Generative KI mit Python

RAG-Anwendungen und Agentensysteme mit Vektordatenbanken und LLMs

Kapitel 6
Vektordatenbanken

»Kannst du das fliegen?« – »Noch nicht«
– Trinity und Neo im Film »The Matrix« (1999)

Vielleicht können Sie sich an diese Szene im Film erinnern – sie hat bei mir einen bleibenden Eindruck hinterlassen. Nach dem Dialog zwischen Neo und Trinity, den Hauptfiguren des Sci-Fi-Films »The Matrix«, wird Wissen über das Fliegen von Hubschraubern in ihr Gehirn geladen. Im Film ist das möglich, weil sie sich in einer computersimulierten Welt befinden. Wäre es nicht cool, wenn wir das im echten Leben machen könnten?

In dieser Form können wir Wissen übertragen: zwar nicht in ein menschliches Gehirn (noch nicht), aber in einen Computer. In diesem Kapitel werden Sie lernen, wie es funktioniert. Die Technologie dahinter nennt sich *Vektordatenbanken*. Sie werden sehen, wie einfach es ist, einem Computer Wissen, zum Beispiel über ganze Bücher, innerhalb von Sekunden beizubringen.

Es gibt ein paar Schritte, die Sie dabei beachten sollten. Alle werden in diesem Kapitel ausführlich besprochen. Am Ende dieses Kapitels werden Sie in der Lage sein, Folgendes zu tun:

- erklären, was eine Vektordatenbank ist
- Daten aus verschiedensten Quellen laden
- verstehen, wie Daten in kleinere Elemente (*Chunks*) aufgeteilt werden können
- für Menschen verständlichen Text in computerlesbare Vektoren umwandeln
- Daten in einer Vektordatenbank speichern
- Daten aus einer Vektordatenbank laden

Zuerst die wichtigsten Dinge. Beginnen wir mit einem Überblick aus der »Hubschrauber-Perspektive«, bevor wir in die kniffligen Details am Boden eintauchen.

6.1 Einleitung

Lassen Sie uns zunächst herausfinden, was eine Vektordatenbank ist und warum sie überhaupt benötigt wird. Es gibt bereits klassische Datenbanken. Wahrscheinlich

kennen Sie Datenbanken wie SQL oder PostgreSQL oder haben zumindest von ihnen gehört. Warum können wir nicht einfach diese verwenden? Diese Datenbanken sind gut darin, *strukturierte Daten* zu speichern und abzurufen, wie Tabellen mit klar definierten Spalten und Zeilen. Aber das spiegelt nicht die große Vielfalt an Formaten wider, die Sie im Internet finden, wie Sie in Abbildung 6.1 sehen können. Dort finden Sie vor allem Textdaten, aber auch Bilder, Audiodateien oder Videos. All diese verschiedenen Formate sollten für KI-Zwecke nun aber ähnlich behandelt und in derselben Datenbank verwaltet werden.

Die bestehenden Datenbanken sind nicht in der Lage, all diese Datentypen effizient zu verarbeiten. Es wird etwas Neues benötigt, das mit allen Arten von Informationen umgehen kann. Und dieses neue Paradigma nennt man *Vektordatenbank*.

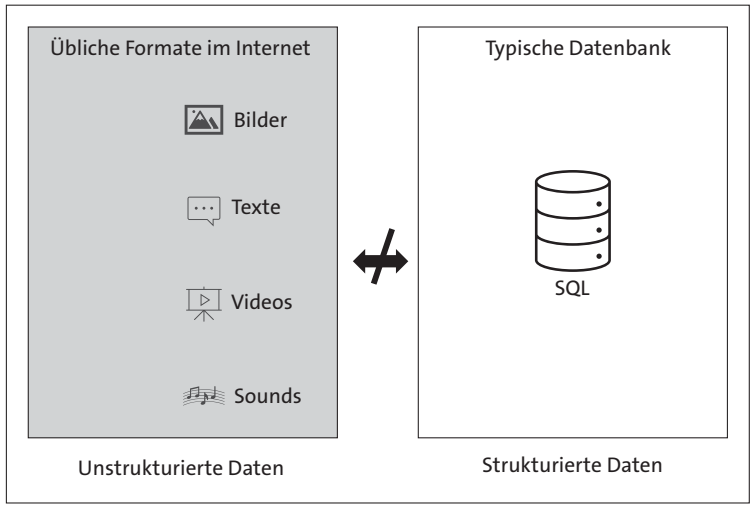


Abbildung 6.1 Unstrukturierte und strukturierte Daten

Das bringt uns zu der Frage, warum dieser neue Datentyp als *Vektor*datenbank bezeichnet wird. Nun, weil alle Datentypen als Vektoren dargestellt werden können. Wenn Sie an Ihren Matheunterricht zurückdenken, erinnern Sie sich vielleicht, was ein Vektor ist – die Linie mit einem Pfeil, die Sie in ein Koordinatensystem zeichnen mussten. Damals mussten Sie diese Linien zeichnen, die durch zwei Zahlen dargestellt werden konnten.

Ein einfaches Beispiel sehen Sie in Abbildung 6.2. Dieses erfundene Diagramm zeigt lebende Wesen und ihre Position in einem zweidimensionalen Diagramm, in dem die beiden Dimensionen die Anzahl der Beine und die Körpergröße darstellen. Indem wir die Informationen auf diese Weise darstellen, lernen wir etwas über die Welt und die semantische Bedeutung von Wörtern, zum Beispiel, dass Katzen und Hunde in Bezug auf diese beiden Eigenschaften ziemlich ähnlich sind.

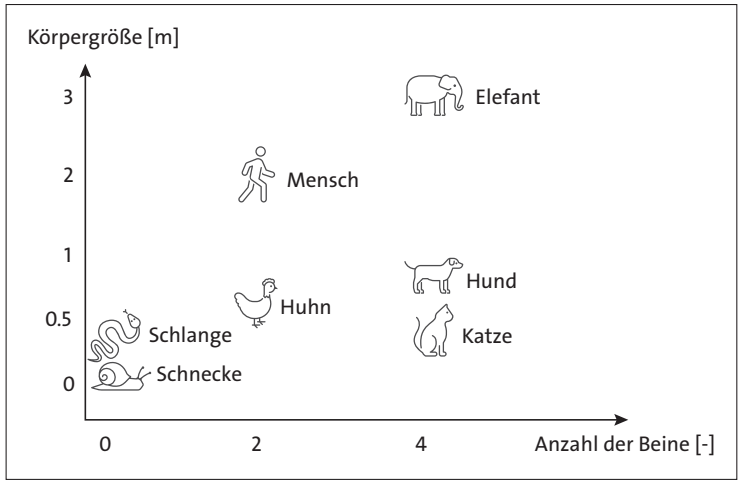


Abbildung 6.2 Beispiel für einen vereinfachten Vektorraum

Wir Menschen können uns einen Punkt in einem zweidimensionalen Raum vorstellen, wie er im Beispiel gezeigt wird, oder in einem dreidimensionalen Raum. Stellen Sie sich vor, wir fügen eine dritte Dimension wie Intelligenz in dieses Diagramm ein. Das ist immer noch verständlich, da wir uns in einer dreidimensionalen Welt bewegen – vierdimensional, wenn wir die Zeit als zusätzliche Dimension nutzen.

Aber wir können uns definitiv keinen 1536- oder 3072-dimensionalen Raum vorstellen. Computer können das. Und das ist hilfreich, denn die semantische Bedeutung von Wörtern, Bildern, Videos, Geräuschen oder jeder anderen Art von Informationen kann in einer höheren Dimension dargestellt werden. Der entscheidende Aspekt hier ist, dass ähnliche Konzepte näher beieinander liegen als Konzepte, die sehr unterschiedlich sind. Das funktioniert wie in unserem Beispiel, in dem wir sehen können, dass Katzen und Hunde oder Schlangen und Schnecken vergleichbar sind – je nach den Eigenschaften, die wir ausgewählt haben. Mit jeder zusätzlichen Dimension bekommt der Computeralgorithmus ein besseres Verständnis von der Bedeutung eines Wortes.

Wir verwenden Sprachen wie Englisch oder Deutsch, um zu kommunizieren und ein Konzept zu klären. Computer arbeiten nicht direkt mit unseren Sprachen, sondern mit ihrem Äquivalent – also mit numerischen, sogenannten *Einbettungsvektoren* (engl. *embedding vectors*).

In unserem Beispiel wird für den Computeralgorithmus z. B. ein Hund durch [4, 1] definiert, eine Katze durch [4, 0.5] und ein Mensch durch [2, 2]. Eine Beispielzuordnung ist in Tabelle 6.1 dargestellt.

Menschliches Konzept	Computeräquivalent/Embedding Vector
Hund	[4, 1]
Katze	[4, 0.5]
Mensch	[4, 2]
Elefant	[4, 3]
Schlange	[0, 0.5]
Schnecke	[0, 0.1]

Tabelle 6.1 Menschliche Konzepte und ihre Computer-Entsprechungen

Und der Prozess, menschliche Texte in Vektoren zu übersetzen, wird *Einbettung* oder *Embedding* genannt. Das ist einer der Schritte der Datenerfassung (engl. *data ingestion*), die wir im nächsten Abschnitt genauer unter die Lupe nehmen werden.

6.2 Der Data-Ingestion-Prozess

Wir beginnen mit einem Überblick über den Prozess, wie man Daten in eine Vektordatenbank einfügt. Die folgenden Schritte werden Sie immer durchlaufen, wenn Sie Daten zu einer Vektordatenbank hinzufügen möchten:

- Daten importieren
- Daten splitten (Chunking)
- Einbettungen (Embeddings) erstellen
- Daten speichern

Es beginnt immer mit einer *Datenquelle*. Ich habe ja bereits die vielfältigen Optionen der Datentypen angesprochen – es kann jede Art von textbasierten Quellen sein, wie Markdown-Dateien, Word-Dokumente oder einfache Textdateien. Aber es kann auch spezieller sein, wie ein YouTube-Video, aus dem wir die Transkription extrahieren wollen. Oder es könnte ein Wikipedia-Artikel sein. Die Möglichkeiten sind nahezu endlos. Glücklicherweise ist die Community von Nutzern, die spezifische Datenhandler beisteuert, ebenfalls groß, sodass Hunderte von verschiedenen Quellen mit einer einzigen Zeile Code direkt verwendet werden können. Basierend darauf haben wir eine Möglichkeit, die Daten auf konsistente Weise zu laden. Das ist wichtig, denn es gibt viele verschiedene Datenquellen, und wir wollen die Daten auf eine konsistente Weise importieren. Infolgedessen haben wir die Daten als langen String von Zeichen importiert.

Dieser lange Textstring muss in kleinere Stücke aufgeteilt werden. Das machen wir, weil wir am Ende eine Nadel im Heuhaufen finden wollen. Basierend auf einigen Benutzereingaben wollen wir ein bestimmtes Stück Information erhalten, das für die Benutzereingabe am relevantesten ist, und nicht einen kompletten Heuhaufen an Informationen. Daher müssen wir den Text in handhabbare, kleine Stücke aufteilen. Dieser Prozess wird *Chunking* genannt. An diesem Punkt haben wir eine Liste von Textdokumenten.

Wie bereits besprochen, sind diese Textdokumente für Menschen verständlich, aber nicht direkt für Computer. Wir müssen sie in ein computerverständliches Format übersetzen. Dafür nutzen wir einen Prozess, der *Embedding* genannt wird: Für jeden Text erhalten wir einen entsprechenden Vektor, der die semantische Bedeutung des Textes repräsentiert.

Damit haben wir die letzte Phase erreicht und können diese Paare (Text | numerischer Vektor) in unserer Vektordatenbank speichern.

6.3 Dokumente importieren

Der erste Teil der Datenerfassungspipeline ist immer das Laden von Dokumenten aus einer Vielzahl von verschiedenen Datenquellen. Diese Quellen können textbasierte Eingaben wie Markdown-Dokumente, Microsoft Word-Dokumente, JSON-Objekte oder einfache Textdateien sein. Aber es kann auch viel exotischer sein. Nur ein paar Beispiele: Sie können WhatsApp-Chats, Wetterdaten, Nachrichtenartikel und vieles mehr importieren.

Auf der LangChain-Dokumentationsseite erfahren Sie mehr darüber, welche verschiedenen Formate verfügbar sind:

https://python.langchain.com/api_reference/community/document_loaders.html

In diesem Abschnitt beginnen wir mit einem Überblick über das Laden von Daten. Anschließend setzen wir praktisch zuerst das Laden einer einzelnen Textdatei um, danach das Laden mehrerer Textdateien. Sie haben dann die Möglichkeit, Ihr Wissen in einigen Übungen anzuwenden, in denen Sie zuerst mehrere Wikipedia-Artikel laden, bevor Sie ein komplettes Buch von Project Gutenberg holen.

6.3.1 Ein erster Überblick

Zu Beginn des Prozesses haben Sie eine oder mehrere Dateien. Diese werden von einer `DataLoader`-Klasse verarbeitet. Für viele verschiedene Dateitypen gibt es `DataLoader`. Der `DataLoader` hilft dabei, den Prozess zu vereinfachen, denn die Ausgabe eines jeden `DataLoaders` ist gleich: Sie erhalten eine Python-Liste mit LangChain-`Document`-Objekten. Dieses Verfahren ist in Abbildung 6.3 dargestellt.

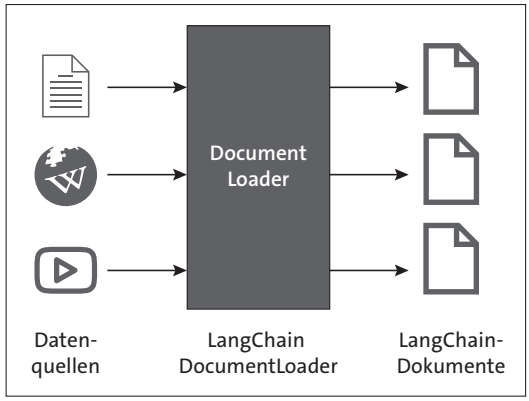


Abbildung 6.3 Der LangChain-Datenimport-Prozess

Ein LangChain-`Document`-Objekt ist ein Container für einen Textdatenabschnitt zusammen mit den zugehörigen Metadaten. Das *Document*-Objekt hat zwei Hauptattribute:

- `page_content`: In diesem Attribut sind die eigentlichen Textinformationen enthalten.
- `metadata`: In diesem Dictionary können zusätzliche Informationen über das Dokument enthalten sein, zum Beispiel die Datenquelle, der Autor, das Erstellungsdatum usw.

6.3.2 Coding: Eine einzelne Textdatei laden

Im ersten Beispiel arbeiten wir mit den Sherlock-Holmes-Büchern, die von *Project Gutenberg* bereitgestellt werden. Sie finden sie im Unterordner *05_VectorDatabases/data*. Wir arbeiten mit Büchern, die als *.txt*-Dateien bereitgestellt werden. Wie Sherlock Holmes werden wir in der Lage sein, die sprichwörtliche Nadel im Heuhaufen zu finden, also die eine Information, die wir wirklich brauchen. Wir beginnen damit, eine einzelne Datei wie folgt zu laden:

1. Bevor wir die Daten importieren können, müssen wir die Pakete laden. Um mit dem Dateipfad und dem Betriebssystem umzugehen, importieren wir das Paket `os`. Damit wir unseren Text importieren können, nutzen wir die Klasse `TextLoader`, die im Untermodul `langchain.document_loaders` gespeichert ist.
2. Wir müssen den Link zum Dateipfad der Buchdatei definieren, die wir importieren wollen. Die Datei befindet sich im obersten Verzeichnis des Ordners *05_VectorDatabases*. Das erreichen wir, indem wir das aktuelle Arbeitsverzeichnis `current_dir` extrahieren. Von dort aus holen wir uns das übergeordnete Verzeichnis `parent_dir`. Jetzt wird der Dateipfad erstellt, indem wir das übergeordnete Ver-

zeichnis mit dem Verzeichnis *data* und der Buchdatei, die wir importieren wollen, zusammenfügen.

3. Jetzt können wir anfangen, die Datei in ein Objekt zu laden, mit dem wir weiterarbeiten können. Zuerst erstellen wir eine Instanz von `TextLoader`. Bei der Instanziierung übergeben wir den Parameter `file_path`. Dieser Parameter ist entscheidend. Um die Daten zu importieren, müssen wir auch die *Textkodierung* festlegen. Textkodierung ist der Prozess, bei dem für Menschen lesbare Zeichen in ein bestimmtes Format umgewandelt werden, das Computer speichern, übertragen und verstehen können, Dazu werden standardisierte Systeme wie ASCII oder UTF-8 verwendet, um Text als binäre Daten darzustellen. In diesem Fall legen wir die Kodierung auf `utf-8` fest. Das Objekt `text_loader` hat eine Methode `load()`, die wir aufrufen. An diesem Punkt wird das Laden durchgeführt.
4. Der Import ist abgeschlossen, und wir können anfangen, besser zu verstehen, was wir eigentlich gemacht haben. Das Objekt `doc` repräsentiert eine Liste mit Elementen vom Typ `Document`. Wir haben nur eine Datei importiert, also gibt es nur ein Element mit dem Index 0. Es hat zwei Eigenschaften: `metadata` und `page_content`. Der Parameter `metadata` bezieht sich auf die Quelle, in unserem Fall auf den Dateipfad. Der Parameter `page_content` bezieht sich auf den tatsächlichen Text des Dokuments, das wir importiert haben.

Das Skript *05_VectorDatabases\10_DataLoader\10_single_text_file.py* zeigt das Laden von Daten aus einer einzelnen Datei. Sie sehen es hier in Listing 6.1:

```
#%% (1) Packages
import os
from langchain.document_loaders import TextLoader

#%% (2) File Handling
# Get the current working directory
file_path = os.path.abspath(__file__)
current_dir = os.path.dirname(file_path)

# Go up one directory level
parent_dir = os.path.dirname(current_dir)

file_path = os.path.join(parent_dir, "data","HoundOfBaskerville.txt")
file_path

#%% (3) Load a single document
text_loader = TextLoader(file_path=file_path, encoding="utf-8")
doc = text_loader.load()
```

```
#%% (4) Understand the document
# Metadata
doc[0].metadata

# %% Page content
doc[0].page_content
```

Listing 6.1 Datenimport aus einer Textdatei (05_VectorDatabases\10_DataLoader\10_single_text_file.py)

Super, wir haben es geschafft, eine einzelne Datei erfolgreich in Python und in ein LangChain-spezifisches Format zu laden! Das wird uns in den nächsten Schritten helfen, sie einfacher weiterzuverarbeiten. Bevor wir dazu kommen, schauen wir uns an, wie man mehrere Textdateien lädt.

6.3.3 Coding: Laden mehrerer Textdateien

Typischerweise arbeiten Sie nicht mit einer einzelnen Datei, sondern mit vielen Dateien desselben Dateityps, und möchten alle Dateien in einem einzigen Prozess importieren. In unserem Beispiel wollen wir alle Textdateien verwenden, die im Ordner *05_VectorDatabases/data* gespeichert sind.

Wir importieren jetzt auch den `DirectoryLoader`, der es uns ermöglicht, über die Dateien in einem Ordner zu iterieren:

```
#%% (1) Packages
import os
from langchain.document_loaders import TextLoader, DirectoryLoader
from pprint import pprint
```

Der Pfad zum Ordner mit den Dateien wird relativ zum Speicherort der Datei definiert. Wenn Sie ihn so einrichten, wie in Listing 6.2 gezeigt, haben Sie die Flexibilität, dass das aktuelle Verzeichnis `current_directory` aus dem Dateipfad extrahiert wird:

```
#%% (2) Path Handling
# Get the current working directory
file_path = os.path.abspath(__file__)
current_dir = os.path.dirname(file_path)

# Go up one directory level
parent_dir = os.path.dirname(current_dir)
text_files_path = os.path.join(parent_dir, "data")
```

Listing 6.2 Pfadbehandlung (05_VectorDatabases\10_DataLoader\20_multiple_text_files.py)

Alle Dateien im Ordner werden dadurch in einem einzigen Prozess geladen. Der `DirectoryLoader` wird angewendet und verwendet den Pfad `path`, der auf die Dateien zeigt. Mit dem `glob`-Parameter (siehe Listing 6.3) können Sie sicherstellen, dass nur bestimmte Dateien ausgewählt werden. Im Beispiel werden nur *.txt*-Dateien berücksichtigt.

`loader_cls` definiert, welche Loader-Klasse auf die Dateien angewendet werden soll. Wir nutzen `TextLoader`, wie im vorherigen Abschnitt definiert. Schließlich müssen wir auch noch die Kodierung festlegen. Die Kodierung haben wir zuvor in der `TextLoader`-Klasse definiert und wollen das hier ebenfalls tun. Wir können sie als `loader_kwargs` übergeben.

Kwargs ist die Kurzform für »Keyword-Argumente«. Das ist eine Python-Konvention, um eine variable Anzahl von benannten Parametern an eine Funktion zu übergeben. Diese Parameter müssen in einem Dictionary übergeben werden.

Listing 6.3 zeigt, wie ein `DirectoryLoader` eingerichtet werden kann, der es ermöglicht, über alle Dateien in einem Ordner zu iterieren:

```
#%% (3) load all files in a directory
dir_loader = DirectoryLoader(path=text_files_path,
    glob="**/*.txt",
    loader_cls=TextLoader,
    loader_kwargs={'encoding': 'utf-8'} )
docs = dir_loader.load()
docs

# Ausgabe:
[Document(metadata={'source': '…', page_content='…'},
 Document(metadata={'source': '…', page_content='…'},
 Document(metadata={'source': '…', page_content='…'}]
```

Listing 6.3 Laden aller Dateien eines Ordners (05_VectorDatabases\10_DataLoader\20_multiple_text_files.py)

Als Ergebnis erhalten wir eine Liste mit `Document`-Objekten, mit denen wir dann weiterarbeiten können. Nun wird es Zeit, mit einer kleinen Übung das neu erworbene Wissen zu überprüfen.

6.3.4 Übung: Laden mehrerer Wikipedia Artikel

In Listing 6.4 sehen Sie mehrere URLs für Artikel über künstliche Intelligenz, künstliche allgemeine Intelligenz und Superintelligenz. Am Ende sollten Sie eine Liste von LangChain-`Document`-Objekten haben, die den URLs entsprechen.

```
#%% (1) Packages

#%% Articles to load
articles = [
    {'title': 'Artificial Intelligence'},
    {'title': 'Artificial General Intelligence'},
    {'title': 'Superintelligence'},
]

# %% (2) Load articles
```

Listing 6.4 Übung zum Daten-Laden – mehrere Wikipedia-Artikel (05_VectorDatabases\10_DataLoader\30_wikipedia_exercise.py)

Aufgabe 1: Pakete laden

Eine wichtige Fähigkeit ist, mit Online-Dokumentationen arbeiten zu können. Ihre erste Aufgabe besteht darin, herauszufinden, ob es einen `DocumentLoader` für Wikipedia-Artikel gibt (Tipp: Den gibt es) und wie Sie das Paket und die Klasse laden.

Aufgabe 2: Artikel importieren

Als Nächstes müssen Sie eine Schleife schreiben, um über alle Artikel zu iterieren. Sie können das Starter-Skript verwenden, das Sie unter dem Pfad *05_VectorDatabases\10_DataLoader\30_wikipedia_exercise.py* finden.

Listing 6.5 zeigt den Inhalt des Starter-Skripts:

```
#%% (1) Packages

#%% Articles to load
articles = [
    {'title': 'Artificial Intelligence'},
    {'title': 'Artificial General Intelligence'},
    {'title': 'Superintelligence'},
]
# %% (2) Load all articles
```

Listing 6.5 Übung zum Daten-Laden – mehrere Wikipedia-Artikel (05_VectorDatabases\10_DataLoader\30_wikipedia_exercise.py)

Bitte versuchen Sie die Aufgabe zu lösen, indem Sie Ihren Code einfügen.

Lösung

Listing 6.6 zeigt meine Musterlösung für diese Übung. Besonders wichtig ist, dass Sie herausgefunden haben, wie man die Daten direkt von Wikipedia mithilfe von `Wiki-`

pediaLoader lädt. Der verbleibende Teil besteht einfach darin, über das articles-Objekt zu iterieren und herauszufinden, wie man eine Liste von Dokumenten erstellt, indem man sie zu docs hinzufügt.

```
#%% Packages
from langchain.document_loaders import WikipediaLoader

#%% Articles to load
articles = [
    {'title': 'Artificial Intelligence'},
    {'title': 'Artificial General Intelligence'},
    {'title': 'Superintelligence'},
]
# %% Load all articles (2)
docs = []
for i in range(len(articles)):
    print(f"Loading article on {articles[i].get('title')}")
    loader = WikipediaLoader(query=articles[i].get("title"),
                             load_all_available_meta=True,
                             doc_content_chars_max=100000,
                             load_max_docs=1)
    doc = loader.load()
    docs.append(doc)
docs
```

Listing 6.6 Übung zum Daten-Laden mit Lösung – mehrere Wikipedia-Artikel laden (05_VectorDatabases\10_DataLoader\40_wikipedia_solution.py)

6.3.5 Übung: Laden von Büchern von Project Gutenberg

In der Übungsdatei, die Sie in Listing 6.7 sehen, finden Sie die Details zu einem Roman in einem Dictionary:

```
# %% The book details
book_details = {
    "title": "The Adventures of Sherlock Holmes",
    "author": "Arthur Conan Doyle",
    "year": 1892,
    "language": "English",
    "genre": "Detective Fiction",
    "url": "https://www.gutenberg.org/cache/epub/1661/pg1661.txt"
}
```

Listing 6.7 Übung zum Daten-Laden – Daten aus dem »Project Gutenberg« beziehen (05_VectorDatabases\10_DataLoader\50_custom_loader_exercise.py)

Aufgabe 1: Pakete importieren

Wie bei der vorherigen Übung sollten Sie sich mit der Online-Dokumentation vertraut machen und herausfinden, welches Paket und welche Klasse Sie importieren müssen, um mit Büchern von *https://www.gutenberg.org/* zu arbeiten. Das *Project Gutenberg* ist eine großartige Ressource, denn dort finden Sie urheberrechtsfreie Klassiker, mit denen Sie arbeiten können.

Aufgabe 2: Metadaten anpassen

Nachdem Sie die Datei geladen haben, bekommen Sie eine Liste von Elementen des Typs `Document`. Ersetzen Sie die Metadaten des `Document`-Objekts, sodass das Dictionary `book_details` die Metadaten darstellt. In Listing 6.8 sehen Sie den Inhalt der Datei, der aus `book_details` besteht:

```
# %% The book details
book_details = {
    "title": "The Adventures of Sherlock Holmes",
    "author": "Arthur Conan Doyle",
    "year": 1892,
    "language": "English",
    "genre": "Detective Fiction",
    "url": "https://www.gutenberg.org/cache/epub/1661/pg1661.txt"
}
```

Listing 6.8 Übung – benutzerdefinierter Import (05_VectorDatabases\10_DataLoader\50_custom_loader_exercise.py)

Lösung

Listing 6.9 zeigt, wie Sie die Daten von Project Gutenberg laden können:

```
from langchain_community.document_loaders import GutenbergLoader
# %% The book details
book_details = {
    "title": "The Adventures of Sherlock Holmes",
    "author": "Arthur Conan Doyle",
    "year": 1892,
    "language": "English",
    "genre": "Detective Fiction",
    "url": "https://www.gutenberg.org/cache/epub/1661/pg1661.txt"
}

loader = GutenbergLoader(book_details.get("url"))
data = loader.load()
```

```
#%% Add metadata from book_details
data[0].metadata = book_details
```

Listing 6.9 Lösung: Datenimport von »Project Gutenberg« (05_VectorDatabases\10_DataLoader\60_custom_loader_solution.py)

Jetzt, da Sie grundsätzlich wissen, wie man Daten auf verschiedene Arten lädt, können wir im folgenden Abschnitt zum nächsten Schritt der Datenaufnahme-Pipeline übergehen – zum Daten-Splitting.

6.4 Dokumente aufteilen

Wir müssen die Daten in kleinere Stücke aufteilen, in sogenannte *Chunks*. Warum ist dieser Schritt notwendig? Es gibt ein paar Gründe:

- Sie bekommen *genauere Ergebnisse*, weil kleinere Dokumentabschnitte eher gezielte und relevante Informationen enthalten.
- Es gibt Einschränkungen bei Einbettungsmodellen. (Keine Sorge, ich werde das später ausführlicher erläutern.) Aber für den Moment sollten Sie sich einfach merken, dass alle Einbettungsmodelle ein *begrenztes Kontextfenster* von Token haben, mit dem sie arbeiten können. Was Token sind, erkläre ich gleich noch.
- Es kann *effizienter* in Bezug auf die Datenbankgröße und Abfragen sein, kleinere Stücke zu speichern.

Eben habe ich schon den Begriff *Token* erwähnt, also sollte ich direkt erklären, was ein Token ist. Ein Token ist die grundlegende Einheit von Text in der Verarbeitung natürlicher Sprache. Typischerweise ist es ein Wort, ein Teil eines Wortes oder manchmal sogar nur ein einzelnes Zeichen. Der Prozess, einen Satz in eine Liste von Token umzuwandeln, wird *Tokenisierung* genannt. Er ist ein entscheidender Schritt in der Vorverarbeitung. Es gibt viele verschiedene Methoden zur Tokenisierung.

Einige Beispiele sollen Ihnen eine Vorstellung vom Prozess geben. Im Satz »Ich liebe generative KI!« könnten die resultierenden Tokens [»Ich«, »liebe«, »gener«, »ative«, »KI«, »!«] sein.

Jedes Sprachmodell besitzt zudem ein eigenes *Wörterbuch*, in dem zu jedem Token die entsprechende Token-ID zugewiesen wird. Somit wird das Modell anstelle der Liste von Tokens [»Ich«, »liebe«, »gener«, »ative«, »KI«, »!«] die Token-IDs [20444, 66483, 2217, 1799, 110949, 0] verwenden. Bei OpenAI können Sie sich unter *https://platform.openai.com/tokenizer* mit dem Tokenizer näher vertraut machen.

In vielen *Tokenisierungsmethoden* werden Wörter in mehrere Tokens zerlegt. Oben haben Sie z. B. gesehen, dass das Wort »generative« in [»gener«, »ative«] tokenisiert wurde.

Warum wird das gemacht? Tokens ermöglichen es KI-Modellen, Text zu verarbeiten, indem sie ihn in numerische Vektoren umwandeln, die von einem Modell verstanden werden. Das ist der nächste Schritt, auch *Einbettung* genannt, den wir später behandeln.

An dieser Stelle ist es nur wichtig zu wissen, dass alle Einbettungsmodelle eine Grenze für die zu verarbeitenden Tokens haben. Wenn Sie versuchen, mehr Tokens in ein Einbettungsmodell zu geben, als es verarbeiten kann, werden überschüssige Tokens abgeschnitten. Die Bedeutung dieser Tokens geht damit verloren.

Es ist also entscheidend, immer weniger Tokens an ein Modell zu übergeben, als es maximal verarbeiten kann. Als Faustregel gilt, dass ein Token ungefähr 0,75 Wörtern entspricht oder 100 Tokens ungefähr 75 Wörtern in der englischen Sprache.

In allen Modellen gibt es eine Grenze, wie viele Daten auf einmal verarbeitet werden können. Diese Einschränkung der Tokens wird auch als *Kontextfenster* bezeichnet.

Die Kontextfenster bei verschiedenen Einbettungsmodellen

Lassen Sie uns einen Eindruck von den typischen Token-Limits beliebter Modelle bekommen. Für alle Modelle finden Sie eine Modellkarte mit relevanten Informationen zum Modell. Suchen Sie einfach danach, zum Beispiel mit »text-embedding-3-small Modellkarte«. Die Ergebnisse fallen recht unterschiedlich aus:

- OpenAI text-embedding-3-small: 8191 Tokens
- OpenAI text-embedding-3-large: 8191 Tokens
- Anthropic voyage-3-large: 32.000 Tokens
- All-MiniLM-L6-v2: 256 Tokens

Sie haben bisher gelernt, dass alle Dokumente in kleinere Teile aufgeteilt werden müssen und dass es wichtig ist, eine angemessene Größe dieser Teile zu definieren. Aber wie werden die Teile nun erstellt? Es gibt viele verschiedene Ansätze. Ich stelle im Folgenden drei sehr beliebte Methoden vor: Chunking mit festen Größen, strukturbasiertes Chunking und semantisches Chunking. Daneben ist aber auch ein benutzerdefiniertes Chunking möglich, das wir uns ebenfalls ansehen werden.

Eine visuelle Darstellung der Konzepte zum *Chunking* von Daten sehen Sie in Abbildung 6.4:

- Der erste Ansatz ist also, eine feste Chunk-Größe zu verwenden und nach einer bestimmten Anzahl von Zeichen zu splitten.

- Ein ausgefeilterer Ansatz besteht darin, der Struktur des Dokuments zu folgen und unterschiedliche Größen der Chunks zuzulassen, wenn das hilft, die Bedeutung jedes Chunks zu bewahren. In der Mitte von Abbildung 6.4 sehen Sie einen Chat-Verlauf von zwei Personen. Es wäre sogar schädlich für das Verständnis, wenn sich die Chunks von zwei verschiedenen Personen überschneiden würden. Stellen Sie sich vor, Sie versuchen, ein Gespräch in einem Buch zu verstehen, aber die Aussagen sind teilweise dem falschen Protagonisten zugeordnet.
- Der fortschrittlichste Ansatz ist das semantische Chunking, bei dem der Text verstanden wird, bevor das Chunking durchgeführt wird.

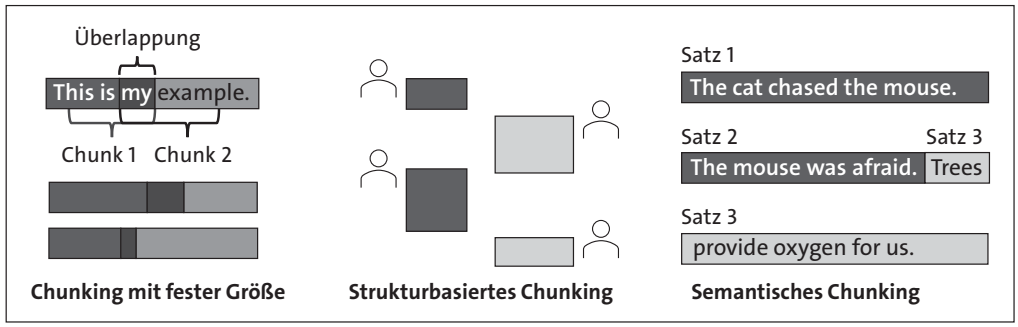


Abbildung 6.4 Chunking-Strategien

Wir fangen mit dem einfachsten Ansatz an, bevor wir unser Vorgehen ständig verbessern. Beginnen wir mit der festen Chunk-Größe.

6.4.1 Coding: Chunking mit festen Größen

Das Chunking mit fester Größe erstellt die Chunks, wobei eine vordefinierte Anzahl von Zeichen genutzt wird. Dadurch werden Chunks mit relativ gleichbleibender Größe erzeugt. Bevor die Chunk-Größe überschritten wird, beginnt der nächste Chunk.

Der Vorteil ist die Einfachheit des Ansatzes. Er ist leicht zu verstehen und umzusetzen. Dieser Ansatz ist auch sehr schnell, da er keine komplexe Analyse der Textbedeutung erfordert.

Sein Nachteil ist das Fehlen von Kontextbewusstsein. Sätze oder Konzepte könnten mittendrin getrennt werden, sodass die semantische Kohärenz verloren geht.

Chunk-Überlappung

Eine Besonderheit, die Sie nutzen sollten, ist die *Chunk-Überlappung*. Als Chunk-Überlappung bezeichnet man die Menge an Text, die zwischen zwei aufeinanderfolgenden Chunks geteilt wird. Sie wird normalerweise in Zeichen oder Tokens gemessen.

Das Ziel bei der Überlappung ist es, den Kontext zwischen den Chunks aufrechtzuerhalten und die Bedeutung zu bewahren, die sonst verloren gehen könnte, wenn der Text willkürlich aufgeteilt wird.

Eine gängige Faustregel ist eine Überlappung von 10 bis 20 %. Das bietet ein gutes Gleichgewicht zwischen der Erhaltung des Kontexts und der Effizienz der Datenbank. Aber das sollte nicht dogmatisch gehandhabt werden. Je nach Art des Textes könnte weniger oder mehr Überlappung sinnvoll sein.

Abbildung 6.5 zeigt das Ergebnis des `CharacterTextSplitter`s. Beide angezeigten Teile sind gleich groß. Außerdem können Sie den überlappenden Text in beiden Teilen sehen.

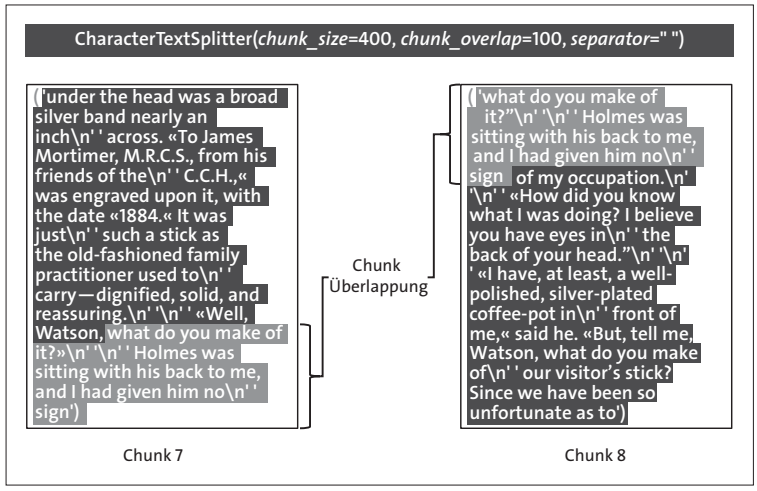


Abbildung 6.5 »CharacterTextSplitter« mit Chunk-Überlappung

Da Sie jetzt ein grundlegendes Verständnis von Chunks und Überlappung haben, können wir anfangen, diese Konzepte umzusetzen. In unserem Beispiel gehen wir zurück zu Sherlock Holmes und erweitern unser vorheriges Skript. Das Laden der Daten zeige ich nicht erneut, sondern wir springen direkt in den Code, nachdem das `docs`-Objekt erstellt wurde.

Wir müssen die Funktionalität laden und die Klasse `CharacterTextSplitter` für das `langchain`-Submodul `text_splitter` verwenden, wie in Listing 6.10 gezeigt. (Das komplette Skript finden Sie unter *05_VectorDatabases\20_Chunking\10_fixed_size_chunking.py*.)

```
# ...
# %%
docs
```

```
# %% Splitting text
# Packages (1)
from langchain.text_splitter import CharacterTextSplitter
```

Listing 6.10 Chunking mit fester Größe – erforderliche Pakete (05_VectorDatabases\20_Chunking\10_fixed_size_chunking.py)

Eine Instanz der Klasse `CharacterTextSplitter` wird erstellt und normalerweise `splitter` genannt. Bei dieser Instanziierung werden die Parameter definiert, wie zum Beispiel die *Chunk-Größe* (also die maximale Anzahl von Zeichen in einem Chunk), die *Chunk-Überlappung* (die maximale Menge an Überlappungszeichen in einem Chunk) oder der *Separator*, der zum Teilen verwendet wird:

```
# Split by characters (2)
splitter = CharacterTextSplitter(chunk_size=256, chunk_overlap=50,
  separator=" ")
```

Die Aufteilung wurde noch nicht durchgeführt. Das machen wir jetzt, indem wir die Methode `splitter.split_documents` aufrufen und die Liste der Dokumente übergeben. Das Ergebnis ist wieder eine Liste von `Document`-Objekten, aber die Anzahl der Elemente hat sich signifikant erhöht. Bis zu diesem Punkt hatten wir ein `Document`-Objekt pro Buch. Jetzt wurde jedes Buch in viele kleinere Teile aufgeteilt, und zwar wie folgt:

```
# %% Apply the splitting (3)
docs_chunks = splitter.split_documents(docs)
```

Wir können nun die Anzahl der Chunks wie folgt überprüfen (und herausfinden, dass sie auf 4140 Dokumente gestiegen ist):

```
# %% Check the number of chunks (4)
len(docs_chunks)
```

Wenn wir uns zwei aufeinanderfolgende Chunks genauer anschauen, können wir die Überlappung der Chunks betrachten, also die Menge der Zeichen, die zwischen beiden Chunks geteilt wird. Wie das geht, sehen Sie in Listing 6.11:

```
# %% check some random Documents (5)
from pprint import pprint
pprint(docs_chunks[100].page_content)
# %%
pprint(docs_chunks[101].page_content)
```

Listing 6.11 Chunking mit festen Größen – Überprüfung der Dokumente (05_VectorDatabases\20_Chunking\10_fixed_size_chunking.py)

Schließlich schauen wir uns mit Listing 6.12 die Chunk-Größen an und visualisieren die Verteilung der Chunk-Längen mit *Matplotlib*. Sie werden sehen, dass die meisten Chunks um 250 zentriert sind, was zu erwarten war.

```
# %% visualize the chunk size (6)
import seaborn as sns
import matplotlib.pyplot as plt
# get number of characters in each chunk
chunk_lengths = [len(chunk.page_content) for chunk in docs_chunks]

sns.histplot(chunk_lengths, bins=50, binrange=(100, 300))
# add title
plt.title("Distribution of chunk lengths")
# add x-axis label
plt.xlabel("Number of characters")
# add y-axis label
plt.ylabel("Number of chunks")
```

Listing 6.12 Visualisierung des Chunkings mit festen Größen (05_VectorDatabases\20_Chunking\10_fixed_size_chunking.py)

Die Verteilung der Chunk-Größen ist in Abbildung 6.6 dargestellt.

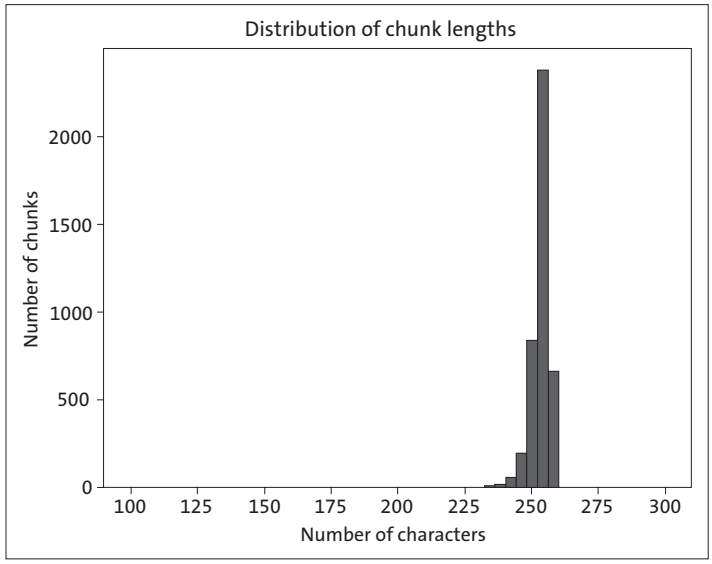


Abbildung 6.6 Verteilungsfunktion für das Chunking mit festen Größen

Die meisten Abschnitte haben 250 Zeichen. Da die Aufteilung nicht mitten im Wort erfolgt, sind einige Abschnitte kürzer und andere etwas länger.

6.4.2 Coding: Strukturbasiertes Chunking

Der kontextuelle Sinn oder die textuelle Kohärenz folgt normalerweise keinen festen Größen, daher spiegelt das Aufteilen von Text in gleich große Teile die Realität nicht gut wider. Wir brauchen daher etwas Anspruchsvolleres. Typischerweise wird ein Text vom Autor in Kapitel, Absätze, Sätze, Satzteile, Wörter usw. strukturiert. Ein Ansatz ist also, den Text in größere Stücke, z. B. Kapitel, zu unterteilen. Wenn das verbleibende Textstück immer noch zu groß ist, wird es in Unterkapitel, dann in Absätze und so weiter aufgeteilt. Das sorgt für ein gutes Gleichgewicht zwischen textlicher Kohärenz und der Einhaltung von Zeichenlimits.

Wie kann ein Algorithmus ein Kapitel oder einen Absatz erkennen? Das hängt von den Entscheidungen ab, die der Autor des Textes getroffen hat. Typischerweise gibt es mehrere leere Zeilen, bevor ein neues Kapitel beginnt. Zwischen Absätzen werden normalerweise kleinere Abstände verwendet. Jetzt, wo wir verstehen, wie es funktioniert, schauen wir uns an, wie es im Code umgesetzt wird.

Schauen wir uns ein Beispiel in Tabelle 6.2 an. Auf der linken Seite sehen Sie, wie die Buchkapitel aussehen. Nach dem Laden des `Document`-Objekts erhalten Sie die Eigenschaft `page_content`, die ein einzelner langer String ist.

Chapter 1 Mr. Sherlock Holmes Chapter 2 The Curse of the Baskervilles Chapter 3 The Problem Chapter 4 Sir Henry Baskerville Chapter 5 Three Broken Threads	`"Chapter 1 Mr. Sherlock Holmes\n Chapter 2 The Curse of the Baskervilles\n Chapter 3 The Problem\n Chapter 4 Sir Henry Baskerville\n Chapter 5 Three Broken Threads\n"`

Tabelle 6.2 Vergleich des Rohtextes (links, Text wie er im Buch dargestellt ist) mit einem Ausschnitt des korrespondierenden Textes nach dem Import (rechts)

In diesem String finden Sie Sonderzeichen wie `\n`. Dieses Sonderzeichen zeigt einen Zeilenumbruch an und sorgt dafür, dass der folgende Text am Anfang einer neuen Zeile beginnt.

Wie kann man wissen, dass `\n` eine neue Zeile anzeigt? Das ist in einem Zeichencodierungsstandard definiert. Es gibt mehrere Codierungsstandards (siehe unten). Ein Zeichencodierungsstandard ist ein System, das eine Zuordnung zwischen Zeichen wie Buchstaben, Zahlen oder Symbolen und spezifischen Codes erstellt. Das soll einen universellen Zeichensatz bereitstellen, der alle Schriftsysteme, Emojis und vieles mehr umfasst. Die Codierung ermöglicht es Computern, Text konsistent zu speichern, zu übertragen und anzuzeigen. Bekannte Codierungsstandards sind ASCII, UTF-8 oder ISO-8859-1. (UTF-8 ist moderner als ASCII.)

Das schauen wir uns in der Infobox etwas genauer an.

Zeichencodierungsstandards

Listing 6.13 enthält einige Beispiele für Codierungsstandards. Sie können diesen Code ausführen, um die Codierung der Zeichen zu sehen. Sie sehen dann die entsprechende Codierung. Wenn Sie versuchen, ein Emoji mit ASCII zu kodieren, wird das fehlschlagen, weil Emojis nicht Teil dieses Zeichensatzes sind. Mit UTF-8 können Sie auch chinesische Zeichen oder Zeichen aus anderen Sprachen kodieren.

```
print("ASCII:")
print("A:", "A".encode("ascii"))
print("\\n:", "\n".encode("ascii"))

# UTF-8 representation
print("\nUTF-8:")
print("A:", "A".encode("utf-8"))
print("\\n:", "\n".encode("utf-8"))
print("€:", "€".encode("utf-8"))
print("你:", "你".encode("utf-8"))
print("☺:", "☺".encode("utf-8"))
```

Listing 6.13 Textcodierungen gemäß ASCII und UTF-8

Lassen Sie uns jetzt auf den Daten aufbauen, die wir in Abschnitt 6.3.2 geladen haben. Die Funktionalität, die wir brauchen, heißt `RecursiveCharacterTextSplitter` und gehört ebenfalls zum Untermodul `langchain.text_splitter`. Sie verwenden sie folgt:

```
# ...
#%% Recursive Chunking
# Packages (1)
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

Mit diesem Code erstellen wir eine Instanz der Klasse `RecursiveCharacterTextSplitter`. Bei der Instanziierung übergeben wir relevante Parameter an den Splitter. Die Parameter sind die `chunk_size`, was für die Anzahl der Zeichen in einem Chunk steht, und der `chunk_overlap`, der den gemeinsamen Text zwischen zwei aufeinanderfolgenden Chunks angibt.

Ein sehr wichtiger Parameter sind die *Separatoren* (`separators`). Die Separatoren (oder auch *Trennzeichen*) sind eine Liste von Zeichen, die verwendet werden, um die Trennungen zu erstellen. Sie sind vom größten zum kleinsten geordnet – wir beginnen also mit einem Separator für Kapitel, dann für Unterkapitel und so weiter. Wenn ein erster Split nicht ausreicht, um die Chunk-Größe auf das Limit zu beschränken, wird nach einem weiteren Separator gesucht, um einen weiteren Split zu erzeugen. Dieser Prozess wird so lange wiederholt, bis die Chunk-Größe innerhalb der vordefinierten Grenzen liegt.

```
# %% Set up the splitter (2)
splitter = RecursiveCharacterTextSplitter(
    chunk_size=256,
    chunk_overlap=50,
    separators=["\n\n", "\n"," ", ".", ","])
```

Der Instanz-Objekt-Splitter hat eine Methode namens `split_documents`, an die wir unsere Liste von Dokumenten übergeben, wie Sie im Folgenden sehen. Als Ergebnis erhalten wir eine Liste von Dokumenten, aber die Anzahl der Dokumente hat sich erheblich erhöht.

```
# %% Create the chunks (3)
docs_chunks = splitter.split_documents(docs)
# %%
len(docs_chunks)
```

Wir überprüfen die Verteilung der Chunk-Größe, indem wir sie visualisieren, wie in Listing 6.14 gezeigt:

```
# %% Visualize the chunk size (4)
import seaborn as sns
import matplotlib.pyplot as plt
# get number of characters in each chunk
chunk_lengths = [len(chunk.page_content) for chunk in docs_chunks]

sns.histplot(chunk_lengths, bins=50, binrange=(0, 300))
# add title
plt.title("Distribution of chunk lengths (RecursiveCharacterTextSplitter)")
# add x-axis label
plt.xlabel("Number of characters")
# add y-axis label
plt.ylabel("Number of chunks")
```

Listing 6.14 Visualisierung der Chunk-Größen (05_VectorDatabases\20_Chunking\20_structure_based_chunking.py)

Abbildung 6.7 zeigt, dass die Chunk-Größen jetzt im Vergleich zur festen Aufteilung viel besser verteilt sind.

Sie sehen, dass jetzt eine breite Palette von Chunk-Größen abgedeckt ist. Diese reicht von fast null bis zu 250 Zeichen. Wenn ein Chunk aufgrund dieses Teilungsansatzes zu klein geraten ist, können Chunks auch zusammengefasst werden, um eine anständige Größe zu erreichen.

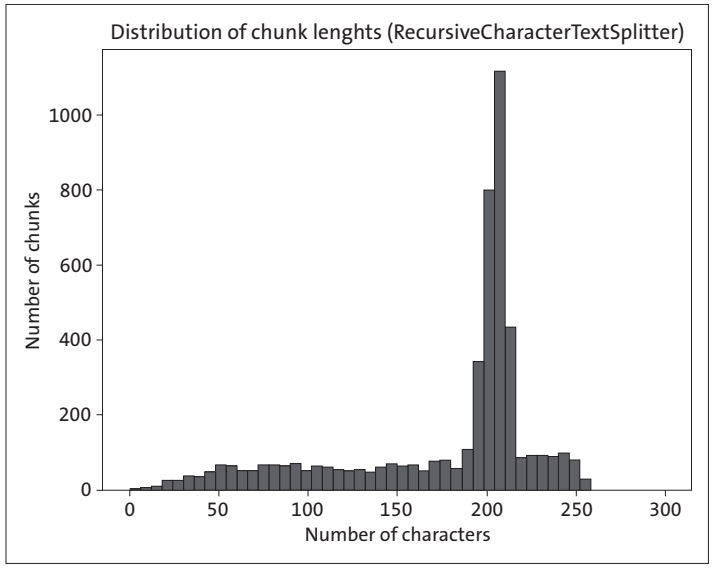


Abbildung 6.7 Verteilungsfunktion für das Chunking mit rekursivem Buchstaben-Splitting

Um die Dinge einfach zu halten, zeige ich das hier nicht. Aber die Umsetzung kann ganz einfach sein – Sie könnten die gesamte Liste der Chunks durchgehen und benachbarte Chunks zusammenfassen, falls ein Chunk zu klein ist und die Zusammenfassung mit dem benachbarten Chunk immer noch innerhalb der erlaubten Chunk-Größe liegt.

6.4.3 Coding: Semantisches Chunking

Bisher haben wir das Chunking basierend auf Chunk-Größen (festes Chunking) und unter Berücksichtigung der Struktur des Textes (strukturbasiertes Chunking) durchgeführt. Der eigentliche Text und seine semantische Struktur wurden dabei jedoch nicht berücksichtigt. Das ändert sich jedoch, wenn semantisches Chunking durchgeführt wird.

Beim semantischen Chunking wird der Text, den wir splitten wollen, zunächst in einzelne Sätze zerlegt. Dann werden die Sätze eingebettet. (Wie das erfolgt, wird das Thema des nächsten Abschnitts sein.) Im Moment müssen Sie nur wissen, dass die Sätze dadurch in numerische Vektoren umgewandelt werden, mit denen bestimmte Berechnungen durchgeführt werden können. Zum Beispiel können die Ähnlichkeiten zwischen aufeinanderfolgenden Sätzen berechnet werden.

Die Idee beim semantischen Chunking ist nun, an natürlichen Pausen zu splitten – das heißt, wenn die Sätze in ihrer Bedeutung zu unterschiedlich sind. Dafür wird die

Distanzmatrix erstellt und werden die Ähnlichkeiten zwischen den Sätzen analysiert. Wenn Sie jemals mit Distanzmatrizen gearbeitet haben, wissen Sie, dass es mehrere Ansätze gibt, um die Distanzen zu berechnen, und folglich gibt es mehrere Möglichkeiten, die textlichen Grenzen zu berechnen, die definieren, wann ein Split durchgeführt wird. Auf solche Aspekte gehe ich zu einem späteren Zeitpunkt ein. Hier verdeutliche ich zunächst nur die allgemeine Idee des semantischen Chunkings, die in Abbildung 6.8 visualisiert ist.

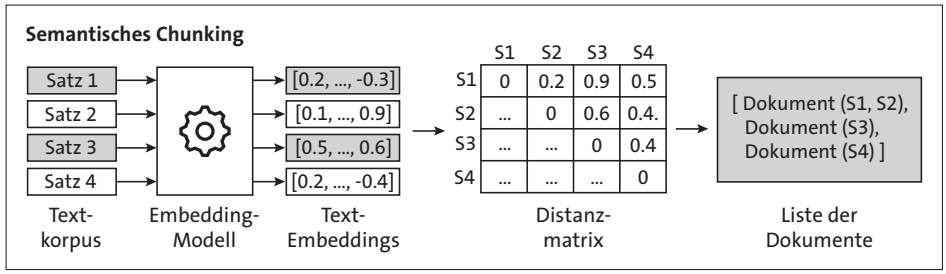


Abbildung 6.8 Semantisches Chunking

Dieser Ansatz bewahrt die semantische Bedeutung von Sätzen und hält verwandte Konzepte zusammen. Er verbessert den Informationsabruf und die Genauigkeit von Suchanfragen. Außerdem ist er sehr anpassungsfähig, da er die Chunk-Größen basierend auf dem Text anpasst, was zu Chunks führt, die natürlicher und leichter für Menschen zu lesen und zu verstehen sind.

Aber es gibt auch Nachteile. Zum einen bringt er einen gewissen Rechenaufwand mit sich und ist langsamer. Sie sollten auch bedenken, dass das Einbetten es erfordert, dass Sie ein neuronales Netzwerk, das Einbettungsmodell, entweder lokal betreiben oder per API nutzen. Das führt zu Kosten für die Erstellung der Einbettungen – obwohl Einbettungsmodelle vergleichsweise günstig sind. Aber Latenz und höhere Systemkomplexität sind Faktoren, die ebenfalls eine wichtige Rolle spielen könnten.

Kommen wir zurück zum Programmieren und schauen wir uns an, wie das funktioniert. Das Skript, das wir entwickeln werden, finden Sie unter *05_VectorDatabases\20_Chunking\30_semantic_chunking.py*.

Der semantische Chunker wurde noch nicht in das Hauptpaket von LangChain aufgenommen und befindet sich in `langchain_experimental`. Um einen Wikipedia-Artikel zu laden, verwenden wir `WikipediaLoader`. Für die Erstellung der Einbettungen nutzen wir OpenAI und das Paket `langchain_openai`. (Um die Dienste von OpenAI nutzen zu können, benötigen Sie ein OpenAI-Konto und einen API-Schlüssel, der in einer Datei namens *.env* gespeichert ist, die sich im selben Ordner wie das Skript befindet.) Wir laden das Skript über `load_dotenv`, wie in Listing 6.15 gezeigt:

```
#%% Packages (1)
from langchain_experimental.text_splitter import SemanticChunker
from langchain.document_loaders import WikipediaLoader
from langchain_openai.embeddings import OpenAIEmbeddings
from pprint import pprint
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv(usecwd=True))
```

Listing 6.15 Semantisches Chunking – erforderliche Pakete (05_VectorDatabases\20_Chunking\30_semantic_chunking.py)

Wir laden den Artikel, und dafür richten wir eine Instanz von `WikipediaLoader` ein und übergeben mehrere Parameter, z. B. den Artikeltitel, an den Parameter `query`. Wir wollen nur einen einzigen Artikel laden, also begrenzen wir die Anzahl mit dem Parameter `load_max_docs`. Außerdem begrenzen wir aus Demonstrationsgründen die Anzahl der Zeichen mit dem Parameter `doc_content_chars_max` so, wie in Listing 6.16 gezeigt:

```
# %% Load the article (2)
ai_article_title = "Artificial_intelligence"
loader = WikipediaLoader(query=ai_article_title,
    load_all_available_meta=True,
    doc_content_chars_max=1000,
    load_max_docs=1)
doc = loader.load()
```

Listing 6.16 Semantisches Chunking – Daten importieren (05_VectorDatabases\20_Chunking\30_semantic_chunking.py)

Nachdem wir die Daten geladen haben, überprüfen wir, ob alles funktioniert. Wir haben unser Objekt `doc`, das eine Liste mit nur einem Dokument ist. Deshalb müssen wir jetzt wie folgt auf `doc[0].page_content` zugreifen, um den Inhalt des Artikels zu bekommen:

```
# %% check the content (3)
pprint(doc[0].page_content)
```

Jetzt sind wir in einem interessanten Bereich angekommen. Wir richten unsere `SemanticChunker`-Instanz so ein, wie in Listing 6.17 gezeigt wird. Der semantische Chunker funktioniert auf Basis von Einbettungen. Hierzu verwenden wir `OpenAIEmbeddings`. Es gibt verschiedene Methoden, um die *Ähnlichkeiten* von Texten zu berechnen, und wir wenden die Kosinus-Ähnlichkeit zusammen mit einem Schwellenwert von 0,5 an, um die Texte zu splitten. Ich werde die Ähnlichkeitsberechnungen in Abschnitt 6.7, »Daten abrufen«, noch viel detaillierter erklären. An dieser Stelle sollte

es ausreichen, zu wissen, dass wir die semantische Bedeutung benachbarter Sätze analysieren müssen, und das geschieht basierend auf Ähnlichkeiten.

```
# %% Create splitter instance (4)
splitter = SemanticChunker(embeddings=OpenAIEmbeddings(),
    breakpoint_threshold_type="cosine",
    breakpoint_threshold=0.5)
```

Listing 6.17 Semantisches Chunking – Splitten der Daten (05_VectorDatabases\20_Chunking\30_semantic_chunking.py)

Der semantische Splitter erstellt dann die Chunks, die wir im gleichnamigen Objekt speichern:

```
# %% Apply semantic chunking (5)
chunks = splitter.split_documents(doc)
```

Schließlich vergleichen wir den Inhalt des ersten und zweiten Chunks, um zu sehen, an welcher Stelle der Schnitt durchgeführt wurde (siehe Listing 6.18):

```
# %% check the results (6)
chunks
# %%
pprint(chunks[0].page_content)
# %%
pprint(chunks[1].page_content)
```

Listing 6.18 Semantisches Chunking – Prüfung der Dokumente (05_VectorDatabases\20_Chunking\30_semantic_chunking.py)

6.4.4 Coding: Benutzerdefiniertes Chunking

Die gezeigten Text-Splitter bieten viel Flexibilität, aber manchmal reicht das einfach nicht aus, und Sie müssen einen benutzerdefinierten Splitter definieren, der Ihren Bedürfnissen gerecht wird.

In unserem Beispiel arbeiten wir mit einer anderen Sherlock-Holmes-Buchdatei. Diese Buchdatei beinhaltet zwölf einzelne Fälle von Sherlock Holmes. Natürlich könnten wir einfach weitermachen und den Text der Datei so aufteilen, wie wir es bisher gehandhabt haben, aber wir wollen jetzt die Informationen über die einzelnen Buchtitel extrahieren und sie zusammen mit den Textstücken speichern. Das wird später hilfreich sein: Wenn wir mit dem Buch arbeiten und Fragen dazu stellen, möchten wir vielleicht wissen, wie Sherlock Holmes den Fall »Ein Skandal in Böhmen« gelöst hat. Dann können wir zuerst auf Basis der Metadaten nach dem relevanten Buch filtern

und anschließend die Informationen abrufen, die uns interessieren. Sie werden das in Abschnitt 6.7 sehen, wenn Sie lernen, wie man Daten abruft.

Sie finden diesen Code in der Datei *05_VectorDatabases\20_Chunking\40_custom_splitter.py*. Das Buch mit dem Titel »Die Abenteuer des Sherlock Holmes« enthält wie gesagt insgesamt zwölf verschiedene Bücher über Sherlock Holmes, und wir wollen alle Titel aus der Datei extrahieren.

Aber wie immer laden wir zu Beginn des Skripts die benötigten Pakete. Die LangChain-Community hilft uns, denn es gibt eine Loader-Klasse für unseren Zweck, die Sie in Listing 6.19 sehen:

```
#%% Packages
import re
from langchain.text_
splitter import CharacterTextSplitter, RecursiveCharacterTextSplitter
from langchain_community.document_loaders import GutenbergLoader
```

Listing 6.19 Benutzerdefiniertes Chunking – erforderliche Pakete (05_VectorDatabases\20_Chunking\40_custom_splitter.py)

Listing 6.20 zeigt, wie wir die Daten direkt über die Klasse `GutenbergLoader` laden können:

```
# %% The book details
book_details = {
    "title": "The Adventures of Sherlock Holmes",
    "author": "Arthur Conan Doyle",
    "year": 1892,
    "language": "English",
    "genre": "Detective Fiction",
    "url": "https://www.gutenberg.org/cache/epub/1661/pg1661.txt"
}
loader = GutenbergLoader(book_details.get("url"))
data = loader.load()
```

Listing 6.20 Benutzerdefiniertes Datensplitten – Daten importieren (05_VectorDatabases\20_Chunking\40_custom_splitter.py)

Das Ergebnis ist eine Liste von Dokumenten, in der wir nur ein `Document`-Objekt haben, und zwar wie folgt:

```
[Document(metadata={'title': 'The Adventures of Sherlock Holmes', 'author':…}
```

Wir wollen alle Metadaten beibehalten und die bestehenden Metadaten des `Document`-Objekts mit unserem Wörterbuch wie folgt erweitern:

```
#%% Add metadata from book_details
data[0].metadata = book_details
```

Durch die Inspektion der Datei stellen wir fest, dass jeder Buchtitel in einer ähnlichen Struktur aufgebaut ist. Zum Beispiel sieht der erste Text für das zweite Buch folgendermaßen aus:

```
('II. THE RED-HEADED LEAGUE\r\n'
 '\n'
 '\n'
 '\r\n'
```

Die innere Struktur von Textstücken zu kennen, ist wichtig, denn damit können wir ein Muster erkennen und extrahieren, das zu allen Titeln passt. Jeder Titel hat die Form, dass auf eine römische Zahl ein Punkt, ein Leerzeichen sowie ein Titel in Großbuchstaben folgen. Nach diesem Muster kann man mit *regulären Ausdrücken* suchen. (Reguläre Ausdrücke sind ein größeres Thema für sich. Sie können es auf Ihre möglicherweise längere Liste von zusätzlichen Dingen setzen, die Sie lernen möchten – oder Sie machen es wie ich: Ich kopiere den Text und bitte ein LLM, mir den entsprechenden regulären Ausdruck zu erstellen, um den Text zu teilen, wann immer ein Titel erkannt wird.)

Das geschieht in Listing 6.21 mit der Funktion `custom_splitter`. Danach überschreiben wir die Methode `split_text` unseres `text_splitter`-Objekts und ersetzen sie durch unseren `custom_splitter`:

```
# %% Custom splitter
def custom_splitter(text):
    # This pattern looks for Roman numerals followed by a title
    pattern = r'\n(?=[IVX]+\.\s[A-Z])'
    return re.split(pattern, text)

text_splitter = CharacterTextSplitter(
    separator="\n",
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
    is_separator_regex=False,
)
# Override the default split method
text_splitter.split_text = custom_splitter
```

Listing 6.21 Benutzerdefiniertes Datensplitten – Funktionsdefinition (05_VectorDatabases\20_Chunking\40_custom_splitter.py)

Super. Probieren wir es aus und rufen wir die Methode `split_documents` auf, die selbst von der Methode `split_text` erbt. Wir erstellen dazu ein Objekt namens `books`:

```
# Assuming you have the full text in a variable called 'full_text'
books = text_splitter.split_documents(data)
```

Dieses Objekt hat 13 Dokumente. Das erste bezieht sich auf kein Sherlock-Holmes-Buch, sondern auf allgemeine Informationen zum Dokument. Dieses erste Listenelement können wir vernachlässigen.

```
# %% remove the first element, because it only holds metadata, not real books
books = books[1: ]
```

Als Nächstes wollen wir in Listing 6.22 für jedes Buch in der Liste die richtige Überschrift herausziehen. Wieder nutzen wir ein reguläres Ausdrucksmuster, und immer, wenn das Muster im Text gefunden wird, wird der Titel extrahiert und zu den Metadaten hinzugefügt:

```
#%% Extract the book title from beginning of page content
for i in range(len(books)):
    print(i)
    # extract title
    pattern = r'\b[IVXLCDM]+\.\s+([A-Z\s\-]+)\r\n'
    match = re.match(pattern, books[i].page_content)
    if match:
        title = match.group(1).replace("\r", "").replace("\n", "")
        print(title)
    # add title to metadata
    books[i].metadata["title"] = title
    print(title)
```

Listing 6.22 Benutzerdefiniertes Datensplitten – Extraktion des Buchtitels (05_VectorDatabases/20_Chunking\40_custom_splitter.py)

Wir sind fast fertig. Das Objekt `books` ist eine Liste von Dokumenten, und jedes hat den richtigen Titel in seinen Metadaten:

```
[Document(metadata={'title': 'A SCANDAL IN BOHEMIA', …,
 Document(metadata={'title': 'THE RED-HEADED LEAGUE', …,
…]
```

Die folgenden Schritte haben Sie schon einmal gesehen. In Listing 6.23 wenden wir einen rekursiven Text-Splitter für Zeichen an, um Teile mit einer vordefinierten maximalen Größe und Überlappung zu erstellen:

```
# %% apply RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
    is_separator_regex=False,
)
chunks = text_splitter.split_documents(books)
```

Listing 6.23 Benutzerdefiniertes Datensplitten – Anwendung des »RecursiveCharacterTextSplitter« (05_VectorDatabases\20_Chunking\40_custom_splitter.py)

Wir haben erfolgreich ein benutzerdefiniertes Splitting durchgeführt. Damit haben Sie das Ende unseres Abschnitts über das Aufteilen von Daten erreicht. Jetzt kommen wir zur nächsten Phase im Datenaufnahmeprozess – zu den Einbettungen.

6.5 Einbettungen erstellen

Führen wir uns noch einmal vor Augen, wo im Prozess wir derzeit stehen. Wir haben ein oder mehrere Dokumente geladen. Und wir haben sie in handhabbare kleine Stücke zerlegt. Aber bis zu diesem Punkt haben wir uns mit Texten beschäftigt, die für Menschen lesbar sind. Jetzt müssen wir sie in eine Darstellung übersetzen, die Maschinen verstehen können. Dieser Prozess wird *Einbettung* (engl. *Embedding*) genannt und ihn geht es in diesem Abschnitt. Wie in Abbildung 6.9 gezeigt, werden während der Einbettung Textstücke in numerische Vektoren umgewandelt.

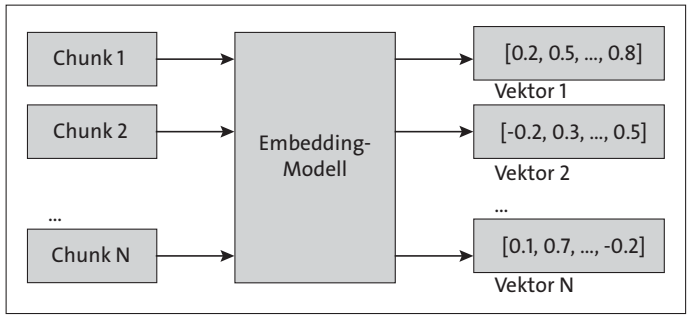


Abbildung 6.9 Der Einbettungsprozess

Ein Einbettungsvektor ist ein numerischer Vektor, der die Bedeutung eines Wortes oder eines kompletten Satzes oder Absatzes darstellt. Dieser Vektor hat nicht zwei oder drei Dimensionen, wie wir es aus der Schule kennen, sondern Hunderte oder sogar Tausende Dimensionen. Das ist notwendig, um die vollständige Bedeutung des zu Textes zu erfassen, den eine Maschine verstehen soll.

In diesem Abschnitt werden Sie herausfinden, wie Einbettungen erstellt werden, und sehen, dass ähnliche Wörter in einer zweidimensionalen Darstellung näher beieinander liegen.

Am Ende dieses Abschnitts werden Sie

- verschiedene Arten von Einbettungen kennen,
- wissen, wie Einbettungen erstellt werden,
- wissen, welche Einbettungsmodelle zur Verfügung stehen, und
- wissen, welche Arten von Einbettungen es gibt.

6.5.1 Überblick

In diesem Buch werden wir hauptsächlich vollständige Sätze einbetten, aber das Konzept der Einbettungen beschränkt sich nicht nur auf Texte. Außer Texten gibt es diverse Modalitäten, die eingebettet werden können:

- **Bild-Einbettungen (image embeddings):** Sie können Einbettungen für komplette Bilder erstellen.
- **Audio-Einbettungen:** Audio-Informationen wie Stimmen oder Musik können in eine numerische Darstellung eingebettet werden.
- **Video-Einbettungen:** Videos sind einfach eine Kombination aus Audiosignalen und einer Abfolge von Bildern, also wird es Sie nicht überraschen, dass auch Video-Einbettungen möglich sind.
- **Graph-Einbettungen:** Ein bekanntes Konzept im maschinellen Lernen sind Graphen. Graphen bestehen aus Knoten, Kanten und Merkmalen. Für bestimmte Daten ist dies die bevorzugte Art der Datenrepräsentation. Graphen können ebenfalls eingebettet werden.
- **Multimodale Einbettungen:** Verschiedene Datenmodalitäten wie Textdaten und Bilddaten können in einem gemeinsamen, einheitlichen Vektorraum eingebettet werden. Ein Beispiel dafür ist das CLIP-Modell von OpenAI, das die gleichzeitige Einbettung von Bildern und dem entsprechenden Text ermöglicht.

Es ist gut zu wissen, dass es möglich ist, diese verschiedenen Datentypen einzubetten, aber warum sollten Sie das tun? Was sind die praktischen Anwendungen von Einbettungen?

- *Natural Language Processing*-Modelle ermöglichen das Einbetten von Texten.
- Im Bereich der Bildverarbeitung (*Computer-Vision*) werden Einbettungen verwendet, um die Bedeutung von Bildern zu modellieren.
- *RAG*-Anwendungen basieren auf *semantischer Suche*. Beide nutzen Einbettungen, um effizient relevante Abschnitte abzurufen.

- Produkte und ihre Eigenschaften können eingebettet werden, und deshalb werden Einbettungsmodelle in *Empfehlungssystemen* verwendet.

Was ist der Stand der Technik bei Einbettungsmodellen?

Der Fortschritt in der generativen KI ist enorm schnell, daher wäre eine statische Liste von Modellen in diesem Buch schnell veraltet. Stattdessen möchte ich Ihnen eine Ressource empfehlen, die aktuelle Rankings bietet.

Das *MTEB-Leaderboard* (*https://huggingface.co/spaces/mteb/leaderboard*) ist ein Bereich auf Hugging Face, der verschiedene Einbettungsmodelle nach unterschiedlichen Parametern bewertet.

MTEB steht für *Massive Text Embedding Benchmark*. Dieser Benchmark-Test basiert auf 8 Embedding-Aufgaben und 58 Datensätzen in 112 Sprachen.

Was sind wichtige Kennzahlen, die Sie beachten sollten, wenn Sie ein Embedding-Modell auswählen möchten? Ein sehr wichtiger Aspekt ist die *Leistung* des Modells. Im MTEB-Leaderboard können Sie zwischen verschiedenen Leistungskennzahlen wählen, wie der »durchschnittlichen Leistung über 56 Datensätze«. Es gibt auch aufgabenbezogene Leistungskennzahlen wie die »durchschnittliche Leistung bei Klassifikation oder Clustering«.

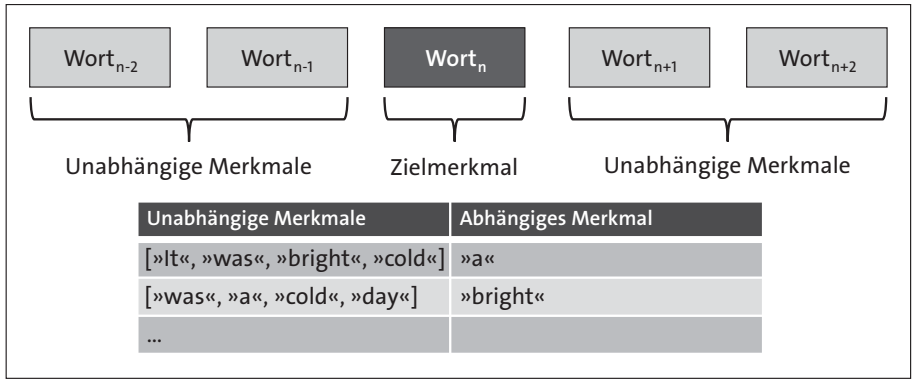
Wenn Sie das Modell selbst hosten möchten, ist es wichtig, die *Modellgröße* und den *Speicherverbrauch* zu berücksichtigen. Je nach Text, mit dem Sie arbeiten, müssen Sie die passende Chunk-Größe definieren. Dann müssen Sie sicherstellen, dass die Chunk-Größe unter dem maximalen Token-Limit des Modells liegt.

Ein weiterer wichtiger Aspekt ist der *Preis*. Viele Unternehmen bieten Zugang zu ihren Modellen über APIs an. Während viele Einbettungsmodelle Open Source und kostenlos sind, kann es dennoch teuer sein, sie auf internen Ressourcen des Unternehmens (also On-Premise) auszuführen: Ihr eigener Server muss ja rund um die Uhr laufen, idealerweise mit einer guten GPU für schnelle Inferenz, und er verbraucht Energie. Daher gibt es feste und variable Kosten, die mit diesem Ansatz verbunden sind. Sie müssen auch ein zusätzliches System warten. Aber der Vorteil ist, dass Sie sich keine Sorgen um den *Datenschutz* machen müssen, da die Daten Ihres Unternehmens Ihr eigenes Netzwerk nicht verlassen.

6.5.2 Coding: Wort-Einbettungen

Der Hauptzweck von Einbettungen besteht darin, die semantische Bedeutung von Wörtern zu erfassen. Wir können das sogar visualisieren und sehen dann, dass ähnliche Wörter näher beieinander liegen als zufällige Wörter. Wie wird das erreicht?

Die bahnbrechende Idee war ein Algorithmus namens *Skip-Gram*. Er wird in Abbildung 6.10 schematisch dargestellt. Die Hauptidee ist im Nachhinein einfach. Ein tatsächlicher Text, wie ein Buch, wird als Trainingsdaten verwendet. Der Fokus liegt immer auf mehreren Wörtern, etwa fünf. Wir beginnen zum Beispiel mit den ersten fünf Wörtern, dann wird der Fokus um ein Wort auf die Wörter 1–6 verschoben, dann auf die Wörter 2–7 und so weiter.



Unabhängige Merkmale	Abhängiges Merkmal
[»It«, »was«, »bright«, »cold«]	»a«
[»was«, »a«, »cold«, »day«]	»bright«
...	

Abbildung 6.10 Skip-Gram-Beispiel

Von dem aktuellen Fokus auf fünf Wörtern ist das zentrale Wort das *Zielmerkmal*, und die beiden benachbarten Wörter links und rechts sind *unabhängige Merkmale* in einem neuronalen Netzwerkmodell. Die unabhängigen Merkmale werden verwendet, um das Zielmerkmal vorherzusagen. Wenn zum Beispiel der aktuelle Fokus auf fünf Wörtern wie »Es war ein heller kalter ...« liegt, dann wird das zentrale Wort »ein« als Ziel betrachtet, und die anderen Wörter werden verwendet, um dieses Wort vorherzusagen. Die Annahme dahinter ist, dass ein Wort durch den Kontext beschrieben wird, in dem es verwendet wird.

Der entsprechende Algorithmus heißt *Word2vec*. Word2vec ist ein frühes Modell für Wort-Einbettungen. Im Kasten finden Sie Informationen über einige andere beliebte Modelle.

Eine kurze Geschichte der Wort-Einbettungsmodelle

Word2vec wurde 2013 von Google entwickelt und weckte großes Interesse an Wort-Embeddings.

Die Forscher von Stanford haben 2014 das Modell *GloVe* entwickelt, was für *Global Vectors for Word Representation* steht. Seine Leistung ist mit der von Word2vec vergleichbar.

Facebook kam 2016 ins Spiel und veröffentlichte *FastText*. Das Modell ist eine Erweiterung von Word2vec und kann sogar mit Wörtern umgehen, die nicht im Wortschatz enthalten sind. Das war besonders nützlich für Sprachen mit vielen Wörtern.

Das einflussreichste Modell ist *BERT* (kurz für *Bidirectional Encoder Representations from Transformers*). Es wurde 2018 von Forschern bei Google entwickelt und basiert auf der *Transformer*-Architektur. Es gehört immer noch zu den beliebtesten Modellen für das Wort-Embedding.

Wir werden nun die Modell-Einbettungen für alle eingebetteten Wörter herunterladen und visuell zeigen, dass ähnliche Wörter näher beieinander liegen als zufällige Wörter. Lassen Sie uns das entwickeln und in einem Diagramm darstellen. Die Code-Datei finden Sie in den Downloadmaterialien unter *05_VectorDatabases\30_Embedding\10_word2vec_similarity.py*.

Wir verwenden das Paket `gensim`, um Word2vec-Embeddings zu erhalten, wie in Listing 6.24 gezeigt. `Seaborn` wird für die Visualisierung verwendet. Um die Daten zu plotten, müssen wir die Dimensionen auf zwei reduzieren. Dafür nutzen wir `PCA` von `sklearn`.

```
#%% (1) Packages
import gensim.downloader as api  # Package for downloading word vectors
import random  # Package for generating random numbers
import seaborn.objects as so # Package for visualizing the embeddings
from sklearn.decomposition import PCA # import PCA
import numpy as np
import pandas as pd
```

Listing 6.24 Wort-Einbettungen – erforderliche Pakete (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Als Nächstes laden wir die Word2vec-Embeddings. Das kann eine Weile dauern, da das Modell `word2vec-google-news-300` 1,5 GByte groß ist.

```
# %% (2) import GloVe word vectors
word_vectors = api.load("word2vec-google-news-300")
```

Wir beschäftigen uns mit dem Wort »mathematics«, aber Sie können damit spielen und ein eigenes Wort wählen. Die Formeigenschaft wird überprüft, und wir bestätigen, dass die Einbettung 300-dimensional ist:

```
# %% (3) get the size of the word vector
studied_word = 'mathematics'
word_vectors[studied_word].shape
```

Um einen Eindruck von den Einbettungen zu bekommen, können wir sie uns anschauen. Das ist ein Vektor mit 300 Doppelzahlen, der die semantische Bedeutung des Wortes »intelligence« repräsentiert.

```
# %% (4) get the word vector for the word 'intelligence'
word_vectors[studied_word]
```

Wir können die Methode `most_similar` verwenden, um die ähnlichsten Wörter zu dem Wort zu bekommen, das wir gerade untersuchen. Wir haben noch nicht erklärt, wie diese Ähnlichkeit berechnet wird. Damit befassen wir uns in Abschnitt 6.7.

```
# %% (5) get similar words to 'intelligence'
word_vectors.most_similar(studied_word)
```

Schließlich beschränken wir uns auf die fünf ähnlichsten Wörter, wie folgt:

```
# %% (6) get a list of strings that are similar to 'intelligence'
words_similar = [w[0] for w in word_vectors.most_similar(studied_word)][:5]
```

Jetzt wählen wir 20 Wörter aus zufälligen Positionen im Einbettungsmodell aus (siehe Listing 6.25):

```
# %% (7) get random words from word vectors
num_random_words = 20
all_words = list(word_vectors.key_to_index.keys())
# set the seed for reproducibility
random.seed(42)
random_words = random.sample(all_words, num_random_words)

# Print the random words
print("Random words extracted:")
for word in random_words:
    print(word)
```

Listing 6.25 Wort-Einbettungen – Zufallswörter erzeugen (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Wir haben 20 zufällige Wörter und 5 Wörter wie »Mathematik«. Für diese 25 Wörter extrahieren wir die Einbettungen und speichern sie in einem Numpy-Array mit den Dimensionen (25, 300). Das sehen Sie in Listing 6.26:

```
# %% (8) get the embeddings for random words and similar words
words_to_plot = random_words + words_similar
embeddings = np.array([])
for word in words_to_plot:
    embeddings = np.vstack([embeddings, word_vectors[
word]]) if embeddings.size else word_vectors[word]
```

Listing 6.26 Wort-Einbettungen – Einbettungen erzeugen (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Da wir einen 300-dimensionalen Vektor nicht visualisieren können, müssen wir die Dimensionen auf eine handhabbare Größe reduzieren. Das geschieht in Listing 6.27 mit der Hauptkomponentenanalyse. Typischerweise sind Plots zweidimensional, also bleiben wir dabei. Es gibt verschiedene Techniken zur *Dimensionsreduktion*. Eine der beliebtesten ist nach wie vor die *Hauptkomponentenanalyse* (engl. *Principal Component Analysis*, *PCA*), die uns dabei helfen wird.

```
# %% (9) create 2D representation via PCA
pca = PCA(n_components=2)
embeddings_2d = pca.fit_transform(embeddings)

df = pd.DataFrame(embeddings_2d, columns=["x", "y"])
df["word"] = words_to_plot
# red for random words, blue for similar words
df["color"] = ["random"] * num_random_words + ["similar"] * len(words_similar)
```

Listing 6.27 Wort-Einbettungen – Hauptkomponentenanalyse (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Schließlich plotten wir die Wörter mit `seaborn` so, wie in Listing 6.28 gezeigt. Wenn Sie noch nicht mit Seaborns neuem Ansatz der Grafikgrammatik (*grammar-of-graphics*) gearbeitet haben, Schauen Sie sich das mal unter *https://seaborn.pydata.org/tutorial/objects_interface.html* an.

Dieser Ansatz ist wirklich interessant, und ich empfehle ihn gegenüber der klassischen Nutzung des Frameworks, weil er funktioniert, als würde man Schichten übereinanderstapeln. Dadurch hat er eine gewisse Ähnlichkeit mit neuronalen Netzwerken. Eine Grafik kann somit modular entwickelt werden.

```
# %% (10) visualize the embeddings using seaborn
(so.Plot(df, x="x", y="y", text="word", color="color")
 .add(so.Text())
 .add(so.Dots())
)
```

Listing 6.28 Wort-Einbettungen – Visualisierung (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Das Ergebnis unserer Arbeit an Wort-Einbettungen und deren Darstellung in zwei Dimensionen sehen Sie in Abbildung 6.11.

Die Wörter, die wie »mathematics« sind, werden in einer Farbe angezeigt, während die zufälligen Wörter in einer anderen Farbe erscheinen. Ähnliche Wörter sind viel näher beieinander als zufällige Wörter. Das wollte ich mit diesem Beispiel zeigen,

denn es bedeutet, dass die semantische Bedeutung von Wörtern durch die Einbettungen dargestellt wird.

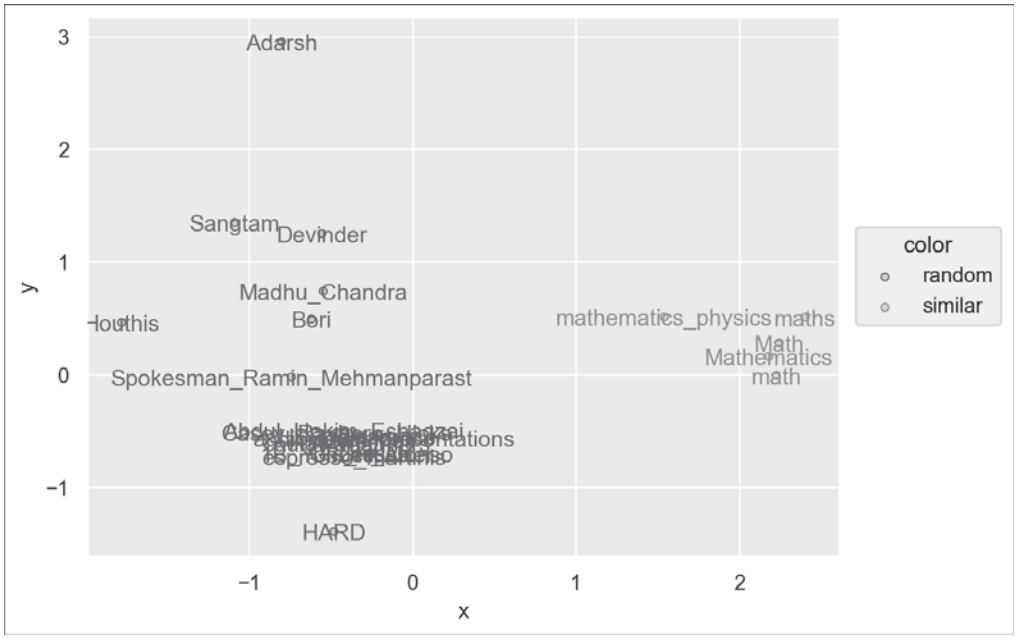


Abbildung 6.11 Word2vec für zufällige Wörter und ähnliche Wörter mit Bezug zu »mathematics«

Jetzt, da wir eine numerische Darstellung haben, können wir mit ihr arbeiten und Beziehungen visualisieren. In den Beispielen in Listing 6.29 erstellen wir Beziehungen zwischen Hauptstädten und den entsprechenden Ländern. Wir holen zunächst die Einbettungen für Berlin, Paris, Madrid und die Länder Deutschland, Frankreich und Spanien. Dann reduzieren wir die Dimensionen der Einbettungen wieder über PCA und plotten die Verbindungen zwischen Hauptstadt und Land als Vektor.

```
# %% visualizing it via lines
df_arithmetic = pd.DataFrame({'word': [
'paris', 'germany', 'france', 'berlin', 'madrid', 'spain']})
# add embeddings and add x- and y-coordinates for PCA
pca = PCA(n_components=2)
embeddings_arithmetic = np.array([])
for word in df_arithmetic['word']:
    embeddings_arithmetic = np.vstack([embeddings_arithmetic, word_vectors[
word]]) if embeddings_arithmetic.size else word_vectors[word]

# apply PCA
embeddings_arithmetic_2d = pca.fit_transform(embeddings_arithmetic)
```

```
df_arithmetic['x'] = embeddings_arithmetic_2d[:, 0]
df_arithmetic['y'] = embeddings_arithmetic_2d[:, 1]

#%% visualise it via matplotlib with lines
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 10))
plt.scatter(df_arithmetic['x'], df_arithmetic['y'], marker='o')

# add vector from paris to france, and berlin to germany
plt.arrow(df_arithmetic['x'][0], df_arithmetic['y'][0],
    df_arithmetic['x'][2] - df_arithmetic['x'][0],
    df_arithmetic['y'][2] - df_arithmetic['y'][0],
    head_width=0.01, head_length=0.01, fc='r', ec='r')
plt.arrow(df_arithmetic['x'][3], df_arithmetic['y'][3],
    df_arithmetic['x'][1] - df_arithmetic['x'][3],
    df_arithmetic['y'][1] - df_arithmetic['y'][3],
    head_width=0.01, head_length=0.01, fc='r', ec='r')
plt.arrow(df_arithmetic['x'][4], df_arithmetic['y'][4],
    df_arithmetic['x'][5] - df_arithmetic['x'][4],
    df_arithmetic['y'][5] - df_arithmetic['y'][4],
    head_width=0.01, head_length=0.01, fc='r', ec='r')
# add labels for words
for i, txt in enumerate(df_arithmetic['word']):
    plt.annotate(txt,(df_arithmetic['x'][i],
        df_arithmetic['y'][i]))
```

Listing 6.29 Word2vec-Einbettungen und Vektor-Repräsentation (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

Die Codeausgabe unserer Vektor-Einbettungen von Hauptstädten und Ländern ist in Abbildung 6.12 zu sehen.

Die Vektoren haben immer eine ähnliche Richtung. Wir können diese Beziehungen nutzen und mit diesen Vektoren so arbeiten, wie in Listing 6.30 gezeigt. Wir können also die folgende Gleichung formulieren:

```
Paris - France + Germany = ???
```

```
#  Paris - France + Germany = Berlin
word_vectors.most_similar(positive = ["paris", "germany"],
    negative= ["france"], topn=1)
```

Listing 6.30 Word2vec – algebraische Berechnungen (05_VectorDatabases\30_Embedding\10_word2vec_similarity.py)

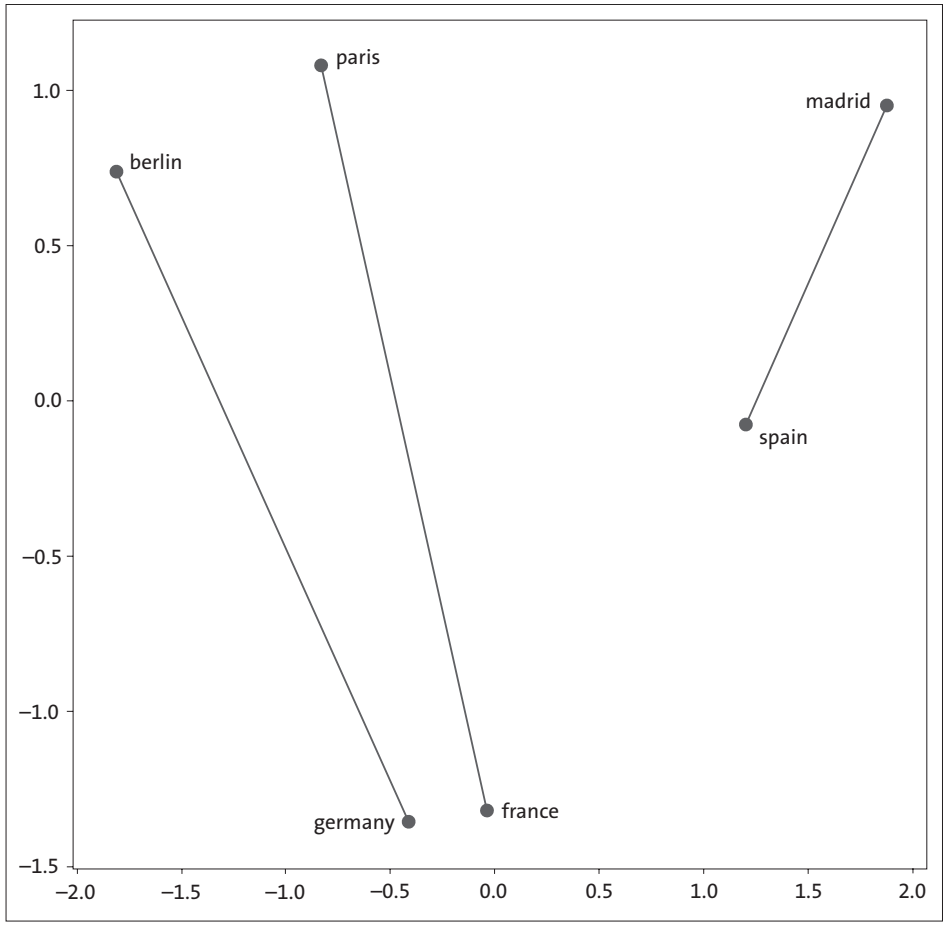


Abbildung 6.12 Word2vec-PCA-Repräsentation für Hauptstädte und Länder

Bitte prüfen Sie das einmal selbst. Der Code sollte `Berlin` als Ergebnis liefern.

Jetzt können wir einzelne Wörter einbetten. Das ist großartig, aber es gibt einige Probleme, sobald wir versuchen, mit vollständigen Sätzen zu arbeiten. Schauen Sie sich die beiden folgenden Sätze an. Beide enthalten das Wort »Bank«, aber beide haben völlig unterschiedliche Bedeutungen:

- Satz 1: »I need to go to the *bank* to deposit this check.«
- Satz 2: »We had a picnic on the *bank* of the river.«

Wir können also festhalten, dass die Bedeutung eines Wortes vom Kontext abhängt, in dem es verwendet wird. Ein einzelnes Embedding für das Wort »bank« ist hier nicht sinnvoll. Daher müssen wir einen Ansatz für vollständige Sätze finden.

6.5.3 Coding: Satzeinbettungen

Satz-Einbettungsmodelle ähneln Wort-Einbettungsmodellen sehr. Aber anstatt nur ein Wort zu nehmen und dessen Einbettungsvektor zu liefern, können wir jetzt einen kompletten Satz an das Modell übergeben und bekommen einen Einbettungsvektor, der den gesamten Satz repräsentiert. Ich habe bereits mehrfach BERT erwähnt, eines der einflussreichsten Wort-Embedding-Modelle. Lassen Sie sich nicht von der Tatsache verwirren, dass es Varianten von BERT gibt, die auch Sätze und komplette Absätze verarbeiten können. Eine typische BERT-Sequenz könnte so aussehen, wie in Abbildung 6.13 gezeigt.

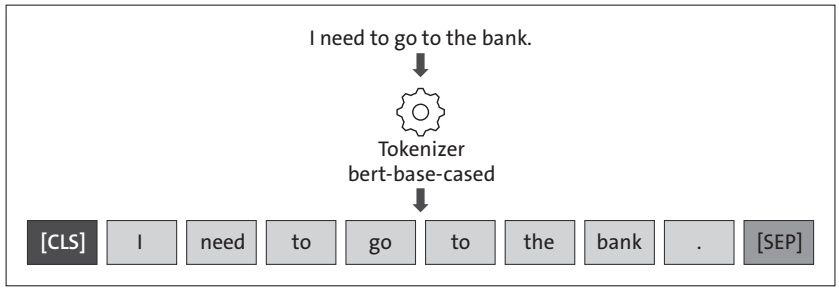


Abbildung 6.13 Beispiel für eine Satzeinbettung mit BERT

Wir definieren einen Satz, lassen ihn durch den Tokenizer (aber noch nicht durch das Einbettungsmodell) laufen und erhalten eine Tokenisierung. Jedes Wort unseres Satzes wird tokenisiert. Auch der Punkt wird für sich tokenisiert. Aber Sie werden auch zwei neue Tokens entdecken: [CLS] und [SEP]. [SEP] trennt jeden Satz. Interessanter ist das [CLS]-Token, was für »Klassifikationstoken« steht oder manchmal als »Klassen-Token« bezeichnet wird. Das ist ein spezielles Token, das in Transformer-basierten Modellen (besonders in BERT) verwendet wird. Es hat die Aufgabe, Informationen aus einer gesamten Eingabesequenz zu aggregieren.

Typischerweise befindet [CLS] sich als erstes Token in der Eingabesequenz, vor den eigentlichen Inhaltstoken. Dieses Token lernt, die Informationen auf Satz- oder Absatzebene zu kodieren. Im ursprünglichen BERT-Papier haben die Autoren vorgeschlagen, den finalen versteckten Zustand des [CLS]-Tokens als Satz-Embedding zu verwenden. Es gibt auch andere Ansätze, aber dieses Beispiel soll Ihnen eine Vorstellung davon geben, wie die Bedeutung eines Satzes von den Modellen behandelt werden könnte.

Die tatsächliche Nutzung von Satz-Embedding-Modellen ist ziemlich einfach, wie Sie im folgenden Beispiel sehen werden. Wir werden ein paar Sätze einbetten und danach ihre Korrelationen überprüfen. Die entsprechende Code-Datei ist *05_VectorDatabases\30_Embedding\20_sentence_similarity.py*.

Ein neues Paket wird verwendet: `sentence_transformers`, wie in Listing 6.31 gezeigt. Über dieses Paket haben Sie Zugriff auf viele verschiedene Modelle, die sich auf das Einbetten kompletter Sätze konzentrieren.

```
#%% (1) Packages
from sentence_transformers import SentenceTransformer
import numpy as np
import seaborn as sns
```

Listing 6.31 Satz-Einbettungen – benötigte Pakete (05_VectorDatabases\30_Embedding\20_sentence_similarity.py)

Wir verwenden nun ein Embedding-Modell, das mehrere Sprachen verarbeiten kann, und erstellen eine Instanz dieses Modells. Das gewählte Modell hat eine Embedding-Größe von 512, sodass jeder einzelne Satz in einen 512-dimensionalen Vektor umgewandelt wird.

```
#%% (2) Load the model
MODEL = 'sentence-transformers/distiluse-base-multilingual-cased-v1'
model = SentenceTransformer(MODEL)
```

Acht Beispielsätze wurden ausgewählt, die Sie in Listing 6.32 sehen. Fühlen Sie sich frei, eigene Sätze zu definieren. Die ersten drei Sätze sind sich ziemlich ähnlich. Das heißt, haben eine sehr ähnliche Bedeutung, obwohl sie unterschiedliche Wörter verwenden, um eine Situation zu beschreiben. Das Gleiche gilt für die letzten drei Sätze. Sie sind tatsächlich Übersetzungen aus dem Englischen ins Französische und Deutsche. Die Sätze 4 und 5 in der Liste unterscheiden sich hingegen sehr vom Rest.

```
# %% (3) Define the sentences
sentences = [
    'The cat lounged lazily on the warm windowsill.',
    'A feline relaxed comfortably on the sun-soaked ledge.',
    'The kitty reclined peacefully on the heated window perch.',
    'Quantum mechanics challenges our understanding of reality.',
    'The chef expertly julienned the carrots for the salad.',
    'The vibrant flowers bloomed in the garden.',
    'Las flores vibrantes florecieron en el jardín. ',
    'Die lebhaften Blumen blühten im Garten.'
]
```

Listing 6.32 Satz-Einbettungen – Beispielsätze (05_VectorDatabases\30_Embedding\20_sentence_similarity.py)

Wir erstellen die Einbettungen, indem wir nun die `encode`-Methode unseres Modells aufrufen und unsere Liste von Sätzen übergeben. Es ist immer hilfreich, die Struktur der Daten zu überprüfen. Wir haben 8 Sätze und dass die Embedding-Größe unseres Modells 512 beträgt, haben Sie bereits gelesen. Daher haben unsere Einbettungen die Form (8, 512).

```
# %% (4) Get the embeddings
sentence_embeddings = model.encode(sentences)
```

Wir wollen die Korrelation zwischen den Sätzen visualisieren. Dazu berechnen wir zuerst den Korrelationskoeffizienten zwischen den acht verschiedenen Sätzen, wie in Listing 6.33 gezeigt:

```
# %% (5) Calculate linear correlation matrix for embeddings
sentence_embeddings_corr = np.corrcoef(sentence_embeddings)
import seaborn as sns
# show annotation with one digit
sns.heatmap(sentence_embeddings_corr, annot=True,
      fmt=".1f",
      xticklabels=sentences,
      yticklabels=sentences)
```

Listing 6.33 Satz-Einbettungen – Korrelation und Visualisierung (05_VectorDatabases\30_Embedding\20_sentence_similarity.py)

Als Ergebnis erhalten wir die Korrelationen zwischen Satzeinbettungen, wie in Abbildung 6.14 dargestellt, mit einer 8×8-Matrix, die wir mit `seaborn` übersichtlich visualisieren können.

Sie sehen hohe Korrelationskoeffizienten in der oberen linken Ecke des Graphen. Das spiegelt die Ähnlichkeit der ersten drei Sätze wider. Ähnlich können Sie hohe Korrelationskoeffizienten in der unteren rechten Ecke des Graphen beobachten. Diese drei Sätze haben genau die gleiche Bedeutung, sie sind nur in drei verschiedenen Sprachen formuliert. Auf der Formebene sind sie also völlig unterschiedlich, während sie inhaltlich fast deckungsgleich sind. Da wir ein Einbettungsmodell verwendet haben, das auf Datensätzen aus verschiedenen Sprachen trainiert wurde, kann es die Inhaltsebene verstehen. Wir schließen daraus, dass ein mehrsprachiges Modell sich nicht um Sprachen kümmert, sondern nur um die inhaltliche Bedeutung des Textes. Die anderen Sätze waren nur zum Vergleich hinzugefügt. Sie zeigen, dass Sätze, die sehr unterschiedlich sind, Korrelationskoeffizienten um null haben.

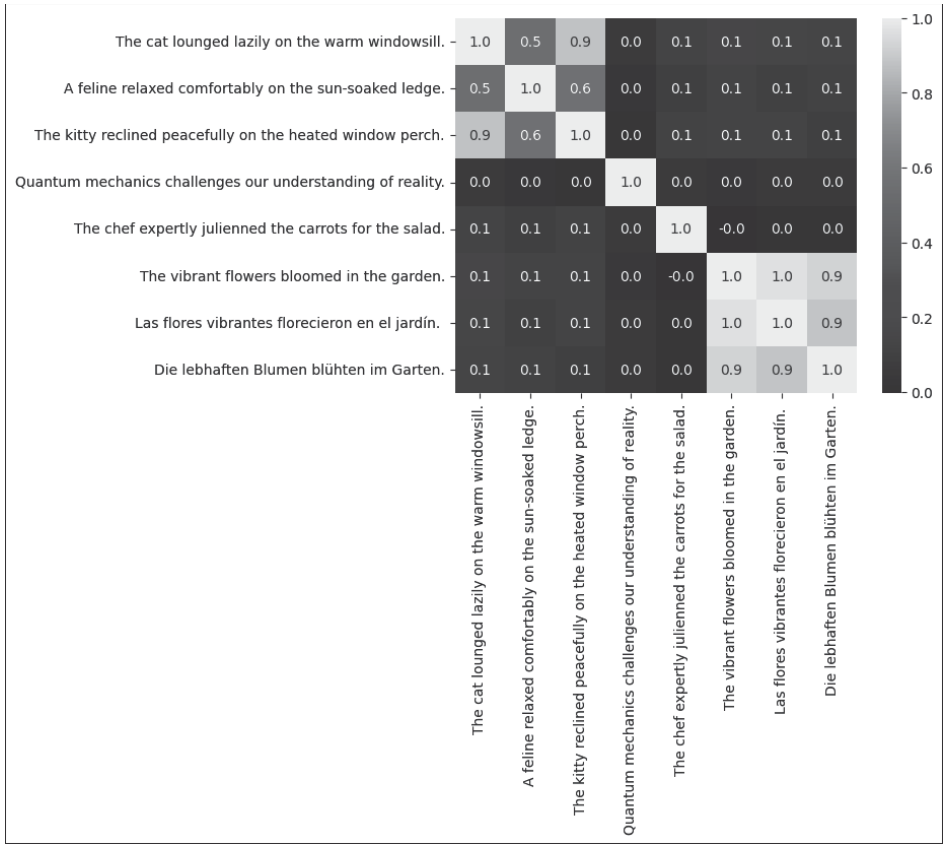


Abbildung 6.14 Korrelationen zwischen Satzeinbettungen (05_VectorDatabases/30_Embedding/20_sentence_similarity.py)

6.5.4 Coding: Einbettungen mit LangChain erzeugen

In Abschnitt 6.4.3 über Semantisches Chunking hatten wir einen Artikel aus der Wikipedia über künstliche Intelligenz geladen. Darauf bauen wir jetzt auf und extrahieren den Text aus unserer Liste von `Document`-Objekten, sodass jeder Text eingebettet wird. Wir fangen mit ein bisschen Boilerplate-Code an und erweitern ihn. Sie finden den Code in *05_VectorDatabases/30_Embedding/30_wikipedia_embeddings.py*.

In diesem Code laden wir eine zusätzliche Klasse, um die OpenAI-Einbettungen nutzen zu können. Der Rest des Codes wird Ihnen mittlerweile vertraut sein. Der Wikipedia-Artikel über künstliche Intelligenz wird geladen und anschließend aufgeteilt.

Um ein OpenAI-Einbettungsmodell zu verwenden, müssen Sie einen API-Schlüssel von OpenAI haben und ihn in der Datei *.env* speichern. Diese Datei wird mit `load_dotenv()` geladen, wie in Listing 6.34 gezeigt:

```
#%% Packages
from langchain.document_loaders import WikipediaLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from pprint import pprint
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv(usecwd=True))
```

Listing 6.34 Einbettungen mit LangChain – erforderliche Pakete (05_VectorDatabases/30_Embedding/30_wikipedia_embeddings.py)

Aus der Wikipedia laden wir nun wieder einen Artikel über künstliche Intelligenz (siehe Listing 6.35):

```
# %% Load the article
ai_article_title = "Artificial_intelligence"
loader = WikipediaLoader(query=ai_article_title,
    load_all_available_meta=True,
    doc_content_chars_max=10000,
    load_max_docs=1)
doc = loader.load()
```

Listing 6.35 Wikipedia-Artikel laden – die Datei 05_VectorDatabases/30_Embedding/30_wikipedia_embeddings.py

Das `Document`-Objekt ist in Teile aufgeteilt, wie Sie in Listing 6.36 sehen können – diesen Schritt kennen Sie mittlerweile.

```
# %% Create splitter instance
splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
    chunk_overlap=200,
    separators=["\n\n", "\n"," ", ".", ","])

# %% Apply semantic chunking
chunks = splitter.split_documents(doc)
# %% Number of Chunks
len(chunks)

# Ausgabe:
16
```

Listing 6.36 Einbettungen mit LangChain – Daten splitten (05_VectorDatabases/30_Embedding/30_wikipedia_embeddings.py)

Als Ergebnis erhalten wir 16 Chunks, die wir einbetten werden. Aber zuerst müssen wir eine Instanz der Embedding-Klasse erstellen. Hier wenden wir ein OpenAI-Embedding-Modell namens `text-embedding-3-small` an. Obwohl "small" nicht ganz zutreffend ist: Dieses Embedding-Modell hat 1536 Dimensionen. Sie könnten auch den großen Bruder dieses Modells, `text-embedding-3-large`, verwenden. Der hat 3072 Dimensionen.

```
# %% Create instance of embedding model
embeddings_model = OpenAIEmbeddings(model="text-embedding-3-small")
```

Aber wir können noch keine Liste von Dokumenten an das Modell übergeben. Daher werden wir die Methode `embed_documents` verwenden, die einen erforderlichen Parameter `text` hat. Dieser Parameter sollte eine Liste von Strings übergeben. Daher müssen wir den `page_content` aus jedem Chunk extrahieren und in dem Objekt `texts` speichern. Dieses Objekt wird dann als Parameter an das Embedding-Modell übergeben, um die Einbettungen zu erstellen:

```
# %% extract the texts from "page_content" attribute of each chunk
texts = [chunk.page_content for chunk in chunks]
# %% create embeddings
embeddings = embeddings_model.embed_documents(texts=texts)
```

Sie können die Länge dieses Objekts (16) überprüfen, was der Anzahl der Textstrings entspricht, die wir dem Einbettungsmodell übergeben haben, wie in Listing 6.37 gezeigt wird. Für jeden Text wurde also ein Einbettungsvektor erstellt, der einen 1536-dimensionalen Vektor hat.

```
# %% get number of embeddings
len(embeddings)

# Ausgabe:
16
# %% check the dimension of the embeddings
len(embeddings[0])

# Ausgabe:
1536
```

Listing 6.37 Einbettungen mit LangChain – Dimension der Einbettungen (05_VectorDatabases/30_Embedding/30_wikipedia_embeddings.py)

Damit sind wir am Ende des Abschnitts über Einbettungen angelangt. Im folgenden Abschnitt arbeiten wir mit dem nächsten Schritt in der Pipeline weiter. Dabei werden Sie lernen, wie man die Daten in einer Vektordatenbank speichert.

Kapitel 7
Retrieval-Augmented Generation

»Manchmal sind es die Menschen, von denen sich niemand etwas vorstellt, die Dinge tun, die sich niemand vorstellen kann.«
– Alan Turing im Film »The Imitation Game«

In diesem Kapitel werden Sie eines der wirkungsvollsten Konzepte im Bereich der großen Sprachmodelle verstehen lernen und erstellen – die *Retrieval-Augmented Generation* (RAG). Bisher haben wir verschiedene ihrer Puzzlestücke – wie LLMs, Prompt Engineering und Vektordatenbanken – gesammelt, die jetzt perfekt zusammenpassen.

Ein wichtiger Aspekt sind Vektordatenbanken (siehe Kapitel 6), die typischerweise das Rückgrat eines *RAG*-Systems darstellen. Um das Modell anzuweisen, wie es das endgültige Ergebnis erstellen soll, werden wir auf das Prompt Engineering zurückkommen und die Konzepte wiederverwenden, die Sie dazu in Kapitel 5 gelernt haben. Anschließend werden Daten an ein großes Sprachmodell übergeben, und an diesem Punkt kommen wir auf das Wissen zurück, das Sie in Kapitel 4 erworben haben. In dieser Hinsicht ist RAG ein Berg, den Sie nur erklimmen können, nachdem Sie in diesen drei verschiedenen Disziplinen fit geworden sind.

Ich beginne das Kapitel mit einer Einführung in das Konzept und bespreche dann die einzelnen Prozessschritte *Retrieval*, *Augmentation* und *Generation*.

Wir werden in Abschnitt 7.2 ein erstes und einfaches RAG-System entwickeln. Dieses System wird überraschend gut funktionieren, aber nicht perfekt sein. Aus diesem Grund gibt Abschnitt 7.3 einen Überblick über fortgeschrittene RAG-Techniken. Diese Verbesserungen können in verschiedenen Phasen des RAG-Systems stattfinden. Fortgeschrittene Techniken für die *Pre-Retrieval*-Phase werden in Abschnitt 7.3.1 vorgestellt. In Abschnitt 7.3.2 zeige ich Ihnen Techniken für die Retrieval-Phase und in Abschnitt 7.3.3 für die Post-Retrieval-Phase.

Eine Alternative zu RAG, die als *Prompt-Caching* bezeichnet wird, sehen wir uns in Abschnitt 7.4 an.

Wenn Sie ein RAG-System entwickeln und es verbessern möchten, benötigen Sie Metriken, die es Ihnen ermöglichen, diese Verbesserungen zu bewerten und zu quantifizieren. Daher widmen wir Abschnitt 7.5 den RAG-Bewertungstechniken.

Lassen Sie uns einfach mit dem beginnen, was RAG ist, warum es benötigt wird und wie es funktioniert.

7.1 Einleitung

Bevor wir uns damit beschäftigen, wie RAG funktioniert, halten wir einen Moment inne und denken darüber nach, warum es überhaupt nötig ist. Stellen Sie sich vor, Sie möchten mit einer großen Wissensquelle arbeiten, z. B. einem langen Dokument, und Sie möchten mit diese Quelle »chatten«.

Der einfachste Ansatz wäre, einen LLM-Aufruf zu machen, in dem Sie Ihre Frage sowie das gesamte Dokument als Kontext übergeben. Dieser Ansatz ist in der Regel jedoch unpraktikabel und unwirtschaftlich, da das große Dokument das Kontextfenster des LLM überschreitet und/oder da es ineffizient wäre, bei jeder Anfrage das komplette Dokument zu senden. Die Anfrage würde vergleichsweise lange dauern, um eine Antwort zu erhalten; und es wäre auch sehr kostspielig, da bei jeder einzelnen Anfrage viele unnötige Tokens gesendet werden.

Eine spezielle Lösung ist die Verwendung des *Prompt-Caching*. Darüber werden wir in Abschnitt 7.4 sprechen. Aber das ist eher eine Lösung für Ausnahmefälle. Der üblichere Ansatz ist die Verwendung von RAG. Mit RAG werden bei jeder LLM-Anfrage nur die relevantesten Dokumente zusammen mit der Benutzeranfrage an das LLM übergeben. Das führt zu einer sehr effizienten und schnellen Lösung. Außerdem überwindet es einige Einschränkungen von LLMs. Mit RAG können Sie also Folgendes ermöglichen:

- Zugang des LLMs zu firmeninternen Dokumenten
- Zugang zu hochaktuellen Informationen, die nach dem Knowledge-Cutoff-Datum des LLM-Trainings erfasst wurden
- faktenbasiertere Ergebnisse

Abbildung 7.1 zeigt den allgemeinen RAG-Prozess. Ein Nutzer erstellt eine Anfrage (*User Query*), die an einen *Retriever* weitergeleitet wird. Der Retriever ist dafür zuständig, relevante Informationen aus einer externen Datenquelle abzurufen. In den meisten Fällen ist diese externe Quelle eine Vektordatenbank, aber Sie sind nicht darauf beschränkt. Alternativ könnten Sie auch als Retriever eine Internetsuche implementieren, um relevante Informationen zu finden.

Der nächste Schritt in der Pipeline ist die *Augmentation*. Dabei nutzen wir Prompt-Engineering, um Anweisungen mit den relevanten Dokumenten zu bündeln. Diese Anweisungen teilen dem Sprachmodell später mit, wie es die relevanten Dokumente verarbeiten soll, aber auch, wie es sich verhalten soll, wenn die relevanten Dokumente nicht gut geeignet sind, um die Informationen herauszuziehen. Als Ergebnis der

Augmentation haben wir Dokumente und die entsprechenden Anweisungen. Diese werden jetzt an die letzte Stufe der Pipeline weitergegeben: an ein großes Sprachmodell.

Das Sprachmodell wird die Anweisungen verarbeiten, Informationen aus den Dokumenten extrahieren und eine Ausgabe gemäß der Nutzeranfrage erstellen.

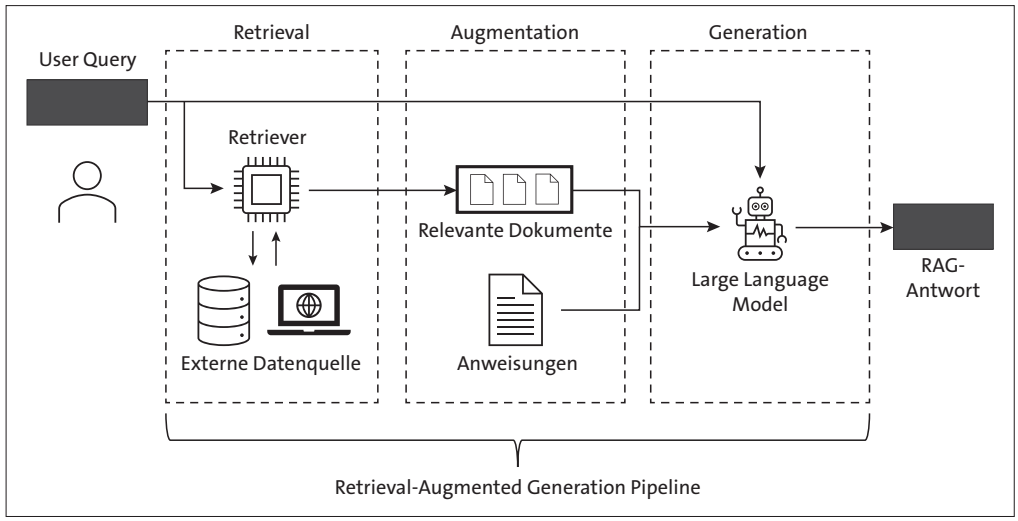


Abbildung 7.1 RAG – allgemeine Pipeline

Die externe Quelle ist in den meisten Fällen eine Vektordatenbank. Schauen wir uns das RAG-System mit einer Vektordatenbank im Backend an – so wie es in Abbildung 7.2 dargestellt ist.

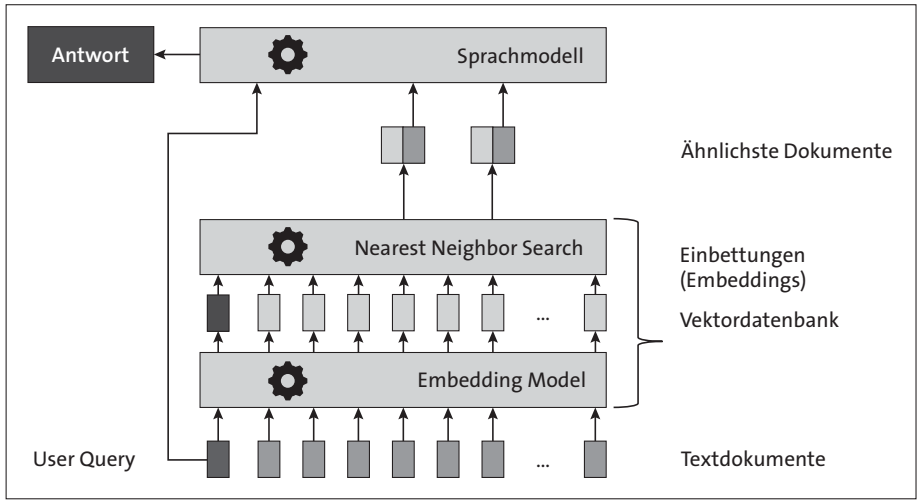


Abbildung 7.2 Einfaches RAG-System auf Basis einer Vektordatenbank im Backend

Die Vektordatenbank, die einen Korpus von Dokumenten und deren Einbettungen repräsentiert, muss zuerst erstellt werden. In Kapitel 6 haben Sie gelernt, wie das geht.

In einem RAG-System, das auf einer Vektordatenbank basiert, sendet der Nutzer eine Anfrage an das System. Diese Anfrage wird basierend auf demselben Embedding-Modell eingebettet, das auch für die Einbettung der Dokumente verwendet wurde. Die ähnlichsten Dokumente zur Anfrage-Einbettung werden aus der Datenbank abgerufen. Im nächsten Schritt wird die Nutzeranfrage zusammen mit den ähnlichsten Dokumenten an das LLM weitergegeben, um die Antwort zu formulieren.

Jetzt, da Sie eine allgemeine Vorstellung haben, wollen wir in die Details eintauchen und sehen uns an, was in den jeweiligen Prozessschritten geschieht.

7.1.1 Retrieval-Prozess

Im *Retrieval-Prozess* oder *Abrufprozess* führt die Benutzeranfrage zu einer Suche in einer bestimmten Datenquelle. Die Benutzeranfrage benötigt mindestens einen Text und eine Einschränkung, wie viele Dokumente zurückgegeben werden sollen (siehe Abbildung 7.3). Es kann in einer Vielzahl von verschiedenen Wissensquellen gesucht werden. Die Wissensquelle repräsentiert eine große Anzahl von Dokumenten. In dieser Wissensquelle werden alle Dokumente basierend auf ihrer Ähnlichkeit zur Benutzeranfrage eingestuft (*Ranking*). Die angeforderte Anzahl der relevantesten Dokumente wird extrahiert und an den nächsten Schritt – die Augmentierung – weitergegeben.

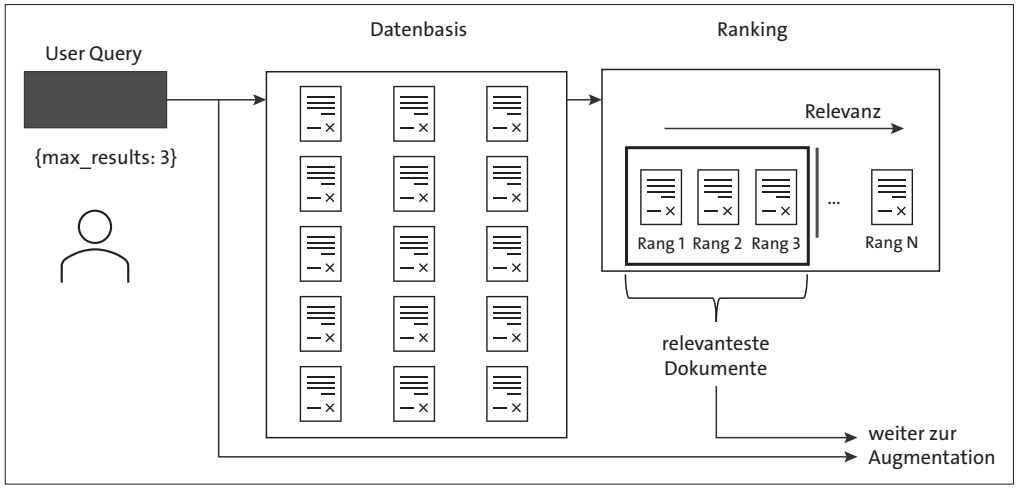


Abbildung 7.3 Retrieval-Prozess

Für die Benutzeranfrage wird ein Embedding erstellt, und die Einbettung wird mit allen anderen Einbettungen der Dokumente in der Wissensquelle verglichen. Basie-

rend auf einer Ähnlichkeitsmetrik wie der Kosinus-Ähnlichkeit werden die Dokumente eingestuft. Eine vordefinierte Anzahl der relevantesten Dokumente wird extrahiert und an die nächste Stufe – die Augmentierung – weitergegeben.

7.1.2 Augmentierung

Der Augmentierungsschritt stellt den Prozess dar, bei dem die abgerufenen Informationen mit der ursprünglichen Anfrage kombiniert werden, um einen verbesserten Prompt zu erstellen. Diese Phase beginnt mit der Integration relevanter Dokumente oder Passagen, die im Abruf-Schritt gefunden wurden.

Typischerweise sind die abgerufenen Informationen klar von der Anfrage des Nutzers zu unterscheiden. An diesem Punkt kommt das *Prompt Engineering* ins Spiel und spielt eine entscheidende Rolle: Sie müssen dem LLM Anweisungen geben, wie es die Informationen verarbeiten soll. Dabei müssen Sie klare Anweisungen geben, um innerhalb der Grenzen der bereitgestellten Informationen zu bleiben, damit Halluzinationen vermieden werden. Außerdem legen Sie deutlich fest, wie das Modell reagieren soll, wenn die gegebenen Informationen nicht helfen, die Nutzeranfrage zu beantworten.

Das Ergebnis ist ein Prompt, der dem Sprachmodell sowohl den *Kontext* bietet, den es benötigt, um eine genaue Antwort zu generieren, als auch die *Anleitung*, die erforderlich ist, um diesen Kontext effektiv zu nutzen. Der auf diese Weise *augmentierte Prompt* dient als Grundlage für die Generierung von Antworten, die relevant für die Nutzeranfrage sind und in den abgerufenen Informationen verankert sind.

Im nächsten Schritt werfen wir einen genaueren Blick auf den Generierungsprozess.

7.1.3 Generierung

Der Generierungsprozess im RAG stellt die letzte entscheidende Phase dar, in der das Sprachmodell den erweiterten Prompt in eine kohärente und kontextuell relevante Antwort umwandelt. Im Gegensatz zur standardmäßigen Ausgabe von LLMs in traditionellen Sprachmodellen integriert die RAG-basierte Generierung die abgerufenen Informationen und bleibt dabei flüssig in der natürlichen Sprache.

Wenn das Sprachmodell den augmentierten Prompt erhält, beginnt es mit der Analyse des abgerufenen Kontexts und der ursprünglichen Anfrage gleichzeitig.

Eine der größten Herausforderungen während der Generierung besteht darin, die Konsistenz mit den abgerufenen Informationen zu wahren und gleichzeitig Halluzinationen zu vermeiden. Das Modell muss ein gutes Gleichgewicht finden zwischen rein extraktiven Antworten, wie dem direkten Zitieren des abgerufenen Inhalts, und abstrakteren Antworten, die die Informationen synthetisieren und umformulieren.

Oft kommen dabei Techniken zum Einsatz, mit denen die Generierung des Modells ausdrücklich so geleitet wird, dass sie innerhalb der Grenzen des bereitgestellten Kontexts bleibt.

Das klingt sehr kompliziert, ist es aber eigentlich nicht. Sie werden es sehen, wenn wir im nächsten Abschnitt mit der Umsetzung beginnen.

7.2 Ein einfaches System zur Retrieval-Augmented Generation

Wir wollen nun unsere allererste RAG-Anwendung entwickeln. Lassen Sie mich skizzieren, was wir erstellen werden. Unser kleines RAG-System wird von einer Vektordatenbank unterstützt. Die Vektordatenbank wird eine Wissensbasis zur Menschheitsgeschichte darstellen. Dieses Wissen wird von Wikipedia abgerufen. Wir laden zu diesem Zweck direkt eine Reihe von Wikipedia-Artikeln zu diesem Thema und speichern sie in der Datenbank. Später werden wir ein RAG-System einrichten, das mit diesen Dokumenten interagiert und Fragen basierend auf diesem repräsentierten Wissen beantwortet. Es wird auch klar angeben, wenn es nicht weiß, wie es die Frage beantworten soll.

7.2.1 Vorbereitung der Wissensbasis

Zuerst packen wir unsere Tasche und legen alle Werkzeuge hinein, die wir brauchen. Listing 7.1 zeigt alle Pakete, die Sie in diesem Skript benötigen. Sie können aus dem Listing schon viele Aspekte unserer Implementierung ableiten, zum Beispiel, dass wir Groq als LLM-Backend verwenden oder dass wir ein OpenAI-Einbettungsmodell für das Einbetten unserer Texte nutzen:

```
#%% packages
import os
from langchain_community.document_loaders import WikipediaLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv(usecwd=True))
from langchain_groq import ChatGroq
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
```

Listing 7.1 Einfaches RAG-System – erforderliche Pakete (06_RAG/10_simple_RAG.py)

Sie müssen auch eine Konfigurationsdatei einrichten, die Ihre API-Anmeldeinformationen für die Interaktion mit dem LLM enthält. Diese Datei heißt *.env* und sieht so aus wie in Listing 7.2:

```
GROQ_API_KEY = …
OPENAI_API_KEY = …
```

Listing 7.2 Einfaches RAG-System – API-Zugangsdaten (».env«)

Der erste Schritt besteht jetzt darin, die Vektordatenbank zu erstellen. Da Sie das schon viel ausführlicher gesehen haben, will ich nicht alle Details wiederholen. Falls Ihnen einige der Schritte hier unklar sind, blättern Sie bitte zurück zu Kapitel 6.

Die allgemeine Einrichtung sieht folgendermaßen aus: Die Datenbank wird in einem Ordner namens *rag_store* gespeichert. Wenn dieser Ordner bereits existiert, gehen wir davon aus, dass Sie den Code schon einmal ausgeführt haben, denn er wird als Teil der vollständigen Erstellung der Datenbank angelegt. Wenn der Ordner noch nicht existiert, wird die Datenbank erstellt. Die Erstellung der Datenbank besteht aus diesen Schritten:

1. Laden der Daten von Wikipedia
2. Aufteilen der Daten in kleinere Einheiten (*Chunks*)
3. Erstellen der Satzeinbettungen mithilfe des OpenAI-Embettungsmodells
4. Speichern der Einbettungen zusammen mit den Chunks in der Vektordatenbank

Listing 7.3 zeigt die Schritte zum Laden des Datensatzes und zum Speichern in einer Vektordatenbank:

```
#%% load dataset
persist_directory = "rag_store"
if os.path.exists(persist_directory):
    vector_store = Chroma(persist_directory=persist_directory, embedding_
function=OpenAIEmbeddings())
else:
    data = WikipediaLoader(
        query="Human History",
        load_max_docs=50,
        doc_content_chars_max=1000000,
    ).load()

    # split the data
    chunks = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=
200).split_documents(data)
```

```
    # create persistent vector store
    vector_store = Chroma.from_documents(chunks, embedding=
OpenAIEmbeddings(), persist_directory="rag_store")
```

Listing 7.3 Einfaches RAG-System – Vektordatenbank erstellen und laden (06_RAG/10_simple_RAG.py)

Als Nächstes arbeiten wir praktisch die drei Schritte von RAG ab, die ich im vorigen Abschnitt schon im theoretischen Überblick vorgestellt hatte: Retrieval, Augmentierung und Generierung.

7.2.2 Retrieval

Um etwas abzurufen, müssen wir einen *Retriever* erstellen. In Listing 7.4 sehen Sie, wie man einen Retriever einrichtet und wie man ihn aufruft. Es ist praktisch, dass der `vector_store` (unsere ChromaDB-Instanz) eine Methode `as_retriever()` hat, die es uns ermöglicht, die Vektordatenbank als unseren Retriever zu definieren. Außerdem können wir einige wichtige Parameter festlegen: zum einen die Ähnlichkeitsfunktion, die wir verwenden wollen, um die relevantesten Dokumente zu finden, zum anderen, wie viele Dokumente zurückgegeben werden sollen. Wir können das ausprobieren, indem wir eine Frage (`question`) formulieren und anschließend den Retriever mit dieser Frage aufrufen.

```
retriever = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3})
question = "what happened in the first world war?"
relevant_docs = retriever.invoke(question)
```

Listing 7.4 Einfaches RAG-System – den Retriever einrichten (06_RAG/10_simple_RAG.py)

Wir überprüfen qualitativ, ob der Abrufprozess funktioniert, indem wir die ähnlichsten Dokumente ausgeben lassen. Zur Vereinfachung beschränken wir die Ausgabe pro Dokument auf nur 100 Zeichen, wie in Listing 7.5 gezeigt:

```
#%% print content of relevant docs
for doc in relevant_docs:
    print(doc.page_content[: 100])
    print("\n--------------")

# Ausgabe:
This transformation was catalyzed by wars of unparalleled scope and
devastation. World War I was a g
--------------
```

```
=== World War I ===
--------------
World War I saw the continent of Europe split into two major opposing
alliances; the Allied Powers,
--------------
A tenuous balance of power among European nations collapsed in 1914 with the
outbreak of the First W --------------
```

Listing 7.5 Einfaches RAG-System – abgerufene Dokumente (06_RAG/10_simple_RAG.py)

Gut, es scheint, dass alle gefundenen Dokumente relevant sind, da sie mit dem Ersten Weltkrieg zu tun haben oder das Jahr 1914 erwähnt wird.

Aber das sind nur Ausschnitte aus größeren Texten. Die Ausgabe würde Ihren Aufwand verringern, die aufgeworfene Frage zu bearbeiten, aber es ist noch keine gute Antwort auf die Frage. Dafür ist RAG da, denn im nächsten Schritt ergänzen wir die Dokumente mit klaren Anweisungen dazu, wie man sie nutzen kann, um die Frage zu beantworten.

7.2.3 Augmentierung

Im Augmentierungsschritt bringen wir die ähnlichsten Dokumente in ein gut definiertes Format: in einen langen zusammenhängenden String, in dem alle Inhalte der Dokumente dargestellt sind. Das erreichen wir mit `join()`. In dieser Funktion übergeben wir eine Liste, die dann einfach zu einem einzigen String zusammengefügt wird. Die Listenelemente werden durch `page_content` unserer `relevant_docs` dargestellt:

```
context = "\n".join([doc.page_content for doc in relevant_docs])
```

Das war die mühsame Arbeit, jetzt müssen wir kreativ werden, um unseren Prompt zu definieren, der dem LLM sagt, wie es eine Antwort formulieren soll. In Listing 7.6 definieren wir die Rolle, die unser LLM einnehmen soll. Wir weisen es an, die Dokumente zur Beantwortung der Frage zu verwenden. Außerdem sagen wir ihm, dass es im Grunde nicht halluzinieren und stattdessen klar angeben soll, dass es die Antwort nicht weiß, falls das der Fall ist.

```
#%% create prompt
messages = [
    ("system", "You are an AI assistant that can answer questions about the
history of human civilization. You are given a question and a list of
documents and need to answer the question. Answer the question only based on
these documents. These documents can help you answer the question: {context}.
If you are not sure about the answer, you can say 'I don't know' or 'I don't
know the answer to that question.'"),
```

```
    ("human", "{question}"),
]
prompt = ChatPromptTemplate.from_messages(messages=messages)
```

Listing 7.6 Einfaches RAG-System – die Augmentierung einrichten (06_RAG/10_simple_RAG.py)

Für den Anfang sollte das zum Thema Augmentieren genug sein. Wir können später zu diesem Punkt zurückkehren, um das System zu verbessern. Im Moment sind wir damit aber zufrieden und machen mit dem nächsten Schritt weiter – der Generierung einer finalen Antwort.

7.2.4 Generierung

Kommen wir also zum letzten Schritt – der Generierung. Wir entscheiden uns zunächst für ein Sprachmodell. In diesem Fall verwenden wir ein Open-Weights-Modell von Google, das *gemma2* heißt. Es ist ein vergleichsweise kleines, aber sehr schnelles Modell. Es wird als kleines Sprachmodell (SLM) klassifiziert. Das ermöglicht eine sehr schnelle Inferenz, was die Benutzerakzeptanz erhöhen könnte, da die Antwort nahezu sofort bereitgestellt wird. Der längste Prozess ist das Abrufen der Dokumente.

Der zweite Schritt wird Ihnen sehr vertraut vorkommen, denn Sie haben ihn bereits in Kapitel 4 gesehen: die Erstellung einer Kette mit LangChain. In diesem Fall beginnt die Kette mit unserem Prompt. Die Ausgabe des Prompts wird an das Modell weitergereicht. Und schließlich wird die Ausgabe des Modells an `StrOutputParser()` übergeben, um nur den Inhaltsbereich der Modellausgabe anzuzeigen:

```
model = ChatGroq(model_name="gemma2-9b-it", temperature=0)
chain = prompt | model | StrOutputParser()
```

Das war das letzte Puzzlestück, und wir können es testen, indem wir die Kette aufrufen und die Antwort überprüfen, wie in Listing 7.7 gezeigt:

```
#%% invoke chain
answer = chain.invoke({"question": question, "context": context})
print(answer)

# Ausgabe:
World War I was a global conflict from 1914 to 1918.

It involved two main alliances:
```

```
* **The Allied Powers:**  Primarily composed of the United Kingdom, France,
Russia, Italy, Japan, Portugal, and various Balkan states.
* **The Central Powers:** Primarily composed of Germany, Austria-Hungary, the
Ottoman Empire, and Bulgaria.

The war resulted in the collapse of four empires: Austro-Hungarian, German,
Ottoman, and Russian. It had a death toll estimated between 10 and 22.5
million people.

The war saw the use of new industrial technologies, making traditional
military tactics obsolete.  It also witnessed horrific events like the
Armenian, Assyrian, and Greek genocides within the Ottoman Empire.
```

Listing 7.7 Einfaches RAG-System – die Generierung der Antwort (06_RAG/10_simple_RAG.py)

Das ist eine relativ gute Antwort. Angenommen, das war unser Test, um zu überprüfen, ob das Modell in der Lage ist, eine gute Antwort basierend auf dem verfügbaren Wissen aus der Quelle zu formulieren. Es ist auch immer wichtig, die gegenteilige Funktionalität zu testen – ob das Modell also seine eigenen Einschränkungen korrekt erkennt und klar angibt, dass es nicht genug weiß, um die Frage zu beantworten.

In einem realen Beispiel möchten wir diese Funktionalität auch in einen breiteren Kontext von Code einbetten. Nun bündeln wir die gesamte Funktionalität in einer Funktion.

7.2.5 Erstellung der RAG-Funktion

Unsere Funktion `simple_rag_system` wird eine Frage `question` entgegennehmen und einen String zurückgeben. Die anderen Schritte sind genau diejenigen Schritte, die wir vorher festgelegt haben:

1. Abruf relevanter Dokumente
2. Erstellen des Kontextes basierend auf den relevanten Dokumenten
3. Erzeugung eines detaillierten Prompts, der dem Modell erklärt, wie es den Kontext sowie die Benutzerfrage anwenden soll
4. Aufbau einer `chain`, die den Prompt, das Modell und die Ausgabenformatierung miteinander verbindet

Listing 7.8 zeigt die Definition unserer `simple_rag_system`-Funktion, die eine `question` als Eingabeparameter verwendet und anschließend die Antwort des RAG-Systems (`answer`) zurückgibt:

```
# %% bundle everything in a function
def simple_rag_system(question: str) -> str:
    relevant_docs = retriever.invoke(question)
    context = "\n".join([doc.page_content for doc in relevant_docs])
    messages = [
        ("system", "You are an AI assistant that can answer questions about
the history of human civilization. You are given a question and a list of
documents and need to answer the question. Answer the question only based on
these documents. These documents can help you answer the question: {context}.
If you are not sure about the answer, you can say 'I don't know' or 'I don't
know the answer to that question.'"),
        ("human", "{question}"),
    ]
    prompt = ChatPromptTemplate.from_messages(messages=messages)
    model = ChatGroq(model_name="gemma2-9b-it", temperature=0)
    chain = prompt | model | StrOutputParser()
    answer = chain.invoke({"question": question, "context": context})
    return answer
```

Listing 7.8 Einfaches RAG-System – Funktion (06_RAG/10_simple_RAG.py)

Jetzt ist es an der Zeit, die Funktion zu testen. Gibt die Funktion korrekt zurück, dass sie keine Antwort kennt? Wie Sie in Listing 7.9 sehen können, tut sie das!

```
# %% Testing the function
question = "What is a black hole?"
simple_rag_system(question=question)

# Ausgabe:
"I don't know the answer to that question. \n"
```

Listing 7.9 Einfaches RAG-System – Test (06_RAG/10_simple_RAG.py)

Zu dieser Frage kann sich das Modell nicht äußern – und das ist genau, was wir erwartet haben. Wir haben das erreicht, indem wir die folgende Anweisung zur Systemnachricht ergänzt haben:

`Answer the question only based on these documents. … If you are not sure about the answer, you can say 'I don't know' or 'I don't know the answer to that question.`

Lassen Sie uns rekapitulieren, was Sie bisher gelernt haben: Wir haben unser erstes RAG-System entwickelt. Es basierte auf einer Vektordatenbank. Bei einer Benutzerfrage wurden die relevantesten Dokumente abgerufen. Und diese abgerufenen Dokumente wurden in Kombination mit einer detaillierten Anleitung für das LLM verwendet, wie es sie nutzen soll.

Sie werden beeindruckt sein, wie gut das funktioniert. Aber nachdem Sie eine Weile damit gearbeitet haben, könnten Sie einige Probleme mit diesem einfachen RAG feststellen. Die Probleme treten entlang der gesamten Pipeline auf, und könnten sich wie folgt äußern:

- **Retrieval:** Die abgerufenen Dokumente könnten irrelevant sein.
- **Augmentierung:** Wenn mehrere Dokumente die gleichen oder sehr ähnliche Informationen enthalten, könnten die abgerufenen Dokumente sich wiederholen und somit nicht sehr informativ sein.

  Falls die Dokumente aus verschiedenen Quellen stammen, ist es schwer, ihren Kontext zu unterscheiden. Das kann dazu führen, dass in einer finalen Antwort verschiedene Dokumente falsch gemischt werden. Stellen Sie sich vor, Sie haben ein System mit umstrittenen Arbeiten und widersprüchlichen Ansichten: Es wird eine wirklich schlechte Antwort liefern, wenn das System die Ansichten den falschen Autoren zuordnet.
- **Generation:** Probleme in diesem Schritt resultieren typischerweise aus den bereits erwähnten Problemen weiter oben.

Es gibt zahlreiche Verbesserungsmöglichkeiten. In den folgenden Abschnitten werden wir einige davon besprechen.

7.3 Fortgeschrittene Techniken

Obwohl das System beeindruckend funktioniert, gibt es noch viel Raum für Verbesserungen. Möglichkeiten zur Verbesserung finden sich in verschiedenen Phasen der Pipeline. Einige Verbesserungen können wir bereits in der Indexierungs-Pipeline vornehmen.

Praktisch können wir die Daten verbessern, bevor wir sie zu unserer Vektordatenbank hinzufügen. Das erläutere ich im Abschnitt über fortgeschrittene Vorab-Abfragemethoden in Abschnitt 7.3.1. Andere Techniken gehen das Problem in der Abrufphase der Pipeline an. Das besprechen wir in Abschnitt 7.3.2. Es wird Sie wahrscheinlich nicht überraschen, dass auch der letzte Schritt der Pipeline verbessert werden kann: Wir werden Verbesserungen im Generierungsschritt in Abschnitt 7.3.3 erkunden.

7.3.1 Fortgeschrittene Prä-Retrieval-Techniken

Wenn das RAG-System die richtigen Dokumente nicht abrufen kann, könnte das Problem in der Datenaufnahme-Pipeline liegen. Es gibt verschiedene Möglichkeiten, die

Daten zu verbessern, bevor sie in der Vektordatenbank gespeichert werden. Abbildung 7.4 zeigt einige gängige Ansätze zur Optimierung der Datenaufnahme-Pipeline.

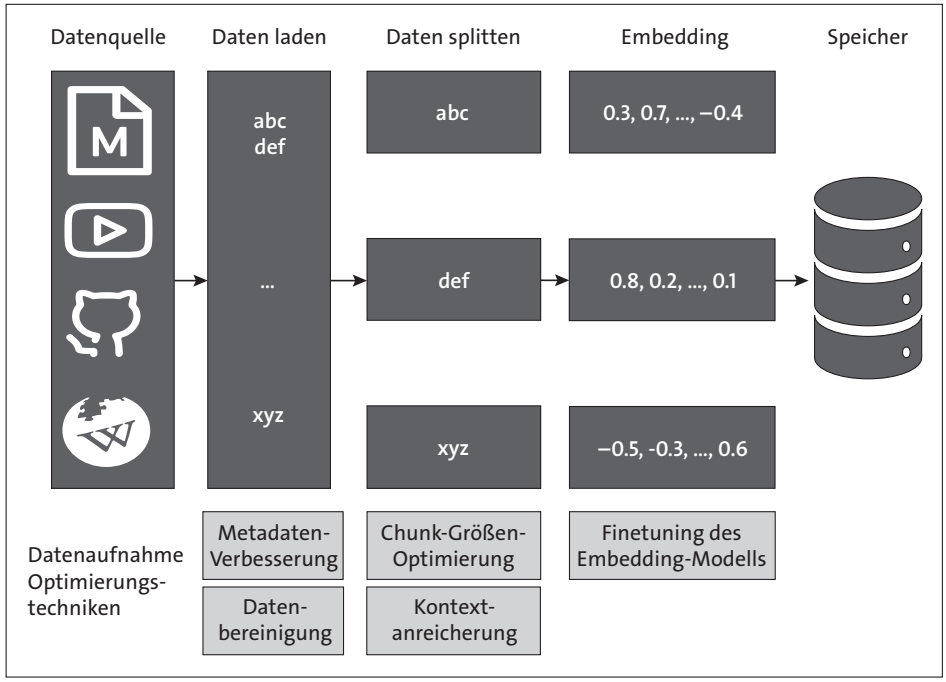


Abbildung 7.4 Prä-Retrieval-Techniken

Die verschiedenen Techniken werden entsprechend dem Schritt angezeigt, in dem sie angewendet werden, und wir werden diese Schritte in den folgenden Abschnitten besprechen.

Datenbereinigung

Die Datenbereinigung in RAG-Systemen umfasst das Vorbereiten und Verfeinern der Quelldokumente, um sie für das Abrufen und die Generierung zu optimieren. Dieser entscheidende Vorverarbeitungsschritt beginnt mit grundlegenden Operationen wie dem Entfernen von doppeltem Inhalt, der Standardisierung von Formaten und dem Umgang mit Sonderzeichen.

Der Text wird dann typischerweise in geeignete Abschnitte unterteilt, die ein Gleichgewicht zwischen der Erhaltung des Kontexts und der Abrufgranularität wahren – Abschnitte, die zu groß sind, können irrelevante Informationen enthalten, während Abschnitte, die zu klein sind, wichtigen Kontext verlieren könnten. Das haben Sie alles ausführlich in Kapitel 6 gesehen. Zu den fortgeschrittenen Bereinigungsschritten könnten folgende gehören:

- Entfernen von Standardcode
- Standardisierung von Datumsformaten
- Abkürzungen auflösen
- Behandlung von mehrsprachigen Inhalten

Der Bereinigungsprozess beinhaltet oft auch die Anreicherung von Metadaten, bei der die Daten mit relevanten Identifikatoren, Zeitstempeln oder Kategorisierungen versehen werden, um die Abrufgenauigkeit zu verbessern. Gut gereinigte Daten bilden die Grundlage für eine effektive RAG-Leistung, da sie direkt die Qualität der abgerufenen Übereinstimmungen und die Kohärenz der generierten Antworten beeinflussen.

Metadaten-Verbesserung

Die Verbesserung von Metadaten in RAG-Systemen bedeutet, dass Dokumentenabschnitte mit zusätzlichen Kontextinformationen angereichert werden, die über den reinen Textinhalt hinausgehen. Dieser Prozess umfasst das Taggen von Dokumenten mit Attributen wie Erstellungsdaten, Autoren, Themen und die Klassifikation ihrer Kategorien.

Eine ausgefeiltere Verbesserung könnte das Generieren von Einbettungen für die semantische Suche, das Berechnen von Lesbarkeitswerten oder das Extrahieren von Schlüsselentitäten und Beziehungen beinhalten. Einige Systeme setzen eine automatisierte Themenmodellierung oder Klassifikation ein, um thematische Tags hinzuzufügen, während andere hierarchische Beziehungen zwischen den Abschnitten beibehalten, um die Dokumentenstruktur zu bewahren.

Diese reichhaltige Schicht von Metadaten verbessert nicht nur die Abrufgenauigkeit, sondern ermöglicht auch eine differenziertere Filterung und ein besseres Verständnis des Kontexts während der Generierungsphase. Das führt letztendlich zu relevanteren und besser fundierten Antworten.

Chunk-Größen-Optimierung

Die Optimierung der Chunk-Größe in RAG-Systemen erfordert ein sorgfältiges Abwägen mehrerer konkurrierender Faktoren, um die Abruf-Effektivität zu maximieren. Kleinere Chunks bieten zwar präzisere Abrufe und reduzieren den Token-Verbrauch, aber sie laufen Gefahr, wichtige Kontexte zu fragmentieren und zusammenhängende Ideen zu zerreißen.

Auf der anderen Seite bewahren größere Chunks mehr Kontext, können aber Rauschen und irrelevante Informationen in die Abruf-Ergebnisse einführen, was die Qualität der generierten Antworten potenziell verwässert. Wie Sie bereits in Kapitel 6

gesehen haben, variiert die optimale Chunk-Größe oft je nach Art des Inhalts. Technische Dokumentationen könnten von kleineren, fokussierten Chunks profitieren, während narrative Inhalte größere Chunks benötigen, um die Kohärenz zu wahren.

Kontextanreicherung

Die Kontextanreicherung (engl. *context enrichment*) in RAG-Systemen bedeutet, dass der Basisinhalt mit zusätzlichen Informationen versehen wird, um seine Nützlichkeit und Abrufbarkeit zu verbessern. Dieser Prozess geht über einfaches Metadaten-Tagging hinaus, indem verwandte Informationen, Querverweise und abgeleitete Erkenntnisse integriert werden, die den Inhalt wertvoller für das Abrufen und Generieren machen.

Fortgeschrittene Anreicherung könnte beinhalten, verwandte Konzepte zu verknüpfen, Definitionen für Fachbegriffe hinzuzufügen oder domänenspezifisches Wissen einzubringen. Zum Beispiel könnte das System beim Verarbeiten medizinischer Dokumente automatisch Abkürzungen erweitern, standardisierte medizinische Codes hinzufügen oder auf relevante Forschungsarbeiten verlinken.

Echte Anwendungen könnten das Hinzufügen geografischer Koordinaten zu standortbasierten Inhalten, zeitliche Beziehungen für ereignisbasierte Informationen oder branchenspezifische Taxonomien umfassen.

Der angereicherte Kontext hilft dem Abrufsystem, intelligentere Übereinstimmungen zu finden, und versorgt das Generierungsmodell mit reichhaltigeren, nuancierteren Informationen, mit denen es arbeiten kann, was letztendlich zu umfassenderen und genaueren Antworten führt.

Finetuning des Embedding-Modells

Das Finetuning von Embedding-Modellen in RAG-Systemen passt allgemeine Embedding-Modelle an, um domänenspezifische semantische Beziehungen und Nuancen besser zu erfassen. Der Prozess beginnt normalerweise mit der Auswahl eines Basis-Embedding-Modells wie BERT oder eines Sentence-Transformers, das dann weiter auf domänenspezifischen Daten trainiert wird, um das Verständnis für spezialisiertes Vokabular, technische Konzepte und branchenspezifische Beziehungen zu verbessern.

Der Finetuning-Prozess erfordert eine sorgfältige Auswahl von Trainingspaaren, die die Arten von semantischen Ähnlichkeiten repräsentieren, die für den spezifischen Anwendungsfall am wichtigsten sind. Zum Beispiel könnte es in rechtlichen Anwendungen darum gehen, verschiedene Formulierungen desselben rechtlichen Konzepts abzugleichen, während es in technischen Dokumentationen darum gehen könnte, Problembeschreibungen mit ihren Lösungen zu verknüpfen.