Member-only story

Master Semantic Search at Scale: Index Millions of Documents with Lightning-Fast Inference Times using FAISS and Sentence Transformers

Dive into an end-to-end demo of a high-performance semantic search engine leveraging GPU acceleration, efficient indexing techniques, and robust sentence encoders on datasets up to 1M documents, achieving 50 ms inference times

Luís Roque · Follow
Published in Towards Data Science
15 min read · Apr 1

Introduction

In search and information retrieval, semantic search has emerged as a game-changer. It allows us to search and retrieve documents based on their meaning or concepts rather than just keyword matching. The semantic search leads to more sophisticated and relevant results than traditional keyword-based search methods. However, the challenge lies in scaling semantic search to handle large corpora of documents without being overwhelmed by the computational complexity of analyzing every semantic content of a document.

In this article, we rise to the challenge of achieving scalable semantic search by harnessing the power of two cutting-edge techniques: FAISS for efficient indexing of semantic vectors and Sentence Transformers for encoding sentences into these vectors. FAISS is an outstanding library designed for the fast retrieval of nearest neighbors in high-dimensional spaces, enabling quick semantic nearest neighbor search even at a large scale. Sentence Transformers, a deep learning model, generates dense vector representations of sentences, effectively capturing their semantic meanings.

This article shows how we can use the synergy of FAISS and Sentence Transformers to build a scalable semantic search engine with remarkable performance. By integrating FAISS and Sentence Transformers, we can index semantic vectors from an extensive corpus of documents, resulting in a rapid and accurate semantic search experience at scale. Our approach can enable new applications such as contextualized question-answering and advanced recommendation systems with inference times as low as 50 ms when searching a corpus of 1M documents. We will guide you through implementing this state-of-the-art end-to-end solution and demonstrate its performance on benchmark datasets.

Figure 1: Search is all you need to navigate this world (source)

This article belongs to “Large Language Models Chronicles: Navigating the NLP Frontier”, a new weekly series of articles that will explore how to leverage the power of large models for various NLP tasks. By diving into these cutting-edge technologies, we aim to empower developers, researchers, and enthusiasts to harness the potential of NLP and unlock new possibilities.

Articles published so far:

1. Summarizing the latest Spotify releases with ChatGPT

As always, the code is available on my Github.

Sentence Transformers for Semantic Encoding

Deep learning brings forth the power of sentence transformers, which craft dense vector representations that capture the essence of a sentence's meaning. Trained on massive amounts of data, these models produce contextualized word embeddings, aiming to reconstruct input sentences accurately and draw semantically similar sentence pairs closer together.

To harness the potential of sentence transformers in semantic encoding, you'll first need to choose a suitable model architecture, such as BERT, RoBERTa, or XLNet. With a model in place, we will feed a corpus of documents into it, generating fixed-length semantic vectors for each sentence. These vectors are compact numerical representations of the core themes and topics within the sentences.

Let's take two sentences as examples: 'The dog chased the cat' and 'The cat chased the dog.' When processed through a sentence transformer, their resulting semantic vectors will be closely related, even with word order differences, because the underlying meaning is similar. On the other hand, a sentence like 'The sky is blue' will yield a more distant vector due to its contrasting meaning.

Using sentence transformers to encode an entire corpus, we obtain a collection of semantic vectors that encapsulate the overarching meanings of the documents. To make this transformed representation ready for efficient retrieval, we index it using FAISS. Stay tuned, as we'll dive into this topic in the next section.

FAISS for Efficient Indexing

FAISS supports various index structures optimized for different use cases. It is a library designed for scenarios where one must quickly find the closest matches to a given query vector in a large collection of vectors.

- Inverted files (IVF): Indexes clusters of similar vectors. Suitable for medium-dimensional vectors.
- Product quantization (PQ): Encodes vectors into quantized subspaces. Suitable for high-dimensional vectors.
- Cluster-based strategies: Organizes vectors into a hierarchical set of clusters for multi-level search. Suitable for very large datasets.

To use FAISS for semantic search, we first load our vector dataset (semantic vectors from sentence transformer encoding) and construct a FAISS index. The specific index structure we choose depends on factors like the dimensionality of our semantic vectors and desired efficiency. We then index the semantic vectors by passing them into the FAISS index, which will efficiently organize them to enable fast retrieval.

For search, we encode a new sentence into a semantic vector query and pass it to the FAISS index. FAISS will retrieve the closest matching semantic vectors and return the most similar sentences. Compared to linear search, which scores the query vector against every indexed vector, FAISS enables much faster retrieval times that typically scale logarithmically with the number of indexed vectors. Additionally, the indexes are highly memory-efficient because they compress the original dense vectors.

Inverted Files Index

The Inverted Files (IVF) index in FAISS clusters similar vectors into 'inverted files' and is suitable for medium-dimensional vectors (e.g., 100–1000 dimensions). Each inverted file contains a subset of vectors that are close together. At search time, FAISS searches only the inverted files closest to the query vector instead of searching through all vectors, enabling efficient search even with many vectors.

To construct an IVF index, we specify the number of inverted files (clusters) we want and the maximum number of vectors per inverted file. Then, FAISS assigns each vector to the closest inverted file until no inverted file exceeds the maximum. The inverted files contain representative points that summarize the vectors within them. At query time, FAISS computes the distance between the query vector and each inverted file representative point and searches only the closest inverted files for the closest matching vectors.

For example, if we have 1024-dimensional image feature vectors and want to perform a fast search over 1 million vectors, we could create an IVF index with 1024 inverted files (clusters) and a maximum of 1000 vectors per inverted file. In this approach, FAISS would search only the closest inverted files to the query, resulting in faster search times than linear search.

Putting It All Together

In this section, we will build a scalable semantic search with FAISS and Sentence Transformers. We will show you how to evaluate the performance benchmarks of

this approach and discuss further improvements and applications.

Scalable Semantic Search Engine

To build a scalable semantic search engine, we first initialize the `ScalableSemanticSearch` class. This class takes care of encoding sentences using Sentence Transformers and indexing them using FAISS for efficient searching. It also provides utility methods for saving and loading indices, measuring time, and memory usage.

```
semantic_search = ScalableSemanticSearch(device="cuda")
```

Next, we encode the large corpus of documents using the encoding method, which returns a numpy array of semantic vectors. The method also creates a mapping between indices and sentences that will be useful later when retrieving the top results.

```
embeddings = semantic_search.encode(corpus)
```

Now, we build the FAISS index using the build_index method, which takes the embeddings as input. This method creates an IndexIVFPQ or IndexFlatL2 index, depending on the number of data points in the embeddings.

```
semantic_search.build_index(embeddings)
```

Selecting Indexing Approaches Based on Dataset Size

We define two indexing approaches: Exact Search with L2 distance and Approximate Search with Product Quantization and L2 distance. We will also discuss the rationale behind selecting the first approach for smaller datasets (less than 1500 documents) and the second for larger datasets.

1. Exact Search with L2 distance

Exact Search with L2 distance is an exact search method that computes the L2 (Euclidean) distance between a query vector and every vector in the dataset. This method guarantees to find the exact nearest neighbors but can be slow for large datasets, as it performs a linear scan of the data.

Use case: This method is suitable for small datasets where the exact nearest neighbors are required, and the computational cost is not a concern.

2. Approximate Search with Product Quantization and L2 distance

Approximate Search with Product Quantization and L2 distance is an approximate nearest neighbor search method that combines an inverted file structure, product quantization, and L2 distance to search for similar vectors in large datasets efficiently. The method first clusters the dataset using k-means (faiss.IndexFlatL2 as the quantizer) and then applies product quantization to compress the residual vectors. This approach allows for a faster search using less memory than brute-force methods.

Use case: This method is suitable for large datasets where the exact nearest neighbors are not strictly required, and the primary focus is on search speed and memory efficiency.

The rationale for selecting different approaches based on dataset size

For datasets containing less than 1500 documents, we set the Exact Search with L2 distance approach because the computational cost is not a significant concern in this case. Furthermore, this approach guarantees to find the nearest neighbors, which is desirable for smaller datasets.

We prefer using the Approximate Search with Product Quantization and L2 distance approach for larger datasets because it offers a more efficient search and consumes less memory than the exact search method. The Approximate Search approach proves to be more suitable for large datasets when prioritizing search speed and memory efficiency over finding the exact nearest neighbors.

Search Procedure

After building the index, we can perform a semantic search by providing an input query and the number of top results to return. The `search` method computes cosine similarity between the input sentence and the indexed embeddings and returns the indices and scores of the top matching sentences.

```
query = "What is the meaning of life?"
top = 5
top_indices, top_scores = semantic_search.search(query, top)
```

Finally, we can retrieve the top sentences using the `get_top_sentences` method, which takes the index to sentence mapping and the top indices as input and returns a list of the top sentences.

```
top_sentences = ScalableSemanticSearch.get_top_sentences(semantic_search.hashma
```

The Complete Model

The complete class of our model looks like the following:

```
class ScalableSemanticSearch:
    """Vector similarity using product quantization with sentence transformers

    def __init__(self, device="cpu"):
        self.device = device
        self.model = SentenceTransformer(
            "sentence-transformers/all-mpnet-base-v2", device=self.device
        )
        self.dimension = self.model.get_sentence_embedding_dimension()
        self.quantizer = None
        self.index = None
        self.hashmap_index_sentence = None

        log_directory = "log"
        if not os.path.exists(log_directory):
            os.makedirs(log_directory)
        log_file_path = os.path.join(log_directory, "scalable_semantic_search.l

        logging.basicConfig(
            filename=log_file_path,
            level=logging.INFO,
            format="%(asctime)s %(levelname)s: %(message)s",
        )
        logging.info("ScalableSemanticSearch initialized with device: %s", self

    @staticmethod
    def calculate_clusters(n_data_points: int) -> int:
        return max(2, min(n_data_points, int(np.sqrt(n_data_points))))
```

```
    def encode(self, data: List[str]) -> np.ndarray:
        """Encode input data using sentence transformer model.

        Args:
            data: List of input sentences.

        Returns:
            Numpy array of encoded sentences.
        """
        embeddings = self.model.encode(data)
        self.hashmap_index_sentence = self.index_to_sentence_map(data)
        return embeddings.astype("float32")

    def build_index(self, embeddings: np.ndarray) -> None:
        """Build the index for FAISS search.

        Args:
            embeddings: Numpy array of encoded sentences.
        """
        n_data_points = len(embeddings)
        if (
            n_data_points >= 1500
        ):  # Adjust this value based on the minimum number of data points requ
            self.quantizer = faiss.IndexFlatL2(self.dimension)
            n_clusters = self.calculate_clusters(n_data_points)
            self.index = faiss.IndexIVFPQ(
                self.quantizer, self.dimension, n_clusters, 8, 4
            )
            logging.info("IndexIVFPQ created with %d clusters", n_clusters)
        else:
            self.index = faiss.IndexFlatL2(self.dimension)
            logging.info("IndexFlatL2 created")

        if isinstance(self.index, faiss.IndexIVFPQ):
            self.index.train(embeddings)
        self.index.add(embeddings)
        logging.info("Index built on device: %s", self.device)

    @staticmethod
    def index_to_sentence_map(data: List[str]) -> Dict[int, str]:
        """Create a mapping between index and sentence.

        Args:
            data: List of sentences.

        Returns:
            Dictionary mapping index to the corresponding sentence.
        """
        return {index: sentence for index, sentence in enumerate(data)}

    @staticmethod
    def get_top_sentences(
```

```
        index_map: Dict[int, str], top_indices: np.ndarray
    ) -> List[str]:
        """Get the top sentences based on the indices.

        Args:
            index_map: Dictionary mapping index to the corresponding sentence.
            top_indices: Numpy array of top indices.

        Returns:
            List of top sentences.
        """
        return [index_map[i] for i in top_indices]

    def search(self, input_sentence: str, top: int) -> Tuple[np.ndarray, np.nda
        """Compute cosine similarity between an input sentence and a collection

        Args:
            input_sentence: The input sentence to compute similarity against.
            top: The number of results to return.

        Returns:
            A tuple containing two numpy arrays. The first array contains the c
            sentence and the embeddings, ordered in descending order. The secon
            corresponding embeddings in the original array, also ordered by des
        """
        vectorized_input = self.model.encode(
            [input_sentence], device=self.device
        ).astype("float32")
        D, I = self.index.search(vectorized_input, top)
        return I[0], 1 - D[0]

    def save_index(self, file_path: str) -> None:
        """Save the FAISS index to disk.

        Args:
            file_path: The path where the index will be saved.
        """
        if hasattr(self, "index"):
            faiss.write_index(self.index, file_path)
        else:
            raise AttributeError(
                "The index has not been built yet. Build the index using `build
            )

    def load_index(self, file_path: str) -> None:
        """Load a previously saved FAISS index from disk.

        Args:
            file_path: The path where the index is stored.
        """
        if os.path.exists(file_path):
            self.index = faiss.read_index(file_path)
        else:
```

```
            raise FileNotFoundError(f"The specified file '{file_path}' does not

    @staticmethod
    def measure_time(func: Callable, *args, **kwargs) -> Tuple[float, Any]:
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        elapsed_time = end_time - start_time
        return elapsed_time, result

    @staticmethod
    def measure_memory_usage() -> float:
        process = psutil.Process(os.getpid())
        ram = process.memory_info().rss
        return ram / (1024**2)

    def timed_train(self, data: List[str]) -> Tuple[float, float]:
        start_time = time.time()
        embeddings = self.encode(data)
        self.build_index(embeddings)
        end_time = time.time()
        elapsed_time = end_time - start_time
        memory_usage = self.measure_memory_usage()
        logging.info(
            "Training time: %.2f seconds on device: %s", elapsed_time, self.dev
        )
        logging.info("Training memory usage: %.2f MB", memory_usage)
        return elapsed_time, memory_usage

    def timed_infer(self, query: str, top: int) -> Tuple[float, float]:
        start_time = time.time()
        _, _ = self.search(query, top)
        end_time = time.time()
        elapsed_time = end_time - start_time
        memory_usage = self.measure_memory_usage()
        logging.info(
            "Inference time: %.2f seconds on device: %s", elapsed_time, self.de
        )
        logging.info("Inference memory usage: %.2f MB", memory_usage)
        return elapsed_time, memory_usage

    def timed_load_index(self, file_path: str) -> float:
        start_time = time.time()
        self.load_index(file_path)
        end_time = time.time()
        elapsed_time = end_time - start_time
        logging.info(
            "Index loading time: %.2f seconds on device: %s", elapsed_time, sel
        )
        return elapsed_time
```

End-to-End Demo

This section will provide an end-to-end demo of the scalable semantic search engine using the SemanticSearchDemo class and the
main function from the code above. The goal is to understand how the different concepts and components combine to create a practical application.

Initializing the SemanticSearchDemo class: To initialize the SemanticSearchDemo class, provide the dataset path, the ScalableSemanticSearch model, an optional index path, and an optional subset size. This flexibility enables using different datasets, models, and subset sizes.

```
demo = SemanticSearchDemo(
    dataset_path, model, index_path=index_path, subset_size=subset_size
)
```

Loading data: The `load_data` function actively reads and processes data from a file, then returns a list of sentences. The system uses this data to train the semantic search model.

```
sentences = demo.load_data(file_name)
subset_sentences = sentences[:subset_size]
```

Training the model: The `train` function trains the semantic search model on the dataset and returns the training process's elapsed time and memory usage.

```
training_time, training_memory_usage = demo.train(subset_sentences)
```

Performing inference: The `infer` function takes a query, a list of sentences to search in, and the number of top results to return. It performs inference on the model and returns the top matching sentences, elapsed time, and memory usage for the inference process.

```
top_sentences, inference_time, inference_memory_usage = demo.infer(
    query, subset_sentences, top=3
)
```

The full class for the demo is below:

```
class SemanticSearchDemo:
    """A demo class for semantic search using the ScalableSemanticSearch model.

    def __init__(
        self,
        dataset_path: str,
        model: ScalableSemanticSearch,
        index_path: Optional[str] = None,
        subset_size: Optional[int] = None,
    ):
        self.dataset_path = dataset_path
        self.model = model
        self.index_path = index_path
        self.subset_size = subset_size

        if self.index_path is not None and os.path.exists(self.index_path):
            self.loading_time = self.model.timed_load_index(self.index_path)
        else:
            self.train()

    def load_data(self, file_name: str) -> List[str]:
        """Load data from a file.

        Args:
            file_name: The name of the file containing the data.

        Returns:
            A list of sentences loaded from the file.
        """
        with open(f"{self.dataset_path}/{file_name}", "r") as f:
            reader = csv.reader(f, delimiter="\t")
            next(reader)  # Skip the header
            sentences = [row[3] for row in reader]  # Extract the sentences
        return sentences

    def train(self, data: Optional[List[str]] = None) -> Tuple[float, float]:
        """Train the semantic search model and measure time and memory usage.

        Args:
            data: A list of sentences to train the model on. If not provided, t
```

```
        Returns:
            A tuple containing the elapsed time in seconds and the memory usage
        """
        if data is None:
            file_name = "GenericsKB-Best.tsv"
            data = self.load_data(file_name)

            if self.subset_size is not None:
                data = data[: self.subset_size]

        elapsed_time, memory_usage = self.model.timed_train(data)

        if self.index_path is not None:
            self.model.save_index(self.index_path)

        return elapsed_time, memory_usage

    def infer(
        self, query: str, data: List[str], top: int
    ) -> Tuple[List[str], float, float]:
        """Perform inference on the semantic search model and measure time and

        Args:
            query: The input query to search for.
            data: A list of sentences to search in.
            top: The number of top results to return.

        Returns:
            A tuple containing the list of top sentences that match the input q
        """
        elapsed_time, memory_usage = self.model.timed_infer(query, top)
        top_indices, _ = self.model.search(query, top)
        index_map = self.model.index_to_sentence_map(data)
        top_sentences = self.model.get_top_sentences(index_map, top_indices)

        return top_sentences, elapsed_time, memory_usage
```

Performance Evaluation of our Scalable Semantic Search Engine

In order to evaluate the performance of our scalable semantic search engine, we can measure the time and memory usage for various operations like training, inference, and loading indices. The `ScalableSemanticSearch` class provides the `timed_train`, `timed_infer`, and `timed_load_index` methods to measure these benchmarks.

```
train_time, train_memory = semantic_search.timed_train(corpus)
```

```
infer_time, infer_memory = semantic_search.timed_infer(query, top)
```

The plots for both the training and inference performance in terms of execution time and memory usage can be found below. We will be discussing and interpreting the results in light of the selection of algorithms based on the size of the corpus we used.

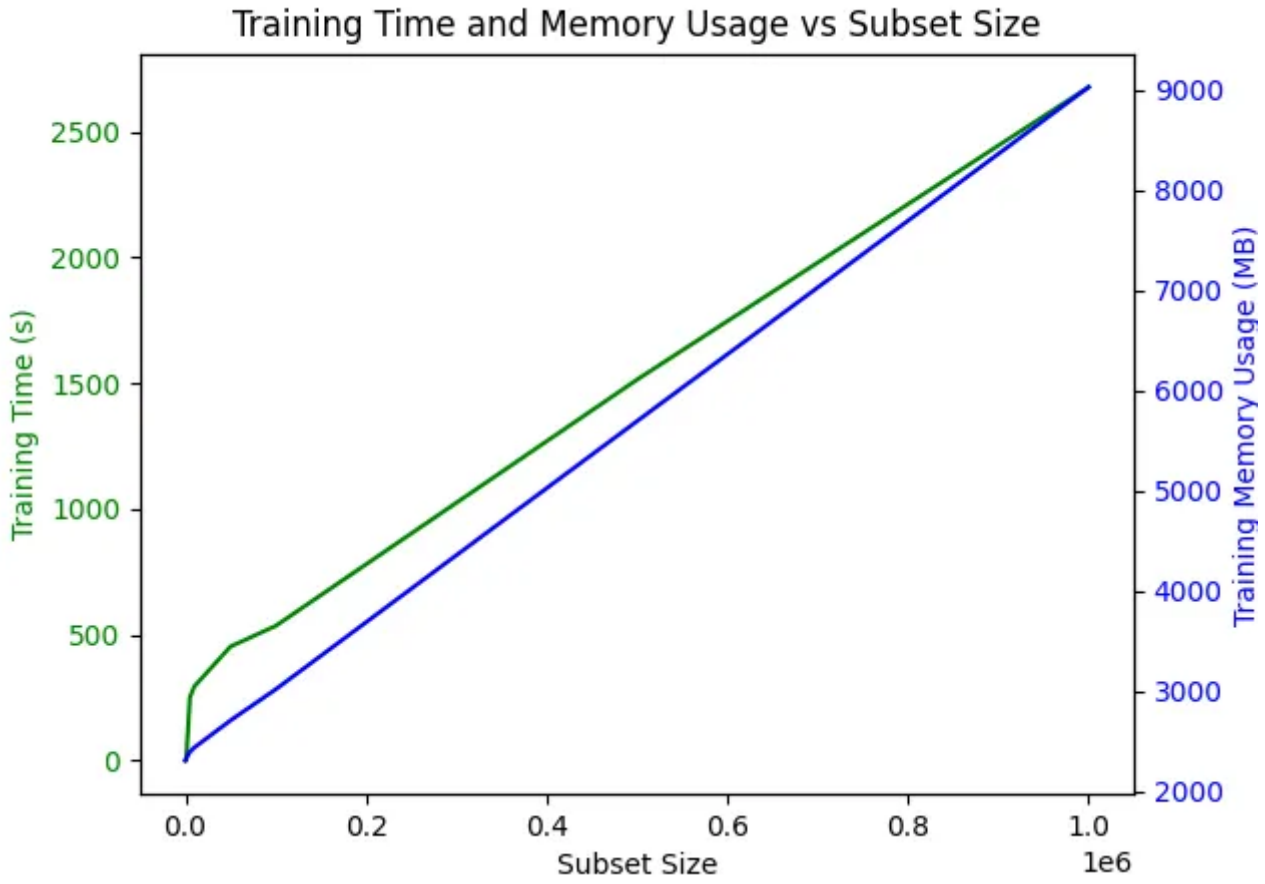


Figure 2: Training time and memory usage for different dataset sizes

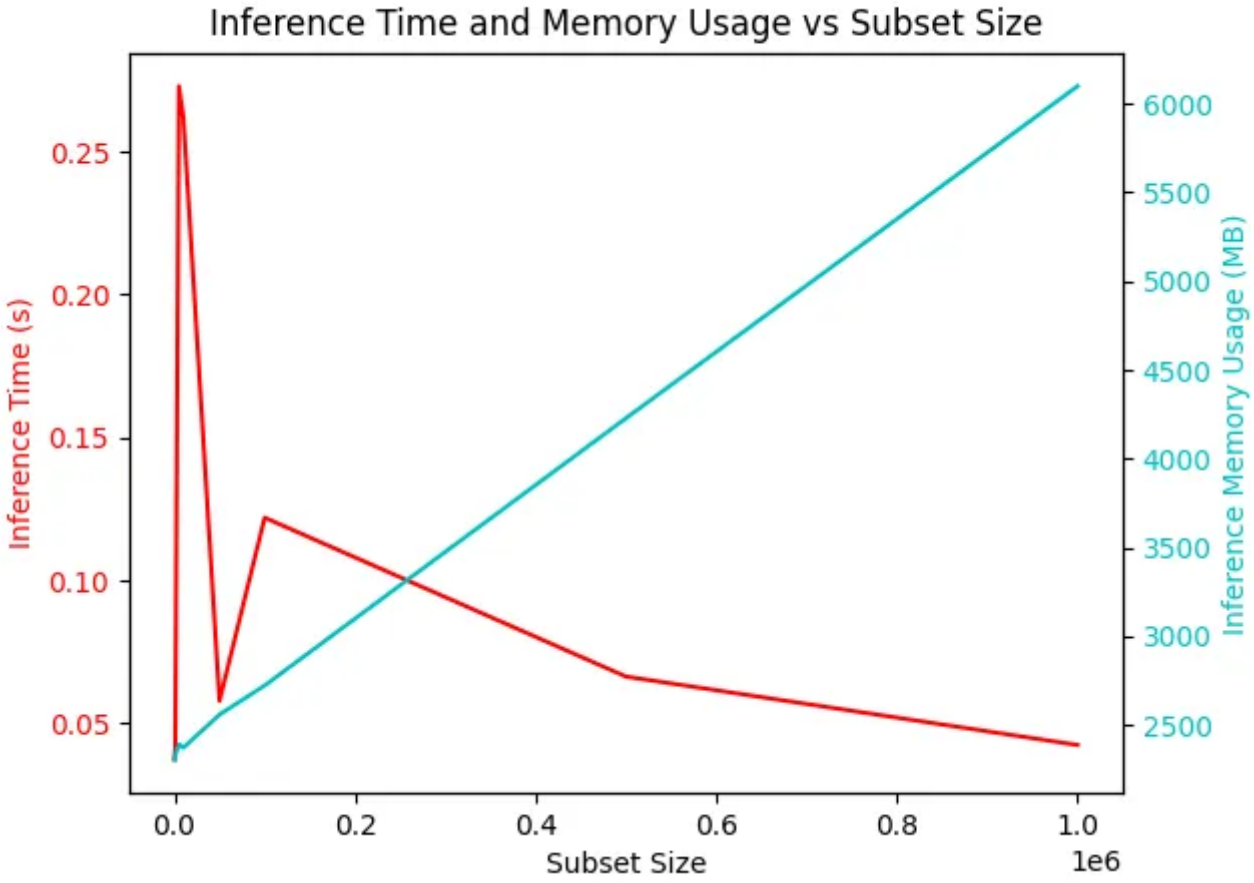


Figure 3: Inference time and memory usage for different dataset sizes

Exact Search using L2 distance

Exact Search using L2 distance is an exhaustive search method that performs a linear scan to find the nearest neighbors.

Theoretical Complexity

- Time Complexity: O(n) — Because it needs to compare the query vector with every vector in the dataset.
- Memory Complexity: O(n) — It stores all the vectors in the dataset.

Observed Complexity

- From the plots above, we can observe that for less than 1500 documents, both the training time and memory usage increase linearly with the number of documents, which matches the expected theoretical complexity.

Approximate Search using Product Quantization and L2 distance

Approximate Search using Product Quantization and L2 distance is an approximate nearest neighbor search method that employs product quantization and an inverted

file structure for improved efficiency. The number of clusters (k) is an essential factor in this method, and it is calculated using the formula: max(2, min(n_data_points, int(np.sqrt(n_data_points)))).

In simpler terms, this formula ensures that:

1. There are at least 2 clusters, providing a minimum level of partitioning.
2. The number of clusters doesn't exceed the number of data points.
3. As a heuristic, the square root of the number of data points is used to balance search accuracy and computational efficiency.

Theoretical Complexity

- Time Complexity (Training): O(n * k) — The complexity of the k-means clustering algorithm used in the training phase.
- Memory Complexity (Training): O(n + k) — It stores the centroids of clusters and residual codes.
- Time Complexity (Inference): O(k + m) — Where m is the number of nearest clusters to be searched. It is faster than linear search due to the hierarchical structure and approximation.
- Memory Complexity (Inference): O(n + k) — It requires storing the inverted file and the centroids.

Observed Complexity

From the plots above, we can observe that for more than 1500 documents:

- Training time complexity: The growth is faster than linear, which matches the expected theoretical complexity of O(n * k) since k grows with the number of data points (n).
- Training memory complexity: The memory usage increases non-linearly with the number of documents, which matches the expected theoretical complexity of O(n + k).
- Inference time complexity: The execution time remains almost constant, which is consistent with the expected theoretical complexity of O(k + m), as m is

usually much smaller than n.

- Inference memory complexity: The memory usage increases linearly with the number of documents, which matches the expected theoretical complexity of O(n + k).

We could also evaluate the accuracy and recall of the search engine by comparing the top results against a manually curated set of ground truth results for a given query. We can calculate the average accuracy and recall for the entire dataset by iterating over various queries and comparing the results.

Conclusion

The demonstrated approach highlights the scalability of semantic search using FAISS and Sentence Transformers while revealing enhancement opportunities. For instance, integrating advanced transformer models for encoding sentences or testing alternative FAISS configurations could speed up the search process. Additionally, investigating state-of-the-art models like GPT-4 or BERT variants might improve semantic search tasks' performance and accuracy.

Several potential applications for the scalable semantic search engine include:

- Retrieving documents in extensive knowledge bases
- Answering questions in automated systems
- Providing personalized recommendations
- Generating chatbot responses

Taking advantage of FAISS and Sentence Transformers, we developed a scalable semantic search engine capable of efficiently processing billions of documents and delivering accurate search results. This innovative approach can significantly influence the future of semantic search and its impact across various industries and applications.

As digital data grows, the demand for efficient and accurate semantic search engines becomes more critical. Based on FAISS and Sentence Transformers, the scalable semantic search engine lays a strong foundation for overcoming these challenges and revolutionizing how we search for and access relevant information.

Future advancements involve incorporating more advanced natural language processing and machine learning techniques to enhance search engine capabilities. These improvements could encompass unsupervised learning methods for better understanding context, intent, and relationships between query words and phrases and approaches for handling ambiguity and variations in language use.

Keep in touch: LinkedIn

Data Science | Machine Learning | Artificial Intelligence | Python | Deep Learning

Follow

Written by Luís Roque

1.4K Followers · Writer for Towards Data Science

Head of Data @ Marley Spoon | Ph.D. Researcher AI @ LIACC | Coordinator DS Masters @ NDS | CoFounder & ex-CEO @ HUUB

More from Luís Roque and Towards Data Science

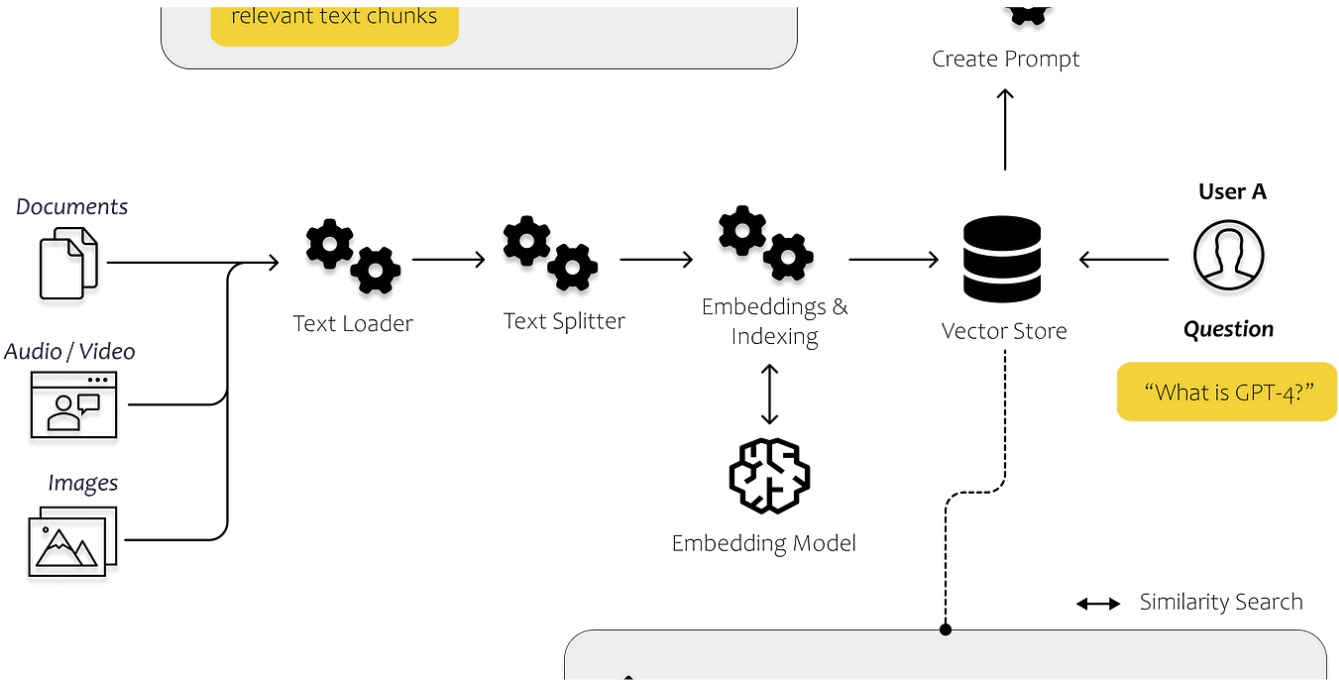
Luís Roque in Towards Data Science

Document-Oriented Agents: A Journey with Vector Databases, LLMs, Langchain, FastAPI, and Docker

Leveraging ChromaDB, Langchain, and ChatGPT: Enhanced Responses and Cited Sources from Large Document Databases

✦ · 11 min read · Jul 5

262 3



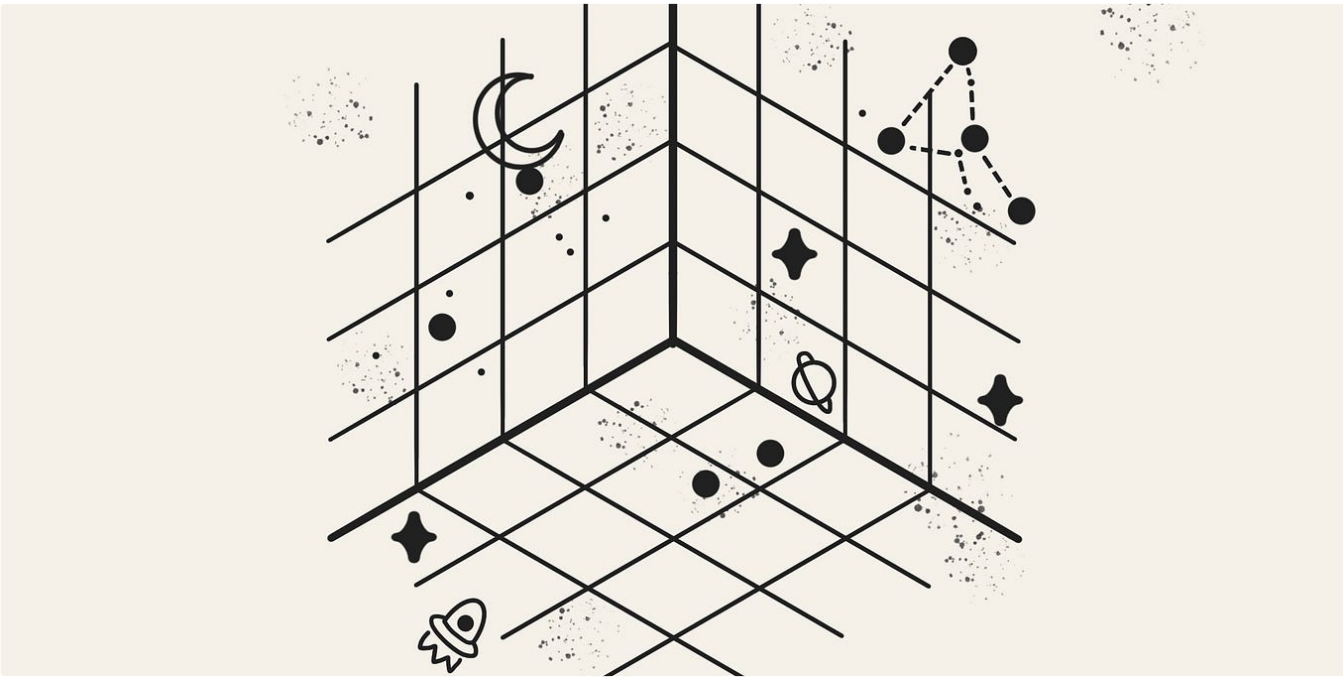
Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 22

3.7K 34



Leonie Monigatti in Towards Data Science

Explaining Vector Databases in 3 Levels of Difficulty

From noob to expert: Demystifying vector databases across different backgrounds

· 8 min read · Jul 4

1.92K 21

Luís Roque in Towards Data Science

Leveraging Llama 2 Features in Real-world Applications: Building Scalable Chatbots with FastAPI...

An In-Depth Exploration: Open vs Closed Source LLMs, Unpacking Llama 2's Unique Features, Mastering the Art of Prompt Engineering, and...

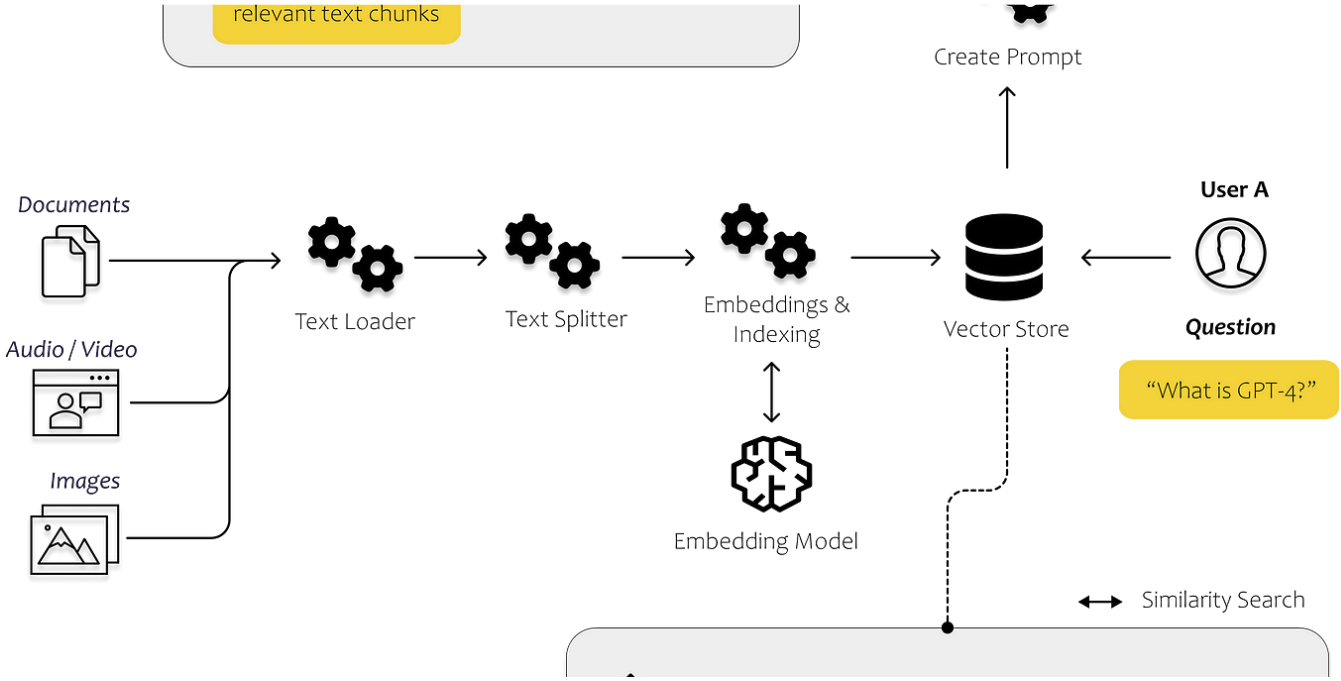
· 14 min read · Jul 24

352 5

See all from Luís Roque

See all from Towards Data Science

Recommended from Medium



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

✦ · 26 min read · Jun 22

3.7K 34

Ryan Nguyen in How AI Built This

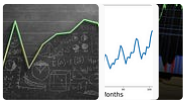
Zero to One: A Guide to Building a First PDF Chatbot with LangChain & LlamaIndex—Part 1

Welcome to Part 1 of our engineering series on building a PDF chatbot with LangChain and LlamaIndex. Don't worry, you don't need to be a...

15 min read · May 11

81 3

Lists



Predictive Modeling w/ Python

18 stories · 195 saves

Practical Guides to Machine Learning

10 stories · 211 saves

Natural Language Processing

455 stories · 89 saves

ChatGPT

21 stories · 79 saves

Lucas McGregor

No One Wants To Talk To Your Chatbot

5 min read · Jul 16

285 5

Sami Maameri in Better Programming

Private LLMs on Your Local Machine and in the Cloud With LangChain, GPT4All, and Cerebrium

The idea of private LLMs resonates with us for sure. The appeal is that we can query and pass information to LLMs without our data or...

17 min read · Jun 15

967 8

CUSTOMER DETAILS

Billing		Delivery	
Nick Bert	P:0401 320 816	Nick Bert	M:0401 320 816
134 Barker Street	M:0401 320 816	134 Barker Street	
NEW FARM Queensland 4005	Account#: WW-833332	NEW FARM Queensland 4005	
Australia		Australia	

DESCRIPTION:	QTY:	UNIT PRICE: (INC TAX)	TOTAL: (EX TAX)	TOTAL: (INC TAX)
Adreno Dive Trip - Wreck of the Marietta Dal & Smiths Rock Double Dive Trip (ID: 179285, AD-DiveMariettaSmiths,)	1	$210.00	$190.91	**$210.00**
Scuba Kit Hire (Tank, Weights, BCD, Regulator & Computer) (ID: 172605, AD-SCUBA-SET,)	1	$70.00	$63.64	**$70.00**
Adreno Wetsuit Hire	1	$30.00	$27.27	**$30.00**

in box

Use of Generative A.I. to achieve better OCR results than traditional tools

As generative AI advances, models like ChatGPT have proven adept at extracting text from unstructured data. These models can grasp the...

3 min read · Apr 12

95

Sushil Khadka

Self-Attention in Transformers

A Beginner-Friendly Guide to Self-Attention Mechanism

11 min read · May 15

93 2