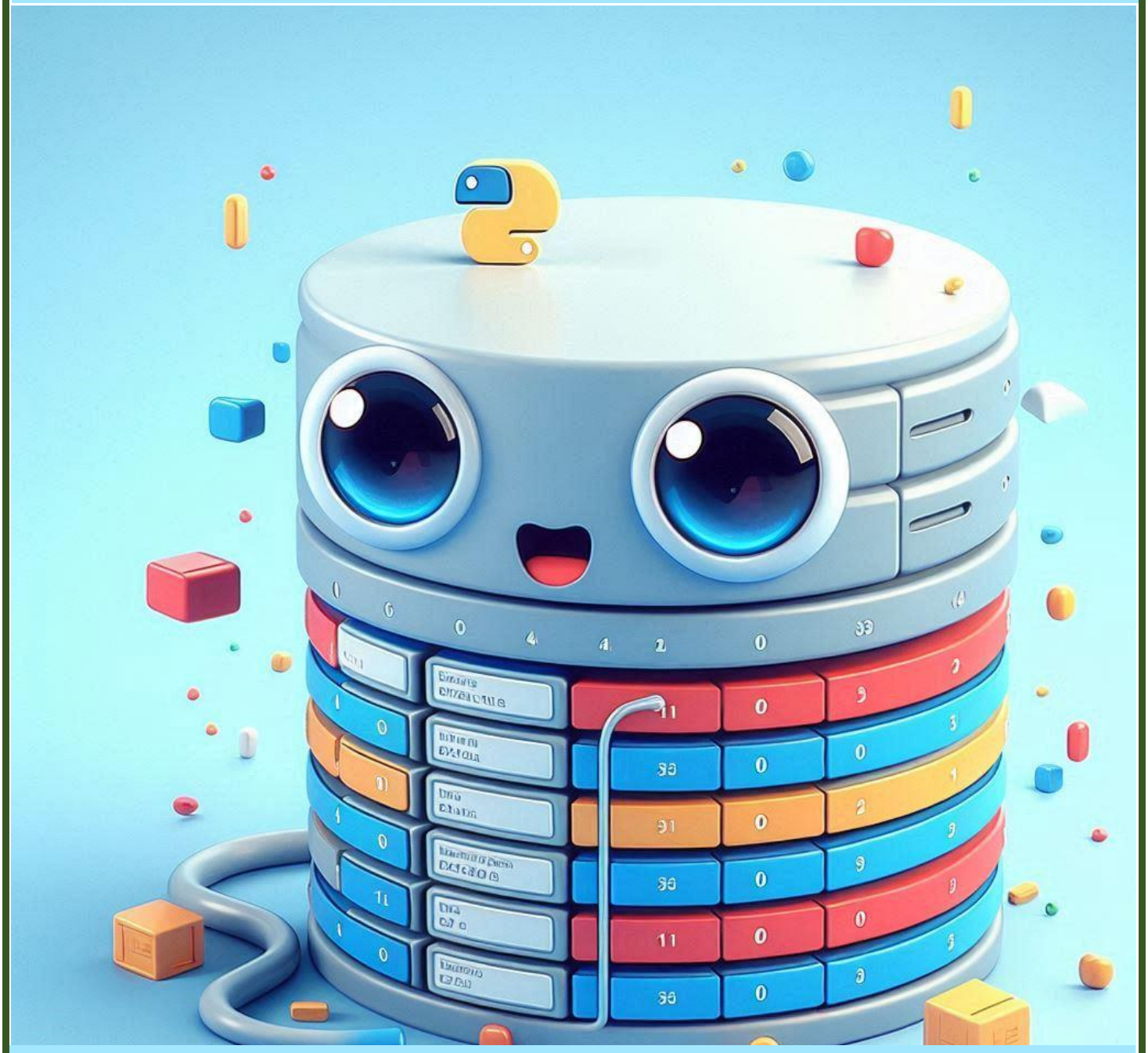
Understanding

Vector Database System

through Python Implementation



Understanding Vector Database System through Python Implementation

1. Introduction

Imagine you have a big box of photos, but instead of pictures, it holds the *meaning* of sentences. This "meaning" is stored as a special code called a vector.
This Guide teaches you how to build that box (a vector database) using the Python computer language.

Here's what you'll learn:
1. What are vector databases? We'll explain how they work and why they're useful for understanding language.
2. Let's build one! We'll use Python code to create a simple vector database that can store and find sentences based on how similar they are.

Think of it like this: you tell the database to find sentences similar to "The cat sat on the mat." It might return sentences like "The dog slept on the floor" because their meanings are close.
This Guide is a great starting point for anyone interested in natural language processing and how computers can understand human language.
This guide will learn how to tokenize sentences, create vocabulary, vectorize sentences, and perform similarity searches.

Overview

Understanding Vector Databases
Vector databases store data in vector form, facilitating tasks like similarity searches. They are crucial in NLP and machine learning for representing words, sentences, or documents as vectors, enabling efficient retrieval based on similarity.
Implementation
We'll create a Python class called **VectorStore** to store vectors and perform similarity searches using cosine similarity.

Code Walkthrough

Step 1: Imports and Setup
First, let's import the necessary libraries and set up our environment:

```
from vector_store import VectorStore  # Import the VectorStore class
import numpy as np  # Import numpy for numerical operations
```

Step 2: Initialize VectorStore
Create an instance of the `VectorStore` class:

```
vector_store = VectorStore()  # Create an instance of VectorStore
```

Step 3: Define Sentences
Define a list of example sentences:

```
sentences = [
    "I eat mango",
    "mango is my favorite fruit",
    "mango, apple, oranges are fruits",
    "fruits are good for health",
]
```

Step 4: Tokenization and Vocabulary Creation

Tokenize the sentences and create a vocabulary set:

```
vocabulary = set()  # Initialize an empty set to store unique words
for sentence in sentences:
    tokens = sentence.lower().split()  # Convert sentence to lowercase and split into words
    vocabulary.update(tokens)  # Add unique words to the vocabulary set

# Create a dictionary mapping words to unique indices
word_to_index = {word: i for i, word in enumerate(vocabulary)}
```

Step 5: Vectorization

Convert each sentence into a vector based on word frequency:

```
sentence_vectors = {}  # Initialize an empty dictionary to store sentence vectors
for sentence in sentences:
    tokens = sentence.lower().split()  # Convert sentence to lowercase and split into words
    vector = np.zeros(len(vocabulary))  # Initialize a zero vector of the same length as the vocabulary
    for token in tokens:
        vector[word_to_index[token]] += 1  # Increment the count of each token in the vector
    sentence_vectors[sentence] = vector  # Store the vector for the sentence in the dictionary
```

Step 6: Store Vectors in VectorStore

Add each sentence vector to the `VectorStore`:

```
for sentence, vector in sentence_vectors.items():
    vector_store.add_vector(sentence, vector)  # Add the sentence vector to VectorStore
```

Step 7: Similarity Search

Define a query sentence and find similar sentences:

```
query_sentence = "Mango is the best fruit"  # Define a query sentence
query_vector = np.zeros(len(vocabulary))  # Initialize a zero vector of the same length as the vocabulary
query_tokens = query_sentence.lower().split()  # Convert query sentence to lowercase and split into words
for token in query_tokens:
    if token in word_to_index:
        query_vector[word_to_index[token]] += 1  # Increment the count of each token in the query vector

similar_sentences = vector_store.find_similar_vectors(query_vector, num_results=2)#Find similar sentences
```

Step 8: Display Results

Print the query sentence and the most similar sentences:

```
print("Query Sentence:", query_sentence)  # Print the query sentence
print("Similar Sentences:")
for sentence, similarity in similar_sentences:
    print(f"{sentence}: Similarity = {similarity:.4f}")  # Print similar sentences with their similarity
scores
```

VectorStore Class

Here is the implementation of the `VectorStore` class:

```
import numpy as np  # Import numpy for numerical operations

class VectorStore:
    def __init__(self):
        self.vector_data = {}  # Dictionary to store vectors
        self.vector_index = {}  # Indexing structure for retrieval

    def add_vector(self, vector_id, vector):
        """
        Add a vector to the store.

        Args:
            vector_id (str or int): A unique identifier for the vector.
            vector (numpy.ndarray): The vector data to be stored.
        """
        self.vector_data[vector_id] = vector  # Store the vector in the dictionary
        self._update_index(vector_id, vector)  # Update the index with the new vector

    def get_vector(self, vector_id):
        """
        Retrieve a vector from the store.

        Args:
            vector_id (str or int): The identifier of the vector to retrieve.

        Returns:
            numpy.ndarray: The vector data if found, or None if not found.
        """
        return self.vector_data.get(vector_id)  # Retrieve the vector from the dictionary

    def _update_index(self, vector_id, vector):
        """
        Update the index with the new vector.

        Args:
            vector_id (str or int): The identifier of the vector.
            vector (numpy.ndarray): The vector data.
        """
        # Use brute-force cosine similarity for indexing
        for existing_id, existing_vector in self.vector_data.items():
            similarity = np.dot(vector, existing_vector) / (np.linalg.norm(vector) * np.linalg.norm(existing_vector))
            if existing_id not in self.vector_index:
                self.vector_index[existing_id] = {}  # Initialize nested dictionary if not present
            self.vector_index[existing_id][vector_id] = similarity  # Store similarity score

    def find_similar_vectors(self, query_vector, num_results=5):
        """
        Find similar vectors to the query vector using brute-force search.

        Args:
            query_vector (numpy.ndarray): The query vector for similarity search.
            num_results (int): The number of similar vectors to return.

        Returns:
            list: A list of (vector_id, similarity_score) tuples for the most similar vectors.
        """
        results = []
        for vector_id, vector in self.vector_data.items():
            similarity = np.dot(query_vector, vector) / (np.linalg.norm(query_vector) * np.linalg.norm(vector))
            results.append((vector_id, similarity))  # Append (vector_id, similarity_score) tuple to results list

        results.sort(key=lambda x: x[1], reverse=True)  # Sort results by similarity score in descending order
        return results[:num_results]  # Return the top N results
```

Workflow

1. Tokenization and Vocabulary Creation:
- Tokenize sentences to break them into individual words.
- Create a vocabulary set to store unique words from all sentences.
- Map each word in the vocabulary to a unique index.

2. Vectorization:
- Convert each sentence into a vector where each element corresponds to the frequency of a word in the vocabulary.

3. Storing in VectorStore:
- Add each sentence vector to an instance of the `VectorStore` class.

4. Similarity Search:
- Create a vector for a query sentence.
- Find similar sentences by calculating cosine similarity between the query vector and stored vectors.

5. Output:
- Print the query sentence and the most similar sentences with their similarity scores.

Conclusion

In this guide, we built a basic vector database in Python. This foundational implementation can be expanded for more complex real-world applications. We covered key concepts, provided a hands-on code walkthrough, and outlined the workflow for constructing and using the vector database.

Constructive comments and feedback are welcomed