DHBW Heidenheim - Wirtschaftsinformatik 2022 UC-KI

Projekt KI - Clustering

Fritz Martin

Maximilian Krämer

Sara Kleszczewski

API-KEY: AIzaSyC-PAnWQs339Lsx9-Z6E_AeiRMdCVXg7Ok

Github Repo: https://github.com/fritzmartin003/RAG-System-Projekt

Colab Laufzeittyp: Python 3 Hardwarebeschleuniger: v2-8 TPU

Inhalt

Erklärung Code

Installierte Bibliotheken

- faiss-cpu: Eine Bibliothek für effiziente Ähnlichkeitssuche.
- transformers: Enthält vortrainierte Modelle und Tokenizer von Hugging Face.
- sentence-transformers: Wird für die Einbettung von Sätzen verwendet.
- pymupdf: Zum Verarbeiten von PDF-Dateien.
- numpy und scipy: Mathematische Bibliotheken für numerische Berechnungen.
- Zusätzlich wird tensorflow deinstalliert, um Konflikte zu vermeiden.

Imports

- fitz (PyMuPDF): Verarbeitung von PDF-Dateien.
- numpy: Matrix- und Vektoroperationen.
- faiss: Schnelle Vektorsuche.
- SentenceTransformer: Für semantische Embeddings.
- sklearn, torch, time: Weitere Hilfsbibliotheken für das Training und die Evaluation.

Extraktion von Text aus einer PDF

```
# PDF-Text extrahieren
def extract_text_from_pdf(pdf_path):
    text = ""
    with fitz.open(pdf_path) as doc:
        for page in doc:
            text += page.get_text("text") + "\n"
    return text

pdf_path = "SakowskiBuch.pdf"
pdf_text = extract_text_from_pdf(pdf_path)

```

- Die fitz-Bibliothek (PyMuPDF) wird verwendet, um eine PDF-Datei zu öffnen.
- Die Funktion extract_text_from_pdf(pdf_path) geht jede Seite der PDF durch und extrahiert den reinen Text mit page.get_text("text").
- Der Text aller Seiten wird zusammengefügt und als text zurückgegeben.
- Hier wird der Pfad zur PDF angegeben und die extract_text_from_pdf-Funktion aufgerufen.

Text in Chunks aufteilen

```
# Text in Chunks teilen
def split_text(text, chunk_size=500, overlap=90):
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start += chunk_size - overlap
    return chunks

chunks = split_text(pdf_text)
print(f" PDF in {len(chunks)} Chunks unterteilt!")
```

- Der extrahierte Text wird in Abschnitte (chunks) mit einer Standardlänge von 500 Zeichen unterteilt.

- Um den Kontext zwischen den Chunks beizubehalten, gibt es eine Überlappung von 90 Zeichen.
- start beginnt bei 0 und wird nach jeder Iteration um chunk_size - overlap verschoben.
- Der Text wird mit split_text(pdf_text) in Chunks zerlegt.
- Die Anzahl der Chunks wird ausgegeben.

Laden des Embedding-Modells

```
# Embedding Modell laden
embedding_model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
```

- Laden des SentenceTransformer-Modell "sentence-transformers/all-MiniLM-L6-v2". Dieses Modell wandelt Texte in Embeddings.

Bestimmen der optimalen Batch-Größe

```
# Optimale Batch-Größe finden
optimal_batch_size = 1600
```

- Batch-Größe von 1600 festlegen. Die Batchsize legt fest wie viele Chunks gleichzeitig verarbeitet werden.

Funktion zur Erstellung eines FAISS-Index

```
def create_faiss_index(chunks):
    start_time = time.time()  # Zeitmessung
```

- create_faiss_index(chunks) beginnt mit einer Zeitmessung, um die Dauer des Indexierungsprozesses zu berechnen.

Embeddings erstellen und normalisieren

```
    # Embeddings erstellen (mit optimierter Batch-Verarbeitung und Tensor-Konvertierung)
    chunk_embeddings_tensor = embedding_model.encode(chunks, batch_size=optimal_batch_size, convert_to_tensor=True)
    chunk_embeddings = chunk_embeddings_tensor.cpu().numpy()
```

- embedding_model.encode(chunks, batch_size=optimal_batch_size, convert_to_tensor=True) sorgt dafür, dass die Chunks als Tensor verarbeitet werden.
- Danach werden die Tensoren mit .cpu().numpy() in ein NumPy-Array umgewandelt, um mit FAISS weiterarbeiten zu können.

```
    # Normalisieren der Embeddings (wichtig für die Distanzmetriken)
    chunk_embeddings = sklearn.preprocessing.normalize(chunk_embeddings)
```

- Embeddings normalisieren. Hier werden alle unnötigen Satzzeichen, welche keinen Einfluss auf den Inhalt haben, rausgelöscht

Initialisierung des FAISS-Index

```
 dimension = chunk_embeddings.shape[1]
```

- Bestimmen der Anzahl der Dimensionen der Embeddings

```
 # FAISS Index erstellen (IndexIVFFlat mit Training)
    nlist = int(np.sqrt(len(chunk_embeddings)))  # Anzahl der Partitionen
```

```
    quantizer = faiss.IndexFlatL2(dimension)  # Quantisierer für die
Clusterzentren
    index = faiss.IndexIVFFlat(quantizer, dimension, nlist,
faiss.METRIC_L2)
```

FAISS verwendet verschiedene Index-Typen zur effizienten Ähnlichkeitssuche.

- IndexFlatL2(dimension): Ein einfacher Index, der L2-Distanzen berechnet.
- IndexIVFFlat(quantizer, dimension, nlist, faiss.METRIC_L2): Ein Index, der das gesamte Suchproblem in nlist Cluster unterteilt, um die Suche zu beschleunigen. Die Anzahl der Cluster wird als Wurzel der Anzahl der Embeddings berechnet.

Training und Hinzufügen der Embeddings

```
    # Training des Index (NOTWENDIG für IndexIVFFlat!)
    index.train(chunk_embeddings)
```

- Training des Index

```
    # Hinzufügen der Embeddings zum Index (NACH dem Training)
    index.add(chunk_embeddings)
```

- Nach dem Training werden die erstellten Embeddings zum FAISS-Index hinzugefügt.

Abschluss und Rückgabe des Index

```
    end_time = time.time()
    print(f"FAISS Vektordatenbank erstellt! (Zeit: {end_time -
start_time:.2f} Sekunden)")
    return index

index = create_faiss_index(chunks)
```

- Die Zeitmessung wird abgeschlossen, die benötigte Zeit und der fertige FAISS-Index werden zurückgegeben.
- Die Funktion create_faiss_index(chunks) wird aufgerufen, um den Index mit den zuvor erstellten Chunks zu erstellen.

Die search Funktion

```
def search(index, query, k=5):
    query_embedding_tensor = embedding_model.encode([query],
convert_to_tensor=True)
    query_embedding = query_embedding_tensor.cpu().numpy()

    query_embedding = sklearn.preprocessing.normalize(query_embedding)
    D, I = index.search(query_embedding, k)  # D: Distanzen, I: Indizes
    return D, I
```

Schritt-für-Schritt-Erklärung:

- index: Der FAISS-Index, der bereits mit Embeddings der Chunks erstellt wurde.
- query: Die Abfrage, nach der im Index gesucht werden soll.
- k=5: Die Anzahl der ähnlichsten Chunks, die zurückgegeben werden sollen.
- Die Abfrage wird durch das embedding_model in ein Embedding umgewandelt. Dabei wird convert_to_tensor=True verwendet, um das Ergebnis als Tensor zu erhalten.
- Die Normalisierung sorgt dafür, dass der Vektor für die Berechnung der Ähnlichkeit korrekt skaliert wird.
- index.search(query_embedding, k) führt die Ähnlichkeitssuche durch und gibt zwei Dinge zurück:
    - D: Die Distanzen zu den k nächstgelegenen Chunks.
    - I: Die Indizes der k nächstgelegenen Chunks im Index.

- Die Funktion gibt die Distanzen (D) und Indizes (I) zurück.

Die get_search_results Funktion

```
def get_search_results(index, query, chunks, k=5):
    D, I = search(index, query, k)
    relevant_chunks = []
    for i, distance in zip(I[0], D[0]):
        relevant_chunks.append(chunks[i])
    return relevant_chunks
```

- index: Der FAISS-Index, der die Embeddings der Chunks enthält.
- query: Die Abfrage, nach der gesucht wird.
- chunks: Eine Liste der Text-Chunks, die zuvor im Index gespeichert wurden.
- k=5: Die Anzahl der zurückgegebenen relevanten Chunks.
- Die Liste results wird mit Dictionaries gefüllt, die sowohl die chunk als auch die distance enthalten.
- Die Ergebnisse werden zur Veranschaulichung ausgegeben.
- Die for Schleife erstellt eine Liste von Chunks, die den am nächsten liegenden Chunks in Bezug auf die Abfrage entsprechen. Diese Liste wird dann zurückgegeben. In der Variable relevant_chunks werden die Chunks gesammelt, die als die relevantesten für die Abfrage angesehen werden.
- Die Funktion gibt eine Liste von relevanten Chunks (relevant_chunks) zurück, die den am besten passenden Textabschnitten zur Abfrage entsprechen.

Antwortgenerierung mit Gemini

```
import google.generativeai as genai

# Konfiguration des API-Schlüssels
genai.configure(api_key= GeminiAPIKey)

# Modell auswählen
model = genai.GenerativeModel('gemini-pro') # Andere Modele falls
überlastet: gemini-1.5-flash oder gemini-2.0-flash-exp
```

- import google.generativeai as genai: Importiert die Google Gemini Bibliothek und benennt sie als genai für einfachere Verwendung.

- genai.configure(api_key=GeminiAPIKey): Konfiguriert die Gemini-Bibliothek mit dem API-Schlüssel, der für den Zugriff auf das Modell benötigt wird.

- model = genai.GenerativeModel('gemini-pro'): Wählt das spezifische Gemini-Modell aus, das für die Textgenerierung verwendet werden soll. Hier ist es 'gemini-pro'. Falls das Modell überlastet sein sollte, muss man ein anderes Gemini-Modell nutzen. Mögliche Modelle sind gemini-1.5-flash oder gemini-2.0-flash-exp

Die Funktion generate_answer(query)

```
def generate_answer(query):
    relevant_chunks = get_search_results(index, query, chunks, k=10)
    context = "\n".join(relevant_chunks)
    prompt = f"Beantworte die Frage basierend auf diesem
Kontext:\n\n{context}\n\nFrage: {query}\nAntwort:"

    # API-Aufruf an Gemini
    # API-Aufruf mit Optionen
    response = model.generate_content(
        prompt,
        generation_config=genai.types.GenerationConfig(
            max_output_tokens=150,
            temperature=0.7,
        )
    )
    return response.text  # Gibt den generierten Text zurück
```

- query: Die vom Benutzer gestellte Frage als String. Diese Frage wird verwendet, um die relevantesten Informationen aus den Text-Chunks zu finden und eine Antwort zu generieren.
- relevant_chunks = get_search_results(index, query, chunks, k=20): Ruft die get_search_results Funktion auf, um die relevantesten Text-Chunks im Zusammenhang mit der query zu finden.
    - index: Der FAISS-Index, der die Embeddings der Text-Chunks enthält.
    - chunks: Die Liste aller Text-Chunks.
    - k=10: Gibt an, dass die 10 relevantesten Chunks zurückgegeben werden sollen.
    - Die Funktion get_search_results gibt eine Liste von Chunks zurück, die basierend auf ihrer semantischen Ähnlichkeit zur query ausgewählt wurden.
- context = "\n".join(relevant_chunks): Verbindet die relevanten Text-Chunks mit einem Zeilenumbruch (\n) zu einem einzigen String. Dieser String bildet den Kontext für die Antwortgenerierung.
- prompt = f"Beantworte die Frage basierend auf diesem Kontext:\n\n{context}\n\nFrage: {query}\nAntwort:": Erstellt den Prompt für das Gemini-Modell. Der Prompt enthält:
    - Eine explizite Anweisung an das Modell: "Beantworte die Frage basierend auf diesem Kontext:"
    - Den context, der die relevanten Informationen aus den Text-Chunks enthält.
    - Die query (die ursprüngliche Frage des Benutzers).
    - "Antwort:", um dem Modell zu signalisieren, wo die Antwort beginnen soll.
- response = model.generate_content(...): Sendet den prompt an das Gemini-Modell und generiert eine Antwort.
    - generation_config=genai.types.GenerationConfig(...): Konfiguriert die Parameter für die Antwortgenerierung.
        - max_output_tokens=150: Begrenzt die Länge der Antwort auf maximal 150 Token.
        - temperature=0.7: Steuert die "Kreativität" des Modells. Ein höherer Wert (z. B. 1.0) führt zu zufälligeren und kreativeren Antworten,

während ein niedrigerer Wert (z. B. 0.2) zu deterministischeren und konservativeren Antworten führt.

- return response.text: Gibt den generierten Text (die Antwort des Modells) als Ergebnis der Funktion zurück.

Antwort für die Frage generieren

```
frage = "Wie hoch ist der gesetzliche Mindestlohn?"
antwort = generate_answer(frage)
print("Antwort:", antwort)
```

- **Frage:**
  - Dies ist die Eingabe für die Frage, die an das Modell übergeben wird

- **generate_answer(frage):**
  - Die Funktion generate_answer wird hier mit der Eingabe ‘frage’ aufgerufen. Der Wert der Variablen ‘frage’ wird an die Funktion übergeben, die dann:
    1. Die relevanten Text-Chunks durch die get_search_results-Funktion abruft.
    2. Den Kontext für die Frage aus den relevanten Chunks erstellt.
    3. Einen Prompt für das Textgenerierungsmodell erstellt, der den Kontext und die Frage enthält.
    4. Das Modell über die Pipeline aufruft, um eine Antwort zu generieren.
  - Das Ergebnis dieser Funktion ist eine Antwort, die im Variablenwert antwort gespeichert wird.
- **print("Antwort:", antwort):**
  - Die generierte Antwort wird mit der Python-Funktion print() ausgegeben. Der Text "Antwort:" wird zusammen mit dem Wert der Variablen antwort ausgegeben.
  - antwort enthält den generierten Text, der die Antwort auf die Frage darstellt.

Herausforderungen bei der Chatboterstellung

Ermittlung der optimalen Chunkgröße:
Um sicherzustellen, dass jeder Chunk ausreichend Inhalt und Kontext enthält, wurde eine geeignete Chunkgröße bestimmt. Bei zu kleinen Chunks konnte das Modell keine aussagekräftigen Antworten generieren, da der inhaltliche Zusammenhang verloren ging. Dies ist besonders bei rechtlichen Texten entscheidend, da der Kontext für das Verständnis essenziell ist. Zudem führten viele kleine Chunks zu erhöhten Ladezeiten. Die Bestimmung der optimalen Chunkgröße stellte eine Herausforderung dar, da ein Gleichgewicht zwischen Detailtiefe und Effizienz gefunden werden musste, ohne dabei unnötig lange Ladezeiten zu verursachen. Die finale Entscheidung für eine Chunkgröße von 500 stellte daher einen Kompromiss zwischen Kontextbewahrung und Performance dar.

Geschwindigkeit der Antwortgenerierung:

Eine weitere Herausforderung bestand darin, die Antwortzeiten des Modells zu optimieren. Anfangs wurden die Chunks nacheinander in das Embedding-Modell geladen, was den Prozess

erheblich verlangsamte. Um dies zu verbessern, wurde das Verfahren angepasst, sodass eine effizientere und schnellere Verarbeitung möglich wurde. Die Anpassung wurde mit der Batchgröße vorgenommen, welche nach mehreren Tests letztendlich auf 1600 gesetzt wurde. Sprich, es werden 1600 Chunks auf einmal geladen. Wie bereits erwähnt, spielte die Wahl der optimalen Chunkgröße eine entscheidende Rolle. Mit etwas weniger als 1.600 Chunks kann das Modell nun mithilfe der FAISS-Datenbank eine schnelle und präzise Ähnlichkeitssuche durchführen. Dennoch bleibt die Balance zwischen Geschwindigkeit und Antwortqualität eine Herausforderung. Ein zu großes Chunk-Set kann die Suche verlangsamen, während zu kleine Chunks den Kontext verlieren lassen. Zudem müssen Embedding-Modelle und die Vektordatenbank effizient zusammenarbeiten, um relevante Informationen schnell und präzise bereitzustellen.

Sicherstellung korrekter Antworten
Eine zentrale Herausforderung bei der Entwicklung des RAG-Chatbots bestand darin, sicherzustellen, dass die generierten Antworten korrekt und kontextbezogen sind. Da das Modell seine Antworten auf Grundlage der gefundenen Chunks generiert, kam es anfangs zu Problemen, da irrelevante oder unvollständige Informationen ausgewählt werden.
Um die Genauigkeit der generierten Antworten zu verbessern, wurden zunächst mehrere Optimierungen vorgenommen. Neben der Optimierung der Chunkgrößen wurde die FAISS-Datenbank für eine präzisere Ähnlichkeitssuche genutzt, sodass nur relevante Chunks ausgewählt wurden. Zudem wurde das Embedding-Modell "all-MiniLM-L6-v2" eingesetzt, um semantische Ähnlichkeiten besser zu erfassen. Außerdem wurden zu Beginn des Projekt nur 3 ähnliche Chunks zur Antwortengenerierung mitgegeben. Dementsprechend waren die Antworten sehr ungenau oder sogar komplett falsch. Hier wurde schließlich viel getestet welche Chunkanzahl am optimalsten für eine vollständige Antwort ist. Hier kam es zu dem Entschluss, dass 20 Chunks zu einer qualitativen Antwort des Modells führen.
Durch die Optimierung der Abfrageverarbeitung und eine gezielte Prompt-Gestaltung wird verhindert, dass das Modell irrelevante oder halluzinierte Informationen generiert.
Diese Maßnahmen führten zu einer deutlichen Verbesserung der Antwortqualität.

Antwortvergleiche

1. Frage: Wie hoch ist der gesetzliche Mindestlohn?

Antwort RAG:
Der gesetzliche Mindestlohn betrug zum 1. Januar 2022 9,82 EUR pro Stunde und erhöhte sich zum 1. Juli 2022 auf 10,45 EUR pro Stunde.

Antwort Gemini mit Kontext:
Der gesetzliche Mindestlohn beträgt seit 1.1.2022 9,82 Euro pro Stunde und erhöht sich ab 1.7.2022 auf 10,45 Euro.

Gemini ohne gegebenen Kontext:
Seit dem 1. Januar 2024 beträgt der gesetzliche Mindestlohn in Deutschland 12,41 Euro brutto pro Stunde. Am 1 Januar 2025 wird dieser auf 12,82 Euro angehoben.

2. Frage: Was ist die Obergrenze der Arbeitszeiten von Minderjährige?

Antwort RAG:
Konkret gilt eine Obergrenze von werktäglich (Mo.-Fr.) acht Stunden bzw. wöchentlich 40 h (§ 8 Abs. 1 JArbSchG).

Antwort Gemini mit Kontext:
Die tägliche Arbeitszeit von Minderjährigen darf acht Stunden werktäglich (Montag bis Freitag) bzw. 40 Stunden wöchentlich nicht überschreiten.

Gemini ohne gegebenen Kontext:
Die Arbeitszeiten von Minderjährigen sind im Jugendarbeitsschutzgesetz (JArbSchG) geregelt. Die tägliche Arbeitszeit darf 8 Stunden und die wöchentliche Arbeitszeit 40 Stunden nicht überschreiten. Minderjährige dürfen grundsätzlich nur zwischen 6 und 20 Uhr arbeiten. Für Jugendliche unter 18 Jahren gelten Sonderregelungen. Z.B. im Gastgewerbe wo bis 22 Uhr gearbeitet werden darf. Desweiteren muss es Pausen von bestimmten längen geben, sowie Mindestruhezeiten.

3. Frage: Gibt es einen Rechtsanspruch auf einen Arbeitsplatz?

Antwort RAG:
Basierend auf dem Kontext gibt es keinen generellen Rechtsanspruch auf einen Arbeitsplatz. Der Text erwähnt jedoch, dass ein Auszubildender unter bestimmten Umständen einen Anspruch darauf haben kann, nach erfolgreichem Abschluss seiner Ausbildung in ein unbefristetes Arbeitsverhältnis übernommen zu werden, nämlich wenn ein "ausbildungsadäquater Arbeitsplatz" zur Verfügung steht, der nicht mit einem Leiharbeiter besetzt ist.

Antwort Gemini mit Kontext:
Nein, einen Rechtsanspruch auf einen Arbeitsplatz gibt es in Deutschland nicht.

Gemini ohne gegebenen Kontext:
Es gibt keinen generellen Rechtsanspruch auf einen bestimmten Arbeitsplatz. Allerdings gibt es in bestimmten Fällen besondere Schutzvorschriften, wie beispielsweise für schwerbehinderte Menschen. Diese haben einen Anspruch auf eine angemessene Beschäftigung.

4. Frage: Wie viele Tage steht Arbeitenden an Urlaub zu?

Antwort RAG:
Die Anzahl der Urlaubstage hängt davon ab, wie die Arbeitszeit auf die einzelnen Wochentage verteilt wird. Wenn der Arbeitnehmer trotz Teilzeitarbeit an fünf Werktagen pro Woche arbeitet, steht ihm dieselbe Anzahl von Urlaubstagen zu wie einem vollzeitbeschäftigten Kollegen. Wenn er an weniger als fünf Wochentagen arbeitet, ist der Urlaub anteilig zu kürzen. Bei zwei Tagen pro Woche erhält er beispielsweise 2/5 des

regulären Urlaubsanspruchs für Vollzeitbeschäftigte. Der volle Urlaubsanspruch wird nach § 4 BUrlG erst nach sechs Monaten erworben.

Antwort Gemini mit Kontext:
Arbeitnehmer haben in jedem Kalenderjahr Anspruch auf mindestens 20 Arbeitstage bezahlten Erholungsurlaub.

Gemini ohne gegebenen Kontext:
Der gesetzliche Mindesturlaub beträgt 24 Werktage im Jahr bei einer 6-Tage-Woche. Bei einer 5-Tage-Woche sind dies 20 Arbeitstage. Viele Tarifverträge und Arbeitsverträge sehen jedoch einen höheren Urlaubsanspruch vor.

5. Frage: Wann kommt es zu einer Freistellung?

Antwort RAG:
Laut dem bereitgestellten Text kommt es zu einer Freistellung in folgenden Fällen:

* **Pflegezeitgesetz (PflegeZG):** Bei der Versorgung pflegebedürftiger naher Angehöriger zu Hause, maximal 10 Arbeitstage bei akut aufgetretener Pflegesituation.
* **Stellensuche:** Wenn sich ein Arbeitnehmer während einer laufenden Kündigungsfrist bei einem neuen Betrieb vorstellt, hat er nach § 629 BGB einen Anspruch auf angemessene Freistellung zur Stellensuche.
* **§ 45 SGB V:** Wenn der Arbeitnehmer wegen der Covid-Pandemie nahe Angehörige oder Eltern ihr krankes (Klein-) Kind zu pflegen hat.

Antwort Gemini mit Kontext:
Eine Freistellung kann in verschiedenen Situationen erfolgen, z. B. bei eigenem Verschulden des Arbeitnehmers, bei Krankheit des Arbeitnehmers, bei Pflegebedürftigkeit naher Angehöriger oder bei bestimmten familiären Krisen- und Notsituationen.

Gemini ohne gegebenen Kontext:
Eine Freistellung bedeutet, dass ein Arbeitnehmer vorübergehend von der Arbeitspflicht befreit wird. Gründe für eine Freistellung können sein: Kündigung (bis zum Ende der Kündigungsfrist) Betriebliche Gründe (z. B. Umstrukturierung) Persönliche Gründe (z. B. Pflege von Angehörigen) Unbezahlte Freistellungen, wenn der Arbeitnehmer dies wünscht.