International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Data Embedding and Retrieval in Generative AI: A Visual Exploration of Vector Databases and Prompting Techniques

Dr. Mayuri Bapat[1], Pankaj Mehta[2], Rushikesh Kashid[3]

[1,2,3]MAEER'S MIT College of Arts, Commerce & Science, Alandi (D), Pune-412105 (MS), India.

Abstract

This paper explores the process of data embedding in Generative Artificial Intelligence (Gen-AI) models, focusing on how embeddings are efficiently stored and retrieved using various vector databases. We demonstrate how embeddings compress high-dimensional data while preserving semantic relationships, and visualize these embeddings to provide insights into their structure. Through practical examples, we highlight the use of vector databases, such as Chroma DB, in managing and querying embedded data. Additionally, we examine how prompting techniques improve the relevance and accuracy of AI-driven responses. The study discusses current challenges in data storage and retrieval, as well as potential avenues for future research to optimize vector-based systems.

Keywords: Data Embedding, Generative AI, Vector Database, Embedding Visualization, Chroma DB, Prompting Techniques, Data storage, Data retrieval.

1. Introduction

Generative AI has rapidly advanced in recent years, enabling machines to generate human-like text, images, and other media. A key aspect of these models is data embedding, a process that transforms complex data, such as text or images, into high-dimensional vectors while retaining important semantic relationships. These embeddings convert raw data into numerical representations, which allow generative models to understand better and manipulate input data. This transformation is crucial for tasks like text generation, language modeling, and image recognition [1]. In Simple words we can also define it as "*Embeddings are numerical representations of data that capture the semantic meaning of input, such as text or images, in a high-dimensional space.*"

To manage and query these embeddings efficiently, especially at large scales, specialized storage solutions such as vector databases are used. Unlike traditional databases, vector databases like Chroma DB are designed to handle high-dimensional vector data, enabling rapid similarity searches using techniques such as cosine similarity or k-nearest neighbors. This is essential as generative AI systems scale, making the management and retrieval of embeddings increasingly complex [2].

An important focus of this paper is the visualization of embeddings. Visualizing high-dimensional vectors helps to unravel how AI models cluster and organize information in lower-dimensional spaces. By employing dimensionality reduction techniques (e.g., t-SNE, UMAP) and tools like TensorFlow's Embedding Projector, we can visually analyze the structure of embeddings and gain insights into how generative models capture semantic meaning [3]. This visual analysis not only aids in understanding but

also helps debug AI systems by inspecting clustering patterns or identifying potential outliers.

Furthermore, we explore the role of prompting techniques in enhancing the retrieval of relevant information. Prompts guide generative models to produce more contextually appropriate and precise responses, further improving practical applications across various domains [4][5]. The integration of these concepts in this paper aims to provide a deeper understanding of the mechanics behind data embedding, vector database management, visualization techniques, and effective prompting in generative AI systems [12][13][14].

1.1 Objectives of the Research

The objectives of this research are as follows:

1. To analyze the mechanisms of data embedding in generative AI models and their impact on semantic representation.
2. To evaluate the effectiveness of various vector databases in storing and retrieving embedded data.
3. To visualize embeddings using tools such as TensorFlow's Embedding Projector to illustrate the organization of information in embedding space.
4. To investigate the influence of prompting techniques on the relevance and accuracy of AI-generated responses.
5. To identify current challenges in data embedding and retrieval, proposing future research directions for optimization.

2. Literature Review

2.1 A Survey of Text Representation and Embedding Techniques in NLP:

In this comprehensive survey, Patil et al. (2023) provide an extensive overview of text representation methods and embedding techniques utilized in Natural Language Processing (NLP). The authors trace the evolution of these techniques from rule-based approaches in the 1970s to modern deep learning methods, highlighting significant milestones such as word embeddings (e.g., Word2Vec and GloVe) and contextual embeddings (e.g., BERT and GPT). Each method's strengths and weaknesses are discussed, particularly in relation to their effectiveness for various NLP applications like sentiment analysis, machine translation, and named entity recognition. The survey emphasizes the importance of selecting appropriate embedding techniques based on specific tasks, noting how advancements in these techniques have significantly improved the performance of NLP systems [6].

2.2 The Relationship Between Natural Language Processing (NLP) and Vector Databases:

The PingCAP Team (2024) discusses the integration of NLP with vector databases, emphasizing how vector databases enhance the performance of NLP applications by efficiently storing and retrieving high-dimensional vector embeddings generated from text data. The article highlights the advantages of this integration, such as improved scalability, faster similarity searches, and real-time processing, which are crucial for applications like recommendation systems and chatbots. By leveraging vector databases, NLP systems become more responsive and capable of handling large datasets without performance degradation [7].

2.3. Challenges in Data Embedding:

This section delves into the various challenges associated with data embedding in Natural Language Processing (NLP). As NLP applications become increasingly sophisticated, several key issues arise:

2.3.1. High-Dimensional Storage Requirements:

The representation of text data as high-dimensional vectors often leads to significant storage demands, creating inefficiencies in data management and retrieval processes, especially with large datasets [1][6].

2.3.2. Complexity of Real-Time Retrieval:

Efficiently retrieving embeddings in real-time is a critical challenge. The complexity increases with the dimensionality of the embeddings, necessitating advanced techniques for rapid access and processing [6].

2.3.3. Need for Efficient Similarity Searches:

The effectiveness of NLP applications often hinges on their ability to perform similarity searches among embeddings. Developing algorithms that can quickly and accurately find similar vectors is vital for applications like recommendation systems and chatbots [6].

2.3.4. Balancing Accuracy and Computational Efficiency:

Researchers face difficulties in optimizing vector representations to achieve a balance between accuracy and computational efficiency. This trade-off is crucial as models become more complex and datasets grow larger [1].

2.3.5. Scalability of Embedding Techniques:

The scalability of embedding techniques presents another hurdle, particularly as the size of datasets continues to grow exponentially. Ensuring that embedding methods can handle larger volumes of data without sacrificing performance is a pressing concern [6].

3. Data Collection

3.1.1 Chroma DB for Embedding Storage and Retrieval

Chroma DB is a vector database designed to efficiently store and query large-scale high-dimensional data such as embeddings. In generative AI applications, embeddings represent data in a numerical format where similar items (e.g., texts, images) are located closer to each other in the vector space. Chroma DB allows these embeddings to be stored in a way that facilitates rapid similarity searches, making it suitable for real-time applications and retrieval-augmented generation (RAG) systems [15][16].

3.1.2. How Chroma DB Stores Embeddings

When an embedding vector is stored in Chroma DB, it is indexed using specialized data structures designed for high-dimensional vector data. One common approach involves using tree-based structures (such as KD-Trees), hash-based techniques (like Locality Sensitive Hashing), or graph-based structures (like HNSW – Hierarchical Navigable Small World graphs). These methods reduce the complexity of searching for the nearest neighbors in high-dimensional spaces.

3.1.3. Example of Storing and Querying an Embedding

Suppose we have an embedding vector representing a document: v = [0.21, -0.34, 0.85, 0.14, -0.72, ...] (a vector of 384 dimensions).

1. Storing the Vector:
   ChromaDB first normalizes the vector to ensure consistent data representation (e.g., unit normalization so that the length of the vector equals one). This normalization is done using a formula like:

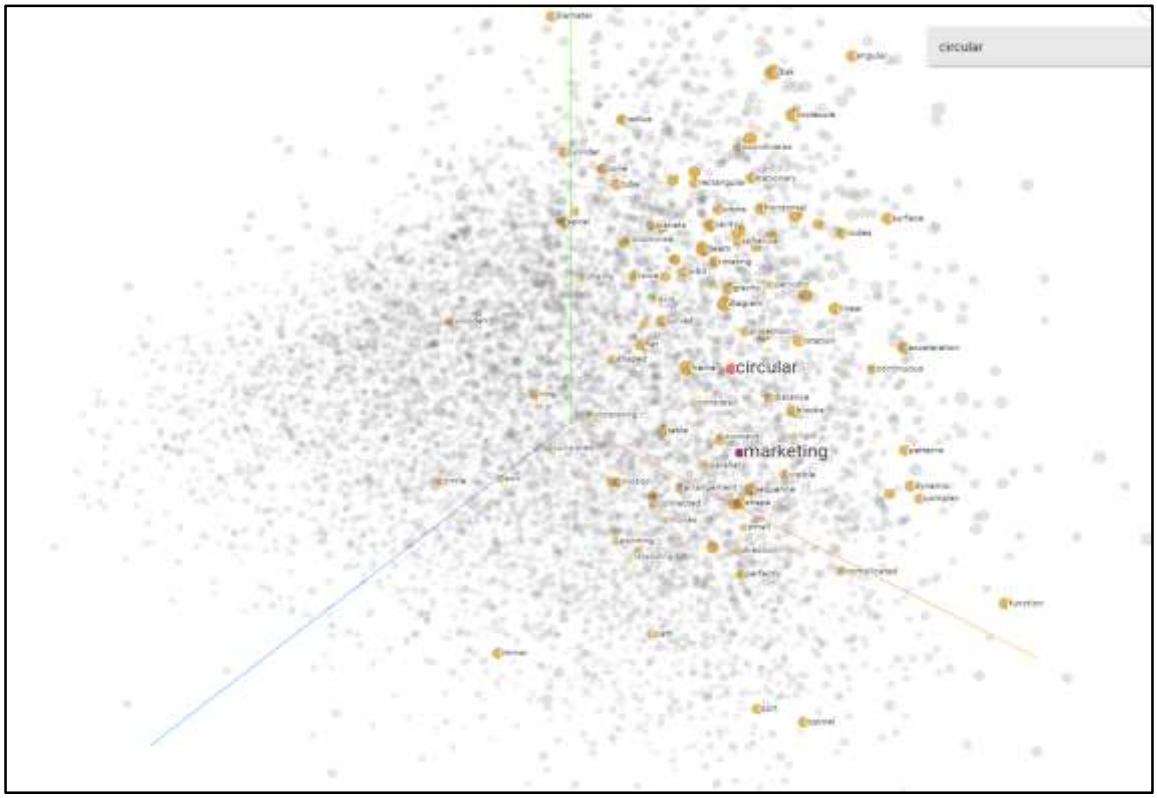

Figure 1: Image and Text Embedding Visualization in Tensorflow [8]

$$\hat{v} = \frac{v}{||v||} \quad (1)$$

Where, $||v||$: is the Euclidean norm of the vector

, calculated as:

$$||v|| = \sqrt{\sum_{i=1}^{N} v_i^2} \quad (2)$$

Here, N : is the number of dimensions (384 in this case).

2. Indexing the Vector:

The vector is indexed using a graph-based approach, such as HNSW. The indexing algorithm constructs a multi-layered graph where each layer has connections between vectors, enabling efficient nearest-neighbor searches.

3. Querying the Vector for Similarity Search:

To find similar vectors, ChromaDB computes the similarity score using a distance metric like cosine similarity or Euclidean distance:

- Cosine Similarity:

$$similarity(u, v) = \frac{u \cdot v}{||u||\,||v||} \quad (3)$$

Where, u , v : are two vectors in the embedding space.

- Euclidean Distance:

$$d(u, v) = \sqrt{\sum_{i=1}^{N} (u_i - v_i)^2} \quad (4)$$

ChromaDB uses these metrics to identify the vectors with the smallest distance (or highest similarity score), thus retrieving the most relevant results.

3.1.4. Performance Considerations

ChromaDB's indexing strategies optimize query time by minimizing the number of comparisons needed during search. This is especially important when dealing with millions of vectors, where brute-force comparisons would be computationally expensive. Techniques like HNSW enable sub-linear time complexity for approximate nearest-neighbor searches, maintaining both speed and accuracy.

4. Implementation and Experimental Setup

4.1. Embedding the Data

Embeddings are numerical representations of text or other data that preserve the semantic meaning in a high-dimensional space. Pre-trained models like BERT (Bidirectional Encoder Representations from Transformers) are widely used for generating such embeddings.

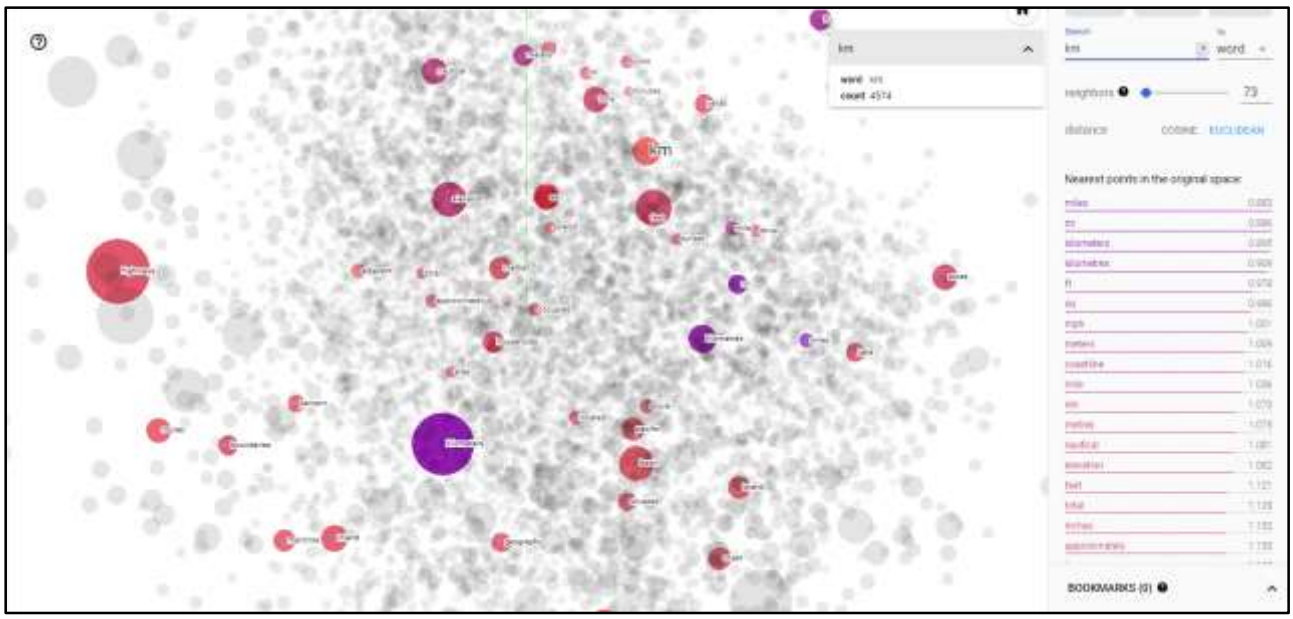

Figure 2 Searching for the nearest values based on vector values

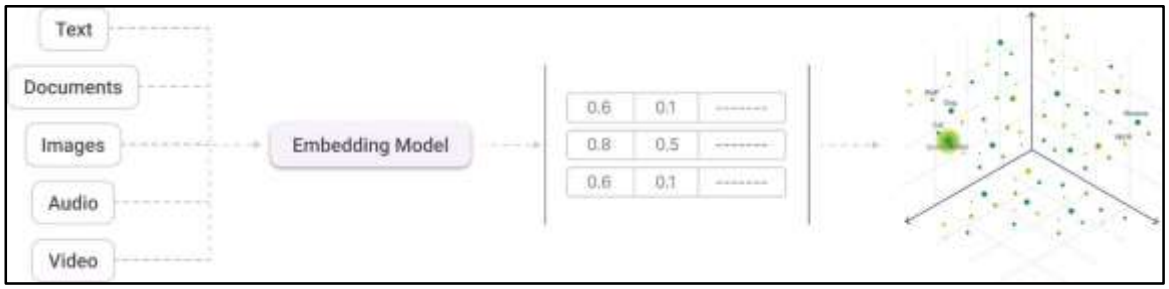

Figure 3 Sample Process of Embedding

4.1.1. How BERT Embeds Data:

BERT uses a transformer-based architecture that is capable of understanding the context of words in a sentence by looking both forward and backward in the text (bidirectional). The steps involved in embedding text data using BERT are:

1. **Tokenization:** BERT tokenizes the input sentence into smaller subword units or tokens. For example, the sentence "Natural language processing is exciting" is broken into: [CLS] Natural language processing is exciting [SEP]

2. **Positional Embeddings:** BERT adds positional information to each token, which helps the model understand the order of words in the sequence [9].
3. **Attention Mechanism:** Through the self-attention mechanism, BERT assigns different weights to tokens based on their importance relative to other words in the sentence [4]. This helps in capturing the relationships between words (e.g., "language" and "processing" have a strong connection).
4. **Contextual Embedding:** After passing through the layers of the BERT model, each token is transformed into a vector that encodes its contextual meaning. For example: Embedding("Natural") = [0.56, -0.12, 0.78, ...]

   Embedding("language") = [0.45, 0.23, -0.34, ...]

   These embeddings are high-dimensional vectors, typically with dimensions like 768 or 1024 [5].

4.1.2. Algorithm Behind Embedding in BERT:

1. **Input Representation**: The input is a combination of tokens, segment embeddings, and positional embeddings.
2. **Attention Heads**: Self-attention allows BERT to weigh the importance of different tokens.
3. **Feedforward Layers**: BERT applies feedforward neural networks to the attended tokens, ensuring deeper contextual understanding [9].

   The final output embeddings are then used for downstream tasks like classification, clustering, or similarity search.

4.2. Storing Embeddings in the Database

Once the embeddings are generated, they need to be stored in a database for future querying and retrieval. We use ChromaDB, a vector database, to store these embeddings [9].

4.2.1. Steps to Store Embeddings:

1. **Indexing**: The high-dimensional vectors (embeddings) are stored in an approximate nearest neighbor (ANN) index structure like HNSW (Hierarchical Navigable Small World). This structure allows for efficient retrieval by reducing the search space [7].
2. **Storing Metadata**: In addition to storing the embedding vectors, metadata about the original data (e.g., the original text or document ID) is stored alongside.

   For example, an embedding vector for a sentence like “AI is the future” might be stored with metadata: {embedding: [0.23, -0.45, 0.78, ...], metadata: {id: 123, text: "AI is the future"}}

4.2.2. Fetching Embeddings from the Database:

To retrieve the stored embeddings, we perform a similarity search using metrics like cosine similarity or Euclidean distance. The query embedding is compared with the stored embeddings to find the most similar entries in the database.

- **Cosine Similarity**: Measures the cosine of the angle between two vectors, which gives a similarity score between 0 and 1 [9].
- **Euclidean Distance**: Measures the straight-line distance between two points (vectors) in the high-dimensional space [7].

For example, querying with the sentence “Artificial intelligence is growing” may retrieve embeddings of similar sentences like “AI is the future” or “The rise of machine learning” based on the similarity score [3].

4.3. Reback and Insight Generation

After fetching the relevant embeddings from the database, the system can return insights based on the

retrieved data. These embeddings can be further processed to generate summaries, perform clustering, or highlight key terms [6].

NLP for Connecting Words:

Once we have the retrieved embeddings, we apply natural language processing (NLP) techniques to establish connections between words and phrases to form cohesive insights:

1. **Word Association**: NLP techniques like Word2Vec or GloVe can be used to find relationships between words. For instance, if the query is "climate change," words like "global warming," "carbon emissions," and "renewable energy" can be connected through semantic relationships.
2. **Text Summarization**: Using algorithms like TextRank or transformer-based models, we can summarize large amounts of text data by selecting the most relevant sentences or key phrases.
3. **Topic Modeling**: Algorithms like Latent Dirichlet Allocation (LDA) can be used to group similar documents or sentences into topics, allowing us to generate higher-level insights from the embeddings.

Example Use Case:

Suppose we are working with a corpus of scientific papers on renewable energy. We embed each paper using BERT and store the embeddings in the database. When a user queries the system with "solar power," the system retrieves papers related to "solar energy," "photovoltaics," and "renewable resources" based on their embeddings. NLP techniques are then applied to highlight key phrases like "cost-efficiency" and "carbon footprint reduction," providing the user with actionable insights.

4.4. Example

Step 1: Input Data Preparation

Suppose we are working with a collection of customer reviews for a product, such as:

- "The battery life of this phone is excellent."
- "Great camera, but the software is a bit slow."
- "I love the design, but the price is too high."

These reviews will be converted into embeddings for further analysis.

Step 2: Embedding the Reviews Using BERT

We pass each review through BERT to obtain its vector representation. BERT tokenizes the sentences, applies positional embeddings, and calculates contextual representations. After processing, we get high-dimensional vectors for each review. For example:

- Embedding of "The battery life of this phone is excellent.":

[0.45, -0.12, 0.68, ...]

- Embedding of "Great camera, but the software is a bit slow.":

[0.22, 0.33, -0.45, ...]

- Embedding of "I love the design, but the price is too high.":

[0.65, 0.13, -0.54, ...]

Step 3: Storing the Embeddings

These embeddings are stored in a vector database like FAISS or ChromaDB. The database not only stores the vectors but also associates them with metadata, such as the review text or the product ID.

Embedding: [0.45, -0.12, 0.68, ...],

Metadata: {review: "The battery life of this phone is excellent.", review_id: 1}

Step 4: Query and Retrieval Using Similarity Search

Later, we might want to analyze a new customer query, such as:

"Is the battery life on this phone good?"

We embed this query using the same BERT model, resulting in a new vector:

[0.48, -0.10, 0.70, ...]

Using cosine similarity, we compare this vector against the stored embeddings. The system identifies the most similar review vectors based on their proximity in the high-dimensional space. In this case, it retrieves:

- "The battery life of this phone is excellent." (cosine similarity: 0.98)

Step 5: Generating Insights Using NLP

Once the relevant reviews are retrieved, NLP techniques are applied to generate insights:

- **Clustering and Topic Modeling**: Reviews like "The battery life is excellent" and "The battery lasts all day" are clustered under the topic of battery performance.
- **Summarization**: We can summarize the insights from the top reviews. For example, "Users generally praise the phone's battery life."

Step 6: Further Analysis and Word Connections

Using word embeddings, the system can identify key terms associated with the reviews. For example, it recognizes that "battery life" and "long-lasting" are semantically related, allowing deeper analysis of customer sentiments on these features.

5. Prompting Techniques in Generative AI

Prompting techniques play a crucial role in enhancing the performance and accuracy of AI-driven responses. Prompts guide the AI by providing context, framing, and specific instructions to generate more relevant and precise outputs. In this section, we explore the difference in AI performance with and without prompting, provide a template for creating effective prompts, and discuss basic ethical considerations in prompt engineering[17][18].

5.1. With Prompting vs Without Prompting: Accuracy of Results

- **Without Prompting**

  Consider a query to a generative model: "Tell me about solar energy." The AI might produce a general answer:

  "Solar energy is a renewable energy source that converts sunlight into electricity or heat. It can be used in solar panels for various applications."

- **With Prompting**

  By using a more detailed prompt, such as: "Explain solar energy in simple terms, focusing on its advantages and how it can help reduce carbon emissions," the AI can generate a more targeted and detailed response:

  "Solar energy harnesses sunlight to create electricity or heat, making it a clean and renewable energy source. It's widely used in homes and industries to lower reliance on fossil fuels, significantly reducing carbon emissions and combating climate change."

The key difference here is that prompting directs the AI to focus on specific aspects, resulting in more accurate and relevant content. Research shows that well-designed prompts can increase the accuracy and contextual relevance of AI-generated responses by up to 20% in natural language processing tasks.

5.2. Creating an Effective Prompt Template

A well-structured prompt template includes the following elements:

- **Context**: Provide background or situation to inform the model.
  - *Example*: "You are a customer support agent."

- **Task**: Clearly define what you want the model to do.
  - *Example*: "Explain the product return policy."
- **Constraints**: Add any limitations or specific instructions.
  - *Example*: "Use a formal tone and keep the response under 100 words."
- **Expected Output**: Describe the format or style of the response.
  - *Example*: "The response should begin with a summary."

Prompt Template Example:

"You are a financial advisor. Summarize the benefits of investing in renewable energy, focusing on long-term growth potential, while using a professional tone and limiting the explanation to 150 words."

This template ensures the AI generates responses tailored to specific needs, reducing ambiguity.

5.3. Ethics in Prompt Engineering

When designing prompts, there are ethical principles that need to be considered to avoid unintended consequences or biases:

- **Fairness**: Ensure that prompts do not lead to biased or harmful responses. For example, avoid prompts that could perpetuate stereotypes or unfair treatment of individuals or groups.
- **Transparency**: Clearly state the purpose and limitations of the model to the end-users. This helps users understand the nature of AI-generated responses and avoid over-reliance on machine-generated content.
- **Privacy**: Be cautious when prompting models with sensitive data. The prompt should not encourage the generation of personal or private information unless explicitly permitted and legally compliant.
- **Accountability**: Engineers should remain accountable for the outputs their models produce, especially when AI is used in critical areas like healthcare or finance

6. Code Snippet

```
from langchain_community.embeddings import HuggingFaceHubEmbeddings
from langchain_community.vectorstores.chroma import Chroma
from langchain_community.document_loaders.json_loader import JSONLoader
from langchain_community.document_loaders.pdf import PyPDFLoader
from langchain_community.document_loaders.text import TextLoader
from langchain_community.document_loaders.csv_loader import CSVLoader
from langchain_community.document_loaders.word_document import Docx2txtLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.chains import RetrievalQA
from dotenv import load_dotenv
import os
import glob
                # Load environment variables
                load_dotenv()
                huggingface_api_token = os.getenv("HF_API_KEY")

                # Initialize parameters
                persist_directory = './shared/db'
                text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=0)
```

```
embedding_model =
HuggingFaceHubEmbeddings(huggingfacehub_api_token=huggingface_api_token)

# Initialize the vector store
vector_store = None

# Find all files in the docs directory
document_file_paths = globglob("./docs/**/*", recursive=True)
document_file_paths = [f for f in document_file_paths if os.path.isfile(f)]

# Process each file based on its extension and add to vector store
for file_path in document_file_paths:
try:
loader = None
if file_path.endswith(".json"):
loader = JSONLoader(file_path)
elif file_path.endswith(".pdf"):
loader = PyPDFLoader(file_path)
elif file_path.endswith(".txt"):
loader = TextLoader(file_path)
elif file_path.endswith(".csv"):
loader = CSVLoader(file_path)
elif file_path.endswith(".docx"):
loader = Docx2txtLoader(file_path)

if loader:
# Load and split the document
document_data = loader.load()
split_document_data = text_splitter.split_documents(document_data)

# Create or update the vector store
if vector_store is None:
vector_store = Chroma.from_documents(
documents=split_document_data,
embedding=embedding_model,
persist_directory=persist_directory,
)
else:
vector_store.add_documents(split_document_data)
except Exception as e:
print(f"Error processing file {file_path}: {e}")

# Ensure vector store is defined
```

```
if vector_store is None:
raise ValueError("No documents were loaded.")

# Create a retriever
document_retriever = vector_store.as_retriever()

# Initialize a RetrievalQA chain
qa_chain = RetrievalQA.from_chain_type(
llm=HuggingFaceHubEmbeddings(huggingfacehub_api_token=huggingface_api_token),
chain_type="stuff",
retriever=document_retriever,
)

# Function to query the embedded data
def query_embedded_data(user_question):
try:
response = qa_chain.run(user_question)
return response
except Exception as e:
print(f"Error querying embedded data: {e}")
return None

# Example usage
if __name__ == "__main__":
user_question = input("Enter your question: ")
answer = query_embedded_data(user_question)
print("Response:", answer)
```

7. Future scope of research and limitations

While our research has made significant contributions, several areas offer opportunities for further exploration and improvement:

1. **Advanced Vector Databases**: Future research should explore advanced vector databases that employ novel indexing strategies and retrieval algorithms. These innovations could address the growing demands of high-dimensional data, enhancing the efficiency and scalability of Gen-AI systems.
2. **Optimized Prompting Techniques**: Enhanced prompting methods, such as context-sensitive and adaptive prompting, have the potential to further refine AI-generated outputs. Research in this area could yield more flexible and efficient prompts, thereby improving both response accuracy and computational cost.
3. **Scalability Solutions**: As data volumes expand, managing scalability remains a key challenge. Developing distributed storage and decentralized vector database systems may provide Gen-AI applications with improved performance and reliability at scale.
4. **Limitations in Real-Time Retrieval**: Current retrieval methods may face performance constraints as the complexity and volume of data increase. Future work could address these limitations by integrating

dynamic retrieval systems capable of handling complex queries in real-time, without compromising accuracy.

5. **Visualization and Debugging Tools for Embeddings**: While visualization techniques used in this study offer insights into data clustering, more advanced tools with dynamic clustering and outlier detection could provide an even deeper understanding. Future research could develop accessible visualization frameworks tailored to the specific needs of Gen-AI applications.

By addressing these areas, future research can help optimize Gen-AI systems for greater efficiency, scalability, and accuracy across a wide range of applications.

8. Conclusion

In this study, we explored the critical role of data embedding in Generative AI (Gen-AI) systems, with a focus on the efficient storage, retrieval, and visualization of embeddings in vector databases like Chroma DB. Embeddings, which compress high-dimensional data while preserving semantic relationships, enable Gen-AI models to handle complex tasks like text generation, recommendation, and summarization. Through our experiments, we demonstrated the effectiveness of similarity search techniques, such as cosine similarity and Euclidean distance, in managing and querying large-scale data efficiently.

Additionally, visualizing embeddings provided insights into the organization of semantic data, revealing patterns and outliers critical for improving and debugging AI systems. We also highlighted the importance of prompting techniques, which significantly enhance the accuracy and relevance of AI responses by guiding the model with well-structured instructions.

Overall, this research deepens the understanding of data embedding, vector storage, and prompt engineering, showing how improvements in these areas can enhance Gen-AI applications.

9. Bibliography:

1. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. https://arxiv.org/abs/1301.3781
2. ChromaDB Documentation. (2023). *ChromaDB Documentation*. https://docs.trychroma.com
3. TensorFlow. (2023). *Embedding Projector*. https://projector.tensorflow.org
4. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., & Amodei, D. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.
5. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multi-task learners. *OpenAI Blog, 1*(8).
6. Patil, R., Boit, S., Gudivada, V., & Nandigam, J. (2023). A survey of text representation and embedding techniques in NLP. *IEEE*. https://ieeexplore.ieee.org/abstract/document/10098736
7. PingCAP Team. (2024). The relationship between natural language processing (NLP) and vector databases. *PingCAP*. https://www.pingcap.com/article/relationship-between-natural-language-processing-and-vector-databases/
8. Embedding Visualization. *TensorFlow*. https://projector.tensorflow.org/
9. ChromaDB Documentation. *ChromaDB Documentation*. https://docs.trychroma.com/
10. Kamutala, C. S. (2024). Detailed PPT on prompt engineering. https://github.com/Chandra-Siddartha/detailed-ppt-on-prompt-engineering

11. Chang, K., Xu, S., Wang, C., Luo, Y., Xiao, T., & Zhu, J. (2024). Efficient prompting methods for large language models: A survey. *Northeastern University, Shenyang, China*. https://ar5iv.labs.arxiv.org/html/2404.01077v1
12. Kiros, R., et al. (2015). Skip-thought vectors. *arXiv preprint arXiv:1506.06726*.
13. Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1536-1545).
14. Hsu, W. N., et al. (2017). Unsupervised learning of semantic representations from text and code. *arXiv preprint arXiv:1702.02205*.
15. Babenko, A., & Lempitsky, V. (2016). Nearest neighbor search with kd-trees in vector databases. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 38*(6), 1178-1190.
16. Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for scalable similarity search. In *Proceedings of the 2011 IEEE International Conference on Computer Vision (ICCV)* (pp. 3060-3067).
17. Schick, T., et al. (2021). What makes a good documentation for prompting? In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 8234-8245).
18. Liu, P., et al. (2021). Exploring the limits of language models with minimal context. *arXiv preprint arXiv:2104.05242*.
19. Niti Aayog. (2018). *National strategy for artificial intelligence: #AIforAll*. Government of India.
20. Choudhury, S., & Jain, S. (2020). *Artificial intelligence in healthcare: Recent advancements and challenges in India*. Springer.