edv-buchversand.de

Praxiseinstieg Large Language Models

Strategien und Best Practices für den Einsatz von ChatGPT und anderen LLMs

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

KAPITEL 2

Semantische Suche mit LLMs

In Kapitel 1 haben wir die Funktionsweise von Sprachmodellen (Language Models) und die Auswirkungen moderner LLMs auf NLP-Aufgaben wie Textklassifizierung, Textgenerierung und maschinelle Übersetzung untersucht. Eine weitere leistungsstarke Anwendung von LLMs hat in den letzten Jahren ebenfalls an Zugkraft gewonnen: die semantische Suche.

Vielleicht denken Sie jetzt, dass es an der Zeit ist, endlich zu lernen, wie man am besten mit ChatGPT und GPT-4 spricht, um optimale Ergebnisse zu erzielen – und damit beginnen wir im nächsten Kapitel. Versprochen. Bis dahin möchte ich Ihnen zeigen, was wir sonst noch auf dieser neuartigen Transformer-Architektur aufbauen können. Während generative Text-zu-Text-Modelle wie GPT an sich schon sehr beeindruckend sind, ist eine der vielseitigsten Lösungen, die KI-Firmen anbieten, die Möglichkeit, Text-Embeddings basierend auf leistungsstarken LLMs zu erzeugen.

Text-Embeddings sind eine Methode, um Wörter oder Phrasen als maschinenlesbare numerische Vektoren in einem mehrdimensionalen Raum darzustellen, im Allgemeinen auf der Grundlage ihrer kontextuellen Bedeutung. Dem liegt die Idee zugrunde, dass, wenn zwei Phrasen ähnlich sind (das Wort »ähnlich« untersuchen wir später in diesem Kapitel ausführlicher), die Vektoren, die diese Phrasen repräsentieren, einem bestimmten Maß (wie dem euklidischen Abstand) entsprechend nahe beieinanderliegen sollten und umgekehrt. Abbildung 2-1 zeigt ein Beispiel für einen einfachen Suchalgorithmus. Wenn ein Benutzer nach einem Artikel sucht, den er kaufen möchte – sagen wir, eine alte Sammelkarte von *Magic: The Gathering* –, könnte er einfach nach »A vintage magic card« suchen. Das System sollte dann diese Abfrage so einbetten, dass zwei nahe beieinanderliegende Text-Embeddings darauf hinweisen, dass die Ausdrücke, mit denen sie erzeugt wurden, ähnlich sind.

Diese Zuordnung von Text zu Vektoren kann man sich als eine Art Hash mit Bedeutung vorstellen. Allerdings können wir die Vektoren nicht wirklich in Text zurückverwandeln. Stattdessen repräsentieren sie den Text, der nun als zusätzlichen Vorteil die Fähigkeit besitzt, Punkte in ihrem codierten Zustand zu vergleichen.

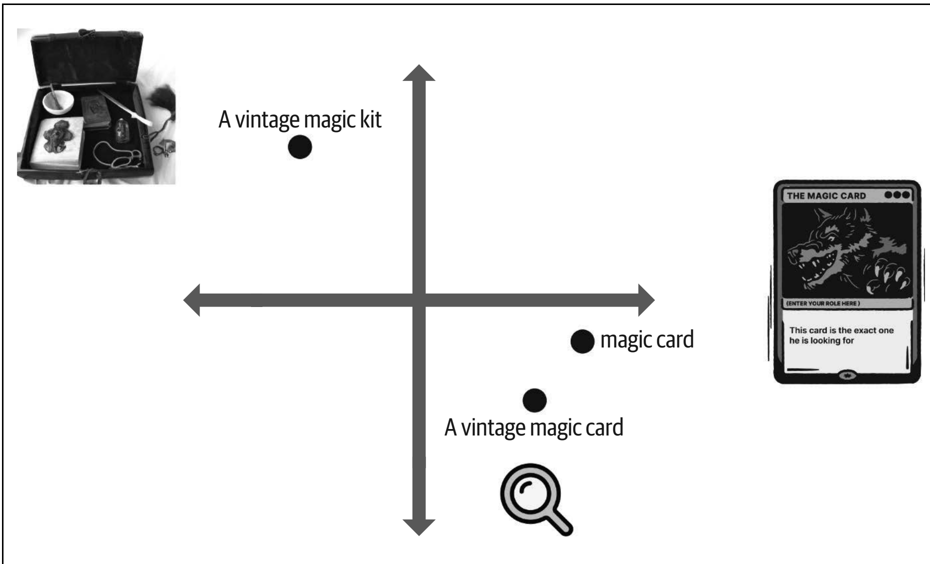

Abbildung 2-1: Vektoren, die ähnliche Ausdrücke darstellen, sollten nahe beieinanderliegen, und solche, die unähnliche Ausdrücke verkörpern, sollten weit voneinander entfernt sein. In diesem Fall könnte ein Benutzer, der eine Sammelkarte haben möchte, nach »A vintage magic card« suchen. Ein geeignetes semantisches Suchsystem sollte die Abfrage so einbetten, dass sie in der Nähe relevanter Ergebnisse (wie »magic card«) und weit entfernt von nicht relevanten Artikeln (wie »A vintage magic kit« – alter Zauberkasten) landet, selbst wenn sie bestimmte Schlüsselwörter gemeinsam haben. (Die Bilder wurden mit DALL·E 2 erzeugt.)

LLM-fähige Text-Embeddings erlauben es uns, den semantischen Wert von Wörtern und Ausdrücken zu erfassen, und zwar über ihre reine Syntax oder Rechtschreibung hinaus. Wir können uns auf das Vortraining und Feintuning von LLMs stützen, um praktisch unbeschränkte Anwendungen darauf aufzubauen, indem wir diese reichhaltige Informationsquelle über den Sprachgebrauch nutzen.

Dieses Kapitel führt in die Welt der semantischen Suche unter Verwendung von LLMs ein, um zu untersuchen, wie sich mit LLMs leistungsstarke Werkzeuge zum Abrufen und Analysieren von Informationen erzeugen lassen. In Kapitel 3 erstellen wir einen Chatbot, der auf GPT-4 aufsetzt und ein vollständig realisiertes semantisches Suchsystem nutzt, das wir in diesem Kapitel aufbauen werden.

Halten wir uns also nicht weiter mit Vorreden auf, sondern fangen wir ohne Umschweife an.

Die Aufgabe

Eine herkömmliche Suchmaschine nimmt in der Regel das, was Sie eingeben, und gibt Ihnen dann eine Reihe von Links zu Websites oder Einträgen, die diese Wörter oder Permutationen der eingegebenen Zeichen enthalten. Wenn Sie zum Beispiel auf der Suche nach dem Sammelkartenspiel »Magic: The Gathering« sind und auf einem

Marktplatz »vintage magic the gathering cards« eingeben, werden Ihnen Artikel angezeigt, deren Titel oder Beschreibung Kombinationen dieser Wörter enthält. Diese Art zu suchen ist durchaus üblich, aber nicht immer die beste. Ich könnte zum Beispiel alle Zauberkästen bekommen, mit denen ich lernen kann, wie man ein Kaninchen aus dem Hut zieht. Lustig, aber nicht das, wonach ich gesucht habe.

Die Begriffe, die Sie in eine Suchmaschine eingeben, stimmen nicht immer *genau* mit den Wörtern überein, die in den von Ihnen gewünschten Artikeln verwendet werden. Es könnte sein, dass die Wörter in der Suchanfrage zu allgemein sind, was zu einer Menge von zusammenhanglosen Treffern führt. Dieses Problem geht oft über die lediglich unterschiedlichen Wörter in den Ergebnissen hinaus; dieselben Wörter können eine andere Bedeutung haben als das, wonach gesucht wurde. An dieser Stelle kommt die semantische Suche ins Spiel, wie in dem bereits erwähnten Szenario *Magic: The Gathering* gezeigt.

Asymmetrische semantische Suche

Ein *semantisches Suchsystem* ist in der Lage, die Bedeutung und den Kontext Ihrer Suchanfrage zu verstehen und sie mit der Bedeutung und dem Kontext der abrufbaren Dokumente abzugleichen. Ein derartiges System kann relevante Ergebnisse in einer Datenbank finden, ohne sich auf eine genaue Übereinstimmung mit Schlüsselwörtern oder n-Grammen verlassen zu müssen. Stattdessen stützt es sich auf ein vortrainiertes LLM, um die Nuancen der Abfrage und der Dokumente zu verstehen (siehe Abbildung 2-2).

Abbildung 2-2: Eine herkömmliche stichwortbasierte Suche könnte einen alten Zauberkasten (A Vintage Magic Kit) mit der gleichen Gewichtung wie den eigentlich gesuchten Artikel einstufen, während ein semantisches Suchsystem das eigentliche Konzept, nach dem wir suchen, verstehen kann.

Der asymmetrische Teil der symmetrischen semantischen Suche bezieht sich auf die Tatsache, dass ein Ungleichgewicht besteht zwischen der semantischen Information (im Grunde der Größe) der Eingabeanfrage und der Dokumente bzw. Informationen, die das Suchsystem abrufen muss. Prinzipiell ist das eine viel kürzer als das andere. Zum Beispiel würde ein Suchsystem, das versucht, »magic the gathering cards« mit längeren Absätzen von Artikelbeschreibungen auf einem Marktplatz abzugleichen, als asymmetrisch betrachtet werden. Die Suchanfrage mit vier Wörtern enthält viel weniger Informationen als die Absätze, aber dennoch ist es das, was wir vergleichen müssen.

Asymmetrische semantische Suchsysteme können sehr genaue und relevante Suchergebnisse liefern, selbst wenn Sie nicht genau die richtigen Wörter in Ihrer Suche verwenden. Die Systeme stützen sich auf die Erkenntnisse der LLMs und nicht darauf, dass die Benutzerin genau weiß, nach welcher Nadel im Heuhaufen sie suchen muss.

Natürlich vereinfache ich die herkömmliche Methode sehr stark. Es gibt viele Möglichkeiten, die Suche performanter zu machen, ohne zu einem komplexeren LLM-Ansatz zu wechseln, und reine semantische Suchsysteme sind nicht immer die Antwort. Sie sind nicht einfach »der bessere Weg, eine Suche durchzuführen«. Semantische Algorithmen haben ihre eigenen Unzulänglichkeiten, darunter die folgenden:

- Sie können übermäßig empfindlich auf kleine Abweichungen im Text reagieren, beispielsweise auf Unterschiede in der Groß-/Kleinschreibung oder Zeichensetzung.
- Sie haben Schwierigkeiten mit nuancierten Konzepten wie Sarkasmus oder Ironie, die auf lokalem kulturellem Wissen beruhen.
- Aus rechentechnischer Sicht kann es teurer als die herkömmliche Methode sein, sie zu implementieren und zu pflegen, vor allem wenn es sich um ein selbst entwickeltes System mit vielen Open-Source-Komponenten handelt.

Semantische Suchsysteme können in bestimmten Kontexten ein wertvolles Werkzeug sein. Beginnen wir also gleich damit, unsere Lösung zu konstruieren.

Die Lösung im Überblick

Der allgemeine Ablauf unseres asymmetrischen semantischen Suchsystems umfasst folgende Schritte:

- Teil I: Einlesen von Dokumenten (siehe Abbildung 2-3)
    1. Sammeln von Dokumenten für das Embedding (z.B. Absätze mit Artikelbeschreibungen).
    2. Erstellen von Text-Embeddings, um semantische Informationen zu codieren.
    3. Speichern der Embeddings in einer Datenbank, um sie später auf eine Anfrage hin abrufen zu können.

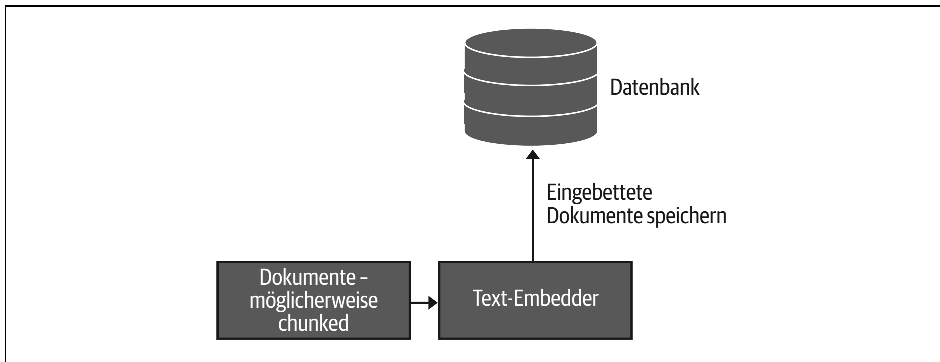

Abbildung 2-3: Detailansicht von Teil I: Um Dokumente zu speichern, werden sie zunächst vorverarbeitet, dann eingebettet und schließlich in einer Datenbank abgelegt.

- Teil II: Dokumente abrufen (siehe Abbildung 2-4)
    1. Der Benutzer hat eine Anfrage, die vorverarbeitet und bereinigt werden kann (z. B. bei einem Benutzer, der nach einem Artikel sucht).
    2. Kandidatendokumente werden anhand von ähnlichen Embeddings (z. B. nach euklidischem Abstand) abgerufen.
    3. Die Kandidatendokumente bei Bedarf neu einstufen (mehr dazu später).
    4. Die endgültigen Suchergebnisse an den Benutzer zurückgeben.

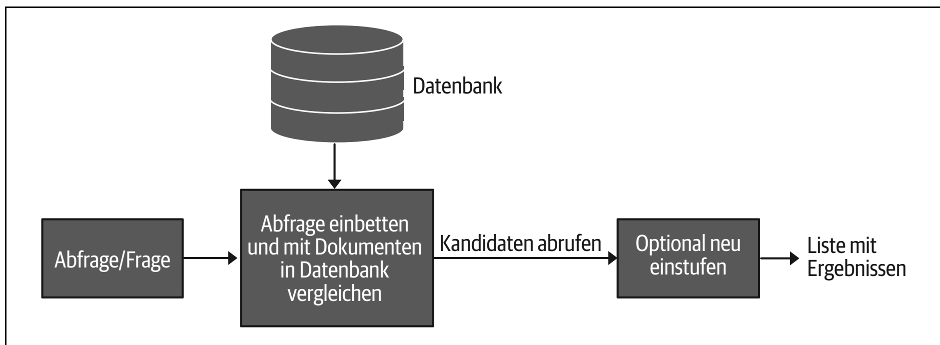

Abbildung 2-4: Detailansicht von Teil II: Um Dokumente abzurufen, müssen wir unsere Abfrage mit dem gleichen Embedding-Schema einbetten, das wir für die Dokumente verwendet haben, sie mit den zuvor gespeicherten Dokumenten vergleichen und dann das beste (am nächsten liegende) Dokument zurückgeben.

Die Komponenten

Sehen wir uns nun die einzelnen Komponenten genauer an, um zu verstehen, welche Entscheidungen wir treffen und welche Überlegungen wir dabei anstellen müssen.

Engines für Text-Embeddings

Das Herzstück jedes semantischen Suchsystems ist das Modul für Text-Embeddings. Diese Komponente übernimmt ein Textdokument, ein einzelnes Wort oder einen Satzteil und wandelt diese Eingabe in einen Vektor um. Der Vektor ist für diesen Text eindeutig und sollte die kontextbezogene Bedeutung des Satzteils erfassen.

Die Wahl des Moduls für das Text-Embedding ist entscheidend, da es die Qualität der Vektordarstellung des Texts bestimmt. Es gibt viele Möglichkeiten, wie wir mit LLMs vektorisieren können, sowohl als Open-Source- als auch als Closed-Source-Code. Um schneller loslegen zu können, werden wir hier für unsere Zwecke das Closed-Source-Produkt *Embeddings* von OpenAI verwenden. In einem späteren Abschnitt komme ich auf einige Open-Source-Optionen zu sprechen.

Das leistungsfähige Tool *Embeddings* von OpenAI ist in der Lage, hoch qualitative Vektoren schnell bereitzustellen. Allerdings ist es ein Closed-Source-Produkt, d.h., wir haben nur begrenzte Kontrolle über seine Implementierung und mögliche Verzerrungen. Insbesondere haben wir bei Closed-Source-Produkten möglicherweise keinen Zugang zu den zugrunde liegenden Algorithmen, was die Fehlerbehebung bei Problemen erschweren kann.

Was macht Textabschnitte »ähnlich«?

Sobald wir unseren Text in Vektoren umgewandelt haben, müssen wir eine mathematische Darstellung finden, um festzustellen, ob Textabschnitte einander »ähnlich« sind. Die Kosinus-Ähnlichkeit ist eine Methode, um zu messen, wie ähnlich sich zwei Dinge sind. Dabei liefert der Winkel zwischen zwei Vektoren ein Maß dafür, wie ausgeprägt zwei Vektoren in die gleiche Richtung zeigen. Wenn die Vektoren genau in die gleiche Richtung zeigen, ist die Kosinus-Ähnlichkeit gleich 1. Stehen die Vektoren senkrecht aufeinander (mit einem Winkel von 90 Grad), ist der Wert 0. Und zeigen sie in entgegengesetzte Richtungen, ist die Kosinus-Ähnlichkeit –1. Die Größe der Vektoren spielt keine Rolle, nur ihre Orientierung ist entscheidend.

Abbildung 2-5 zeigt, wie der Vergleich per Kosinus-Ähnlichkeit dabei hilft, Dokumente für eine gegebene Abfrage abzurufen.

Wir könnten auch auf andere Ähnlichkeitsmetriken zurückgreifen, beispielsweise das Punktprodukt oder den euklidischen Abstand. Allerdings haben die OpenAI-Embeddings eine spezielle Eigenschaft. Die Größen (Längen) ihrer Vektoren sind auf die Länge 1 normiert. Das bedeutet im Grunde genommen, dass wir mathematisch an zwei Fronten profitieren:

- Die Kosinus-Ähnlichkeit ist identisch mit dem Punktprodukt.
- Die Kosinus-Ähnlichkeit und der euklidische Abstand ergeben die gleiche Rangfolge.

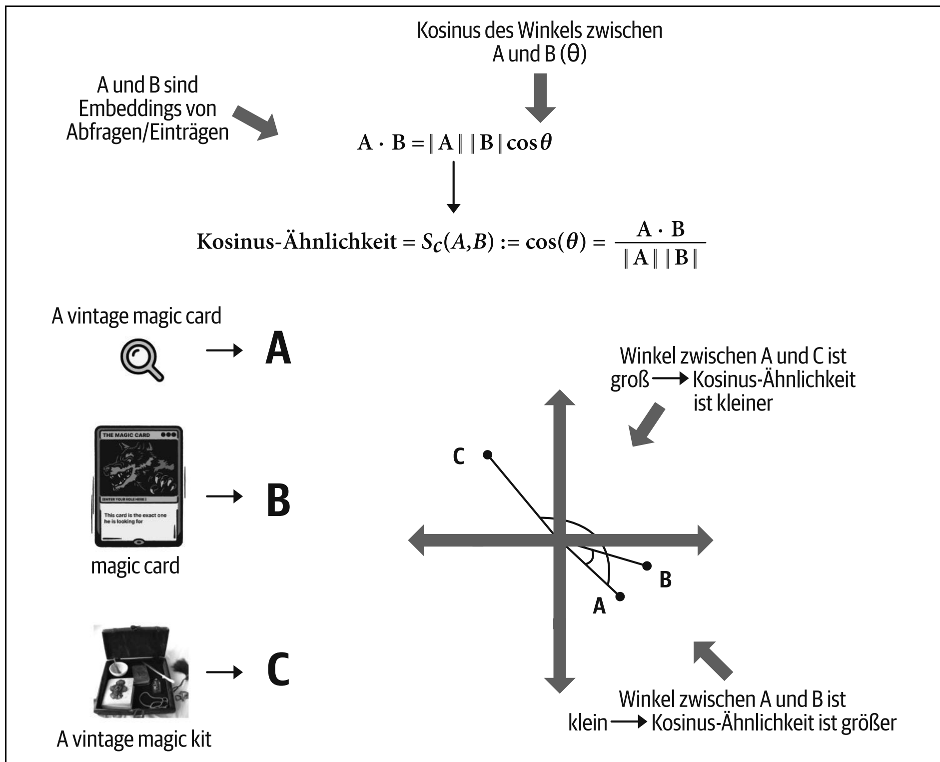

Abbildung 2-5: In einem idealen Szenario der semantischen Suche bietet die Kosinus-Ähnlichkeit (siehe Formel ganz oben) eine rechnerisch effiziente Methode, um Textabschnitte auf einer Skala zu vergleichen, da die Embeddings so abgestimmt sind, dass semantisch ähnliche Textabschnitte nahe beieinanderliegen (unten). Wir beginnen mit dem Embedding aller Einträge – einschließlich der Abfrage (links unten) – und überprüfen dann den Winkel zwischen ihnen. Je kleiner der Winkel ist, desto größer ist die Kosinus-Ähnlichkeit (rechts unten).

Normalisierte Vektoren (die alle die Größe 1 haben) eignen sich hervorragend, weil wir mit einer einfachen Kosinus-Berechnung feststellen können, wie nahe zwei Vektoren beieinanderliegen, und damit auch über die Kosinus-Ähnlichkeit sehen, wie nahe sich zwei Textabschnitte semantisch sind.

Embedding-Engines von OpenAI

Um Embeddings von OpenAI zu bekommen, genügen ein paar Zeilen Code (siehe Beispiel 2-1). Wie schon erwähnt, beruht das gesamte System auf einem Embedding-Mechanismus, der semantisch ähnliche Einträge nahe beieinander platziert, sodass die Kosinus-Ähnlichkeit groß ist, wenn die Einträge tatsächlich ähnlich sind. Diese Embeddings könnten wir nach verschiedenen Methoden erstellen, doch im Moment verlassen wir uns auf die *Embedding-Engines* von OpenAI, die uns diese Arbeit abnehmen. Wir werden die neueste Engine des Unternehmens verwenden, die es für die meisten Anwendungsfälle empfiehlt.

Beispiel 2-1: Text-Embeddings von OpenAI abrufen

```
# Die erforderlichen Module importieren, um das Skript auszuführen
import openai
from openai.embeddings_utils import get_embeddings, get_embedding

# Den OpenAI-Schlüssel auf den in der Umgebungsvariablen
# 'OPENAI_API_KEY' gespeicherten Wert setzen
openai.api_key = os.environ.get('OPENAI_API_KEY')

# Die für Text-Embeddings zu verwendende Engine festlegen
ENGINE = 'text-embedding-ada-002'

# Die Vektordarstellung des gegebenen Texts mit der angegebenen Engine erzeugen
embedded_text = get_embedding('I love to be vectorized', engine=ENGINE)

# Die Länge des resultierenden Vektors kontrollieren, um sicherzustellen, dass es
# sich um die erwartete Größe (1536) handelt.
len(embedded_text) == '1536'
```

OpenAI bietet mehrere Optionen für Embedding-Engines, die sich für Text-Embeddings eignen. Jede Engine kann für verschiedene Genauigkeitsebenen ausgelegt und für verschiedene Arten von Textdaten optimiert sein. Zum Entstehungszeitpunkt dieses Buchs war die im Codeblock verwendete Engine die neueste Version und auch diejenige, die OpenAI empfiehlt.

Darüber hinaus ist es möglich, mehrere Textabschnitte auf einmal an die Funktion `get_embeddings` zu übergeben, die Embeddings für alle Abschnitte in einem einzigen API-Aufruf generieren kann. Dies ist möglicherweise effizienter, als `get_embedding` mehrmals für jeden einzelnen Text aufzurufen. Ein Beispiel dafür werden Sie später sehen.

Alternative Open-Source-Embeddings

Während zum einen OpenAI und andere Unternehmen leistungsstarke Produkte für das Text-Embedding anbieten, sind zum anderen auch mehrere Open-Source-Alternativen für das Text-Embedding verfügbar. Eine beliebte Option ist der Bi-Encoder mit BERT, ein leistungsstarker auf Deep Learning basierender Algorithmus, der bei einer Reihe von Aufgaben zur Verarbeitung natürlicher Sprache nachweislich Ergebnisse nach dem Stand der Technik liefert. Vortrainierte Bi-Encoder sind in vielen Open-Source-Repositorys zu finden, darunter auch die Bibliothek *Sentence Transformers*, die von Haus aus vortrainierte Modelle für eine Vielzahl von NLP-Aufgaben bereitstellt.

Bei einem Bi-Encoder werden zwei BERT-Modelle trainiert: eines, um den Eingabetext zu codieren, und das andere, um den Ausgabetext zu codieren (siehe Abbildung 2-6). Beide Modelle werden gleichzeitig mit einem großen Korpus von Textdaten trainiert, wobei das Ziel darin besteht, die Ähnlichkeit zwischen entsprechenden Paaren von Eingabe- und Ausgabetext zu maximieren. Die resultierenden Embeddings erfassen die semantische Beziehung zwischen dem Eingabe- und dem Ausgabetext.

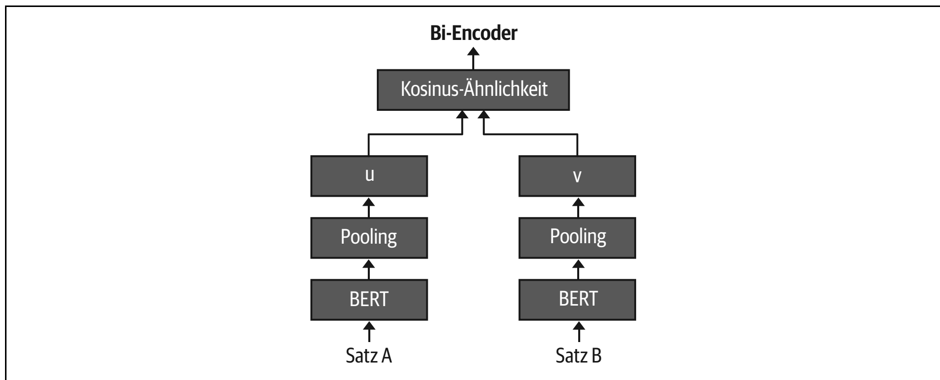

Abbildung 2-6: Ein Bi-Encoder wird auf einzigartige Weise trainiert, wobei zwei Klone eines einzelnen LLM parallel trainiert werden, um Ähnlichkeiten zwischen Dokumenten zu lernen. Zum Beispiel kann ein Bi-Encoder lernen, Fragen mit Absätzen zu assoziieren, sodass sie in einem Vektorraum nahe beieinander erscheinen.

Beispiel 2-2 zeigt ein Beispiel für das Embedding von Text mit einem vortrainierten Bi-Encoder aus dem Paket sentence_transformers.

Beispiel 2-2: Text-Embeddings von einem vortrainierten Open-Source-Bi-Encoder abrufen

```
# Die Bibliothek SentenceTransformer importieren
from sentence_transformers import SentenceTransformer

# Ein SentenceTransformer-Modell mit dem vortrainierten
# Modell 'multi-qa-mpnet-base-cos-v1' initialisieren
pre-trained model
model = SentenceTransformer(
  'sentence-transformers/multi-qa-mpnet-base-cos-v1')

# Eine Liste mit Dokumenten definieren, für die Embeddings
# zu generieren sind
docs = [
          "Around 9 million people live in London",
          "London is known for its financial district"
       ]

# Vektor-Embeddings für die Dokumente generieren
doc_emb = model.encode(
    docs,                  # Unsere Dokumente (iterierbare Strings)
    batch_size=32,         # Die Embeddings in dieser Größe stapeln
    show_progress_bar=True  # Eine Fortschrittsleiste anzeigen
)

# Die Embeddings haben die Form (2, 768), d.h. eine Länge von 768 und zwei
# generierte Embeddings.
doc_emb.shape  #  == (2, 768)
```

Dieser Code erzeugt eine Instanz der Klasse `SentenceTransformer`, die mit dem vortrainierten Modell `multi-qa-mpnet-base-cos-v1` initialisiert wird. Dieses Modell wurde für Multitasking-Lernen entwickelt, insbesondere für Aufgaben wie Beantwortung von Fragen (Question Answering) und Textklassifizierung. Da es mit asymmetrischen Daten vortrainiert wurde, wissen wir, dass es sowohl kurze Abfragen als auch

lange Dokumente verarbeiten kann und in der Lage ist, sie entsprechend zu vergleichen. Wir rufen die Funktion `encode` der Klasse `SentenceTransformer` auf, um Vektor-Embeddings für die Dokumente zu erzeugen, und speichern die resultierenden Embeddings in der Variablen `doc_emb`. Verschiedene Algorithmen schneiden bei verschiedenen Arten von Textdaten möglicherweise besser ab und verwenden unterschiedliche Vektorgrößen. Die Wahl des Algorithmus kann einen erheblichen Einfluss auf die Qualität der resultierenden Embeddings haben. Zudem erfordern Open-Source-Algorithmen möglicherweise mehr Anpassungen und Feintuning als Closed-Source-Produkte, bieten aber auch mehr Flexibilität und mehr Kontrolle über den Embedding-Prozess.

Chunking von Dokumenten

Nachdem wir unsere Engine für Text-Embeddings eingerichtet haben, müssen wir uns mit der Herausforderung befassen, große Dokumente einzubetten. Oftmals ist es nicht praktikabel, ganze Dokumente als einen einzigen Vektor einzubetten, insbesondere wenn es sich um lange Dokumente wie Bücher oder Forschungsarbeiten handelt. Eine Lösung für dieses Problem ist das Chunking von Dokumenten, d. h. das Aufteilen eines großen Dokuments in kleinere, besser handhabbare Teile für das Embedding.

Max-Token-Window-Chunking

Ein Ansatz für das Chunking von Dokumenten ist das Max-Token-Window-Chunking. Als eine der am einfachsten zu implementierenden Methoden teilt sie das Dokument in Chunks einer gegebenen maximalen Größe auf. Wenn wir zum Beispiel ein Token-Fenster von 500 festlegen, sollte jeder Chunk etwas weniger als 500 Token umfassen. Erstellt man die Chunks mit ungefähr der gleichen Größe, trägt das auch dazu bei, das System konsistenter zu machen.

Häufig befürchtet man bei dieser Methode, dass versehentlich einige wichtige Textteile zwischen den Chunks abgeschnitten werden und so der Kontext zerrissen wird. Um dieses Problem zu entschärfen, können wir überlappende Fenster mit einer bestimmten Anzahl von Token so festlegen, dass es zwischen den Chunks gemeinsame Token gibt. Natürlich bringt dies eine gewisse Redundanz mit sich, doch ist das im Interesse einer besseren Genauigkeit und Latenz oft in Ordnung.

Schauen wir uns ein Beispiel für das Chunking mit überlappenden Fenstern anhand eines Beispieltexts an (siehe Beispiel 2-3). Zunächst lesen wir ein großes Dokument ein. Wie wäre es mit einem Buch, das ich kürzlich geschrieben habe und das mehr als 400 Seiten umfasst?

Beispiel 2-3: Ein ganzes Lehrbuch einlesen

```
# Eine PDF-Datei mithilfe der Bibliothek PyPDF2 lesen
import PyPDF2

# Die PDF-Datei im binären Modus nur zum Lesen öffnen
with open('../data/pds2.pdf', 'rb') as file:
```

```
    # Ein PDF-reader-Objekt erstellen
    reader = PyPDF2.PdfReader(file)

    # Einen leeren String initialisieren, um den Text aufzunehmen
    principles_of_ds = ''

    # Die einzelnen Seiten in der PDF-Datei in einer Schleife durchlaufen
    for page in tqdm(reader.pages):

        # Den Text aus der Seite extrahieren
        text = page.extract_text()
        # Den Anfangspunkt des Texts suchen, den wir extrahieren wollen
        # In diesem Fall extrahieren wir den Text ab dem String ' ]'
        principles_of_ds += '\n\n' + text[text.find(' ]')+2:]

# Alle führenden oder nachgestellten Whitespace-Zeichen aus dem resultierenden
# String entfernen
principles_of_ds = principles_of_ds.strip()
```

Als Nächstes teilen wir dieses Dokument in Chunks, die höchstens eine bestimmte Token-Größe haben dürfen (siehe Beispiel 2-4).

Beispiel 2-4: Das Lehrbuch mit und ohne Überlappung in Chunks aufteilen

```
# Funktion, um den Text in Chunks einer maximalen Anzahl von Token aufzuteilen,
# inspiriert von OpenAI
def overlapping_chunks(text, max_tokens = 500, overlapping_factor = 5):
    '''
    max_tokens: Anzahl der Token, die wir pro Chunk haben wollen
    overlapping_factor: Anzahl der Sätze, mit denen jeder Chunk beginnen soll, der sich
        mit dem vorherigen Chunk überschneidet
    '''

    # Den Text anhand von Interpunktionszeichen trennen
    sentences = re.split(r'[.?!]', text)

    # Die Anzahl der Token für jeden Satz ermitteln
    n_tokens = [len(tokenizer.encode(" " + sentence)) for sentence in sentences]

    chunks, tokens_so_far, chunk = [], 0, []

    # Schleife durch die Sätze und Token, die in einem Tupel zusammengefasst sind
    for sentence, token in zip(sentences, n_tokens):

        # Wenn die Anzahl der bisherigen Token plus die Anzahl der Token im aktuellen
        # Satz größer als die maximale Anzahl der Token ist, dann den Chunk zur Liste
        # der Chunks hinzufügen und die Werte für die bisherigen Chunks und Token
        # zurücksetzen.
        if tokens_so_far + token > max_tokens:
            chunks.append(". ".join(chunk) + ".")
            if overlapping_factor > 0:
                chunk = chunk[-overlapping_factor:]
                tokens_so_far = sum([len(tokenizer.encode(c)) for c in chunk])
            else:
                chunk = []
                tokens_so_far = 0
        # Wenn die Anzahl der Token im aktuellen Satz größer als die maximale
        # Anzahl der Token ist, zum nächsten Satz übergehen.
```

```
        if token > max_tokens:
            continue

        # Andernfalls den Satz zum Chunk hinzufügen und die Anzahl der Token
        # zur Gesamtanzahl addieren.
        chunk.append(sentence)
        tokens_so_far += token + 1

    return chunks

split = overlapping_chunks(principles_of_ds, overlapping_factor=0)
avg_length = sum([len(tokenizer.encode(t)) for t in split]) / len(split)
print(f'non-
overlapping chunking approach has {len(split)} documents with average length {avg_
length:.1f} tokens')
```
non-overlapping chunking approach has 286 documents with average length 474.1 tokens

```
# mit 5 überlappenden Sätzen pro Chunk
split = overlapping_chunks(principles_of_ds, overlapping_factor=5)
avg_length = sum([len(tokenizer.encode(t)) for t in split]) / len(split)
print(f'overlapping chunking approach has {len(split)} documents with average length
{avg_length:.1f} tokens')
```
overlapping chunking approach has 391 documents with average length 485.4 tokens

Mit Überlappung steigt die Anzahl der Dokument-Chunks, die aber alle ungefähr gleich groß sind. Je höher der Überlappungsfaktor, desto mehr Redundanz bringen wir in das System ein. Die Max-Token-Window-Methode berücksichtigt jedoch nicht die natürliche Struktur des Dokuments, und das kann dazu führen, dass die Informationen zwischen Chunks und Chunks mit überlappenden Informationen aufgeteilt werden, was das Abrufsystem verwirrt.

Benutzerdefinierte Begrenzerzeichen suchen. Um unsere Chunking-Methode zu unterstützen, könnten wir nach benutzerdefinierten natürlichen Trennzeichen wie zum Beispiel Seitenumbrüchen in einem PDF-Dokument oder Zeilenschaltungen zwischen Absätzen suchen. Für ein bestimmtes Dokument würden wir natürliche Whitespace-Zeichen innerhalb von Text identifizieren und diesen heranziehen, um sinnvollere Texteinheiten zu bilden. Diese landen dann in Dokument-Chunks, die schließlich eingebettet werden (siehe Abbildung 2-7).

Suchen wir nun nach den üblichen Whitespace-Zeichen im Lehrbuch (siehe Beispiel 2-5).

Beispiel 2-5: Chunking des Lehrbuchs anhand von natürlichen Whitespace-Zeichen

```
# Die Bibliotheken Counter und re importieren
from collections import Counter
import re

# Alle Vorkommen von einem oder mehreren Leerzeichen in 'principles_of_ds' suchen
matches = re.findall(r'[\s]{1,}', principles_of_ds)

# Die 5 häufigsten Leerzeichen, die im Dokument auftreten
most_common_spaces = Counter(matches).most_common(5)
```

```
# Die häufigsten Leerzeichen mit ihren Häufigkeiten ausgeben
print(most_common_spaces)

[(' ', 82259),
 ('\n', 9220),
 ('  ', 1592),
 ('\n\n', 333),
 ('\n   ', 250)]
```

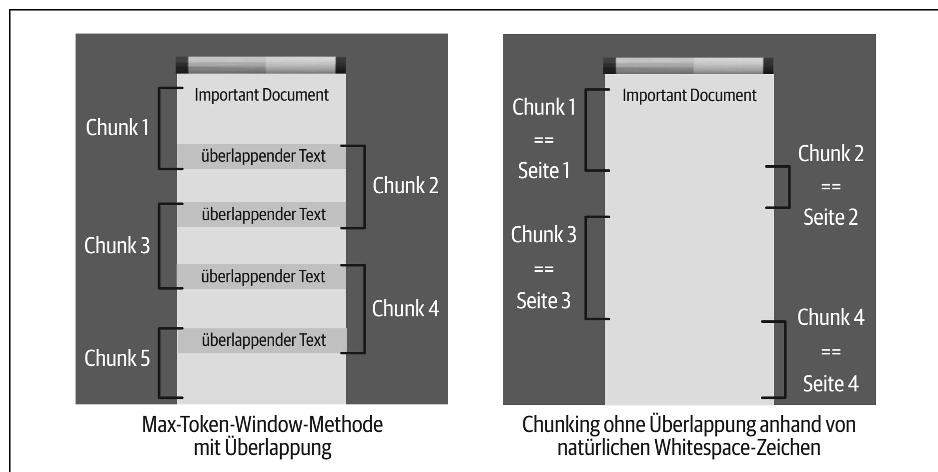

Abbildung 2-7: Max-Token-Chunking und Chunking anhand von natürlichen Whitespace-Zeichen lässt sich mit oder ohne Überlappung ausführen. Chunking anhand von natürlichen Whitespace-Zeichen führt tendenziell zu uneinheitlichen Chunk-Größen.

Das am häufigsten vorkommende doppelte Whitespace-Zeichen wird gebildet aus zwei Zeilenvorschubzeichen nacheinander. Auf diese Weise habe ich früher auch zwischen Seiten unterschieden. Das ergibt Sinn, weil das natürlichste Whitespace-Zeichen in einem Buch die Seitentrennung ist. In anderen Fällen haben wir vielleicht auch natürliche Whitespace-Zeichen zwischen Absätzen gefunden. Diese Methode ist sehr praxisorientiert und verlangt, dass man mit den Quelldokumenten vertraut ist und diese wirklich kennt.

Wir können auch auf mehr maschinelles Lernen setzen, um etwas kreativer zu werden, wenn es darum geht, wie wir Dokument-Chunks gestalten.

Semantische Dokumente per Clustering erstellen

Ein weiterer Chunking-Ansatz für Dokumente greift auf das Clustern zurück, um semantische Dokumente zu erstellen. Dabei entstehen neue Dokumente, indem kleine, semantisch ähnliche Informationsblöcke kombiniert werden (siehe Abbildung 2-8). Hier ist etwas Kreativität gefragt, da jede Änderung an den Dokument-Chunks den resultierenden Vektor verändert. Wir könnten eine Instanz des agglomerativen Clusterings aus scikit-learn verwenden, bei dem ähnliche Sätze oder Absätze gruppiert werden, um neue Dokumente zu erzeugen.

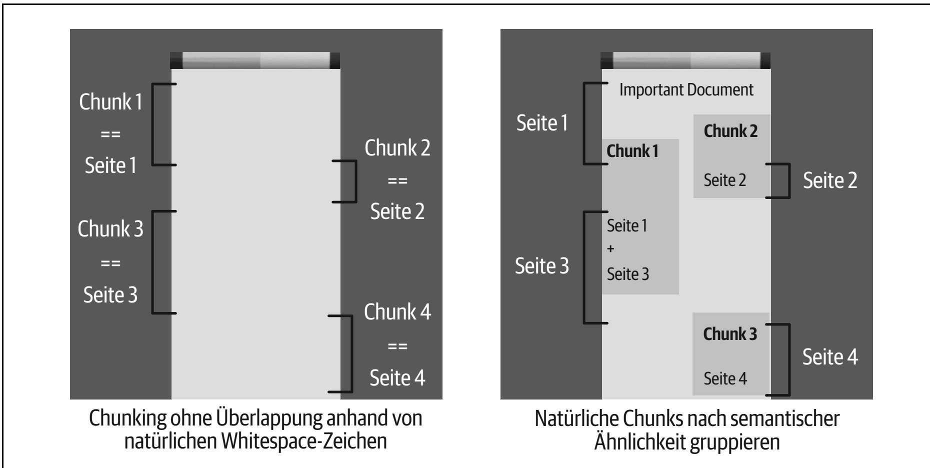

Abbildung 2-8: Alle Arten von Dokument-Chunks können wir mit einem separaten semantischen Clustering-System (rechts dargestellt) gruppieren, um gänzlich neue Dokumente zu erstellen, deren Informationsblöcke einander ähnlich sind.

Versuchen wir in Beispiel 2-6 nun, die im letzten Abschnitt gefundenen Chunks aus dem Lehrbuch zu clustern.

Beispiel 2-6: Clustering von Seiten des Dokuments nach semantischer Ähnlichkeit

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Angenommen sei eine Liste von Text-Embeddings namens 'embeddings'.
# Zuerst die Kosinus-Ähnlichkeitsmatrix für alle Embedding-Paare berechnen
cosine_sim_matrix = cosine_similarity(embeddings)

# Das AgglomerativeClustering-Modell instanziieren
agg_clustering = AgglomerativeClustering(
    n_clusters=None,         # Der Algorithmus bestimmt die optimale Anzahl von Clustern
                             # basierend auf den Daten.
    distance_threshold=0.1,  # Cluster werden gebildet, bis alle paarweisen Abstände
                             # zwischen den Clustern größer als 0.1 sind.
    affinity='precomputed',  # Wir stellen eine vorberechnete Abstandsmatrix
                             # (1 - similarity matrix) als Eingabe bereit.
    linkage='complete'       # Cluster bilden, indem iterativ die kleinsten Cluster
                             # basierend auf dem maximalen Abstand zwischen ihren
                             # Komponenten zusammengeführt werden.
)

# Das Modell an die Kosinus-Distanzmatrix (1 - similarity matrix) anpassen
agg_clustering.fit(1 - cosine_sim_matrix)

# Die Cluster-Bezeichnungen für jedes Embedding abrufen
cluster_labels = agg_clustering.labels_

# Die Anzahl der Embeddings in jedem Cluster ausgeben
unique_labels, counts = np.unique(cluster_labels, return_counts=True)
```

```
for label, count in zip(unique_labels, counts):
    print(f'Cluster {label}: {count} embeddings')

Cluster 0: 2 embeddings
Cluster 1: 3 embeddings
Cluster 2: 4 embeddings
...
```

Dieser Ansatz führt in der Regel zu Chunks, die semantisch kohärenter sind, aber darunter leiden, dass Teile des Inhalts nicht im Kontext mit dem umgebenden Text stehen. Er funktioniert gut, wenn bekannt ist, dass die Chunks, mit denen Sie beginnen, nicht unbedingt in Beziehung zueinander stehen – d.h., wenn die Chunks eher unabhängig voneinander sind.

Ganze Dokumente ohne Chunking verwenden. Alternativ ist es möglich, ganze Dokumente ohne Chunking zu verwenden. Dieser Ansatz ist die wahrscheinlich einfachste Option überhaupt, hat aber Nachteile, wenn das Dokument viel zu lang ist und wir beim Embedding des Texts an eine Grenze des Kontextfensters stoßen. Zudem kann es passieren, dass das Dokument mit fremden, ungleichartigen Kontextpunkten gefüllt wird, und die resultierenden Embeddings versuchen möglicherweise, zu viel zu codieren, wodurch die Qualität leidet. Diese Nachteile verstärken sich bei sehr großen (mehrseitigen) Dokumenten.

Wichtig ist, bei der Entscheidung für einen Ansatz zum Embedding von Dokumenten die Kompromisse zwischen dem Chunking und der Verwendung ganzer Dokumente zu berücksichtigen (siehe Tabelle 2-1). Sobald wir uns entschieden haben, wie wir unsere Dokumente chunken wollen, brauchen wir ein Zuhause für die von uns erstellten Embeddings. Auf lokaler Ebene können wir uns auf Matrixoperationen für schnelles Abrufen stützen. Da wir hier jedoch für die Cloud arbeiten, sollten wir uns unsere Datenbankoptionen ansehen.

Tabelle 2-1: Überblick über die verschiedenen Methoden beim Dokument-Chunking mit Vor- und Nachteilen

Chunking-Typ	Beschreibung	Vorteile	Nachteile
Max-Token-Window-Chunking ohne Überlappung	Das Dokument wird in Fenster fester Größe zerlegt, wobei jedes Fenster einen separaten Dokument-Chunk darstellt.	Einfach und leicht zu implementieren.	Kann Kontext zwischen Chunks abschneiden, was zu Informationsverlust führt.
Max-Token-Window-Chunking mit Überlappung	Das Dokument wird in überlappende Fenster fester Größe aufgeteilt.	Einfach und leicht zu implementieren.	Kann zu redundanten Informationen über verschiedene Chunks führen.
Chunking anhand natürlicher Trennzeichen	Natürliche Whitespace-Zeichen im Dokument werden verwendet, um die Grenzen jedes Chunks zu bestimmen.	Kann in bedeutungsvolleren Chunks resultieren, die natürlichen Lücken im Dokument entsprechen.	Es kann zeitaufwendig sein, die richtigen Begrenzungszeichen zu finden.

Tabelle 2-1: Überblick über die verschiedenen Methoden beim Dokument-Chunking mit Vor- und Nachteilen (Fortsetzung)

Chunking-Typ	Beschreibung	Vorteile	Nachteile
Clustering, um semantische Dokumente zu erzeugen	Ähnliche Dokument-Chunks werden kombiniert, um größere semantische Dokumente zu bilden.	Kann bedeutungsvollere Dokumente erzeugen, die die Gesamtbedeutung des Dokuments erfassen.	Erfordert mehr Rechenressourcen und kann komplexer zu implementieren sein.
Ganze Dokumente ohne Chunking	Das gesamte Dokument wird als einzelner Chunk behandelt.	Einfach und leicht zu implementieren.	Kann an einem Kontextfenster für das Embedding scheitern, wodurch fremder Kontext entsteht, der die Qualität des Embeddings beeinträchtigt.

Vektordatenbanken

Eine Vektordatenbank ist ein Dateispeichersystem, das speziell dafür ausgelegt ist, Vektoren schnell sowohl zu speichern als auch abzurufen. Eine derartige Datenbank ist nützlich, um die von einem LLM generierten Embeddings zu speichern, die die semantische Bedeutung unserer Dokumente oder Chunks von Dokumenten codieren und speichern. Da wir Embeddings in einer Vektordatenbank speichern, können wir effiziente Suchen nach den nächsten Nachbarn durchführen, um ähnliche Textabschnitte basierend auf ihrer semantischen Bedeutung abzurufen.

Pinecone

Pinecone ist eine Vektordatenbank, die für kleine bis mittelgroße Datensets konzipiert ist (normalerweise ideal für weniger als eine Million Einträge). Der Einstieg in Pinecone ist einfach und kostenlos, aber es gibt auch kostenpflichtige Versionen, die zusätzliche Features und erhöhte Skalierbarkeit bieten. Pinecone ist für eine schnelle Vektorsuche und -abfrage optimiert und eignet sich daher hervorragend für Anwendungen, die eine Suche mit geringer Latenz benötigen, wie zum Beispiel Empfehlungssysteme, Suchmaschinen und Chatbots.

Open-Source-Alternativen

Um eine Vektordatenbank für LLM- Embeddings zu erstellen, können Sie auch auf mehrere Open-Source-Alternativen zu Pinecode zurückgreifen. Eine dieser Alternativen ist Pgvector, eine PostgreSQL-Erweiterung, die Vektordatentypen unterstützt und schnelle Vektoroperationen bietet. Eine andere Option ist Weaviate, eine Cloud-native Open-Source-Vektordatenbank, die für Anwendungen des Machine Learning konzipiert ist. Weaviate unterstützt semantische Suche und lässt sich in andere Tools für maschinelles Lernen integrieren, wie zum Beispiel TensorFlow und PyTorch. ANNOY ist eine Open-Source-Bibliothek für die approximative Suche nach nächsten Nachbarn, die für große Datensets optimiert ist. Damit haben Sie die Möglichkeit, eine benutzerdefinierte Vektordatenbank zu erstellen, die auf spezifische Anwendungsfälle zugeschnitten ist.

Neueinstufen der abgerufenen Ergebnisse

Nachdem man für eine gegebene Abfrage mögliche Ergebnisse aus einer Vektordatenbank abgerufen hat, die auf Ähnlichkeitsvergleichen (z.B. Kosinus-Ähnlichkeit) beruhen, ist es oftmals sinnvoll, die Ergebnisse neu zu ordnen, um sicherzustellen, dass dem Benutzer die relevantesten Ergebnisse präsentiert werden (siehe Abbildung 2-9). Die Ergebnisse lassen sich zum Beispiel mit einem Cross-Encoder neu ordnen, d.h. mit einem Transformer-Modell, das Paare von Eingabesequenzen übernimmt und eine Einstufung vorhersagt, die angibt, wie relevant die zweite Sequenz für die erste ist. Wenn wir die Suchergebnisse mit einem Cross-Encoder umordnen, können wir den gesamten Abfragekontext berücksichtigen und nicht nur einzelne Schlüsselwörter. Natürlich bedeutet das einen gewissen Mehraufwand und verschlechtert unsere Latenz, könnte aber auch zu einer verbesserten Performance beitragen. In einem späteren Abschnitt vergleichen wir Methoden, die einen Cross-Encoder verwenden bzw. nicht verwenden, um festzustellen, wie diese Ansätze abschneiden.

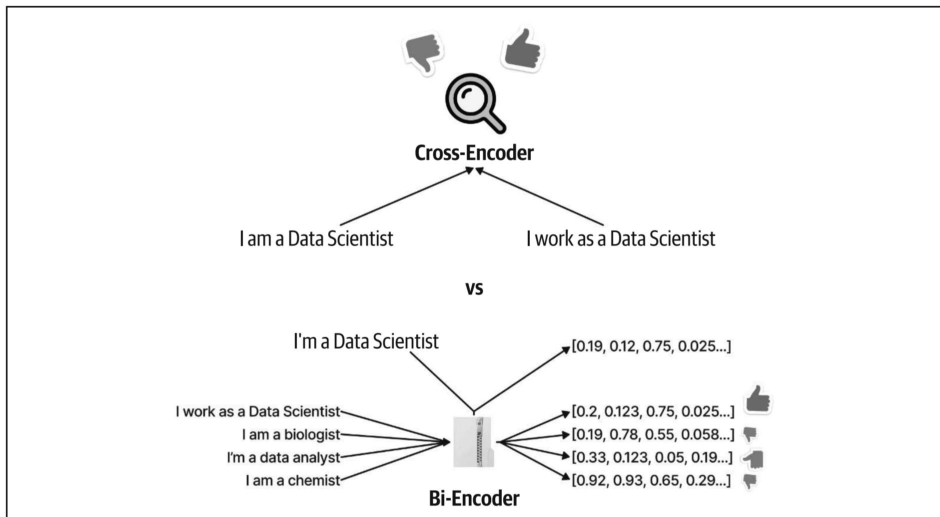

Abbildung 2-9: Ein Cross-Encoder übernimmt zwei Textabschnitte und gibt einen Ähnlichkeitswert aus, ohne ein vektorisiertes Format des Texts zurückzugeben. Ein Bi-Encoder bettet eine Reihe von Textteilen im Voraus in Vektoren ein und ruft sie dann später in Echtzeit bei einer Abfrage ab (z.B. bei der Suche nach »I'm a Data Scientist«).

Eine bekannte Quelle für Cross-Encoder-Modelle ist die Bibliothek *Sentence Transformers*, aus der die weiter oben erwähnten Bi-Encoder stammen. Wir können auch ein vortrainiertes Cross-Encoder-Modell auf unserem aufgabenspezifischen Datenset feintunen, um die Relevanz der Suchergebnisse zu verbessern und genauere Empfehlungen zu geben.

Eine weitere Option für das Neueinstufen von Suchergebnissen verwendet ein herkömmliches Abrufmodell wie BM25, das Ergebnisse nach der Häufigkeit der Abfragebegriffe im Dokument ordnet und die Begriffsnähe sowie die inverse Dokument-

häufigkeit berücksichtigt. Auch wenn BM25 nicht den gesamten Abfragekontext in Betracht zieht, kann es dennoch eine nützliche Methode sein, um Suchergebnisse neu zu ordnen und die allgemeine Relevanz der Ergebnisse zu verbessern.

API

Wir brauchen nun einen Ort, an dem wir alle diese Komponenten unterbringen können, damit die Benutzerinnen und Benutzer schnell, sicher und einfach auf die Dokumente zugreifen können. Zu diesem Zweck werden wir eine API erstellen.

FastAPI

FastAPI ist ein Web-Framework, das darauf ausgelegt ist, APIs mit Python schnell aufzubauen. Konzeptionell soll es sowohl schnell als auch einfach einzurichten sein, was es zu einer hervorragenden Wahl für unsere semantische Such-API macht. FastAPI validiert mithilfe der Bibliothek Pydantic die Anfrage- und Antwortdaten und nutzt auch den hochperformanten ASGI-Server *uvicorn*.

Ein FastAPI-Projekt lässt sich unkompliziert erstellen und erfordert eine nur minimale Konfiguration. Darüber hinaus bietet FastAPI eine automatische Dokumentationserzeugung nach dem OpenAPI-Standard, was das Erstellen von API-Dokumentation und Clientbibliotheken erleichtert. Beispiel 2-7 zeigt das Gerüst einer solchen Datei.

Beispiel 2-7: FastAPI-Gerüstcode

```
import hashlib
import os
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

openai.api_key = os.environ.get('OPENAI_API_KEY', '')
pinecone_key = os.environ.get('PINECONE_KEY', '')

# In Pincecone einen Index mit den erforderlichen Eigenschaften erstellen
def my_hash(s):
    # Den MD5-Hash des Eingabestrings als hexadezimalen String zurückgeben
    return hashlib.md5(s.encode()).hexdigest()

class DocumentInputRequest(BaseModel):
    # Eingabe in /document/ingest definieren

class DocumentInputResponse(BaseModel):
    # Ausgabe von /document/ingest definieren

class DocumentRetrieveRequest(BaseModel):
    # Eingabe in /document/retrieve definieren

class DocumentRetrieveResponse(BaseModel):
    # Ausgabe von /document/retrieve definieren
```

```
# API-Route zum Einlesen von Dokumenten
@app.post("/document/ingest", response_model=DocumentInputResponse)
async def document_ingest(request: DocumentInputRequest):
    # Anfragedaten parsen und chunken
    # Embeddings und Metadaten für jeden Chunk erstellen
    # Embeddings und Metadaten in Pinecone hochladen
    # Anzahl der hochgeladenen Chunks zurückgeben
    return DocumentInputResponse(chunks_count=num_chunks)

# API-Route zum Abrufen von Dokumenten
@app.post("/document/retrieve", response_model=DocumentRetrieveResponse)
async def document_retrieve(request: DocumentRetrieveRequest):
    # Anforderungsdaten parsen und Pinecone nach passenden Embeddings
    # abfragen. Ergebnisse basierend auf Strategie für Neueinstufung
    # (falls vorhanden) sortieren.
    # Eine Liste von Dokumentantworten zurückgeben.
    return DocumentRetrieveResponse(documents=documents)

if __name__ == "__main__":
    uvicorn.run("api:app", host="0.0.0.0", port=8000, reload=True)
```

Die vollständige Datei finden Sie im Code-Repository für dieses Buch (*https://github.com/sinanuozdemir/quickstart-guide-to-llms*).

Alles zusammen

Wir haben nun eine Lösung für alle unsere Komponenten. Werfen wir einen Blick darauf, wo wir in unserer Lösung stehen. Die fett geschriebenen Punkte sind neu, seit wir diese Lösung das letzte Mal skizziert haben:

- Teil I: Einlesen von Dokumenten
    1. Sammeln von Dokumenten für das Embedding – **Jedes Dokument chunken, um es besser handhaben zu können**.
    2. Erstellen von Text-Embeddings, um semantische Informationen zu codieren – **Embeddings von OpenAI**.
    3. Speichern der Embeddings in einer Datenbank, um sie später auf eine Anfrage hin abrufen zu können – **Pinecone**.
- Teil II: Dokumente abrufen
    1. Der Benutzer hat eine Anfrage, die vorverarbeitet und bereinigt werden kann – **FastAPI**.
    2. Kandidatendokumente abrufen – **Embeddings von OpenAI und Pinecone**.
    3. Die Kandidatendokumente bei Bedarf neu einstufen – **Cross-Encoder**.
    4. Die endgültigen Suchergebnisse an den Benutzer zurückgeben – **FastAPI**.

Mit all diesen Teilen werfen wir nun einen Blick auf unsere endgültige Systemarchitektur (siehe Abbildung 2-10).

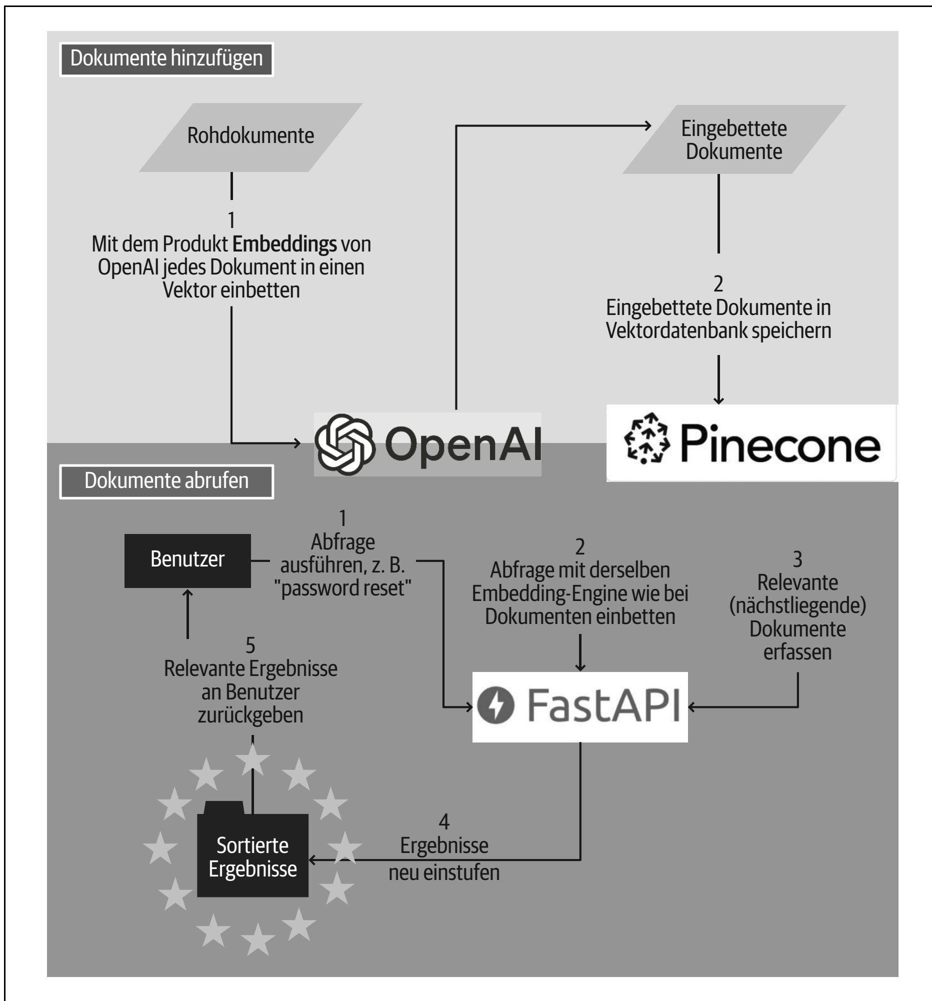

Abbildung 2-10: Unsere komplette Sucharchitektur mit zwei Closed-Source-Systemen (OpenAI und Pinecone) und einem Open-Source-API-Framework (FastAPI)

Wir verfügen nun über eine vollständige End-to-End-Lösung für unsere semantische Suche. Sehen wir uns an, wie gut das System mit einem Validierungsdatenset abschneidet.

Performance

Ich habe eine Lösung für das Problem der semantischen Suche skizziert, möchte aber auch darüber sprechen, wie man testen kann, wie die verschiedenen Komponenten zusammenwirken. Zu diesem Zweck verwenden wir ein schon bekanntes Datenset, um damit die Tests durchzuführen: das **BoolQ**-Datenset – ein Fragen-Antworten-Datenset für Ja-Nein-Fragen mit nahezu 16.000 Beispielen. Das Datenset enthält Paare von Fragen und Textpassagen (*Question*, *Passage*), die für eine ge-

gebene Frage anzeigen, ob die jeweilige Textstelle die beste Antwort auf die Frage wäre.

Tabelle 2-2 gibt einen Überblick über einige Versuche, die ich für dieses Buch durchgeführt und programmiert habe: Kombinationen aus Embeddings, Lösungen für die Neueinstufung und etwas Feintuning, um zu sehen, wie gut das System an zwei Fronten funktioniert:

1. *Performance*: Die Werte sind in der Spalte *Beste Ergebnisgenauigkeit* angegeben. Für jedes bekannte Paar von (Question, Passage) – Frage, Textstelle – in unserem BoolQ-Validierungssatz (3.270 Beispiele) testen wir, ob das beste Ergebnis des Systems die gewünschte Textstelle ist. Wir hätten auch eine andere Metrik heranziehen können. Die Bibliothek *Sentence Transformers* definiert weitere Metriken wie Bewertung der Rangfolge, Bewertung der Korrelation und viele mehr.
2. *Latenz*: Gibt an, wie lange es dauert, diese Beispiele mithilfe von Pinecone auszuführen. Für jeden Embedder habe ich den Index zurückgesetzt, neue Vektoren hochgeladen und im Arbeitsspeicher meines Laptops Cross-Encoder verwendet, um die Dinge einfach und standardisiert zu halten. Die in Minuten gemessene Latenzzeit gibt an, wie lange es dauert, den Validierungssatz des BoolQ-Datensets auszuführen.

Tabelle 2-2: Performanceergebnisse bei verschiedenen Kombinationen mit dem BoolQ-Validierungsdatenset

Embedder	Methode zur Neueinstufung	Beste Ergebnisgenauigkeit	Dauer der Auswertung (mit Pinecone)	Anmerkungen
OpenAI (Closed Source)	keine	0,85229	18 Minuten	Bei Weitem am einfachsten auszuführen.
OpenAI (Closed Source)	Cross-encoder/mmarco-mMini-LMv2-L12-H384-v1 (Open Source)	0,83731	27 Minuten	Etwa 50 % langsamer gegenüber der Ausführung ohne Cross-Encoder und ohne Verbesserung der Genauigkeit.
OpenAI (Closed Source)	Cross-encoder/ms-marco-MiniLM-L-12-v2 (Open Source)	0,84190	27 Minuten	Ein neuerer Cross-Encoder schnitt bei der Aufgabe besser ab, konnte aber trotzdem die Version nicht schlagen, die allein OpenAI verwendet.

Tabelle 2-2: Performanceergebnisse bei verschiedenen Kombinationen mit dem BoolQ-Validierungsdatenset (Fortsetzung)

Embedder	Methode zur Neueinstufung	Beste Ergebnisgenauigkeit	Dauer der Auswertung (mit Pinecone)	Anmerkungen
OpenAI (Closed Source)	Cross-encoder/ms-marco-MiniLM-L-12-v2 (Open Source und feingetunt mit zwei Epochen der BoolQ-Trainingsdaten)	0,84954	27 Minuten	Immer noch nicht besser als ausschließlich OpenAI, wobei aber die Genauigkeit des Cross-Encoders gegenüber der vorherigen Zeile verbessert wurde
Sentence-transformers/multi-qa-mp-net-base-cos-v1 (Open Source)	keine	0,85260	16 Minuten	Schlägt knapp das Standard-Embedding von OpenAI ohne Feintuning am Bi-Encoder. Diese Version ist auch etwas schneller, weil das Embedding durch Berechnung und nicht über die API durchgeführt wird.
Sentence-transformers/multi-qa-mp-net-base-cos-v1 (Open-Source)	Cross-encoder/ms-marco-MiniLM-L-12-v2 (Open Source und feingetunt für zwei Epochen auf BoolQ-Trainingsdaten)	0,84343	25 Minuten	Feingetunter Cross-Encoder zeigt keine Leistungssteigerung.

Einige Experimente habe ich nicht ausgeführt, darunter die folgenden:

1. Feintuning des Cross-Encoders mit mehr Epochen und mehr Investieren von mehr Zeit, um die optimalen Lernparameter zu ermitteln (z.B. Reduzierung der Gewichtungen, Planung der Lernrate).
2. Verwendung anderer OpenAI-Engines für das Embedding.
3. Feintuning eines Open-Source-Bi-Encoders mit dem Trainingsset.

Die Modelle, die ich für den Cross-Encoder und den Bi-Encoder verwendet habe, wurden beide speziell mit den Daten in einer Weise vortrainiert, die der asymmetrischen semantischen Suche ähnelt. Dies ist wichtig, weil der Embedder Vektoren sowohl für kurze Abfragen als auch für lange Dokumente erzeugen soll und um sie nahe beieinander zu platzieren, wenn sie in einer gewissen Beziehung zueinander stehen.

Da wir die Dinge einfach halten wollen, um unser Projekt in Gang zu bringen, verwenden wir nur den Embedder von OpenAI und führen keine Neueinstufung (Zeile 1) in unserer Anwendung durch. Wir sollten jetzt die Kosten betrachten, die mit FastAPI, Pinecone und OpenAI für Text-Embeddings entstehen.

Die Kosten von Closed-Source-Komponenten

Wir haben einige Komponenten im Spiel, und nicht alle sind kostenlos. Erfreulicherweise ist FastAPI ein Open-Source-Framework, für das keine Lizenzgebühren anfallen. Unsere Kosten bei FastAPI sind mit dem Hosting verbunden – und entfallen gegebenenfalls, was abhängig vom verwendeten Dienst ist. Ich bevorzuge Render, der sowohl eine kostenlose Version bietet als auch Preismodelle von 7 Dollar pro Monat für eine 100%ige Betriebszeit. Derzeit bietet Pinecone eine kostenlose Version mit einem Limit von 100.000 Embeddings und bis zu drei Indizes. Darüber hinaus richten sich die Gebühren nach der Anzahl der verwendeten Embeddings und Indizes. Der Standardtarif von Pinecone sieht 49 Dollar für bis zu eine Million Embeddings und zehn Indizes vor.

Die kostenlose Version der OpenAI-Dienste für das Text-Embedding ist auf 100.000 Anfragen pro Monat begrenzt. Darüber hinaus fallen 0,0004 Dollar pro 1.000 Token für die von uns verwendete Embedding-Engine (Ada-002) an. Geht man von durchschnittlich 500 Token pro Dokument aus, würden die Kosten pro Dokument 0,0002 Dollar betragen. Wenn wir beispielsweise eine Million Dokumente einbetten wollten, würden etwa 200 Dollar fällig.

Wenn wir ein System mit einer Million Embeddings aufbauen wollen und davon ausgehen, dass der Index einmal pro Monat mit komplett neuen Embeddings aktualisiert wird, berechnen sich die Gesamtkosten folgendermaßen:

Pinecone = 49 Dollar

OpenAI = 200 Dollar

FastAPI = 7 Dollar

Gesamtkosten = 49 Dollar + 200 Dollar + 7 Dollar = 256 Dollar pro Monat

Das ist eine schöne Binärzahl :-) Nicht beabsichtigt, aber dennoch amüsant.

Diese Kosten können schnell anwachsen, wenn das System skaliert. Es dürfte sich lohnen, Open-Source-Alternativen oder andere Strategien zu untersuchen, um die Kosten zu reduzieren – zum Beispiel Open-Source-Bi-Encoder für das Embedding einsetzen oder Pgvector als Vektordatenbank nutzen.

Zusammenfassung

Nachdem wir alle diese Komponenten berücksichtigt, unsere Groschen zusammengezählt und bei jedem Realisierungsschritt nach verfügbaren Alternativen geschaut haben, überlasse ich nun Ihnen das Feld. Viel Spaß beim Einrichten Ihres neuen semantischen Suchsystems. Schauen Sie sich unbedingt den vollständigen Code dafür im Code-Repository für dieses Buch an – einschließlich einer voll funktionsfähigen FastAPI-App mit einer Anleitung, wie man sie einsetzt. Experimentieren Sie nach Herzenslust, damit diese Lösung für Ihre bereichsspezifischen Daten so gut wie möglich funktioniert.

Seien Sie gespannt auf das nächste Kapitel, in dem wir auf dieser API aufbauend einen Chatbot basierend auf GPT-4 und unserem Abrufsystem kreieren werden.