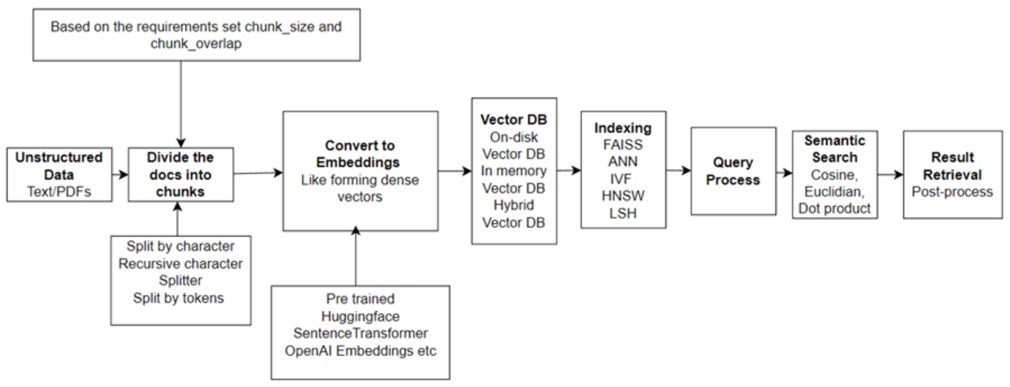
Puneet Hegde
[LinkedIn]
[GitHub]

CUSTOM CHATBOT -- $0 embeddings, open source (NO OPENAI API)

Vector DB Architecture:


Based on the requirements set chunk_size and chunk_overlap
Unstructured Data Text/PDFs
Divide the docs into chunks
Split by character Recursive character Splitter Split by tokens
Convert to Embeddings Like forming dense vectors
Pre trained Huggingface SentenceTransformer OpenAI Embeddings etc
Vector DB On-disk Vector DB In memory Vector DB Hybrid Vector DB
Indexing FAISS ANN IVF HNSW LSH
Query Process
Semantic Search Cosine, Euclidian, Dot product
Result Retrieval Post-process

Unstructured data is fed into langchain library which will divide the whole document into smaller chunks based on value we have given.

1. **Divide into smaller chunks**:

- Split by character.
- Recursive character Splitter (Efficient one)
- Split by tokens.

- **Split by character:** This method is the simplest to implement, but it is also the least efficient. It requires looping through the entire string and checking each character. This can be slow, especially for long strings.
- **Recursive character splitter:** This method is more efficient than split by character, but it is still not as efficient as split by tokens. It uses recursion to split the string into smaller and smaller pieces until it is split into individual characters. This can be faster than looping through the entire string, but it can still be slow for long strings.
- **Split by tokens:** This method is the most efficient way to split a string. It uses a regular expression to split the string into tokens. This can be very fast, even for long strings.

Splitting method	Efficiency	Complexity	Use cases
Split by character	Least efficient	Simple	Simple strings
Recursive character splitter	More efficient	Medium	Complex strings
Split by tokens	Most efficient	Complex	Strings with regular expressions

Here are some additional considerations when choosing a splitting method:

- **The length of the string:** If the string is very long, then split by tokens will be the most efficient method.
- **The complexity of the string:** If the string contains regular expressions, then split by tokens will be the most efficient method.
- **The specific use case:** If there is a need to split the string into individual characters, then split by character is the best option.

3. **Convert to Embeddings** like forming dense vectors: We can use Pre trained Huggingface SentenceTransformer, OpenAI Embeddings etc.

Text embedding models | 🦜🔗 Langchain

4. **Vector Database:** On-disk, In memory & Hybrid VDBs

In memory VDB is very fast and efficient as it stores whole data in the RAM.

Pipeline:

- **Indexing**: The vector database indexes vectors using an algorithm such as LSH, or HNSW (more on these below). This step maps the vectors to a data structure that will enable faster searching.
- **Querying**: The vector database compares the indexed query vector to the indexed vectors in the dataset to find the nearest neighbors (applying a similarity metric used by that index)
- **Post Processing**: In some cases, the vector database retrieves the final nearest neighbors from the dataset and post-processes them to return the final results. This step can include re-ranking the nearest neighbors using a different similarity measure.

For example in Chroma DB, SQLite is used to store the metadata for the database. This includes information such as the schema, the indexes, and the data types of the columns. SQLite is also used to store the configuration settings for the database.

Here are some of the benefits of using SQLite in ChromaDB:

- **Lightweight:** SQLite is a very lightweight database, which means that it takes up very little space on disk. This is important for applications that are running on devices with limited resources.
- **Embedded:** SQLite is an embedded database, which means that it can be embedded in the application code. This makes it easy to use SQLite with other libraries and frameworks.
- **Reliable:** SQLite is a very reliable database, and it has been used in a variety of applications. This makes it a good choice for applications that need a reliable database.

Overall, SQLite is a good choice for storing the metadata for Chroma DB. It is lightweight, embedded, and reliable.
Reasons:

- **To store the schema:** The schema of a database describes the structure of the database. This includes information about the data types of the columns, the relationships between the columns, and the constraints on the data. SQLite is a good choice for storing the schema because it is a lightweight and reliable database.
- **To store the indexes:** Indexes are used to speed up queries on the database. SQLite is a good choice for storing indexes because it is a fast and efficient database.
- **To store the configuration settings:** The configuration settings of a database control how the database works. SQLite is a good choice for storing the configuration settings because it is a simple and easy-to-use database.

Chroma DB stores indexes in SQLite using a B-tree data structure. A B-tree is a self-balancing tree data structure that is used to store sorted data. B-trees are very efficient for storing and retrieving data, and they are a good choice for storing indexes.

The indexes in ChromaDB are stored in a separate table in the SQLite database. This table contains the following columns:

- **Column:** The name of the column that the index is for.
- **Type:** The type of the column that the index is for.
- **Unique:** Whether the index is a unique index.
- **Values:** The values that are stored in the index.

The values in the index are stored in sorted order. This allows the index to be used to quickly find the rows in the database that match a given value.

The indexes in ChromaDB are used to speed up queries on the database. When a query is executed, the indexes are used to find the rows in the database that match the query criteria. This can significantly improve the performance of the query.

4. **Indexing:** Faiss (Facebook AI Similarity Search) is a library for efficient similarity search in vector spaces. It provides a variety of indexing algorithms, including LSH, IVF, and HNSW. Faiss is designed to be highly efficient, and it can be used to index large datasets.

   In vector database similarity search, Faiss can be used to create an index that maps each vector to a set of similar vectors. This index can then be used to efficiently find similar vectors.

- **LSH:** LSH (Locality-Sensitive Hashing) is a family of indexing algorithms that are based on the idea of hashing. LSH algorithms are very efficient, and they can be used to index large datasets.
- **IVF:** IVF (Inverted File Vectors) is a variant of LSH that is specifically designed for vector indexing. IVF algorithms are even more efficient than LSH algorithms, and they can be used to index very large datasets.
- **HNSW:** HNSW (Hierarchical Navigable Small World) is a graph-based indexing algorithm that is designed for efficient search queries. HNSW algorithms are very accurate, and they can be used to find similar vectors even if they are not very close together.

5. **Query process:** Convert the query into embeddings by the same technique used to convert the input data and do semantic search.

6. **Semantic Search:**
   Cosine similarity, Euclidean distance, and dot product are all similarity measures that can be used for semantic text retrieval. However, they have different strengths and weaknesses, so the best choice for a particular application will depend on the specific requirements.
   - **Cosine similarity:** Cosine similarity is a measure of the similarity between two vectors. It is calculated by taking the dot product of the two vectors and dividing by the product of their norms. Cosine similarity is a good choice for semantic text retrieval because it is insensitive to the order of the terms in the vectors. This means that it can be used to compare documents that are written in different ways, but that have the same meaning.
   - **Euclidean distance:** Euclidean distance is a measure of the distance between two points in a Euclidean space. It is calculated by taking the square root of the sum of the squared differences between the corresponding elements of the two vectors. Euclidean distance is a good choice for semantic text retrieval when the documents are similar in length. This is because Euclidean distance is sensitive to the magnitude of the terms in the vectors.
   - **Dot product:** The dot product is a measure of the similarity between two vectors. It is calculated by taking the product of the corresponding elements of the two vectors. The dot product is a good choice for semantic text retrieval when the documents are similar in meaning, but not necessarily in length. This is because the dot product is not sensitive to the magnitude of the terms in the vectors.

   In general, cosine similarity is a good choice for semantic text retrieval because it is insensitive to the order of the terms in the vectors. Euclidean distance is a good choice for semantic text retrieval when the documents are similar in length. The dot product is a good choice for semantic text retrieval when the documents are similar in meaning, but not necessarily in length.

Cosine similarity, Euclidean distance, and dot product are all measures of similarity between two vectors. They are all calculated differently, and they have different strengths and weaknesses.

- **Cosine similarity** measures the angle between two vectors. It is a measure of how similar the directions of the vectors are, regardless of their magnitudes. Cosine similarity is a good measure of similarity when the magnitudes of the vectors are not important. For example, it can be used to measure the similarity of two documents, even if the documents are different lengths.
- **Euclidean distance** measures the distance between two vectors in Euclidean space. It is a measure of how close the vectors are in terms of their magnitudes. Euclidean distance is a good measure of similarity when the magnitudes of the vectors are important. For example, it can be used to measure the similarity of two images, even if the images are different sizes.
- **Dot product** is the product of two vectors. It is a measure of how similar the directions of the vectors are, and how similar their magnitudes are. Dot product is a good measure of similarity when both the directions and magnitudes of the vectors are important. For example, it can be used to measure the similarity of two vectors in a high-dimensional space.

Which one to use when?

The best measure of similarity to use depends on the specific application. If the magnitudes of the vectors are not important, then cosine similarity is a good choice. If the magnitudes of the vectors are important, then Euclidean distance is a good choice. If both the directions and magnitudes of the vectors are important, then dot product is a good choice.

Which one is efficient?

Cosine similarity and dot product are both computationally efficient. Euclidean distance is less efficient than cosine similarity and dot product, but it is still relatively efficient.

Measure of similarity	Description	Strengths	Weaknesses
Cosine similarity	Measures the angle between two vectors	• Does not consider the magnitudes of the vectors	• Can be misleading if the vectors have different magnitudes
Euclidean distance	Measures the distance between two vectors in Euclidean space	• Considers the magnitudes of the vectors	• Can be misleading if the vectors have different directions
Dot product	Measures the product of two vectors	• Considers the directions and magnitudes of the vectors	• Can be misleading if the vectors have different magnitudes

Here is a table that summarizes the differences between cosine similarity, Euclidean distance, and dot product:

Cosine distance

The cosine similarity measures the angle between two vectors in a multi-dimensional space – with the idea that similar vectors point in a similar direction. Cosine similarity is commonly used in Natural Language Processing (NLP). It measures the similarity between documents regardless of the magnitude. This is advantageous because if two documents are far apart by the Euclidean distance, the angle between them could still be small. For example, if the word 'fruit' appears 30 times in one document and 10 in the other, that is a clear difference in magnitude, but the documents can still be similar if we only consider the angle. The smaller the angle is, the more similar the documents are.

The **cosine similarity and cosine distance have an inverse relationship**. As the **distance** between two vectors **increases**, the **similarity** will **decrease**. Likewise, if the distance decreases, then the similarity between the two vectors increases.

The cosine similarity is calculated as:

A·B is the product (dot) of the vectors A and B

||A|| and ||B|| is the length of the two vectors

||A|| * ||B|| is the cross product of the two vectors

The **cosine distance** is then calculated as: 1 - Cosine Similarity

Let's use an example to calculate the similarity between two fruits – strawberries (vector A) and blueberries (vector B). Since our data is already represented as a vector, we can calculate the distance.

Strawberry → `[4, 0, 1]`
Blueberry → `[3, 0, 1]`

$$A \cdot B = 4 * 3 + 0 * 0 + 1 * 1 = 13$$

$$||A|| = \sqrt{4^2 + 0^2 + 1^2} = 4.12$$

$$||B|| = \sqrt{3^2 + 0^2 + 1^2} = 3.16$$

$$\text{Cos}(A,B) = 13/\ (4.12 * 3.16) = 0.998$$

$$\text{Cosine Distance} = 1 - 0.998 = 0.002$$

A distance of 0 indicates that the vectors are identical, whereas a distance of 2 represents opposite vectors. The similarity between the two vectors is 0.998 and the distance is 0.002. This means that strawberries and blueberries are closely related.

Euclidean distance

Euclidean distance is calculated by taking the square root of the sum of the squares of the differences between the corresponding elements of the two vectors.

For example, let's say we have the same two vectors, *v1* and *v2*, as we used for cosine similarity. The Euclidean distance between *v1* and *v2* is calculated as follows:

```
v1 = [1, 2, 0, 0]
```

```
v2 = [0, 0, 1, 2]
```

|v1 - v2|| = sqrt((1 - 0)^2 + (2 - 0)^2 + (0 - 1)^2 + (0 - 2)^2) = sqrt(5)

Dataset Preparation:

1. **Data Preprocessing**: Before training or fine-tuning, it's important to preprocess meeting data.
   - Tokenization: Split the meeting summaries into individual tokens or words.
   - Cleaning: Remove any unnecessary characters, punctuation, or special symbols.
   - Lowercasing: Convert all text to lowercase for consistency.
   - Stop word Removal: Remove common words that do not contribute much to the meaning.
   - Lemmatization or Stemming: Reduce words to their base form to capture their core meaning.
2. **Data Organization**: Once the meeting summaries are preprocessed, we need to organize the data in a suitable format for training or fine-tuning.
   - Sentence Pairs: If we have paired meeting summaries (e.g., question and answer pairs), we can organize your data as sentence pairs, where each pair consists of a question and its corresponding answer. This format is useful for tasks like question-answering or conversational AI.
   - Contextual Embeddings: If we want to capture the context of the entire meeting summary, we can organize our data as individual meeting summaries. Each meeting summary would be treated as a separate data instance during training.
3. **Embedding Generation**: To train or fine-tune using pre-trained sentence transformer embeddings, we need to generate embeddings for your preprocessed meeting data.
   - Use a pre-trained sentence transformer model (e.g., `all-MiniLM-L12-v2`) to encode our preprocessed meeting summaries into embeddings. The model will map each summary to a fixed-length vector representation.
   - Iterate through your preprocessed data and encode each meeting summary using the sentence transformer model. This will result in a corresponding embedding for each summary.
4. **Training or Fine-tuning**: Once we have the embeddings for our meeting data, we can proceed with training or fine-tuning your model.
   - Training from Scratch: As we have a large amount of meeting data, we can train a new sentence transformer model from scratch using your embeddings as input. This allows the model to learn the representations specifically tailored to your meeting data.
   - Fine-tuning: Or for smaller dataset, we can fine-tune a pre-trained sentence transformer model using our embeddings. This process involves initializing the model with the pre-trained weights and further training it on our meeting data to adapt it to our specific task or domain.

sentence-transformers/all-MiniLM-L6-v2 at main

config.json--

- `_name_or_path`: This parameter indicates the name or path of the pre-trained model. In this case, the model is named "nreimers/MiniLM-L6-H384-uncased." It suggests that this model is based on the MiniLM architecture with 6 layers and a hidden size of 384, and it has been trained on uncased text data.

- `architectures`: This parameter specifies the model architecture used, which is "BertModel" in this case. The underlying architecture may be similar to BERT (Bidirectional Encoder Representations from Transformers).
- `attention_probs_dropout_prob`: This parameter determines the dropout probability for attention probabilities during training. It controls the amount of regularization applied to the attention mechanism, preventing overfitting.
- `gradient_checkpointing`: This parameter controls whether gradient checkpointing is enabled. When set to `false`, full backpropagation is used. Gradient checkpointing trades off memory usage for computation time during training.
- `hidden_act`: This parameter specifies the activation function used within the model's hidden layers. In this case, "gelu" (Gaussian Error Linear Unit) is used.
- `hidden_dropout_prob`: This parameter determines the dropout probability for hidden layers during training. It controls the amount of regularization applied to the hidden layers, preventing overfitting.
- `hidden_size`: This parameter represents the dimensionality of the hidden layers in the model. It is set to 384 in this configuration.
- `initializer_range`: This parameter specifies the range for weight initialization in the model.
- `intermediate_size`: This parameter indicates the dimensionality of the intermediate (feed-forward) layers in the model.
- `layer_norm_eps`: This parameter represents the epsilon value used for layer normalization.
- `max_position_embeddings`: This parameter indicates the maximum sequence length that the model can handle. It is set to 512 in this configuration.
- `model_type`: This parameter denotes the type of the model architecture. In this case, it is "bert."
- `num_attention_heads`: This parameter specifies the number of attention heads in the model.
- `num_hidden_layers`: This parameter indicates the number of hidden layers in the model. It is set to 6 in this configuration.
- `pad_token_id`: This parameter represents the token ID used for padding sequences. It is set to 0 in this configuration.
- `position_embedding_type`: This parameter denotes the type of position embeddings used. In this case, "absolute" position embeddings are used.
- `transformers_version`: This parameter indicates the version of the transformers library used.
- `type_vocab_size`: This parameter specifies the size of the token type vocabulary. It is set to 2 in this configuration.
- `use_cache`: This parameter controls whether cache is used during model inference or not.
- `vocab_size`: This parameter indicates the size of the vocabulary used by the model.

These parameters collectively define the architecture, configuration, and hyperparameters of the "nreimers/MiniLM-L6-H384-uncased" model. They determine the behavior and characteristics of the model during training and inference.

data_config.json

`pytorch_model.bin` is a binary file that contains the weights of a PyTorch model. In this case, it is part of the sentence-transformers/all-MiniLM-L6-v2 model.

- `"name"`: This parameter represents the name of the dataset file. Each dataset corresponds to a specific Stack Exchange domain and is stored in a compressed JSON file (e.g., `skeptics.stackexchange.com.jsonl.gz`).
- `"lines"`: This parameter indicates the number of lines or records in the dataset file. It provides an estimation of the dataset size.
- `"weight"`: This parameter assigns a weight to each dataset. The weight can be used to balance the influence of different datasets during training. In this case, each dataset is given a weight of 1, indicating equal importance for all datasets.

sentence_bert_config.json

The file `sentence_bert_config.json` is a configuration file for the sentence-transformers library. It contains a number of parameters that control the behavior of the library, including the maximum sequence length and whether or not to lowercase the input text.

The `max_seq_length` parameter specifies the maximum length of the input sequences that the library will accept. If the input sequence is longer than the maximum length, it will be truncated. The default value for the `max_seq_length` parameter is 256.

The `do_lower_case` parameter specifies whether or not the library should lowercase the input text. If this parameter is set to `true`, the library will lowercase all of the words in the input text before processing it. The default value for the `do_lower_case` parameter is `false`.

In the example you provided, the `max_seq_length` parameter is set to 256 and the `do_lower_case` parameter is set to `false`. This means that the library will accept input sequences up to 256 tokens long and will not lowercase the input text.

Here is a more detailed explanation of the `max_seq_length` and `do_lower_case` parameters:

- `max_seq_length`: This parameter specifies the maximum length of the input sequences that the library will accept. If the input sequence is longer than the maximum length, it will be truncated. The default value for the `max_seq_length` parameter is 256.
- `do_lower_case`: This parameter specifies whether or not the library should lowercase the input text. If this parameter is set to `true`, the library will lowercase all of the words in the input text before processing it. The default value for the `do_lower_case` parameter is `false`.

special_tokens_map.json

- `unk_token`: This token is used to represent unknown words. If a word is not found in the vocabulary of the sentence-transformers library, it will be replaced with the `unk_token`.

- `sep_token`: This token is used to mark the end of a sentence. The `sep_token` is often used to separate sentences in a sequence.
- `pad_token`: This token is used to pad sequences to a fixed length. The `pad_token` is often used to ensure that all sequences have the same length, which can be helpful for training and evaluating models.
- `cls_token`: This token is used to represent the beginning of a sentence. The `cls_token` is often used to identify the beginning of a sentence in a sequence.

`mask_token`: This token is used to mask out words in a sequence. The `mask_token` is often used to train models to ignore certain words in a sequence.

tokenizer.json

The `tokenizer.json` file is a JSON file that stores the configuration of a tokenizer. A tokenizer is a tool that is used to split text into tokens, which are individual words or subwords. The `tokenizer.json` file contains a number of parameters that control the behavior of the tokenizer, including the following:

- The vocabulary of the tokenizer.
- The special tokens used by the tokenizer.
- The rules for splitting text into tokens.

The `tokenizer.json` file is used by the sentence-transformers library to load and use a tokenizer. The library can also be used to train a new tokenizer, and the `tokenizer.json` file will be saved as part of the training process.

Here is a more detailed explanation of the contents of the `tokenizer.json` file:

- **Vocabulary:** The vocabulary of a tokenizer is a list of all of the words that the tokenizer can recognize. The vocabulary is stored as a list of strings, and each string represents a single word.
- **Special tokens:** The special tokens are a set of tokens that are used by the tokenizer for special purposes. These tokens include the start of sentence token, the end of sentence token, and the unknown word token.
- **Rules for splitting text into tokens:** The rules for splitting text into tokens are a set of rules that are used by the tokenizer to determine how to split text into tokens. These rules are often based on the morphology of the language, and they can be very complex.