Semantic Search Capabilities of LLMs

Bhupen Sinha

25-07-2024

TABLE OF CONTENTS

WHAT IS SEMANTIC SEARCH?

TEXT EMBEDDINGS OVERVIEW

- Represent words or phrases as numerical vectors in a multidimensional space based on contextual meaning.
- Similar phrases are represented by vectors close together; dissimilar phrases by vectors far apart.

Example of Text Embeddings in Action

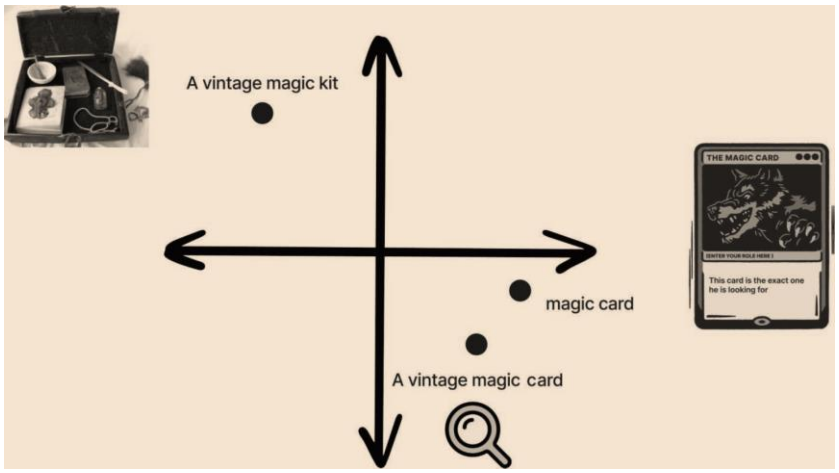
- A user searching for "a vintage magic card" should get relevant results like "magic card" if text embeddings are used effectively.
- The system embeds queries so that similar phrases have similar vector representations.

Representation and Comparison

- Vectors act like a meaningful hash, allowing comparison of text in its encoded state, though vectors cannot be directly reversed to text.

SEMANTIC SEARCH AND LLMS

- Proper semantic search systems use embeddings to differentiate between similar and dissimilar phrases, even if they share keywords.
- LLM-enabled embeddings capture semantic value beyond surface-level syntax or spelling, enabling advanced applications.

Query Example • If a user requests "a vintage magic card," the system should embed this query effectively. **Embedding Results** • Relevant results (e.g., "magic card") should be close to the embedded query. • Non-relevant items (e.g., "a vintage magic kit") should be far from the query, despite shared keywords.	

Semantic Search Capabilities of LLMs

TRADITIONAL SEARCH ENGINES

- Return links to websites or items based on exact matches or permutations of the query words.
- For example, searching “vintage magic the gathering cards” yields items with those words in the title/description.

Limitations of Traditional Search

- May return irrelevant results, such as “vintage magic sets” related to magic tricks rather than trading cards.
- Queries might not align with the exact words used in the desired items, leading to unrelated findings.

SEMANTIC SEARCH SOLUTION

- Addresses issues of mismatched words and differing meanings by understanding context.
- Enhances search results by focusing on the intended meaning.

Asymmetric Semantic Search Overview

- A semantic search system understands the meaning and context of search queries and matches them with the meaning and context of documents in a database.
- It relies on a pre-trained LLM (Large Language Model) to grasp the nuances of both queries and documents, rather than exact keyword or n-gram matching.
- **Definition**: Asymmetric semantic search involves an imbalance between the size of the input query and the size of the documents or information being retrieved.
- **Example**: Matching a short query like “magic the gathering cards” to lengthy paragraphs of item descriptions in a marketplace. The short query has significantly less information compared to the detailed paragraphs.

Key Points

- **Understanding Context**: The system leverages semantic understanding to find relevant results based on context rather than exact word matches.
- **Handling Imbalance**: The search system must effectively compare a short, concise query with more extensive and detailed document descriptions.

ASYMMETRIC SEMANTIC SEARCH SYSTEM

Collect Documents for Embedding

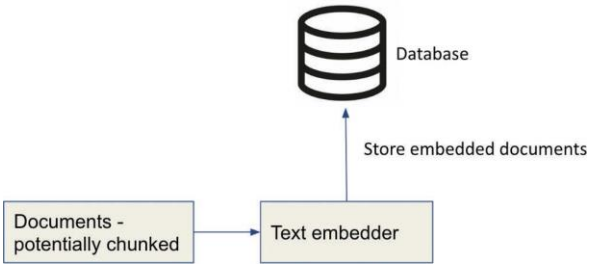
- Gather paragraph descriptions of items or relevant documents that need to be searched.

Create Text Embeddings

- Encode the semantic information of the collected documents into numerical vectors using text embeddings.

Store Embeddings

- Save the generated embeddings in a database to facilitate efficient retrieval when a query is made.



Preprocess and Clean Query

- The user's query is preprocessed and cleaned to ensure it's in a suitable format for searching (e.g., standardizing terms or removing noise).

Retrieve Candidate Documents

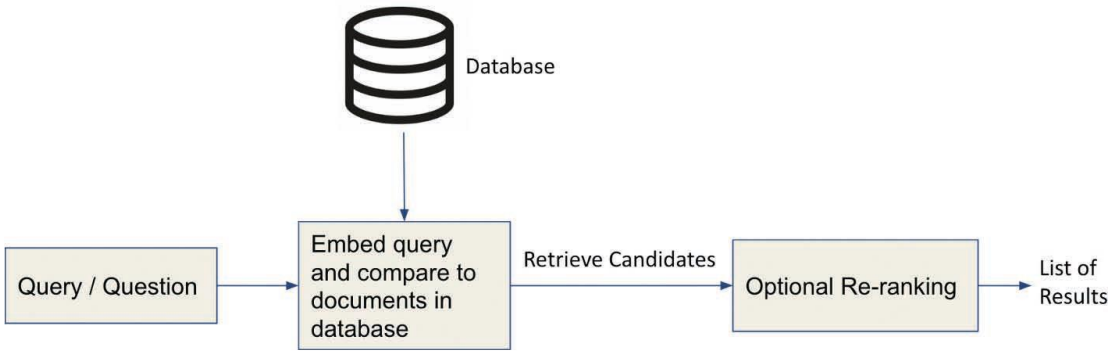
- Use embedding similarity, such as Euclidean distance, to find candidate documents that closely match the query's semantic information.

Re-rank Candidates

- Optionally re-rank the retrieved documents based on additional criteria or relevance metrics (details will be covered later).

Return Final Results

- Present the final, ranked search results to the user.



SEMANTIC SEARCH - THE COMPONENTS

TEXT EMBEDDER

What is it?

- **Function**: Converts text (e.g., documents, words, phrases) into unique vectors that capture the contextual meaning of the text.
- **Importance**: The quality of the vector representation directly impacts the effectiveness of the semantic search system.

Choice of Text Embedder

- **Critical Factor**: The selection of the text embedder affects the quality of the vector representation. Options include both open and closed-source solutions.
- **Example**: For quick setup, OpenAI's closed-source "Embeddings" product is used, known for providing high-quality vectors efficiently.

OpenAI's "Embeddings"

- **Advantages**: Provides high-quality vectors quickly and efficiently.
- **Limitations**: Being a closed-source product, it offers limited control over implementation and potential biases.

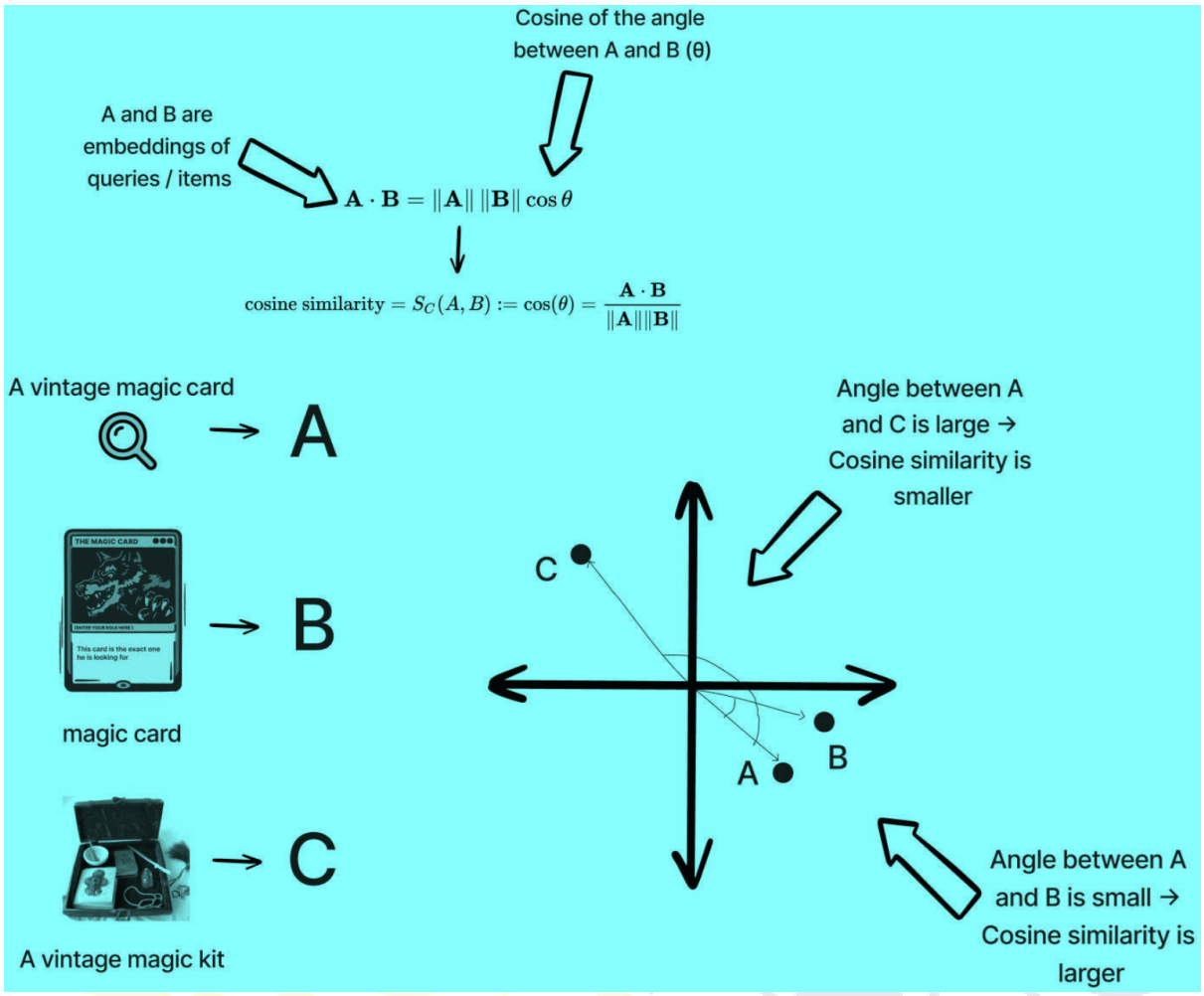
WHAT MAKES PIECES OF TEXT "SIMILAR"

Measuring Text Similarity

- **Objective**: Determine how similar two pieces of text are after converting them into vectors.

Cosine Similarity

- **Definition**: A metric used to measure similarity between two vectors based on their orientation rather than magnitude.
- **Calculation**:
  - **Score of 1**: Vectors point in the same direction (identical context).
  - **Score of 0**: Vectors are perpendicular (no similarity).
  - **Score of -1**: Vectors point in opposite directions (completely dissimilar).
- **Key Aspect**: Only the angle between vectors is considered, not their size.



Define the Vectors: Let's use the following example vectors:

A=[a1,a2,a3,a4]

B=[b1,b2,b3,b4]

Dot Product Formula: The dot product of vectors A and B is calculated as:

A·B = a1·b1 + a2·b2 + a3·b3 + a4·b4

Example Calculation: Let's choose:

A=[1,2,3,4]

B=[5,6,7,8]

Compute the dot product:

A·B=(1·5)+(2·6)+(3·7)+(4·8) = 5 + 12 + 21 + 32 = 70

Magnitude Calculation

1. **Definition**: The magnitude (or norm) of a vector measures its length. For a vector with four dimensions, the magnitude is calculated by taking the square root of the sum of the squares of its components.

2. **Formula**:

Vector A: $[1, 2, 3, 4]$

- Square Each Component:
  - $1^2 = 1$
  - $2^2 = 4$
  - $3^2 = 9$
  - $4^2 = 16$
- Sum the Squares:
  - $1 + 4 + 9 + 16 = 30$
- Take the Square Root:
  - $\sqrt{30} \approx 5.477$

So, the magnitude of Vector A is approximately **5.477**.

Vector B: $[5, 6, 7, 8]$

- Square Each Component:
  - $5^2 = 25$
  - $6^2 = 36$
  - $7^2 = 49$
  - $8^2 = 64$
- Sum the Squares:
  - $25 + 36 + 49 + 64 = 174$
- Take the Square Root:
  - $\sqrt{174} \approx 13.228$

So, the magnitude of Vector B is approximately **13.228**.

Cosine Similarity Formula

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \times \|\mathbf{B}\|}$$

where:

- $\mathbf{A} \cdot \mathbf{B}$ is the dot product of vectors $\mathbf{A}$ and $\mathbf{B}$.
- $\|\mathbf{A}\|$ is the magnitude of vector $\mathbf{A}$.
- $\|\mathbf{B}\|$ is the magnitude of vector $\mathbf{B}$.

Given Values

- **Dot Product**: 70 (from previous calculation)
- **Magnitude of $\mathbf{A}$**: 5.477
- **Magnitude of $\mathbf{B}$**: 13.228

Cosine Similarity Calculation

1. **Calculate Cosine Similarity**:

$$\text{Cosine Similarity} = \frac{70}{5.477 \times 13.228}$$

2. **Multiply the Magnitudes**:

$$5.477 \times 13.228 \approx 72.49$$

3. **Divide the Dot Product by the Product of the Magnitudes**:

$$\text{Cosine Similarity} = \frac{70}{72.49} \approx 0.965$$

Result

The cosine similarity between vectors A and B is approximately **0.965**. This value indicates that the vectors are quite similar and point in a similar direction in the vector space.

Semantic Search Capabilities of LLMs

Special Property of OpenAI Embeddings: OpenAI embeddings have vectors normalized to length 1.

- **Cosine Similarity and Dot Product**: With normalized vectors, cosine similarity is equivalent to the dot product.
- **Cosine Similarity and Euclidean Distance**: Both cosine similarity and Euclidean distance produce identical rankings.

Advantages of Normalized Vectors:

- Using normalized vectors allows us to efficiently calculate cosine similarity to measure semantic closeness between phrases.

[DEMO : OpenAI Embeddings]

OPEN-SOURCE EMBEDDING ALTERNATIVES

(1) **Sentence Transformers** work by leveraging pre-trained transformer models, such as BERT, RoBERTa, or others, to encode sentences into high-dimensional vectors (embeddings) that capture their semantic meaning. These embeddings can then be used for various natural language processing (NLP) tasks such as semantic search, clustering, paraphrase identification, and more. Here's a detailed breakdown of how Sentence Transformers work:

Key Concepts

1. **Transformer Models**: Sentence Transformers use models based on the transformer architecture, which is highly effective at understanding the context and meaning of words in a sentence. Popular transformer models include BERT, RoBERTa, DistilBERT, and others.
2. **Sentence Embeddings**: The core idea is to convert sentences into fixed-size dense vectors that capture their semantic information. These embeddings can then be compared using similarity measures like cosine similarity.
3. **Pre-trained Models**: The library provides various pre-trained models that are fine-tuned on specific tasks or datasets to enhance their performance for particular applications.

How It Works

1. **Encoding Sentences**:
    - A Sentence Transformer model takes a sentence as input and processes it through multiple layers of transformers.
    - Each transformer layer consists of self-attention mechanisms and feed-forward neural networks that allow the model to understand the context and relationships between words in the sentence.

2. **Output Embeddings**:
    - After passing through the transformer layers, the model outputs a high-dimensional vector (embedding) for each token in the sentence.
    - These token embeddings are then aggregated (usually by taking the mean or using the embedding of the special [CLS] token) to form a single fixed-size vector representing the entire sentence.
3. **Similarity Calculation**:
    - Once sentences are encoded into embeddings, their similarity can be measured using metrics like cosine similarity.
    - This allows for various downstream tasks, such as finding similar sentences, clustering, and semantic search.

Training and Fine-Tuning

- **Training Data**: Sentence Transformers can be trained or fine-tuned on various datasets depending on the task, such as semantic textual similarity (STS), natural language inference (NLI), or custom datasets.
- **Loss Functions**: Common loss functions used include contrastive loss, triplet loss, and multiple negative ranking loss, which help the model learn to distinguish between similar and dissimilar sentences.

Applications

1. **Semantic Search**: Finding documents or sentences that are semantically similar to a given query.
2. **Clustering**: Grouping similar sentences or documents together.
3. **Paraphrase Identification**: Detecting sentences with similar meanings.
4. **Summarization**: Identifying key sentences in a document.
5. **Question Answering**: Enhancing the understanding of questions and finding relevant answers.

[DEMO : sentence_transformers – simple embeddings]

(2) **Cross Encoders** are a type of model used in natural language processing (NLP) that focus on computing similarity or relevance scores between **pairs of texts** by jointly processing them through a transformer model.

Key Concepts

1. **Joint Processing**: Cross Encoders take two texts as input and process them together through a transformer model. This allows the model to capture fine-grained interactions between the two texts, leading to more accurate similarity or relevance scores.
2. **Transformer Architecture**: Like Sentence Transformers, Cross Encoders use pre-trained transformer models such as BERT, RoBERTa, or others, but they leverage the full power of these models by considering both texts simultaneously.

How It Works

1. **Input Pairing**:
    - Cross Encoders receive a pair of texts (e.g., a query and a candidate sentence) as input.
    - The texts are concatenated and passed as a single input to the transformer model, typically separated by a special token (like [SEP] in BERT).
2. **Embedding and Interaction**:
    - The transformer model processes the concatenated input, allowing each token in both texts to attend to every other token.
    - This joint attention mechanism enables the model to capture rich interactions and dependencies between the two texts.
3. **Output Score**:
    - The output from the transformer is usually pooled or reduced to a single vector (often using the embedding of the [CLS] token).
    - A classification layer (usually a simple feed-forward neural network) is then applied to this vector to produce a similarity or relevance score.

Advantages

1. **Higher Accuracy**:
    - Because Cross Encoders evaluate the interaction between texts directly, they often achieve higher accuracy on tasks that require fine-grained understanding, such as semantic similarity, paraphrase identification, and ranking.
2. **Effective for Ranking Tasks**:
    - Cross Encoders are particularly effective for tasks where precise ranking of pairs is crucial, such as re-ranking search results or evaluating candidate responses in conversational AI.

Limitations

1. **Computational Cost**:
    - Cross Encoders are computationally expensive because each pair of texts must be processed together. This makes them less suitable for large-scale retrieval tasks where many comparisons are required.
2. **Slower Inference**:
    - The need to process each text pair individually results in slower inference times compared to bi-encoders, which can pre-compute embeddings and compare them efficiently.

Practical Use Case - Re-ranking

Due to their computational cost, Cross Encoders are often used to re-rank the top-k results from a faster, less accurate model like a bi-encoder. The typical workflow involves:

1. **Initial Retrieval with Bi-encoder**: Use a bi-encoder to quickly retrieve a shortlist of candidate texts.
2. **Re-ranking with Cross Encoder**: Apply the Cross Encoder to the shortlisted candidates to get precise similarity scores and re-rank them accordingly.

BI-ENCODER AND CROSS ENCODER

Aspect	Bi-Encoders	Cross Encoders
Encoding Method	Encode each text independently into fixed-size vectors (embeddings)	Encode text pairs jointly in a single forward pass through the model
Similarity Computation	Compute similarity between embeddings using metrics like cosine similarity	Directly output similarity or relevance scores for text pairs
Efficiency	High efficiency and scalability for large-scale retrieval tasks	Computationally expensive, less scalable for large-scale retrieval
Contextual Interaction	Limited contextual interaction between texts	Captures fine-grained interactions and dependencies between texts
Accuracy	May miss subtle nuances and interactions	Higher accuracy for tasks requiring detailed understanding
Inference Speed	Fast inference, suitable for quick retrieval	Slower inference due to processing each pair individually
Typical Use Cases	Semantic search, large-scale retrieval, clustering	Re-ranking, precise similarity scoring, contextual tasks
Example Usage	bi_encoder = SentenceTransformer('all-MiniLM-L6-v2')	cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-12-v2')
	embeddings = bi_encoder.encode(sentences)	similarity_scores = cross_encoder.predict(sentence_pairs)
	similarities = util.pytorch_cos_sim(embeddings, embeddings)	

Aspect	Cosine Similarity (Bi-Encoders)	Similarity Scores (Cross Encoders)
Calculation Method	Computes the cosine of the angle between two vectors	Directly output by the model based on joint text processing
Input	Independent embeddings of two texts	Pairs of texts processed jointly through the model
Typical Use Cases	Large-scale retrieval, semantic search, clustering	Re-ranking, precise similarity scoring, contextual tasks requiring detailed comparison

[DEMO : sentence_transformers – cross encoders]

SIMILARITY SCORES IN CROSS ENCODERS

Text Pair Example:

- Sentence 1: "The cat sits on the mat."
- Sentence 2: "A cat is sitting on a mat."

Steps in Cross Encoder Similarity Score Calculation

1. **Input Preparation**
    - The two sentences are combined into a single input sequence with a special separator token [SEP].
    - In BERT-like models, the input sequence would look like: [CLS] The cat sits on the mat. [SEP] A cat is sitting on a mat. [SEP]
2. **Tokenization**
    - The combined sequence is tokenized. Each word or sub-word is converted into its corresponding token ID.
    - Example tokens: [CLS] the cat sits on the mat [SEP] a cat is sitting on a mat [SEP]
3. **Embedding Generation**
    - The token IDs are converted into embeddings using the embedding layer of the transformer model.
    - Positional embeddings are added to account for the position of each token in the sequence.
4. **Transformer Encoding**
    - The entire token sequence, including both sentences, is fed into the transformer model.
    - The transformer model processes the sequence through multiple layers of self-attention and feed-forward neural networks, capturing interactions between tokens across both sentences.
5. **[CLS] Token Representation**:
    - The final hidden state corresponding to the [CLS] token is taken as the aggregate representation of the entire input sequence. This hidden state now contains information about the relationship between the two sentences.
6. **Similarity Score Computation**:
    - A classification or regression head (often a simple feed-forward neural network) is applied to the [CLS] token representation to produce the similarity score.

- This head is trained to output a single scalar value that represents the similarity or relevance between the two sentences.

FEED-FORWARD NEURAL NETWORK (HEAD)

During the training of a Cross Encoder, the feed-forward neural network (classification or regression head) learns to map the [CLS] token representation (input) to a similarity or relevance score (output). Let's break down the training process in detail:

Inputs and Outputs During Training

1. **X (Input)**:
    - **[CLS] Token Representation**: The input to the head during training is the final hidden state of the [CLS] token from the transformer model.
    - This hidden state is a high-dimensional vector that summarizes the combined input sequence of the text pair.
2. **y (Output)**:
    - **Similarity or Relevance Score**: The output during training is the actual similarity or relevance score for the text pair.
    - These scores are typically provided in a Labeled training dataset.

Training Process

1. **Prepare Training Data**:
    - Collect or create a labeled dataset of text pairs along with their corresponding similarity or relevance scores.
    - Example dataset entry: ("The cat sits on the mat.", "A cat is sitting on a mat.", 0.9)
2. **Forward Pass Through Transformer**:
    - Each text pair is tokenized, embedded, and processed jointly by the transformer model.
    - The final hidden state of the [CLS] token is extracted as the input to the head.
3. **Forward Pass Through Head**:
    - The [CLS] token's hidden state is passed through the feed-forward neural network (head) to produce a predicted similarity score.
4. **Loss Calculation**:
    - Compute the loss between the predicted similarity score and the actual similarity score from the dataset.
    - Example loss function: Mean Squared Error (MSE) for regression tasks or Binary Cross-Entropy (BCE) for binary classification tasks.

5. **Backward Pass and Optimization**:
    - Perform backpropagation to compute gradients.
    - Update the model parameters using an optimization algorithm (e.g., Adam)

Illustrative Example

Let's go through a simplified example step-by-step:

1. **Training Data**:
    - Example entry: ("The cat sits on the mat.", "A cat is sitting on a mat.", 0.9)
    - 0.9 is the actual similarity score (y).
2. **Forward Pass**:
    - Input sequence: [CLS] The cat sits on the mat. [SEP] A cat is sitting on a mat. [SEP]
    - Transformer model processes the sequence.
    - Extract the [CLS] token's hidden state (input X).
3. **Feed-Forward Head**:
    - Pass the [CLS] token's hidden state through the feed-forward neural network.
    - Obtain the predicted similarity score.
4. **Loss Calculation**:
    - Compare the predicted score with the actual score (0.9) using a loss function.
5. **Backward Pass and Optimization**:
    - Update the model parameters to minimize the loss.

Summary

- **X (Input)**: The final hidden state of the [CLS] token from the transformer model, representing the combined input sequence of the text pair.
- **y (Output)**: The actual similarity or relevance score from the labeled dataset.

WHO DECIDES THE SIMILARITY SCORE?

1. **Human Annotators**:
    - **Manual Annotation**: Human annotators review pairs of texts and assign similarity or relevance scores based on predefined criteria.
    - **Guidelines**: Annotators follow guidelines to ensure consistency and objectivity in scoring. For example, a score of 1 might indicate that the texts are nearly identical in meaning, while a score of 0 might indicate no similarity.
2. **Automated Systems**:
    - **Heuristic Methods**: Automated systems can use heuristics to assign similarity scores. For instance, cosine similarity between TF-IDF vectors or other traditional NLP methods can provide an initial scoring.
    - **Pre-trained Models**: Existing models trained on similar tasks can be used to generate scores, which can then be refined through further training or human review.

Best Practices for Assigning and Using Similarity Scores

1. **Clear Annotation Guidelines**:
    - Provide detailed instructions and examples to human annotators to ensure they understand how to rate the similarity or relevance of text pairs.
    - Use a well-defined scale (e.g., 0 to 1, 1 to 5) and describe what each point on the scale represents.
2. **Consistency and Quality Control**:
    - Ensure consistency among annotators by conducting training sessions and providing feedback.
    - Implement quality control measures, such as having multiple annotators rate the same pairs and using inter-annotator agreement metrics to assess consistency.
3. **Balanced Dataset**:
    - Create a balanced dataset that includes a wide range of similarity scores. This helps the model learn to distinguish between different levels of similarity.
    - Ensure diversity in the types of text pairs to cover various scenarios the model might encounter in real applications.
4. **Use of Pre-trained Models**:
    - Leverage pre-trained models as a starting point. Fine-tune them on your specific dataset to adapt them to your particular domain or task.
    - Pre-trained models like BERT, RoBERTa, or models from the sentence-transformers library can provide robust embeddings that are fine-tuned for specific similarity scoring tasks.

5. **Evaluation and Validation**:
    - Split your data into training, validation, and test sets to ensure robust evaluation.
    - Use metrics like Mean Squared Error (MSE) for regression tasks, or precision, recall, and F1-score for classification tasks, to evaluate model performance.
6. **Iterative Improvement**:
    - Continuously improve the quality of your annotations and dataset.
    - Iteratively refine your model by retraining with updated data and incorporating feedback from real-world use.

DOCUMENT CHUNKING

The maximum token limit in a large language model (LLM) refers to the maximum number of tokens (words or subwords) that the model can process in a single input. Exceeding this limit requires techniques like document chunking to manage longer texts.

Token Limits for Popular LLMs

1. **GPT-3**:
    - Maximum token limit: 4096 tokens
2. **GPT-4**:
    - Maximum token limit: 8192 tokens (for standard models)
    - There are extended models with token limits up to 32,768 tokens for specific use cases.
3. **BERT (Bidirectional Encoder Representations from Transformers)**:
    - Maximum token limit: 512 tokens
4. **T5 (Text-to-Text Transfer Transformer)**:
    - Maximum token limit: 512 tokens
5. **RoBERTa (A Robustly Optimized BERT Pretraining Approach)**:
    - Maximum token limit: 512 tokens

Importance of Token Limits

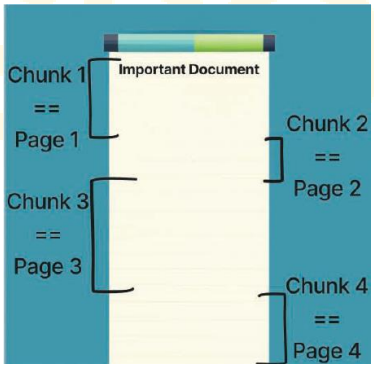
1. **Model Constraints**: Token limits are defined by the architecture and design of the model, ensuring efficient processing and memory management.
2. **Performance**: Keeping within token limits ensures optimal performance and accuracy of the model's predictions.
3. **Context Preservation**: For tasks requiring long context, staying within token limits while preserving context is crucial for maintaining the quality of the output.

Why Document Chunking is Important

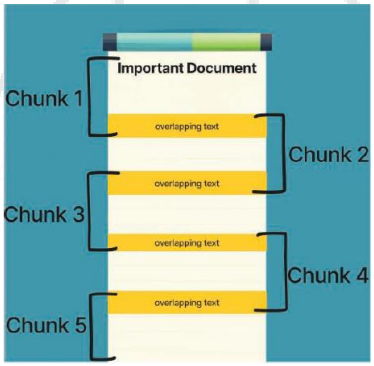
1. **Token Limitations**: Most LLMs, such as GPT-4, have a maximum token limit (e.g., 4096 tokens). Documents longer than this limit need to be split to be fully processed.
2. **Memory Management**: Handling smaller chunks helps in managing memory efficiently, avoiding the risk of running out of memory when processing large documents.
3. **Accuracy**: By chunking documents, the model can maintain high accuracy by focusing on smaller, contextually coherent sections rather than being overwhelmed by a long text.

Methods of Document Chunking

1. **Fixed-Length Chunking**: Splitting the document into chunks of a fixed number of tokens.
    1. **Chunk 1**: "This is a long document that needs to be chunked."
    2. **Chunk 2**: "It contains multiple paragraphs and sections."
    3. **Chunk 3**: "The goal is to split this document into smaller, manageable parts without losing the context."
2. **Overlap Chunking**: Overlapping chunks to ensure that context is preserved across chunk boundaries.
    1. **Chunk 1**: "This is a long document that needs to be chunked. It contains multiple paragraphs."
    2. **Chunk 2**: "It contains multiple paragraphs and sections. The goal is to split this document."
    3. **Chunk 3**: "The goal is to split this document into smaller, manageable parts without losing the context."



Max token with no overlap



Max token with overlap

[DEMO : Chunking – simple python example]

Best Practices

- **Contextual Overlap**: If maintaining context between chunks is critical, you can implement overlapping chunking, where consecutive chunks share some common tokens.
- **Adjust Max Tokens**: Make sure that the sum of input tokens and expected output tokens does not exceed the model's token limit.
- **API Usage Limits**: Be aware of your API usage limits, as chunking and sending multiple requests can quickly accumulate API calls.
- **Batch Processing**: If you have many chunks, consider processing them in batches to manage API rate limits.

Type of Chunking	Description	Pros	Cons
Max token window chunking with no overlap	The document is split into fixed-size windows, with each window representing a separate document chunk.	Simple and easy to implement.	May cut off context in between chunks, resulting in loss of information.
Max token window chunking with overlap	The document is split into fixed-size overlapping windows.	Simple and easy to implement.	May result in redundant information across different chunks.
Chunking on natural delimiters	Natural whitespace in the document is used to determine the boundaries of each chunk.	Can result in more meaningful chunks that correspond to natural breaks in the document.	May be time-consuming to find the right delimiters.
Clustering to create semantic documents	Similar document chunks are combined to form larger semantic documents.	Can create more meaningful documents that capture the overall meaning of the document.	Requires more computational resources and may be more complex to implement.
Use entire documents without chunking	The entire document is treated as a single chunk.	Simple and easy to implement.	May suffer from a context window for embedding, resulting in extraneous context that affects the quality of the embedding.

VECTOR DATABASES OVERVIEW

A **vector database** is a specialized data storage system designed to efficiently store and retrieve vectors, typically high-dimensional vectors used in machine learning and AI. Vectors in this context often represent embeddings, which are numerical representations of data (like text, images, or audio) that capture semantic meaning.

Purpose of Vector Databases

- **Storage of Embeddings**: Vector databases are optimized to store embeddings generated by models, such as those from large language models (LLMs). These embeddings encapsulate the semantic meaning of data, enabling advanced operations like similarity search.
- **Efficient Retrieval**: They are particularly useful for performing fast and accurate nearest-neighbor searches. This capability allows you to quickly find data points that are similar to a query vector, which is crucial for tasks like semantic search, recommendation systems, and clustering.

Key Features of Vector Databases

1. **High-Dimensional Vector Storage**:
    - Vector databases are designed to handle and efficiently store large amounts of high-dimensional data. This makes them ideal for applications like NLP, computer vision, and recommendation engines where embeddings are commonly used.
2. **Nearest-Neighbor Search**:
    - The primary operation supported by vector databases is nearest-neighbor search. Given a query vector, the database can quickly retrieve vectors (and their associated data) that are closest in terms of distance metrics like cosine similarity or Euclidean distance.
3. **Scalability**:
    - Vector databases are built to scale with the amount of data, allowing you to handle millions or even billions of vectors. This is important as modern machine learning models often produce vast quantities of embeddings.
4. **Indexing Techniques**:
    - Vector databases employ advanced indexing techniques, such as HNSW (Hierarchical Navigable Small World), IVF (Inverted File Index), or PQ (Product Quantization), to enable fast searches even in large datasets.
5. **Integration with Machine Learning Pipelines**:
    - These databases are designed to integrate seamlessly with machine learning pipelines, allowing for the continuous updating and querying of vectors as models evolve or new data becomes available.

Use Cases

- **Semantic Search**: Retrieving documents, images, or other data that are semantically similar to a given query. For example, in a text-based search engine, a vector database can help find articles that are contextually related to the user's query.
- **Recommendation Systems**: Suggesting similar items to users based on their preferences by finding items whose embeddings are close to those of items the user likes.
- **Anomaly Detection**: Identifying outliers or anomalies by finding vectors that are significantly distant from the majority of the data points.
- **Clustering**: Grouping similar data points together based on their embeddings, which can be useful for categorization or understanding data structure.

Why Use a Vector Database?

1. **Performance**: Vector databases are optimized for the type of queries common in machine learning tasks, providing much faster search and retrieval times compared to traditional databases.
2. **Precision**: They are designed to maintain high precision in nearest-neighbor searches, which is crucial when working with embeddings that represent subtle nuances in data.
3. **Scalability**: As datasets grow, vector databases provide the necessary infrastructure to handle large-scale data efficiently without compromising on speed or accuracy.

Some popular vector databases

Here are some popular vector databases used for storing, indexing, and searching high-dimensional vectors, which are commonly used in machine learning and AI applications like similarity search, recommendation systems, and NLP tasks:

1. Pinecone

- **Description**: Pinecone is a fully managed vector database that is designed for high-performance, scalable, and real-time similarity search. It handles the complexities of vector indexing, storage, and search, allowing developers to focus on building applications.
- **Key Features**: Real-time indexing, low-latency queries, high scalability, easy integration with machine learning models.

2. Milvus

- **Description**: Milvus is an open-source vector database built for AI applications, enabling efficient similarity search on massive datasets. It supports various machine learning and deep learning models.
- **Key Features**: High scalability, support for multiple indexing methods (e.g., IVF, HNSW), distributed architecture, easy integration with popular AI frameworks.

3. Weaviate

- **Description**: Weaviate is an open-source vector search engine that uses machine learning models to create vectors from data and perform semantic search. It is schema-less and supports various data types, including text, images, and audio.
- **Key Features**: Flexible data models, semantic search, context-aware search, RESTful API, built-in machine learning capabilities.

4. Vespa

- **Description**: Vespa is an open-source search engine and vector database developed by Yahoo. It is designed for large-scale machine learning and search applications, offering both structured and unstructured data processing.
- **Key Features**: Real-time data indexing, distributed architecture, support for large-scale ML models, hybrid search combining vectors and traditional text.

5. FAISS (Facebook AI Similarity Search)

- **Description**: FAISS is a library developed by Facebook AI Research that provides efficient similarity search and clustering of dense vectors. It is particularly well-suited for large-scale datasets.
- **Key Features**: High-performance indexing methods (e.g., IVF, PQ, HNSW), multi-GPU support, customizable, extensive documentation.

6. Annoy (Approximate Nearest Neighbors Oh Yeah)

- **Description**: Annoy is an open-source library developed by Spotify for efficient nearest neighbor search. It is designed for large-scale, high-dimensional datasets, particularly in recommendation systems.
- **Key Features**: Fast indexing and query time, low memory usage, support for disk-based indexes, easy to use with Python.

7. ElasticSearch with KNN plugin

- **Description**: ElasticSearch is a popular distributed search engine that can also perform vector similarity search using the K-Nearest Neighbors (KNN) plugin. It combines traditional keyword search with vector search.
- **Key Features**: Supports hybrid search (text and vectors), scalable, integration with the Elastic stack, large community and ecosystem.

8. Qdrant

- **Description**: Qdrant is an open-source vector similarity search engine and database, optimized for the performance of machine learning models. It provides RESTful APIs for easy integration.
- **Key Features**: High performance, real-time updates, cloud-native, support for various deployment options (e.g., Docker, Kubernetes).

9. Zilliz Cloud

- **Description**: Zilliz Cloud is a managed service based on Milvus, offering vector database capabilities with added scalability and management features. It is suitable for enterprises that need a robust vector search solution without managing the infrastructure.
- **Key Features**: Managed service, high scalability, integration with various AI frameworks, advanced search capabilities.

[DEMO : show slides on Vector databases]

RE-RANKING THE RETRIEVED RESULTS

Re-ranking Results, in Large Language Models (LLMs)

1. Description

Re-ranking in the context of LLMs refers to the process of refining the order of search or retrieval results to ensure that the most relevant and contextually appropriate results are presented to the user. This is typically done after an initial set of results has been retrieved, often using a vector database or a similarity search method. The goal of re-ranking is to enhance the quality of the final output by considering more complex and nuanced aspects of the query and the results.

2. Why Re-ranking is Needed

- **Enhanced Relevance**: Initial retrieval methods, like those based on vector similarities (e.g., cosine similarity), primarily capture the broad semantic relationships between the query and the results. However, they might not fully account for the entire context of the query or the subtle differences between potential results. Re-ranking helps prioritize the most contextually relevant results.
- **Improved User Experience**: By re-ranking the results, users are more likely to receive answers or documents that are more aligned with their intent, thereby improving satisfaction and the overall effectiveness of the search system.
- **Addressing Limitations of Initial Retrieval**: Methods like vector search may return results that are semantically similar but not necessarily the most useful or informative. Re-ranking can address these limitations by employing models or techniques that consider additional factors like contextual fit, term proximity, or relevance scores.

3. Common Approaches and Tools

a) Cross-Encoder Re-ranking

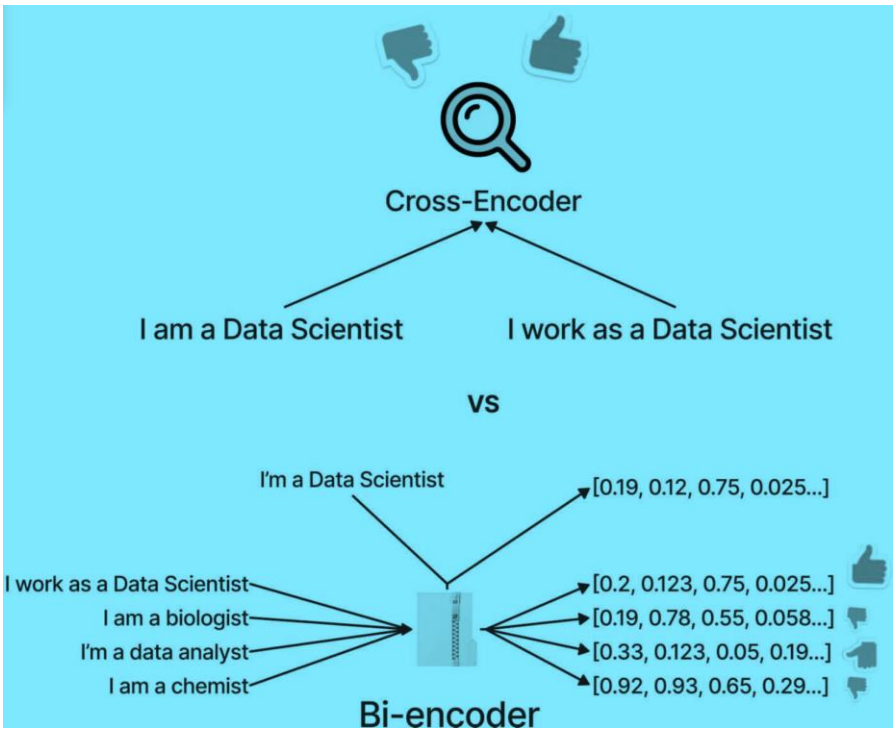
- **Description**: A cross-encoder is a type of Transformer model that evaluates the relevance of a pair of sequences (such as a query and a document) by jointly encoding them and then predicting a relevance score. This score is used to re-rank the initial set of results.
- **Why It's Used**: Cross-encoders take into account the entire context of the query and the candidate result, allowing for more precise relevance judgments than methods that consider the query and results independently.
- **Tool**:
    - **Sentence Transformers Library**: This library provides pre-trained cross-encoder models that can be used for re-ranking tasks. It also allows fine-tuning of these models on task-specific datasets to further improve performance.
    - **Benefits**: Better context understanding, leading to more accurate relevance predictions.
    - **Drawbacks**: Increased computational overhead and latency due to the complexity of jointly encoding the sequences.

b) Traditional Retrieval Models (e.g., BM25)

- **Description**: BM25 is a traditional information retrieval model that ranks documents based on the frequency of query terms within them, taking into account term proximity and inverse document frequency (IDF). It provides a statistical approach to scoring the relevance of a document to a query.
- **Why It's Used**: BM25 is effective for scenarios where term frequency and proximity are strong indicators of relevance. It's a robust and efficient method, especially when combined with more modern techniques.
- **Tool**:
    - **BM25 Implementations**: Available in libraries like ElasticSearch, Whoosh, or Pyserini.
    - **Benefits**: Fast and computationally efficient, especially for large datasets. It doesn't require significant additional computational resources beyond the initial retrieval.
    - **Drawbacks**: BM25 doesn't fully account for the semantic context of the query and results, making it less effective for complex queries that require nuanced understanding.

c) Hybrid Approaches

- **Description**: Combining modern methods like cross-encoders with traditional methods like BM25 can create a hybrid re-ranking approach. For example, BM25 might be used for an initial ranking, and a cross-encoder can then re-rank the top results.
- **Why It's Used**: Hybrid approaches leverage the strengths of multiple methods, improving accuracy without fully sacrificing performance or adding excessive computational costs.
- **Tools**:
    - **ElasticSearch + Cross-Encoder**: ElasticSearch can handle the initial retrieval and BM25 scoring, while a cross-encoder model (from Sentence Transformers) can fine-tune the ranking.
    - **Benefits**: Balances efficiency and effectiveness, potentially yielding superior results compared to using a single method.
    - **Drawbacks**: Can be complex to implement and optimize, requiring careful tuning of both components.



USEFUL LINKS AND REFERENCES

-

-

INDEX

No index entries found.