompt { get; set; }

        [JsonPropertyName("max_tokens")]
        public int MaxTokens { get; set; }

        [JsonPropertyName("temperature")]
        public float Temperature { get; set; }

        [JsonPropertyName("top_p")]
        public float TopP { get; set; }

        [JsonPropertyName("presence_penalty")]
        public float PresencePenalty { get; set; }

        [JsonPropertyName("frequency_penalty")]
        public float FrequencyPenalty { get; set; }
    }
}
```

Listing 3: Implementieren der Klasse ResponseGPT

```
using System.Collections.Generic;
using System.Text.Json.Serialization;

namespace ExampleChatGPTApplication
{
    public class ResponseGPT
    {
        [JsonPropertyName("id")]
        public string? Id { get; set; }

        [JsonPropertyName("object")]
        public string? @Object { get; set; }

        [JsonPropertyName("created")]
        public int Created { get; set; }

        [JsonPropertyName("model")]
        public string? Model { get; set; }

        [JsonPropertyName("choices")]
```

```
        public List<ChatGPTChoice>? Choices { get; set; }

        [JsonPropertyName("usage")]
        public ChatGPTUsage? Usage { get; set; }
    }
}

public class ChatGPTUsage
{
    [JsonPropertyName("prompt_tokens")]
    public int PromptTokens { get; set; }

    [JsonPropertyName("completion_tokens")]
    public int CompletionTokens { get; set; }

    [JsonPropertyName("total_tokens")]
    public int TotalTokens { get; set; }
}

public class ChatGPTChoice
{
    [JsonPropertyName("text")]
    public string? Text { get; set; }

    [JsonPropertyName("index")]
    public int Index { get; set; }

    [JsonPropertyName("logprobs")]
    public object? LogProbabilities { get; set; }

    [JsonPropertyName("finish_reason")]
    public string? FinishReason { get; set; }
}
```

Listing 4: Ausschnitt aus der Implementierung der Klasse RequestChatGPT

```
using System.Collections.Generic;
using System.Text.Json.Serialization;

namespace ExampleChatGPTApplication
{
```

```
    public class RequestChatGPT
    {
        [JsonPropertyName("model")]
        public string Model { get; set; } = null!;

        [JsonPropertyName("messages")]
         public IList<ChatCompletionMessage> Messages { get;
set; } = new List<ChatCompletionMessage>();...
```

Listing 5: Implementieren der Klasse ResponseChatGPT

```
using System.Collections.Generic;
using System.Text.Json.Serialization;

namespace ExampleChatGPTApplication
{
    public class CreateChatResponse
    {
        [JsonPropertyName("id")]
        public string Id { get; set; } = null!;

        [JsonPropertyName("object")]
        public string Object { get; set; } = null!;

        [JsonPropertyName("created")]
        public int Created { get; set; }

        [JsonPropertyName("model")]
        public string Model { get; set; } = null!;

        [JsonPropertyName("choices")]
         public List<ChatCompletionChoice> Choices { get; set;
} = new();

        [JsonPropertyName("usage")]
        public ChatCompletionUsage? Usage { get; set; }
    }
    public class ChatCompletionChoice
    {
        [JsonPropertyName("delta")]
        public ChatCompletionMessage? Delta
```

```
        {
            get => Message;
            set => Message = value;
        }

        [JsonPropertyName("message")]
        public ChatCompletionMessage? Message { get; set; }

        [JsonPropertyName("index")]
        public int Index { get; set; }

        [JsonPropertyName("finish_reason")]
        public string FinishReason { get; set; } = null!;
    }
    public class ChatCompletionUsage
    {
        [JsonPropertyName("prompt_tokens")]
        public int PromptTokens { get; set; }

        [JsonPropertyName("completion_tokens")]
        public int CompletionTokens { get; set; }

        [JsonPropertyName("total_tokens")]
        public int TotalTokens { get; set; }
    }
}
```

Listing 6: Implementieren der Klasse ChatCompletionMessage

```
using System.Text.Json.Serialization;

namespace ExampleChatGPTApplication
{
    public class ChatCompletionMessage
    {
          public ChatCompletionMessage(string role, string
content)
        {
            Role = role;
            Content = content;
        }
```

```
        [JsonPropertyName("role")]
        public string Role { get; set; }

        [JsonPropertyName("content")]
        public string Content { get; set; }
    }
}
```

In der Request-Klasse bleiben die Eigenschaftsnamen der JSON-Ausgabe für die spätere HTTP-Anfrage über die API unverändert. Die Klasse liefert ein Request-Objekt, in dem sich das gewünschte Modell und Parameter festlegen lassen. Bei der Response-Klasse legt man auch die Eigenschaftsnamen als Attribute fest. Des Weiteren lassen sich hier die Klassen für das Verwenden der Token (ChatGPTUsage) und der Auswahl (ChatGPTChoice) beziehungsweise Message (ChatCompletionChoice) festlegen. In der MainWindow.xaml.cs implementiert man die Programmlogik zum Aufruf und zur Rückgabe über die OpenAI-API. Listing 7 zeigt die benötigten Methoden für den Zugriff auf die Sprachmodelle von OpenAI.

Listing 7: Implementieren der Zugriffslogik auf die API

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Net.Http;
using System.Text;
using System.Text.Json;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace ExampleChatGPTApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
```

```
    {
        string OpenAI_ApiKey = "Your API-KEY";
        string modelName = string.Empty;
        int maxTokens = 2048;
        string user = "Anfrage > ";
        string textGPT = "Antwort GPT > ";
        bool isChatGPTModel = false;

        public MainWindow()
        {
            InitializeComponent();
        }

            private void Window_Loaded(object sender,
RoutedEventArgs e)
        {
            if(OpenAI_ApiKey == string.Empty)
            {
                  MessageBox.Show("Bitte OpenAI API Key
eintragen!");
            }
        }

         private void comboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            int selectedIndex = comboBox.SelectedIndex;
            switch (selectedIndex)
            {
                case 0:
                    modelName = "text-davinci-002";
                    maxTokens = 2048;
                    isChatGPTModel = false;
                    break;
                case 1:
                    modelName = "text-davinci-003";
                    maxTokens = 4000;
                    isChatGPTModel = false;
                    break;
                case 2:
                    modelName = "code-davinci-002";
```

```
                    maxTokens = 2048;
                    isChatGPTModel = false;
                    break;
                case 3:
                    modelName = "gpt-3.5-turbo";
                    maxTokens = 4096;
                    isChatGPTModel = true;
                    break;
                default:
                    modelName = "text-davinci-003";
                    maxTokens = 4000;
                    isChatGPTModel = false;
                    break;
            }
        }

         private void sendMessageButton_Click(object sender,
RoutedEventArgs e)
        {
           if(!isChatGPTModel)
            {
                CallGPTRequest();
            }
           else
            {
                CallChatGPTModel();
            }
        }

        private async void CallGPTRequest()
        {

            RequestGPT completionReqGTP = new RequestGPT
            {
                Model = modelName,
                   Temperature = float.Parse(textBox.Text,
CultureInfo.InvariantCulture.NumberFormat),
                MaxTokens = maxTokens,
                     TopP = float.Parse(textBox3.Text,
CultureInfo.InvariantCulture.NumberFormat),
                 FrequencyPenalty = float.Parse(textBox1.Text,
```

```
CultureInfo.InvariantCulture.NumberFormat),
                PresencePenalty = float.Parse(textBox2.Text,
CultureInfo.InvariantCulture.NumberFormat)
            };

            using (HttpClient httpClient = new HttpClient())
            {
                try
                {
                    string? userResponse = messageText.Text;
                                chatBox.Foreground = new
SolidColorBrush(Colors.Red);
                    chatBox.Text = user + userResponse;
                    completionReqGTP.Prompt = userResponse;
                    ResponseGPT? responseGPT = null;

                     using (HttpRequestMessage httpReq = new
HttpRequestMessage(HttpMethod.Post,
"https://api.openai.com/v1/completions"))
                    {
                         httpReq.Headers.Add("Authorization",
$"Bearer {OpenAI_ApiKey}");
                                    string requestString =
JsonSerializer.Serialize(completionReqGTP);
                                     httpReq.Content = new
StringContent(requestString,         Encoding.UTF8,
"application/json");

                                using (HttpResponseMessage?
httpResponse = await httpClient.SendAsync(httpReq))
                        {
                            if (httpResponse is not null)
                            {
                                 string responseString = await
httpResponse.Content.ReadAsStringAsync();
                                                            if
(httpResponse.IsSuccessStatusCode                           &&
!string.IsNullOrWhiteSpace(responseString))
                                    {
                                                    responseGPT =
JsonSerializer.Deserialize<ResponseGPT>(responseString);
```

```
                                }
                            }
                        }
                    }

                    if (responseGPT != null)
                    {
                                    string? responseText =
responseGPT.Choices?[0]?.Text;
                                chatBox.Foreground = new
SolidColorBrush(Colors.DarkBlue);
                        chatBox.Text = textGPT+responseText;
                    }

                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
            }
        }

        private async void CallChatGPTModel()
        {
            var request = new RequestChatGPT
            {
                Model = modelName,
                Stream = true,
                MaxTokens = maxTokens,
                Messages = new List<ChatCompletionMessage>
                {
                    new("user", messageText.Text)
                }
            };

            HttpClient httpClient = new HttpClient();
            CreateChatResponse? responseChatGPT = null;

               using (HttpRequestMessage httpReq = new
HttpRequestMessage(HttpMethod.Post,
"https://api.openai.com/v1/chat/completions"))
```

```
            {
                httpReq.Headers.Add("Authorization", $"Bearer
{OpenAI_ApiKey}");
                                string requestString =
JsonSerializer.Serialize(request);
                                 httpReq.Content = new
StringContent(requestString,         Encoding.UTF8,
"application/json");

                 using (HttpResponseMessage? httpResponse =
await httpClient.SendAsync(httpReq))
                {
                    if (httpResponse is not null)
                    {
                            string responseString = await
httpResponse.Content.ReadAsStringAsync();
                         if (httpResponse.IsSuccessStatusCode
&& !string.IsNullOrWhiteSpace(responseString))
                        {
                                                responseChatGPT =
JsonSerializer.Deserialize<CreateChatResponse>(responseString)
;
                        }
                    }
                }
            }

            if (responseChatGPT != null)
            {
                                string? responseText =
responseChatGPT.Choices?[0]?.Message?.ToString();
                              chatBox.Foreground = new
SolidColorBrush(Colors.DarkBlue);
                chatBox.Text = textGPT + responseText;
            }
        }
    }
}
```

Als Erstes stehen die Felder für den Gültigkeitsbereich der deklarierten Variablen in der Klasse. Die String-Variable

OpenAI_ApiKey enthält den Authentifikations-Key für die API. Die Event-Handler-Methode Window_Loaded überprüft den String und gibt eine Meldung aus, wenn der Key nicht gesetzt ist.

Die Methode comboBox_Selection Changed verwendet eine Auswahlanweisung vom Typ switch für die Musterübereinstimmung des gewählten Modells und der Tokenanzahl. Die Event-Handler-Methode sendMessageButton_Click erstellt ein Objekt vom Typ RequestGPT für die HTTP-Anfrage mit dem Modellnamen und dazugehörige Parameter. Des Weiteren ruft die Übergabe des RequestGPT-Objektes die Methode CallGPTRequest auf.

Mit der Methode CallGPTRequest sendet das Programm eine HTTPRequestMessage über das HTTPClient-Objekt an den API-Endpunkt https://api.openai.com/v1/completions. Wählt man das Modell gpt-3.5-turbo aus, so erzeugt das Programm das entsprechende RequestChatGpt-Objekt und verwendet den API-Endpunkt https://api.openai.com/v1/chat/completions für die Chatkonversation. Die Authentifikation erfolgt über ein Bearer Token (Inhabertoken), das nicht an eine bestimmte Identität gebunden ist.

Der Endpunkt Completions bietet eine einfache Schnittstelle, um Anfragen an die API zu stellen. Der Endpunkt hat Zugriff auf die verschiedenen Modelle von OpenAI und ermöglicht es, Text basierend auf einer bestimmten Eingabeaufforderung zu generieren. Die API stellt noch weitere Endpunkte bereit. So liefert der Aufruf get https://api.openai.com/v1/models eine Liste der verfügbaren Modelle mit grundlegenden Informationen zu jedem Modell, wie zum Beispiel den Eigentümer und die Verfügbarkeit. Die vollständige API-Referenz findet sich unter ix.de/zhhp.

Die Anwendung übergibt das Request-Objekt als serialisiertes JSON-Objekt an die API. Das Programm deserialisiert das JSON-Objekt aus der HTTPResponseMessage vom HTTPClient-Objekt und weist es dem ResponeGPT– oder ResponseChatGPT-Objekt zu. Zum Schluss stellt die Choices-Collection den empfangenen Text in

der TextBox in der WPF-Oberfläche dar. Ist die Implementierung abgeschlossen, lässt sich das Projekt kompilieren, um die WPF-Applikation direkt aus Visual Studio zu starten und zu testen.

Programmablauf und Ergebnis

Die Programmoberfläche erlaubt es, eine Eingabeaufforderung als Text an das ausgewählte Sprachmodell zu senden. Die API gibt einen vervollständigten Text zurück, der versucht, den Befehlen oder dem Kontext zu entsprechen, den die Eingabe vorgegeben hat.

Die Benutzereingabe heißt bei OpenAI Prompt. Die Prompts sollten so konkret und ausführlich wie möglich formuliert sein. OpenAI spricht hier auch direkt von Prompt Engineering: Je besser die Anfrage, umso genauer das Ergebnis. Traditionell verwenden GPT-Modelle unstrukturierten Text, der für das Modell als eine Folge von Token dargestellt wird. Die neuen ChatGPT-Modelle verarbeiten stattdessen eine Folge von Nachrichten zusammen mit Metadaten. Dieses neue Format bezeichnet OpenAI als Chat Markup Language (ChatML). Das Programmbeispiel unterstützt die Prompt-Eingabe für alle Modelle.

OpenAI bietet eine Reihe unterschiedlicher Sprachmodelle: ChatGPT, Davinci, Codex, Curie, Babbage und Ada. Die Auswahl in der Programmoberfläche beschränkt sich auf die Modelle, die OpenAI zu GPT-3.5 zählt. Alle diese Modelle können natürliche Sprache verstehen und erzeugen. Das neueste Modell ist gpt-3.5-turbo. Es basiert auf Trainingsdaten bis September 2021 und kann jede Aufgabe erledigen, die die anderen Modelle ausführen können, oft mit höherer Qualität, längerer Ausgabe und besserer Befehlsfolge. gpt-3.5-turbo verarbeitet bis zu 4096 Eingabetoken. Es stellt zurzeit das leistungsfähigste Modell dar und ist für die Konversation im Chat optimiert. Außerdem ist es derzeit das kostengünstigste der GPT-Modelle.

Unter Codex befinden sich eine Reihe von Modellen, die Code

verstehen und generieren können, einschließlich der Übersetzung natürlicher Sprache in Code. Das Modell code-davinci-002 steht zurzeit als Beta bereit und gehört ebenfalls zur Generation GPT-3.5. Die Trainingsdaten enthalten sowohl natürliche Sprache als auch Milliarden von Zeilen öffentlichen Codes von GitHub. Die Modelle code-davinci-002 und gpt-3.5-turbo befinden sich noch in der Betaphase, sind nicht immer verfügbar und liefern hin und wieder auch nur groben Unfug zurück. Es könnte sein, dass OpenAI den API-Endpunkt noch anpasst. In diesem Fall lässt sich der Request-Body dementsprechend erweitern.

Token und Parameter

Die Modelle von OpenAI verarbeiten Text, indem sie ihn in Token zerlegen. Die Anzahl der Token, die eine bestimmte API-Abfrage verarbeitet, hängt von der Länge der Eingabe und Ausgabe ab. Ein Token entspricht ungefähr vier Zeichen oder 0,75 Wörtern für englischen Text. 2048 Token sind in etwa 1500 Wörter.

Alle Sprachmodelle von OpenAI verfügen über Parameter, deren Werte sich bei der Anfrage setzen lassen, um die Ausgabe zu optimieren. Temperature ist eine der wichtigsten Einstellungen beim Steuern der Ausgabe des Modells, denn sie steuert die Zufälligkeit des generierten Textes. Bei einem Wert von 0 ist das Modell deterministisch, das heißt, es erzeugt für einen bestimmten Eingabetext immer die gleiche Ausgabe. Bei einem Wert von 1 geht das Modell die meisten Risiken ein und ist bei der Ausgabe sehr kreativ. Durch das Verwenden der beiden Parameter Presence Penalty und Frequency Penalty lässt sich der Grad der Wortwiederholung in den Antworten der Modelle steuern. Die Angabe der Frequency Penalty senkt die Wahrscheinlichkeit, dass ein Wort erneut ausgewählt wird, je öfter es bereits verwendet wurde. Bei Presence Penalty berücksichtigt das Programm nicht, wie häufig es ein Wort verwendet hat, sondern nur, ob der Wert im Text bereits

vorkommt.

Der Parameter TopP ist eine alternative Möglichkeit, die Zufälligkeit und Kreativität des erzeugten Textes zu steuern. Die OpenAI-Dokumentation empfiehlt, nur eine der beiden Optionen Temperature oder TopP zu verwenden. Wenn man einen der beiden Parameter variiert, sollte der andere auf 1 gesetzt sein. Es gibt noch eine Vielzahl von Verbesserungsmöglichkeiten und Optimierungen für das Programmbeispiel. Durch schrittweises Herantasten kann man das Programm entsprechend anpassen oder erweitern und dabei mit verschiedenen Parametern und Sprachmodellen experimentieren.

Fazit

Dieses Programmierbeispiel gibt einen kleinen Überblick über die Möglichkeiten der OpenAI-API. Es zeigt, wie einfach es ist, die API in C# zu verwenden und so die Modelle von OpenAI in eigene Applikationen zu integrieren – also Text oder Code in natürlicher Sprache zu generieren. Besonders das Spielen mit den Modellparametern zeigt die Vielseitigkeit der Sprach-KI. Das Beispiel macht deutlich, was dieser Technikstack in Zukunft bieten wird und wie Entwickler damit umgehen können. (pst@ix.de)

1. Quellen
2. Das GitHub-Repository mit dem Quellcode dieses Artikels und weitere Informationen zur OpenAI-API finden sich unter ix.de/zhhp.

Daniel Basler

arbeitet als Softwareentwickler bei der Solarlux GmbH. Seine Schwerpunkte liegen auf Cross-Platform-Apps, Android, JavaScript und Microsoft-Technologien.