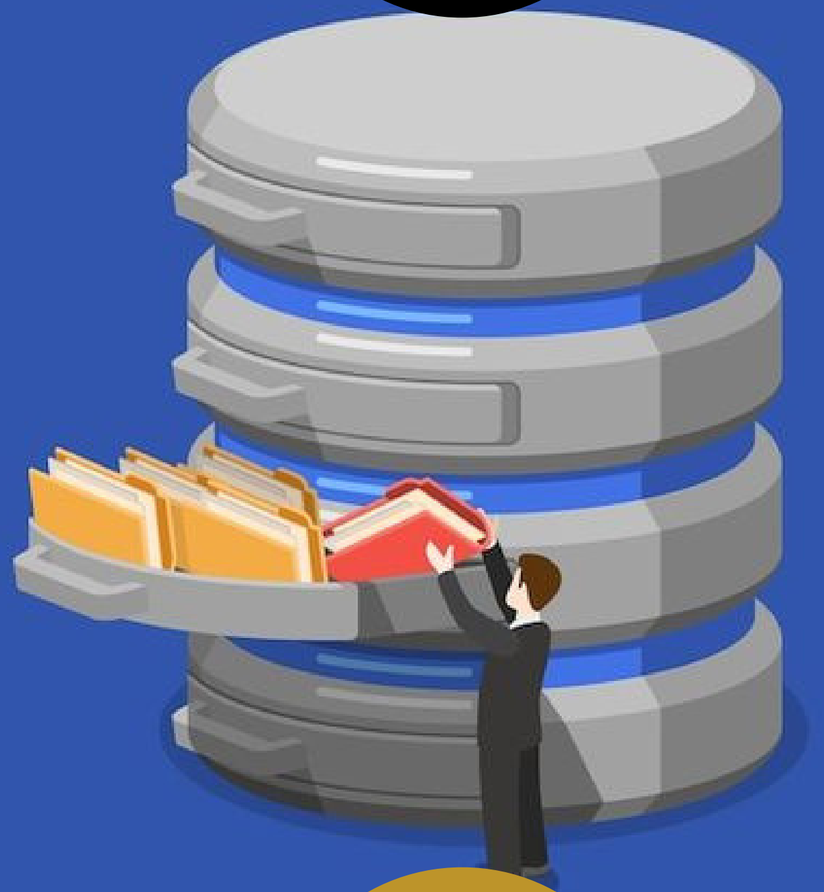

RV University

VECTOR DATABASES

Presentation by

Karmishtha Patnaik

Abstract

In this presentation, we explore the transformative power of vector databases in handling diverse unstructured data types. Through real-world examples and practical insights, we delve into their efficiency, scalability, and real-time analytics capabilities. From understanding the basics to tackling challenges and exploring future trends, attendees will gain a concise yet comprehensive understanding of how vector databases are reshaping the data landscape. Join us for a focused journey into the heart of modern data science.

Introduction

Definition: Vector databases are specialized databases designed to handle vector data, which consists of **numerical representations** of objects used in machine learning, artificial intelligence, and data analytics.

Example: Think of a music streaming service. Traditional databases can store user information, but vector databases can store audio embeddings (vectors representing songs), enabling quick and accurate song recommendations based on similar user preferences.

Advantages of Vector Databases

Speed and Efficiency

Vector databases leverage advanced algorithms to retrieve similar items rapidly, enabling real-time applications like recommendation systems

Scalability

These databases can handle large-scale datasets by distributing data across multiple nodes, ensuring seamless scalability as data volumes grow.

Use Cases

Showcase specific use cases in e-commerce (product recommendations), healthcare (medical image analysis), and security (facial recognition) to emphasize practical applications.

Example

Consider an e-commerce platform. A vector database can store product images as vectors. When a user searches for a specific item, the system can quickly find visually similar products using vector similarity algorithms, enhancing the user experience.

How Vector Databases Work

01

Vectorization:

- Explanation of how raw data is transformed into numerical vectors, making it suitable for mathematical operations and comparisons.

02

Indexing and Searching:

- Detailed overview of indexing techniques such as tree-based indexes and hashing methods, explaining how these facilitate efficient similarity searches.

Example

Imagine a language translation service. Texts from different languages are converted into vectors. Vector databases use algorithms like LSH to find similar vectors, enabling accurate translation suggestions based on similar sentence structures and contexts..

Problems

- Data Quality
- Integration
- Security
- Vendor Selection

Example

For a healthcare application using facial recognition, ensuring data quality means accurate representation of patient facial features. Robust security measures, like encryption in Amazon DynamoDB, protect patient data. Integration challenges might involve integrating the facial recognition system with existing hospital databases, emphasizing the need for seamless data flow.

WORD EMBEDDINGS

Word embeddings are numerical representations of words in a high-dimensional space, where words with similar meanings are closer to each other. These representations capture semantic relationships between words, making them useful in various natural language processing tasks.

Word2Vec is a popular word embedding technique that learns word vectors from large amounts of text data. It does so by considering the context in which words appear in the text. There are two main approaches in Word2Vec: **Continuous Bag of Words (CBOW)** and **Skip-gram**.

CBOW

Continuous Bag of Words (CBOW):

CBOW is a type of Word2Vec model that predicts a target word based on its surrounding context words. Here's how it works:

1. **Input:** CBOW takes a sequence of context words as input, for example, in the sentence "The cat sat on the ___," the context words are "The," "cat," "on," and "the."
2. **Prediction:** The model predicts the target word, which is "sat" in this case. It learns to predict the target word using the provided context words.
3. **Architecture:** CBOW uses a neural network with an input layer, a hidden layer, and an output layer. The input layer represents the context words, the hidden layer captures the relationships between words, and the output layer predicts the target word.
4. **Training:** During training, the model adjusts its parameters (word vectors) so that the predicted word is close to the actual target word. It does this by minimizing the difference between predicted and actual word representations.

Skip Gram

Skip-gram:

Skip-gram, on the other hand, works in the opposite way. Instead of predicting a target word from context words, Skip-gram predicts context words from a target word. Here's how it works:

1. **Input:** Skip-gram takes a target word as input, for example, in the sentence "The cat sat on the ___," the target word is "sat."
2. **Prediction:** The model predicts the context words, which are "The," "cat," "on," and "the." It learns the likelihood of each context word given the target word.
3. **Architecture:** Similar to CBOW, Skip-gram also uses a neural network with an input layer, a hidden layer, and an output layer. The input layer represents the target word, the hidden layer captures word relationships, and the output layer predicts context words.
4. **Training:** Skip-gram is trained by adjusting its parameters so that the predicted context words are close to the actual context words for the given target word. It learns to represent words in a way that captures their contextual usage.

GloVe (Global Vectors for Word Representation):

1. **Unsupervised Learning Algorithm:** GloVe is an unsupervised learning algorithm that generates vector representations (embeddings) for words.
2. **Count-Based Model:** Unlike Word2Vec, which is predictive, GloVe is a count-based model. It learns word embeddings by analyzing the global statistics of word co-occurrence in a corpus.
3. **Co-occurrence Matrix:** GloVe constructs a word-word co-occurrence matrix from input text data. Each matrix entry represents the frequency of co-occurrence of two words within a specified context window.
4. **Inference of Word Meaning:** The key idea is that word meaning can be inferred from the co-occurrence patterns of words in a large text corpus.
5. **Factorization:** GloVe learns to map words into a continuous vector space by factorizing the co-occurrence matrix. This process captures semantic relationships between words based on their statistical co-occurrence patterns.
6. **Semantic Relationships:** The resulting word embeddings are capable of preserving semantic relationships. Words with similar meanings have similar vector representations, allowing mathematical operations on these vectors to reveal relationships like analogies (e.g., king - man + woman = queen).

FastText

1. **Word Embedding Extension:** FastText is an extension of the Word2Vec model developed by Facebook's AI Research (FAIR) lab.
2. **Character n-grams Representation:** FastText represents words as bags of character n-grams. An n-gram is a contiguous sequence of n characters from a word. For example, for the word "apple" and n=2, the character n-grams are <ap, pp, pl, le>. Both start and end of the word are also considered.
3. **Averaging Character n-grams:** These character n-grams are then averaged to create the word representation. This method captures morphological and subword information of the words.
4. **Handling Morphologically Rich Languages:** FastText is especially useful for languages with complex word forms and morphological variations. It can represent words even if they are rare or out-of-vocabulary by aggregating character n-grams.
5. **Efficiency and Robustness:** FastText is efficient and robust, making it suitable for large-scale applications. It is widely used in tasks like machine translation and sentiment analysis, where handling diverse and complex linguistic patterns is crucial.
6. **Subword Information:** By considering character n-grams, FastText can understand the meaning of words even when they share common prefixes or suffixes. This ability to capture subword information is valuable for understanding the semantics of words in various contexts.

TORCH library

PyTorch and Vector Embeddings:

1. **Embedding Layers:** PyTorch provides **nn.Embedding**, a module specifically designed for word embeddings. It maps word indices to dense vectors, crucial for natural language processing tasks.
2. **Pre-trained Embeddings:** PyTorch allows loading pre-trained word embeddings like GloVe or FastText, enabling the use of external semantic knowledge in models.
3. **Custom Embeddings:** You can initialize and train custom embeddings specific to your dataset. PyTorch facilitates the training of embeddings alongside neural networks during model training.
4. **Handling Sequences:** For text-related tasks, PyTorch integrates embeddings into models processing sequential data, such as RNNs and transformers, by converting word indices into dense vectors.

In essence, PyTorch's **torch** library is instrumental in managing, utilizing, and training word embeddings for various natural language processing applications.

OPTIM

In PyTorch, the **optim** module is used to optimize various types of parameters, including the parameters of embedding layers. When training neural networks with embedding layers, you can use the **optim** module to update the embedding vectors based on the gradients computed during backpropagation.

1. **Optimizer Selection:** Choose appropriate optimizer (e.g., SGD, Adam) for updating embeddings.
2. **Parameter Initialization:** Initialize optimizer with embedding parameters.
3. **Gradient Computation:** Gradients for embeddings are computed during backpropagation.
4. **Learning Rate:** Adjust learning rate for optimizer convergence (e.g., lr=0.01).
5. **Optimization Step:** Update embeddings using **optimizer.step()** based on computed gradients.

Pytorch : Strenghts and Limitation

- **Strengths:**
  - Flexible Framework: PyTorch provides a versatile platform for building diverse neural network architectures, crucial for advanced LLMs like BERT and GPT.
  - Dynamic Computation Graphs: PyTorch's dynamic graphs enable adaptive network structures, vital for tasks where input data influences the model's architecture.
  - GPU Acceleration: Offers robust GPU support, ensuring accelerated training, particularly for complex models.
- **Considerations:**
  - Expertise Needed: Developing sophisticated LLMs in PyTorch requires expertise in designing architectures and managing intricate contextual nuances.

Word2Vec:Strengths and Limitations

Word2Vec:

- Strengths:
  - Simplicity: Word2Vec provides a straightforward approach to generating word embeddings, which can be used as features for LLMs.
  - Resource Efficiency: Computationally lighter than deep LLMs, making it suitable for resource-constrained environments.
  - Semantic Context: Captures basic semantic relationships, although not as intricate as advanced LLMs.
- Considerations:
  - Contextual Limitations: Limited in capturing complex contextual nuances present in advanced LLMs like BERT.

Consideration for LLMs:

- For Cutting-edge Applications: Opt for PyTorch or similar frameworks when aiming for cutting-edge research or applications demanding complex contextual understanding.
- Simplicity and Efficiency: Choose Word2Vec when prioritizing simplicity, efficiency, and computational resources, especially for less complex tasks or resource-constrained environments.

Why can’t we use openAI embedding models :Acts like an advantage and disadvantage at the same time

Limiting access helps OpenAI ensure data privacy and security, especially when handling sensitive information from users.
While OpenAI's advanced models are not open source like Word2Vec, OpenAI has provided some models and tools for research purposes. Additionally, they have released smaller versions of their models for free access, allowing developers to experiment and learn with the technology.

These AI embedding models are not always freely accessible due to their computational complexity, requiring significant resources for training and inference. Additionally, they are often proprietary technologies developed by organizations investing substantial intellectual and financial capital. Controlled access ensures responsible usage, protects intellectual property, and helps maintain ethical standards in AI applications. While some versions might be accessible for free, commercial usage often involves subscription models to sustain ongoing research, development, and maintenance efforts.

Potential Fields in the domain of text embedding

1. **Enhanced Conversations: Text embeddings will make chatbots and AI conversations more natural and human-like.**
2. **Explainable AI: They will help explain complex AI decisions, building trust in AI systems.**
3. **Medical Discoveries: Text embeddings will aid medical research, supporting discoveries and personalized medicine.**
4. **Smart Content Creation: Authors and creators will use embeddings for tailored, engaging content.**
5. **Emotionally Aware AI: AI systems will understand and respond to human emotions in text.**
6. **Educational Tools: Text embeddings will enhance personalized learning and educational content.**
7. **Cultural Preservation: They can preserve endangered languages and cultural heritage.**
8. **Climate Change Analysis: Text embeddings will analyze climate-related data and research papers.**
9. **Disaster Response: They can help emergency responders by processing real-time data during disasters.**
10. **Virtual Reality: In VR and AR applications, embeddings will enhance natural language interactions.**

These advancements signify a future where text embeddings will revolutionize communication, learning, research, and various other domains.

Activity 1:
Word2Vec Implementation

https://colab.research.google.com/drive/1vDpA_d00EHTd_WlfpIR8dwMzrmy2r2ok?usp=sharing

```
!pip install --upgrade gensim
```

This line attempts to upgrade the **gensim** package using the pip package manager.

```
import gensim.downloader as api
```

The line **import gensim.downloader as api** imports the **api** module from the **gensim.downloader** package in Python. This module provides an interface to download pre-trained word vectors and models from the Gensim data repository.

```
info=api.info()
for model_name, model_data in sorted(info['models'].items()):
    print(
        '%s (%d records): %s' % (
            model_name,
            model_data.get('num_records',-1),
            model_data['description'][:40]+ '...',
            )
    )
```

This line calls the **info()** function from the **gensim.downloader** module (aliased as **api**). The **info()** function provides information about the available models and datasets that can be downloaded using Gensim. This information includes details about the models' names, descriptions, and the number of records they contain, among other things.

This line sets up a **for** loop that iterates through the available models and their corresponding data. It uses the **sorted()** function to sort the models in alphabetical order.

- **model_name**: This variable represents the name of the model. In the loop, it will take on each available model's name.
- **model_data**: This variable represents the data associated with the model. This data includes details about the model, such as the number of records and a description.

```
wv=api.load('word2vec-google-news-300')
```

This line downloads and loads the pre-trained Word2Vec model trained on Google News data. The model contains word vectors with a dimensionality of 300. The wv variable now holds the loaded Word2Vec model, allowing you to perform various operations such as finding word similarities or analyzing word relationships.

```
glove=api.load("glove-twitter-50")
```

This line downloads and loads the GloVe (Global Vectors for Word Representation) model trained on Twitter data. The model contains word vectors with a dimensionality of 50. The glove variable now holds the loaded GloVe model, which can be used for tasks like word analogy or word similarity comparisons.

```
fasttext=api.load("fasttext-wiki-news-subwords-300")
```

```
[==================================================] 100.0% 958.5/958.4MB downloaded
```

This line downloads and loads the FastText model trained on Wikipedia data. FastText represents words as bags of character n-grams, allowing it to handle out-of-vocabulary words well. The model contains word vectors with a dimensionality of 300. The fasttext variable now holds the loaded FastText model, enabling you to use it for tasks such as text classification or word similarity calculation

```
wv.most_similar("tea")
```

```
[('Tea', 0.7009038329124451),
 ('teas', 0.6727380156517029),
 ('shape_Angius', 0.632348299026489 3),
 ('activist_Jamie_Radtke', 0.586386024951934 8),
 ('decaffeinated_brew', 0.583953619003295 9),
 ('planter_bungalow', 0.57582902908325 2),
 ('herbal_tea', 0.573117434978485 1),
 ('coffee', 0.563529193401336 7),
 ('jasmine_tea', 0.548339307308197),
 ('Tea_NASDAQ_PEET', 0.540254414081573 5)]
```

The **most_similar** function in Gensim's Word2Vec model allows you to find words that are most similar to a given word according to the trained word embeddings. In this case, you are trying to find words that are most similar to the word "tea" using the **wv** variable, which represents the loaded Word2Vec model.

In Gensim's Word2Vec models, there is no direct **distance** function to compute the cosine distance between two words. However, you can calculate the cosine distance between two word vectors using the following formula:

Cosine Distance=1−Cosine Similarity

```
wv.distance("tea","coffee")
```

```
0.43647080659866333
```

Activity 2:

Custom Word Embedding Training

https://colab.research.google.com/drive/1MDtnbrPxgw28MSd6IA5lduEpb6ee8ax4?usp=sharing

This line attempts to upgrade the **gensim** package using the pip package manager.

```
!pip install --upgrade gensim
```

```
import torch
import torch.nn as nn
import torch.optim as optim
```

These lines import the necessary modules from the PyTorch library, which is used for defining and training neural networks.

- **This line defines a new class called MyEmbeddingLayer that inherits from nn.Module. This class will represent your custom embedding layer.**
- In the constructor (__**init**__) of the **MyEmbeddingLayer** class, this code initializes the embedding layer. **num_embeddings** specifies the size of the vocabulary (number of unique words), and **embedding_dim** specifies the dimensionality of the word embeddings. **nn.Parameter** is used to define a learnable parameter (the embedding matrix) that will be optimized during training.
- **This method defines the forward pass of the embedding layer. Given an input index (input), it returns the corresponding row from the embedding matrix. In other words, it looks up the word embeddings for the given input indices.**

```
class MyEmbeddingLayer(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        super(MyEmbeddingLayer, self).__init__()
        self.embeddings = nn.Parameter(torch.randn(num_embeddings, embedding_dim))

    def forward(self, input):
        return self.embeddings[input]
```

- This line creates a dictionary **word_to_index** where words ('food', 'animal', 'human') are mapped to their respective indices (0, 1, 2).

- These lines define **positive_samples** and **negative_samples** - pairs of word indices representing words that appear near each other and words that do not appear near each other, respectively.

```
# Assume the indices of the words we're interested in are [0, 1, 2]
word_to_index = {'food': 0, 'animal': 1, 'human': 2}

# Assume some training data (pairs of indices of words that appear near each other)
positive_samples = [(0, 1), (1, 2), (2, 0)]  # Example pairs of words that appear near each other
negative_samples = [(0, 2), (1, 0), (2, 1)]  # Example pairs of words that do not appear near each other
```

Your paragraph text

- This line creates an instance of the MyEmbeddingLayer class with a vocabulary size of 10 and embedding dimensionality of 3.

- This line defines the stochastic gradient descent (SGD) optimizer to optimize the parameters (embedding matrix) of the embedding_layer. The learning rate is set to 0.1.

```
# Create the embedding layer
embedding_layer = MyEmbeddingLayer(num_embeddings=10, embedding_dim=3)

# Define the loss function and optimizer
optimizer = optim.SGD(embedding_layer.parameters(), lr=0.1)
```

```python
# Function to compute cosine similarity
def cosine_similarity(a, b):
    return torch.dot(a, b) / (torch.norm(a) * torch.norm(b))

# Compare the words using the trained embeddings
word_embeddings = embedding_layer(torch.tensor([0, 1, 2]))
similarity_food_animal = cosine_similarity(word_embeddings[0], word_embeddings[1])
similarity_food_human = cosine_similarity(word_embeddings[0], word_embeddings[2])
similarity_animal_human = cosine_similarity(word_embeddings[1], word_embeddings[2])

print(f"Similarity between food and animal: {similarity_food_animal.item():.2f}")
print(f"Similarity between food and human: {similarity_food_human.item():.2f}")
print(f"Similarity between animal and human: {similarity_animal_human.item():.2f}")
```

```
Similarity between food and animal: 0.00
Similarity between food and human: 0.11
Similarity between animal and human: -0.01
```

The rest of the code performs training using negative sampling, where it calculates the loss for positive and negative samples, backpropagates the gradients, and updates the embeddings using the optimizer. After training, it computes cosine similarities between specific words using the trained embeddings and prints the results.