Chattbot med RAG

Kapitel 1

Chattbot

1.1 Chattbottar

I detta kapitel kommer vi lära oss mer om hur chattbottar såsom ChatGPT fungerar och hur man ställer effektiva frågor till dem, vilket kallas *prompt engineering*. Vi kommer därefter lära oss hur vi bygger en lokal chattbot som vi kan använda på vår egna dator. Kapitlet avslutas med att gå igenom RAG vilket låter oss anpassa chattbottens svar utifrån en given kontext, exempelvis utifrån egna dokument.

1.1.1 ChatGPT och *prompt engineering*

Chattbottar, som till exempel ChatGPT, används av många människor för en rad olika syften. Alltifrån att skapa kreativt innehåll, få förslag på förbättringar av skriven text eller programmeringskod till att lösa mer komplexa problem.

I korthet har chattbottar tränats på enorma datamängder och har från detta lärt sig vad som är rimliga sekvenser av ord. Exempelvis vet vi att om någon säger "Hej, hur mår ____" så brukar det sista ordet vara "du" för att meningen ska bli "Hej, hur mår du". När vi ställer en fråga till ChatGPT så uppskattar den sannolikheter för nästkommande ord och ger alltså ett svar som är sannolikt utifrån den datan som den blivit tränad på. I korthet kan vi säga att det fungerar som en "stokastisk papegoja". "Stokastisk" för att den uppskattar sannolikheter och "papegoja" för att den upprepar vad den lärt sig baserat på datan den blivit tränad på.

Det är uppenbart att AI-modeller och svaren de ger är beroende av datan de tränats på och alltså inte "sanna" i en egentlig mening. Läsaren uppmanas till att tänka igenom vilka etiska implikationer och risker detta kan ha.

Den läsare som vill fördjupa sig i hur ChatGPT funkar kan läsa igenom följande relativt korta bok: "What Is ChatGPT Doing … and Why Does It Work?" av Stephen Wolfram. https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/.

Ett gammalt talessätt säger "som man frågar får man svar" och detta gäller även chattbottar. Generellt sett kan vi ställa de frågor vi har och få rimliga svar utan att behöva tänka allt för mycket.

Om vi däremot vill ha "bättre" svar kan vi försöka ställa "effektiva frågor". Hur detta rent praktiskt går till är temat för det som kallas för *prompt engineering*.

Olika chattbottar kan fungera på olika sätt men som tumregel är det bra att vara så specifik, deskriptiv och detaljerad som möjligt om önskad kontext, utfall, längd, format, stil med mera.

Därför är följande prompt effektivare: "Skriv en kort och inspirerande dikt om AI som fokuserar på dess möjligheter och risker inom utbildning. Stilen ska likna Shakespears." än "skriv en dikt om AI".

Den läsare som vill lära sig mer om prompt engineering kan exempelvis kolla på följande guide kopplat till ChatGPT: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api.

1.2 Molnbaserade och lokala språkmodeller

Att träna en *Large Language Model* (LLM) kostar både tid och pengar och kräver stora mängder energi. För de allra flesta är det därför bättre att använda en förtränad modell. Det kan vi åstadkomma antingen genom att koppla upp mot en modell i molnet via ett API, eller genom att ladda ned en förtränad modell och köra den lokalt på vår egna dator.

Nedan beskrivs några för- och nackdelar md respektive tillvägagångssätt och därefter demonstreras hur vi skapar en chattbot som kopplar upp mot en modell i molnet.

1.2.1 Molnbaserade modeller

Att använda sig av molnbaserade modeller har sina för- och nackdelar.

Fördelar

- Enkla att implementera i ett befintligt system tack vare API
- Skalar automatiskt efter behov
- Uppdateras i takt med att teknologin utvecklas

Nackdelar

- Ökad användning kan leda till ökade kostnader
- Risker med hantering av känslig data
- Kräver extern anslutning till internet, hastigheten kan påverkas av problem med nätverket

1.2.2 Lokala modeller

Även lokala modeller har sina för- och nackdelar.

Fördelar

- Större kontroll över känslig data
- Kan tränas ytterligare för att skapa specialiserade modeller
- Svarar snabbare och kräver ingen extern anslutning till internet

Nackdelar

- Investeringar i infrastruktur kan innebära höga kostnader initialt
- Uppskalning kan innebära ytterligare investeringar

En en stor samling av modeller finns tillgängliga på Hugging Face (https://huggingface.co), som också fungerar som ett community för utveckling och implementering av AI.

1.3 Bygga en chattbot

I följande avsnitt ska vi steg för steg bygga en enkel chattbot som körs i terminalen. Vi kommer att använda Googles Gemini-modell som LLM.

> **!** För att kunna använda Gemini behöver vi en API-nyckel, vilket vi kan få genom att logga in på https://aistudio.google.com/ och skapa en nyckel.
>
> För att inte dela med sig av sin privata API-nyckel brukar man vanligtvis spara den som en miljövariabel och läsa in den med `os.getenv()`.
>
> Vi kommer inte i det här kapitlet gå igenom hur man hanterar miljövariabler i Python.

Vi installerar biblioteket `google-genai` via `pip`. Vi kan också använda exempelvis `conda` eller `uv`.

```
> pip install google-genai
```

Vi börjar med att kolla så att vi kan få Gemini att generera svar.

```
import os
from google import genai

client = genai.Client(api_key=os.getenv("API_KEY"))                    ①
```

① Skapa en instans av `Client`-klassen från `genai`-biblioteket.

```
response = client.models.generate_content(                             ①
    model="gemini-2.0-flash",
    contents="Hej där, vem pratar jag med?"
)

print(response.text)
```

① Generera ett svar med `generate_content()`-metoden från vår `client`-instans.

```
Hej! Du pratar med en stor språkmodell, tränad av Google. Du kan kalla
mig för vad du vill, jag har inget namn. Hur kan jag hjälpa dig idag?
```

Med hjälp av en `while`-loop och Pythons `input()`-funktion kan vi nu skapa en enkel textbaserad chattbot som vi kan köra i terminalen. Koden nedan demonstrerar detta och Figur 1.1 visar hur det kan se ut.

```python
# chatbot.py

import os
from google import genai

client = genai.Client(api_key=os.getenv("API_KEY"))

if __name__ == "__main__":
    print("*** Gemini chat ***")
    print("Type <q> to exit chat.")
    while True:
        prompt = input("User: ")
        if prompt == "q":
            break
        else:
            response = client.models.generate_content(
                model="gemini-2.0-flash",
                contents=prompt
            )
            print("Gemini: " + response.text)
```

```
(chatbot) linus@magnolia:~/documents/kapitel/chatbot$ python chatbot.py
*** Gemini chat ***
Type <q> to exit chat.
User: Hej, vem pratar jag med?
Gemini: Jag är en stor språkmodell, tränad av Google.
```

Figur 1.1: En enkel chattbot i terminalen

1.4 RAG

Om vi vill att vår Gemini-chattbot ska hålla sig till ett visst ämne kan vi använda en teknik som kallas RAG (*Retrieval Augmented Generation*). Det går ut på att vi ger modellen en kontext, ofta ett eller flera dokument eller liknande, att förhålla sig till. I en prompt säger vi åt modellen att enbart svara utifrån den givna kontexten. Om modellen inte hittar svaret i kontexten ska den säga det istället för att försöka hitta svaren någon annanstans eller gissa.

En RAG-modell innehåller alltså två delar.

Den första delen är en *retriever*, som söker efter relevanta stycken i en större text. Dessa stycken skickas sedan vidare som kontext till den andra delen, en *generator* som genererar svaren utifrån den givna kontexten.

För att ge modellen en kontext behöver vi läsa in data (exempelvis ett eller flera PDF-dokument) och bearbeta den så att vi kan göra en *semantisk sökning* i datan, det vill säga leta upp de stycken i kontexten som verkar ha mest med själva frågan att göra. Dessa stycken skickar vi sedan med till språkmodellen när vi ställer vår fråga.

Det finns ett antal ramverk för att implementera RAG, bland annat LangChain. Vi kommer inte använda något ramverk i det här kapitlet utan istället använda oss av vanligt förekommande Python-bibliotek som `pypdf` för att läsa in text från ett PDF-dokument och `numpy` för beräkningar samt en enkel *vector store* för att hantera vår data.

I resten av det här kapitlet kommer vi att använda texten från det här kapitlet som kontext när vi ställer frågor till vår chattbot.

Kodexemplet som följer bygger på kod från ett Github-repository som är en mycket bra källa till vidare läsning och experimentering. Repositoryt kan hittas på https://github.com/FareedKhan-dev/all-rag-techniques.

1.4.1 Läs in data

RAG-tekniken kan användas med många olika typer av data. I det här exemplet kommer vi arbeta med text-data.

Vi läser in ett PDF-dokument och extraherar all text i dokumentet.

```
from pypdf import PdfReader

reader = PdfReader("chattbot.pdf")

text = ""

for page in reader.pages:
    text += page.extract_text()
```

1.4.2 Chunking

Just nu är vår variabel `text` en enda lång textsträng som innehåller hela kapitlet. Vi behöver dela upp den i mindre delar, som brukar kallas *chunks*. Det är för att vi så småningom kommer att söka efter de delar av texten som innehåller den information vi är intresserade av, så kallad semantisk sökning.

Det finns ett antal olika strategier för *chunking*. Vi kommer inte gå igenom dem alla i det här kapitlet utan fokuserar på några av de vanligaste: *fixed-length chunking*, *sentence-based chunking*, och *semantic chunking*.

Fixed-length chunking

En vanlig strategi är den som kallas *fixed-length chunking*. Vi delar då upp texten i ett antal lika stora delar baserat på *tokens*, ord eller tecken. Det är en effektiv och enkel strategi, men den görs utan hänsyn till innehållet i texten. Det kan leda till att vi tappar information och att våra semantiska sökningar blir sämre.

När vi använder *fixed-length chunking* låter vi vanligtvis delarna överlappa med ett antal tecken. Hur stora delarna ska vara och hur många tecken som ska överlappa beror på typen av text och är något vi kan behöva prova oss fram till när vi bygger en RAG-modell.

I kodexemplet nedan skapar vi *chunks* som innehåller 1000 tecken vardera, med 200 teckens överlappning mellan delarna.

```
chunks = []
n = 1000
overlap = 200

for i in range(0, len(text), n - overlap):
    chunks.append(text[i:i + n])

print(f"Antal chunks: {len(chunks)}.")
```

```
Antal chunks: 30.
```

Sentence-based chunking

Ett alternativ till *fixed-length chunking* är *sentence-based chunking*. Vi delar upp texten i meningar, till exempel genom Pythons `split()`-funktion.

```
sentences = text.split(". ")

print(f"Antal meningar: {len(sentences)}.")
```

```
Antal meningar: 60.
```

Vi återkommer till *semantic chunking* längre fram.

I det här kapitlet kommer vi använda *fixed-length chunking*.

1.4.3 *Embeddings*

Vi behöver dock köra vår textdata genom ytterligare ett steg innan den är redo att användas som kontext till vår RAG-modell. Vi behöver skapa *embeddings*, det vill säga numeriska representationer av texten. *Embeddings* fångar textens betydelse, vilket gör att *chunks* som ligger närmare varandra i betydelse också har *embeddings* som ligger närmare varandra rent matematiskt.

Vi definierar en funktion `create_embeddings()`, som använder `embed_content()`-metoden från vår `client`-instans.

```
from google.genai import types

def create_embeddings(text,
    model="text-embedding-004",
    task_type="SEMANTIC_SIMILARITY"):                                    ①
        return client.models.embed_content(                              ②
            model=model,
            contents=text,
            config=types.EmbedContentConfig(task_type=task_type))
```

① Ange text, vilken modell vi vill använda samt i vilket syfte vi vill skapa *embeddings*.
② Funktionen kör i sin tur metoden `embed_content()`.

```
embeddings = create_embeddings(chunks)
```

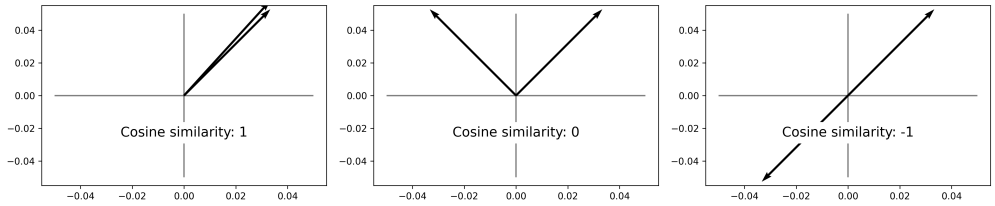
Våra *embeddings* lagras som vektorer. I Avsnitt 1.5 skapar vi en enkel *vector store* för att hantera våra *chunks* och *embeddings*.

1.4.4 Semantisk sökning

När vi har skapat våra *embeddings* är vi redo att göra semantiska sökningar för att hitta de *chunks* som verkar ligga närmast vår fråga i betydelse.

Det finns lite olika algoritmer vi kan använda oss av när vi jämför texter i en semantisk sökning. En av de vanligaste är cosinuslikheten (*cosine similarity*), som beskriver vinkeln mellan två vektorer. Cosinuslikheten är ett tal i intervallet $[-1, 1]$. Figur 1.2 visar exempel på tre olika par av vektorer med olika grad av cosinuslikhet.

Notera att de vektorer som utgörs av våra *embeddings* har många fler dimensioner än de som visas i Figur 1.2.



(a) Två proportionella vektorer *(b) Två ortogonella vektorer* *(c) Två motsatta vektorer*

Figur 1.2: Exempel på cosinuslikheter

Vi kan skriva en funktion som räknar ut cosinuslikheten enligt exemplet nedan.

```
import numpy as np

def cosine_similarity(vec1, vec2):
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) *
     ↪  np.linalg.norm(vec2))
```

Den här funktionen kan vi nu använda för att utföra en semantisk sökning bland våra *embeddings*.

```
def semantic_search(query, chunks, embeddings, k=5):
    query_embedding =
 ↪  create_embeddings(query).embeddings[0].values                    ①
    similarity_scores = []
```

```
    for i, chunk_embedding in enumerate(embeddings.embeddings):
        similarity_score = cosine_similarity(
            query_embedding,
            chunk_embedding.values
        )
        similarity_scores.append((i, similarity_score))

    similarity_scores.sort(key=lambda x: x[1], reverse=True)
    top_indices = [index for index, _ in similarity_scores[:k]]

    return [chunks[index] for index in top_indices]
```

① `embed_content()` returnerar ett `EmbedContentResponse`-objekt som innehåller olika data. Vi är bara intresserade av själva värdena på våra *embeddings*, vilka vi hittar i `embeddings`-attributet.

Vi får ut de fem *chunks* som har störst semantisk likhet med vår *query*. Vi kollar på det första resultatet.

```
query = "Vad innebär chunking?"
print(f"Fråga: {query}")
print("Svar:")
print(semantic_search(query, chunks, embeddings)[0])
```

```
Fråga: Vad innebär chunking?
Svar:
 vi
hittar i embeddings-attributet.
Vi får ut de fem chunks som har störst semantisk likhet med vår query.
Vi kollar på det
första resultatet.
query = "Vad innebär chunking?"
print(f"Fråga: {query}")
print("Svar:")
print(semantic_search(query, chunks, embeddings)[0])
Fråga: Vad innebär chunking?
Svar:
```

```
som innehåller oli-
ka data. Vi är bara intresserade av själva värdena på våra embeddings,
vilka vi
hittar i embeddings-attributet.
Vi får ut de fem chunks som har störst semantisk likhet med vår query.
Vi kollar på det
första resultatet.
query = "Vad innebär chunking?"
print(f"Fråga: {query}")
print("Svar:")
print(semantic_search(query, chunks, embeddings)[0])
12Fråga: Vad innebär chunking?
Svar:
n görs utan
12hänsyn till innehållet i texten. Det kan leda till att man tappar
information och att våra semantiska sökningar blir sämre.
När man använder fixed-length chunking låter man vanligtvis delarna
överlappa med ett antal tecken. Hur stora delarna ska vara och hur
många tecken som ska öv
```

> ***Semantic chunking***
>
> Nu när vi känner till begreppet semantisk sökning kan vi nämna ytterligare en metod för *chunking*, nämligen *semantic chunking*.
>
> *Semantic chunking* innebär att vi börjar med en *sentence based chunking*. Sedan skapar vi *embeddings* av våra meningar och jämför deras innehåll med en semantisk sökning. Utifrån resultaten skapar vi sedan *chunks* som innehåller meningar som ligger nära varandra i betydelse. Det är en bra metod om vi har flera olika dokument som tar upp samma saker.

1.4.5 Generera svar

Vår semantiska sökning verkar fungera ganska bra, men den återger ju bara texten så som den står i kapitlet. Nu är vi redo att låta språkmodellen generera svar som låter mer naturliga.

Vi börjar med att definiera en *system prompt*, som talar om för språkmodellen att den bara får utgå från informationen i kontexten som vi kommer skicka med när vi ställer

själva frågan i vår *user prompt*.

```
system_prompt = """Jag kommer ställa dig en fråga, och jag vill
 ↪  att du svarar
baserat bara på kontexten jag skickar med, och ingen annan
 ↪  information.
Om det inte finns nog med information i kontexten för att svara på
 ↪  frågan,
säg "Det vet jag inte". Försök inte att gissa.
Formulera dig enkelt och dela upp svaret i fina stycken. """
```

Nu kan vi definiera en funktion, `generate_user_prompt()`, som tar en fråga, skapar en kontext genom en semantisk sökning med vår fråga, och bakar ihop allting till en enda prompt som vi kan skicka till språkmodellen.

```
def generate_user_prompt(query):
    context = "\n".join(semantic_search(query, chunks,
 ↪  embeddings))

    user_prompt = f"Frågan är {query}. Här är kontexten:
 ↪  {context}."

    return user_prompt
```

Nu kan vi ställa en fråga till språkmodellen. Vi definierar ytterligare en funktion, `generate_response()`.

```
def generate_response(system_prompt, user_message,
 ↪  model="gemini-2.0-flash"):
    response = client.models.generate_content(
        model=model,
        config=types.GenerateContentConfig(
            system_instruction=system_prompt),
            contents=generate_user_prompt(user_message)
        )
    return response
```

```
print(generate_response(system_prompt, "Vad är semantic
 ↪  chunking?").text)
```

```
Semantic chunking är en metod för att dela upp text i mindre delar
baserat på den semantiska betydelsen.

Det innebär att skapa delar som innehåller meningar som ligger nära
varandra i betydelse.

Metoden innefattar sentence-based chunking, följt av skapandet av
embeddings av meningarna och en semantisk sökning för att jämföra deras
innehåll.

Resultaten används sedan för att skapa chunks som innehåller meningar
som är semantiskt relaterade.
```

1.4.6 Evaluering

Hur vet vi om vår RAG-modell fungerar som vi vill? Vi kan naturligtvis ställa den ett antal frågor och själva bilda oss en uppfattning, men det är ofta en bättre idé att skapa ett system för att utvärdera modellen med kod.

Vi skriver ett antal frågor, samt svar på frågorna som är i linje med vad vi önskar att modellen svarar. Sedan kan vi med hjälp av en annan *system prompt* låta modellen utvärdera om svaren på frågorna är i linje med det vi önskar, och sätta "betyg" på svaren. Det ger oss data att jämföra olika modeller med.

Vi börjar med att skriva ett antal testfrågor, och de svar vi önskar att modellen ska ge oss. En testfråga kan se ut som i exemplet nedan.

```
validation_data = [
    {
        "question": "Vilka delar utgör en RAG-modell?",
        "ideal_answer": "En RAG-modell innehåller två delar: en
         ↪  retriever som söker efter relevanta stycken i en text,
         ↪  och en generator som generar svar utifrån den givna
         ↪  kontexten."
    }
```

```
]
```

```
validation_data[0]["question"]
```

```
'Vilka delar utgör en RAG-modell?'
```

```
validation_data[0]["ideal_answer"]
```

```
'En RAG-modell innehåller två delar: en retriever som söker efter
relevanta stycken i en text, och en generator som generar svar utifrån
den givna kontexten.'
```

Nu skriver vi en ny *system prompt* som talar om för språkmodellen att vi vill att den ska utvärdera svaret.

```
evaluation_system_prompt = """Du är ett intelligent
 ↪  utvärderingssystem vars uppgift är att utvärdera en
 ↪  AI-assistents svar.
Om svaret är väldigt nära det önskade svaret, sätt poängen 1. Om
 ↪  svaret är felaktigt eller inte bra nog, sätt poängen 0.
Om svaret är delvis i linje med det önskade svaret, sätt poängen
 ↪  0.5. Motivera kort varför du sätter den poäng du gör.
"""
```

```
query = validation_data[0]["question"]
response = generate_response(system_prompt, query)

evaluation_prompt = f"""Fråga: {query}
AI-assistentens svar: {response.text}
Önskat svar: {validation_data[0]['ideal_answer']}"""

evaluation_response = generate_response(evaluation_system_prompt,
 ↪  evaluation_prompt)

print(evaluation_response.text)
```

```
Poäng: 1
Motivering: AI-assistentens svar överensstämmer helt med det önskade
svaret och innehåller all relevant information.
```

Vi kan också ge modellen en fråga som ligger utanför den givna kontexten och försäkra oss om att den inte försöker hitta på ett svar.

```
validation_data[2]
```

```
{'question': 'Hur många bultar finns det i Ölandsbron?',
 'ideal_answer': 'Det vet jag inte.'}
```

```
query = validation_data[2]["question"]
response = generate_response(system_prompt, query)

evaluation_prompt = f"""Fråga: {query}
AI-assistentens svar: {response.text}
Önskat svar: {validation_data[2]['ideal_answer']}"""

evaluation_response = generate_response(evaluation_system_prompt,
 ↪  evaluation_prompt)

print(evaluation_response.text)
```

```
Poäng: 1
Motivering: AI-assistenten svarade korrekt "Det vet jag inte." i
enlighet med instruktionerna.
```

1.5 Vector store

I det här kapitlet har vi arbetat med en liten mängd data. I fall där kontexten består av många dokument, bilder, filmer eller ljudfiler kan det ta lång tid att generera *embeddings*. Då är det en bra idé att spara dem till en fil. Det gör det också möjligt att dela våra *embeddings* med andra, eller att använda en molntjänst för att generera dem och sedan ladda ned dem och arbeta med dem lokalt på vår egen dator.

Det är vanligt att man använder en vektordatabas för att lagra *embeddings*. Exempel på vektordatabaser är faiss, qdrant och ChromaDB.

För enklare projekt är det dock inte nödvändigt att använda en vektordatabas. Vi kan implementera en enkel *vector store* med `numpy`, och använda datahanteringsbiblioteket `polars` för att spara våra *embeddings*, tillsammans med deras metadata, i Parquet-format. Vi använder `polars` istället för `pandas` på grund av att `polars` har bättre stöd för nästlad data, alltså kolumner som innehåller exempelvis listor.

```
import polars as pl

class VectorStore:
    def __init__(self):
        self.vectors = []
        self.texts = []
        self.metadata = []

    def add_item(self, text, embedding, metadata=None):
        self.vectors.append(np.array(embedding))
        self.texts.append(text)
        self.metadata.append(metadata or {})

    def semantic_search(self, query_embedding, k=5):
        if not self.vectors:
            return []

        query_vector = np.array(query_embedding)

        similarities = []
        for i, vector in enumerate(self.vectors):
            similarity = np.dot(query_vector, vector) /
↪   (np.linalg.norm(query_vector) * np.linalg.norm(vector))      ①
            similarities.append((i, similarity))

        similarities.sort(key=lambda x: x[1], reverse=True)

        results = []
        for i in range(min(k, len(similarities))):
            idx, score = similarities[i]
            results.append({
```

```
                "text": self.texts[idx],
                "metadata": self.metadata[idx],
                "similarity": score
            })

        return results

    def save(self):
        df = pl.DataFrame(
            dict(
                vectors=self.vectors,
                texts=self.texts,
                metadata=self.metadata)
            )
        df.write_parquet("embeddings.parquet")

    def load(self, file):
        df = pl.read_parquet(file, columns=["vectors", "texts",
↪   "metadata"])
        self.vectors = df["vectors"].to_list()
        self.texts = df["texts"].to_list()
        self.metadata = df["metadata"].to_list()
```

① Det här är `cosine_similarity()`-funktionen från tidigare.

Nu kan vi lagra våra *embeddings* i en instans av vår `VectorStore`-klass.

```
vector_store = VectorStore()

for i, chunk in enumerate(chunks):
    chunk_embedding =
↪   create_embeddings(chunk).embeddings[0].values
    vector_store.add_item(
        text=chunk,
        embedding=chunk_embedding,
        metadata={"type": "chunk", "index": i}
    )
```

Vi använder metoden `semantic_search()` för att utföra semantiska sökningar bland våra *chunks*.

```
query = "Vad är en semantisk sökning?"
query_embedding = create_embeddings(query).embeddings[0].values

vector_store.semantic_search(query_embedding=query_embedding)
```

Med metoden `save()` kan vi lagra våra *embeddings*, tillsammans med texten och metadatan. Filen `embeddings.parquet` kan vi sedan till exempel dela med oss av till andra som vill använda våra *embeddings*.

```
vector_store.save()
```

Metoden `load()` låter oss läsa in en lagrad fil.

```
vector_store2 = VectorStore()
vector_store2.load("embeddings.parquet")
```

Vår *vector store* kan vi nu använda istället för att generera kontext som vi kan skicka med till språkmodellen.

1.6 Vidare arbete med chattbottar

Detta kapitlet har gett en introduktion till hur vi bygger egna chattbottar. Den läsare som vill fördjupa sig och bland annat använda etablerade och välkända bibliotek kan undersöka till exempel LangChain för att hantera språkmodeller, och DeepEval för att evaluera dem.

1.7 Sammanfattning

I det här kapitlet har vi gått igenom språkmodeller och sett hur vi kan använda RAG-tekniken för att begränsa en språkmodell till att enbart generera svar utifrån en given kontext.

Vi har också sett exempel på hur vi kan skapa en *vector store* för att hantera och lagra våra *embeddings*.