ASIF QAMAR

SUPPORT VECTORS

LLM PROJECTS BOOTCAMP

Caveat Emptor!

This is currently a work in progress, and incomplete: therefore, please do not circulate it yet.

supportvectors ai lab technical series

www.supportvectors.com

First draft, February 25, 2024.

Cover art: a young guy among books and clouds, the world of dreams, reading literature, created by a neural network, Generative AI technology. (Licensed from Adobe stock: https://adobe.ly/45k1zy6)

Typeset in LaTeX.

Contents

"The journey of a thousand miles begins with a single step."

Lao Tzu's Dao De Jing, Chapter 64

1

Journey of a thousand miles...

Contents

千里之行，始于足下

1.1 Welcome, AI Visionaries!

Welcome to the in-depth boot camp on large language models (LLMs) — the frontier of artificial intelligence. As you stand on the threshold of this transformative journey, know that the path ahead is arduous. The intricacies of LLMs are vast and multifaceted. However, the promise of what you'll uncover, the insights you'll gain, and the skills you'll hone will be deeply, profoundly enriching.

Over the next few weeks, you will dive into the inner workings of some of our field's most potent computational entities. LLMs are not just tools; they are a testament to human ingenuity, innovation, and the unyielding spirit of exploration. And you, brave engineers, are here to master their potential.

The full translation of the Tao chapter:

A journey of a thousand miles starts under one's feet.
A tree with a full span's girth begins from a tiny sprout.
A nine-storied terrace rises from a basketful of earth.
A journey of a thousand miles starts right at the foot which is below one.
To act with desire is to be agenda-ridden, and thus to err, to go astray.
To act without agitating is to grip things firmly.
To act with desire is to want something, and thus not to be content with things as they are.
To act without desires is to be free from such wants, to be open and self-revealing, self-illuminating.

By the time this bootcamp concludes, you will not only have worked on two dozen high-value, real-life AI projects, but you will also emerge with a deep-rooted expertise in LLMs. Every challenging module, every hands-on task, every discussion, and every brainstorming session will forge a version of you that's sharper, more knowledgeable, and brimming with confidence in your abilities.

Remember, the realms of AI are not for the faint-hearted. But with grit, dedication, and the right guidance, they hold a treasure of knowledge and possibilities. This bootcamp is your map. Embark on this quest with an open mind and a resilient spirit, and let's redefine what's possible together.

"The journey of a thousand miles begins with a single step."

Let the journey begin!

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Antoine de Saint-Exupery

2

Tl;dr

Contents

What mischief brews hither?

2.1 Introduction

This is the eagerly awaited hackathon-style, coding-centered boot camp. The goal is to make you deeply confident and fluent in applying LLMs to solve a large range of real-world problems in multi-modal learning, video comprehension, audio-processing, natural language processing, tabular data, anomaly and fraud detection, AI-search, healthcare applications, and clever techniques of prompt engineering.

You will work in teams of 4-5 engineers working in an environment that closely reproduces the innovation and development that is the quintessence of Silicon Valley spirit.

You will have access to a 4-GPU massive AI server at the admin-privilege level for the three-month duration of this boot camp.[1] Besides this, you will also have the benefit of 10-gigabit networking and the large server cluster to take your projects to actual production. SupportVectors will let you use the compute resources for an additional 4 weeks if you need to finish any remaining aspects of your projects.

[1] Each team will get its specific timeshare for model training on the AI server.

SupportVectors Creative Studio will help you create professional videos explaining your projects and will provide creative writing help in deriving detailed technical articles around your projects.

This AI boot camp comprises about a dozen projects, mapping roughly to one project a week. Each project, internally, will comprise of at-least two or three AI sub-projects, as well considerable amount of data engineering, application development, MLOps and software architecture.

Why Develop in-depth expertise in crafting real-world, enterprise and web scale AI systems.

What Each Saturday, you will gather with your team and collaborate to build an AI-system over the course of day (and the subsequent week).

When This course spans 12 weeks, on Saturday of each week (10 AM to 10 PM Pacific Standard Time), starting on September the 9th.

Where You can do it in-person (the preferred way), remotely if you are far away, or in a hybrid mode.

2.2 It is a team effort!

These projects build AI-systems over a very compacted time-frame. As such, they are inherently non-trivial in scope and specification. This is deliberate, to give you an intense learning experience. Therefore, it would be very hard for any working professional to carve out enough spare time to do the entire project on their own.[2] The cohort will be partitioned into teams – usually, the teams are already formed prior to the start of boot camp. To see which team you are a part of, log into the learning portal.

[2] **Caveat Emptor!** If you really want to be a lone wolf, and are confident you have the time and talent to pull it off completely yourself, then proceed.

Considering the scope of work each project entails, it is worth reiterating the importance of team-work.

> “Getting good players is easy.
>
> Getting ‘em to play together is the hard part.”
>
> CASEY STENGEL

2.3 The learning portal

The learning portal associated with this boot camp is: https://supportvectors.io. Solutions, and code-samples will be posted to this portal, along with instructions for the projects. We will continue to use the SupportVectors AI lab slack channel for continually running discussions.

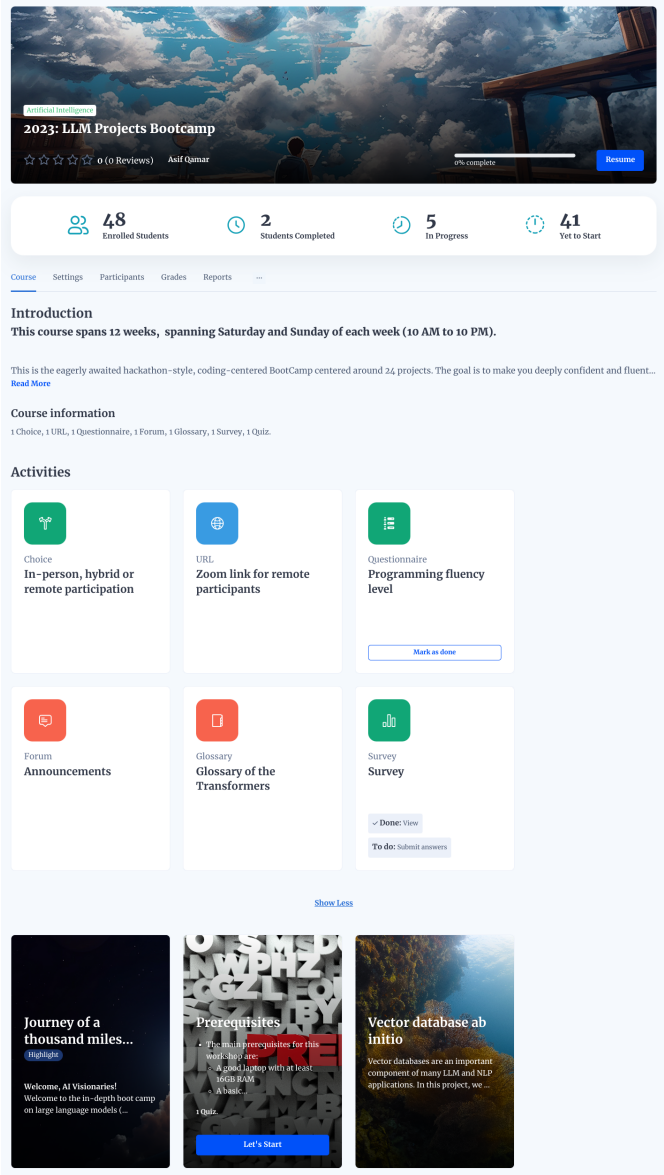


Figure 2.1: The learning portal

Well begun is half done.

Shakespeare

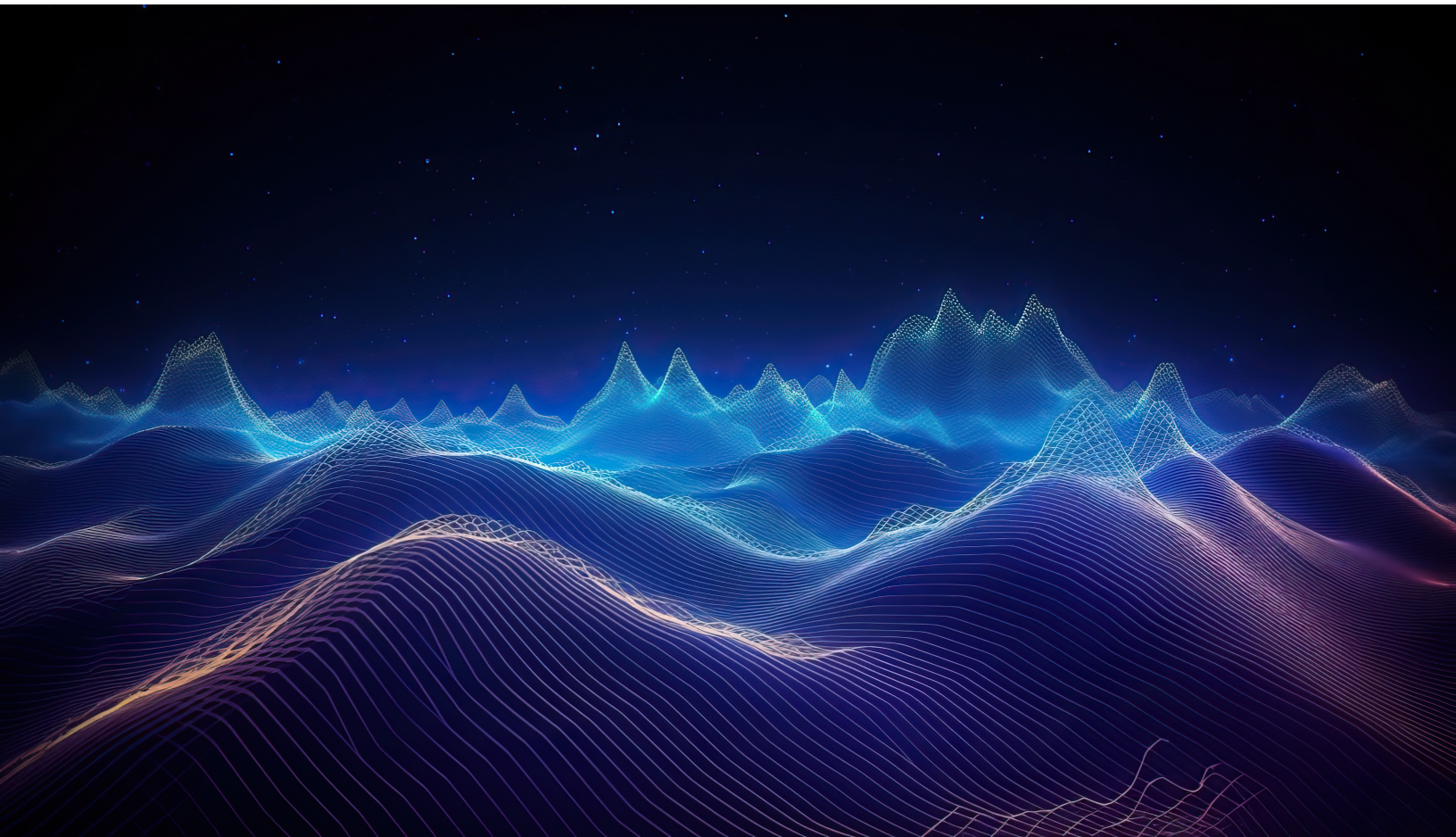
"Failure after long perseverance is much grander than never to have a striving good enough to be called a failure."

George Eliot

3

Latent metropolis of meanings

We learn to build a vector database of our own, and call it the Latent metropolis.



Contents

3.1 Introduction

A peculiar paradox permeates the AI world: it is relatively easy to prototype a proof-of-concept artificial intelligence-based application that serves as a stellar demo. The vast ecosystem of open-source models and ever-increasing commercial AI services available as API reinforce the impression that it has

become vastly simpler and easier to create incredible solutions to real-world problems.

However, taking these proof-of-concept demos and then transmuting them into an enterprise-scale or web-scale application has proved extraordinarily difficult. Indeed, despite a plethora of few-liner code snippets available on the web that purportedly work miracles, most real-world enterprise AI applications and platforms take an entirely different, methodical, architecture-driven approach to crafting AI systems.

This vast chasm between the effort to prototype versus the resources needed to build the entire AI system will remain a leitmotif through the months ahead with us across the various projects. Prototyping a successful proof-of-concept is often the domain of AI researchers and engineers. On the other hand, building an entire AI system requires the collaboration of at least four tribes:

- the AI researchers/engineers,
- the MLOps and system engineers,
- the data engineers,
- and the application development engineers.

Building an AI system that synthesizes the contributions of these tribes is the skill of an AI system architect – and developing this skill is the core learning goal of this boot camp.

In this first project, we will build an AI system for semantic search and, thus, an end-to-end implementation of all the parts. It has its roots in the very first lab of the two prior workshops:[1]

NLP with transformers Recall that we did a Jupyter lab notebook centered around sentence embedding and semantic search.

Neural Architectures Likewise, we covered the topic of sentence embedding and multimodal search in considerable theoretical detail in this workshop.

[1] **If you haven't taken the priors:** All is not lost if you have not taken the previous workshops! Those were theory centered. Many participants prefer a more hands-on approach: learn by doing and building, then return to understanding the foundations behind it all.

Work with your teammates to fill you in with the background context from the prior workshops wherever necessary.

In this project, we will translate what we did in the Jupyter notebooks into a production-grade AI-driven semantic search engine. So we will build all the parts needed.

3.1.1 *Scope for improvement*

The first AI-systems architecture we will build here favors simplicity over other considerations. For example, we do not try to embed metadata into the index; instead, we post-filter the search retrieval along the metadata criteria.

In later projects of this boot camp, we will have an opportunity to revisit all of this and evolve it.

3.2 Recap of the background theory

In this section, we will recapitulate the foundational concepts behind AI-based semantic search.[2]

[2] **Okay to skip ahead!**
If you are well versed in these aspects – say, you have already reviewed the lab notebooks of the previous workshops – then consider skipping this section altogether.

3.2.1 *Semantic search*

Traditionally, one would search through a corpus of documents using a keywords-based search engine like Lucene, Solr, ElasticSearch, etc. While the technology has matured, the basic underlying approach behind keyword search engines is to maintain an *inverted-index* mapping keywords to a list of documents that contain them, with associated relevances.

In general, the keywords-based search approach has been quite successful over the years, and have matured with added features and linguistic capabilities.

However, this approach has had its limitations. The principal cause of it goes to the fact that when we enter keywords, it is a human tendency to describe the

intent of what we are looking for. For example, if we enter "breakfast places", we implicitly also mean restaurants, cafe, etc that serve items appropriate for breakfast. There may be a restaurant described as a shop for expresso, or crepe, that a keywords-search will likely miss, since its keywords do not match the query terms. And yet, we would hope to see it near the top of the search results.

Semantic search is an NLP approach largely relying on deep-neural networks, and in particular, the transformers that make it possible to more closely infer the human intent behind the search terms, the relationship between the words, and the underlying context. It allows for entire sentences – and even paragraphs – describing what the searcher's intent is, and retrieves results more relevant or aligned to it.

3.2.2 How would we do this NLP task with AI?

Let us represent the functional behavior we expect:

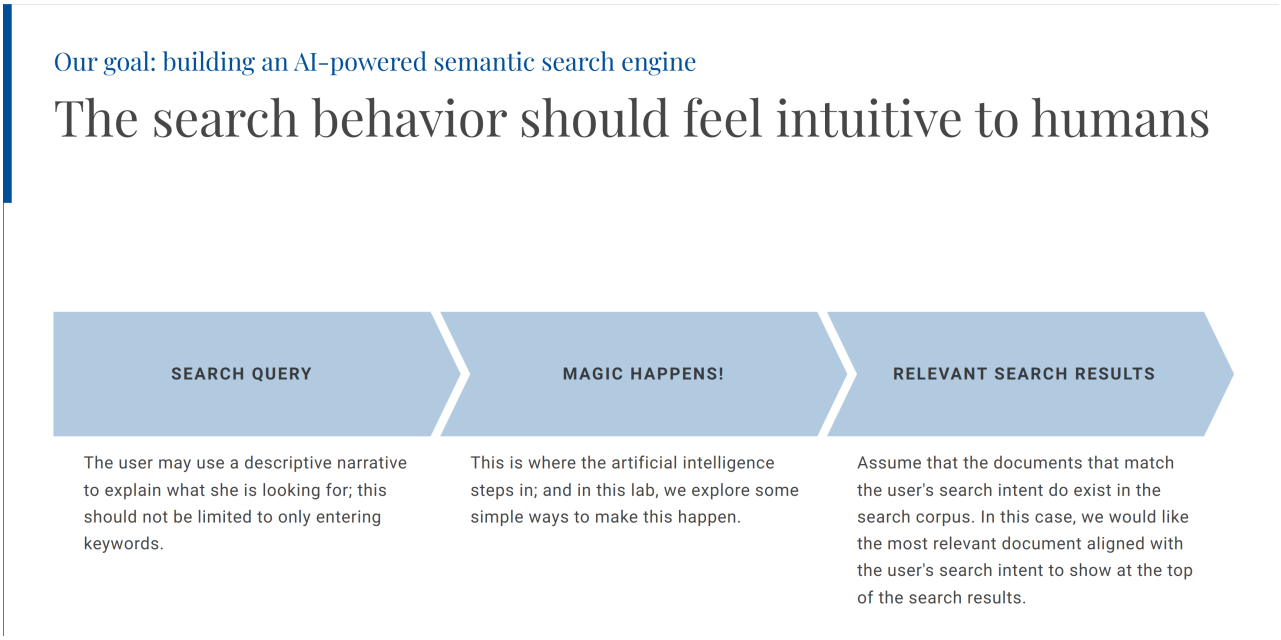


Figure 3.1: A more intuitive search experience for the user.

3.2.3 Magic happens: breaking it down into steps

We recall that machine-learning algorithms work with vectors ($\mathbf{X}$) representation of data.

So the first order of business would be to map each of the document texts D_i to its corresponding vector $\mathbf{X}_i$ in an appropriate d-dimensional space, $\mathbb{R}^d$, i.e.

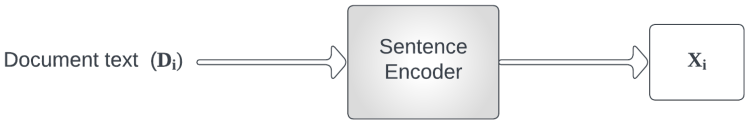


Figure 3.2: The sentence encoder creates a vector embedding, in a semantically learned latent embedding $d-$dimensional real-valued space, $\mathbb{R}^d$.

$$D_i \longrightarrow \mathbf{X}_i \in \mathbb{R}^d \tag{3.1}$$

In order to train a neural network to create semantically learned vector embedding a sentence, a contrastive loss function is used to train on a collection of triplets, where each triplet is:

$$< \mathbf{X}_a, \mathbf{X}_b, \mathbf{X}_c >$$

where

$\mathbf{X}_a$: a given sentence

$\mathbf{X}_b$: a similar sentence to$\mathbf{X}_a$

$\mathbf{X}_c$: a random sentence, presumably unrelated or dissimilar to $\mathbf{X}_a$ (3.2)

This resulting vectors are called **sentence embeddings**.[3] Once these embeddings are for each of the documents, we can store the collection of tuples $[< D_1, \mathbf{X}_1 >, < D_2, \mathbf{X}_2 >, ..., < D_n, \mathbf{X}_n >]$. Here each tuple corresponds to a document and its sentence embedding.

This collection of tuples, therefore, becomes our database: we need a **search index** to quickly retrieve the tuples we are looking for. Much as a book has an

[3] **Caveat Emptor**

Note that we have a rather relaxed definition of a *sentence* in NLP: it diverges from a grammatical definition of a sentence somewhat. For example, in the English language, we would consider a sentence to be terminated with a punctuation, such as a period, question-mark or exclamation. However, in NLP, we loosely consider the entire text – whether it is just a word, or a few keywords, or an English sentence, or a few sentences together – as one **sentence** for the purposes of natual language processing task.

index to help us quickly search for what we are looking for, databases facilitate fast retrieval through an index.

3.2.4 Search

Now, when the user described what she is looking for, we consider the entire query text as a "sentence". The Sentence-Encoder converts the query into the corresponding latent-vector. A nearest neighbor search within the collection of previously gathered vectors therefore gives us the search results.

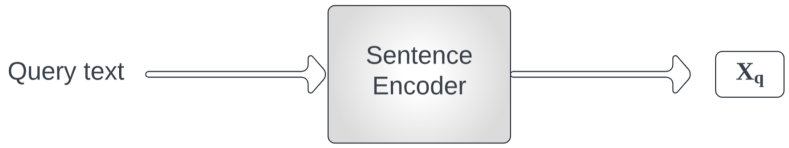


Figure 3.3: Application of the Sentence-Encoder on the query text gives us the latent-vector in the embedding space, where one can search for k-nearest neighbors.

3.2.5 Vector Similarity

Once we have this, we simply need to compare the query vector $\mathbf{X}_q$ with each of the document vectors $\mathbf{X}_i$, and sort the document vectors in descending order of similarity.

The rest is trivial: pick the top-k in sorted sentence vectors list. Then for each vector, look up its corresponding document, and return the list as sorted search result of relevant document.

We expect that these sentences will exhibit high semantic similarity with the search query, assuming that the search index did contain such documents.

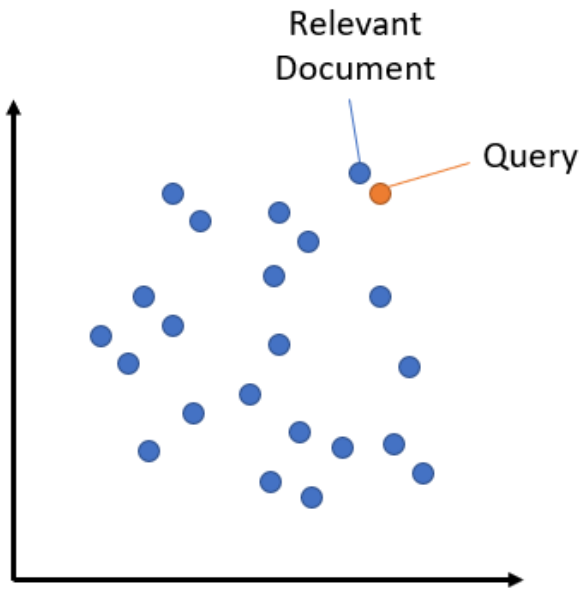


Figure 3.4: Semantic similarity as vector proximity in the embedding space.(Figure source: Sbert.net documentation).

3.2.6 *Similarity measures*

The sentence embedding vectors typically exist in very large dimensional space (e.g., 300 dimensions). In such large dimensional spaces, the notion of euclidean distance is not as effective. Therefore, it is far more common to use one of the two below measures for vector similarity:

dot-product the (inner) dot-product between the embedding vectors.

$$\text{dot-similarity} = \langle \mathbf{X}_i, \mathbf{X}_j \rangle \tag{3.3}$$

cosine-similarity the $\cos(\theta_{ij})$ gives degree of directional alignment between the vectors, but ignores their magnitudes. Here, θ_{ij} is the angle between $\mathbf{X}_i$ and $\mathbf{X}_j$ (embedding) vectors.

$$\text{cosine-similarity} = \frac{\langle \mathbf{X}_i, \mathbf{X}_j \rangle}{\|\mathbf{X}_i\| \|\mathbf{X}_j\|} \tag{3.4}$$

Important: Sentence transformer models trained with cosine-similarity tend to favor the shorter document texts in the search results, whereas the models trained on the dot-product similarity tend to favor longer texts.

3.2.7 Symmetric vs asymmetric search

One of the technical aspects to be careful of is the relative textual length of the query sentence compared to the actual documents. Different sentence-transformer models have been trained specifically for each of these use-cases.

Symmetric search when we expect the query-sentence to be approximately the same length as the document sentences.

Asymmetric search when we expect the document texts to be significantly larger in length to the query sentence.

3.3 Architecture

We will use a hybrid search approach in this project, and to quite some extent, leverage the power of existing database architectures.

We use a hybrid search architecture to maximize the benefits separately derived from AI-based semantic search, and the traditional keywords-based search. Under varying queries, each will produce results of different quality or relevance. Therefore, by hybridizing the results, we get the best of both.

RDBMS For keeping track of the documents, we will use a relational. This is where we will keep the text extractions from documents in three separate tables.[4]

File-store The file-store, stores the original document files. Typically, it can be either the operating-system's file-system or it can be a form of object-store in the cloud. Here, we will simply use the Linux file-system as original document store.

Search microservice The main facade of the Semantic search engine, which which the clients engage using the REST-api calls.

Client The client, which could be a web user-interface, or it can be some other service, invoking the main search microservice using the REST-api.

ElasticSearch This is necessary to ensure that traditional keyword search results too are considered during retrieval.[5]

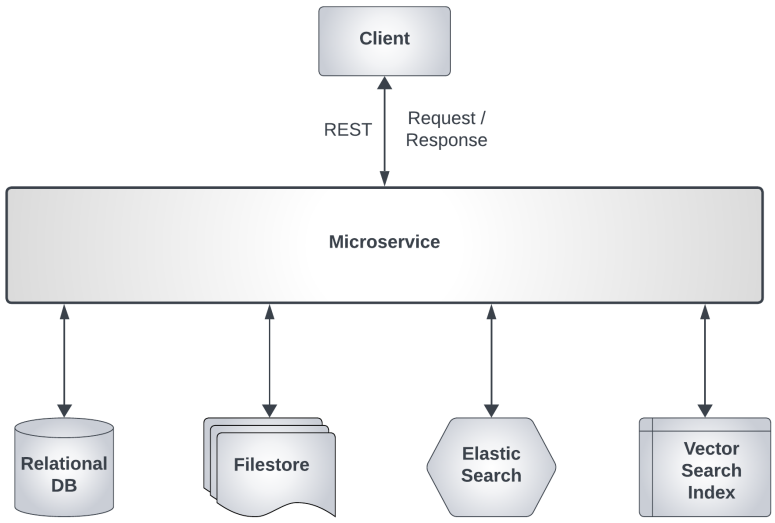
Vector Search Index This derives from an ANN indexing of the latent vectors.[6].

[4] There are alternatives possible: for example, one could argue that a document database, such as MongoDB, is a more appropriate choice. As always in engineering, it is a matter of trade-offs: we leave it as an exercise for the boot-camp participants to pick whichever database orientation they favor for this problem, and experiment. In the team presentations, we expect them to discuss the pros and cons of their choices.

[5] Alternatively, you can use Solr, OpenSearch, etc. if you so prefer.

[6] Approximate k-Nearest Neighbors. High-performance, scalable implementations of k-NN search require some form of indexing of the data into an appropriate structure, such as clusters, or quantization, etc. Such an indexing leads to, as a performance trade-off, a nominal loss of exactitude of search results. Thus these approaches are called ANN, i.e. approximate neighbors search. (The k is implicit and often dropped).

Figure 3.5: The overall search architecture



3.4 Data Pipeline

The data pipeline is a daisy-chain of jobs or steps that start with raw documents of various formats, process them, and eventually produce the vector embeddings and their indexing.

Since this needs to be a scalable pipeline, dealing with **millions** of documents in production, therefore, you must design accordingly.

There are many ways to go about it, and mature open-source frameworks exist for each of these approaches. Consider whichever framework best aligns with your experience. If you are learning a new one, consider Apache Spark[7], perhaps the most pervasive for this purpose.

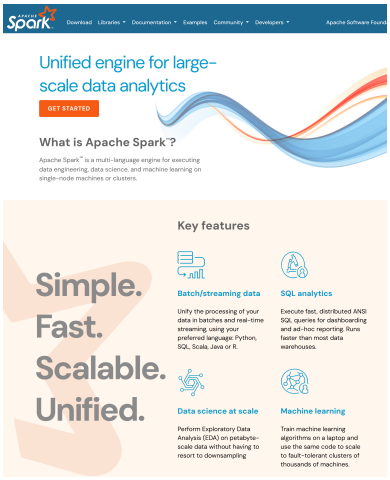


Figure 3.6: Apache Spark: a popular data engineering framework that hands big-data scale transformations.

[7] spark.apache.org

3.4.1 *Plain text extraction*

Documents come in hundreds of formats, such as pdf, google-docs, html, markdown, and so forth. The Sentence-Encoder[8] can, however, only take

[8] A neural architecture that can embed a sentence into a vector space where the mapping has been learned from the semantics of the text.

textual inputs – the “sentences” whose total length is upper-bounded by the permissible token length of the Sentence-Encoder.

Therefore, a crucial part of our data pipeline, early on, has to be a document parser that can extract plain text from the given documents.

In this process, we take a given directory of documents. For each document, we extract the plain text from it, using a document parser. (We recommend using Apache Tika because it can handle over a thousand document formats, and has been a well-established, battle-tested tool for decades. However, feel free to use whatever tool you would like.)

In order to accommodate for the token-length restriction of a Sentence-Encoder, we need to decompose the document into chunks.

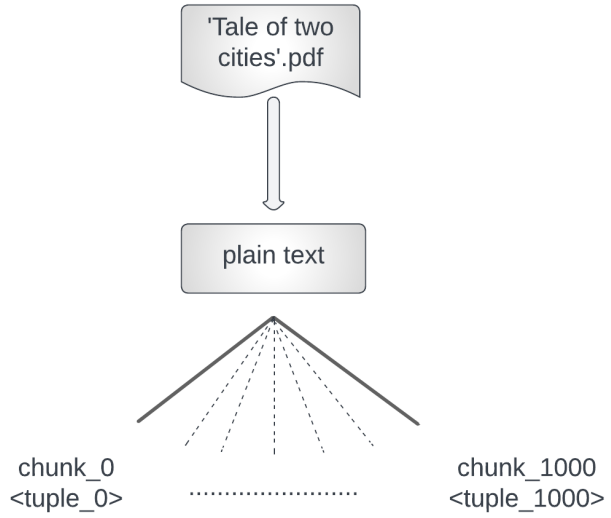


Figure 3.7: Semantic chunking of the document into smaller, meaningful parts that respect the token-length restriction of the Sentence-Encoder.

We can accomplish this efficiently with a distributed parallel-processing framework like Spark/PySpark.

Figure 3.8: The document parsing job, as part of the data pipeline.

Caveat Emptor While it is common to decompose a text into fixed-length chunks, it is perhaps a rather poor way of doing it. Potentially, one may end up splitting mid-sentence or mid-paragraph. Therefore, it is better to break ideally along an appropriate paragraph boundary, so that the chunk-sizes are approximately (and not exactly respected.) In the very least, chunk size should respect sentence boundary. We call our way of careful chunking **Semantic chunking** (perhaps there is a well-known term for this; we leave it as an exercise for the reader to find that out if it exists, and educate the teaching staff!)

Once these chunks have been created, we store them back into the relational datastore.

Important: Remember to move a processed document to another folder (`/processed`) so that you don't keep processing the same file over and over again.

3.4.2 Vector-embedding

Here, we read the chunks from the RDBMS as a collection of tuples. For each textual chunk, we create latent vectors using an appropriate sentence-encoder. Then we store the embeddings back into the relational store.

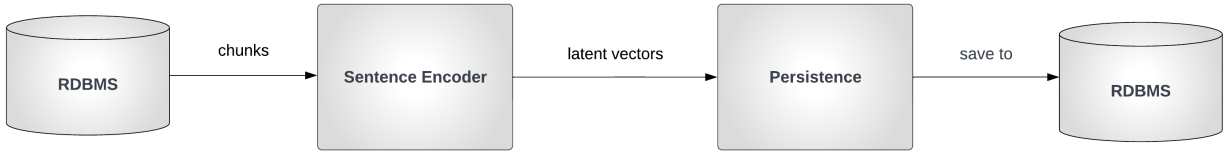


Figure 3.9: Encoding of the text to latent, semantic vectors.

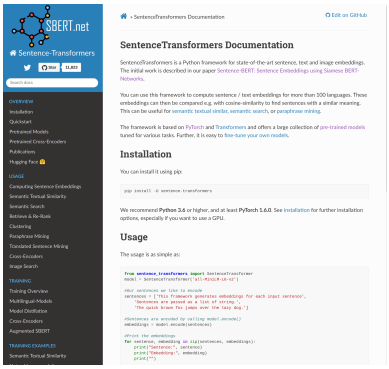


Figure 3.10: Sentence Bert website
[9] huggingface.co/spaces/mteb/leaderboard

Note that the proper choice of a sentence-encoder is essential: you must take the following into consideration:

Model Performance How good is your model of choice at the task you aim to accomplish? Look into the leaderboard for MTEB.[9]

Inference latency How long does it take to produce an embedding for a given sentence? Is that acceptable latency for your production use-case?

Computational load How much computational load does it put on your infrastructure? Is it cost-effective, and does the differential benefit over smaller models justify this cost?

Throughput What is the sustained throughput of this model on your infrastructure? In other words, how many vector embeddings emerge per second? How do you prevent GPU-out-of-memory errors.

Symmetric vs Asymmetric Some models are optimized for symmetric search – i.e. the queries are about the same length as the indexed sentences. On the other hands, if you anticipate that your queries would be far shorter than the text you create embeddings for, then models trained on such asymmetric data is preferable.

Storage & retrieval Each model produces a vector of difference dimension. Consider the impact of different dimensionalities on your storage, retrieval and model performance.

3.4.3 Search Indexing

Now that we have a repository of latent vectors for the document chunks, we need to create a search index for high-performance, low-latency retrieval. For this, a vector indexer that uses an appropriate ANN (Approximate k-Nearest Neighbors) search indexing method. FAISS, SCANN etc. are popular implementations.

At the same time, we need to insert the text into a traditional keyword-search database, such as ElasticSearch.

Figure 3.11: Indexing the latent vectors into an ANN index.

3.4.4 *The complete data processing pipeline*

A concatenation of the above tasks leads to the MLOps data processing pipeline, depicted below:

Figure 3.12: The complete MLOps data processing pipeline for indexing the documents for fast semantic searches.

3.5 Schema design

Consider now that exact document file you encounter will have a document-type and filename. You will use a parser to extract plain-text from it. You will store it in the relational schema, say `documents`. Then you will decompose each of the extracted texts to chunks, with the size of each chunk reasonable enough for your AI models to handle.

With all of these considerations, now design the schema of needed tables. While there are a variety of designs you can arrive at, strive for brevity and simplicity. Exclude extraneous details for now.

3.5.1 Reference schema design

A possible realization of the relational schema would have the following three tables:

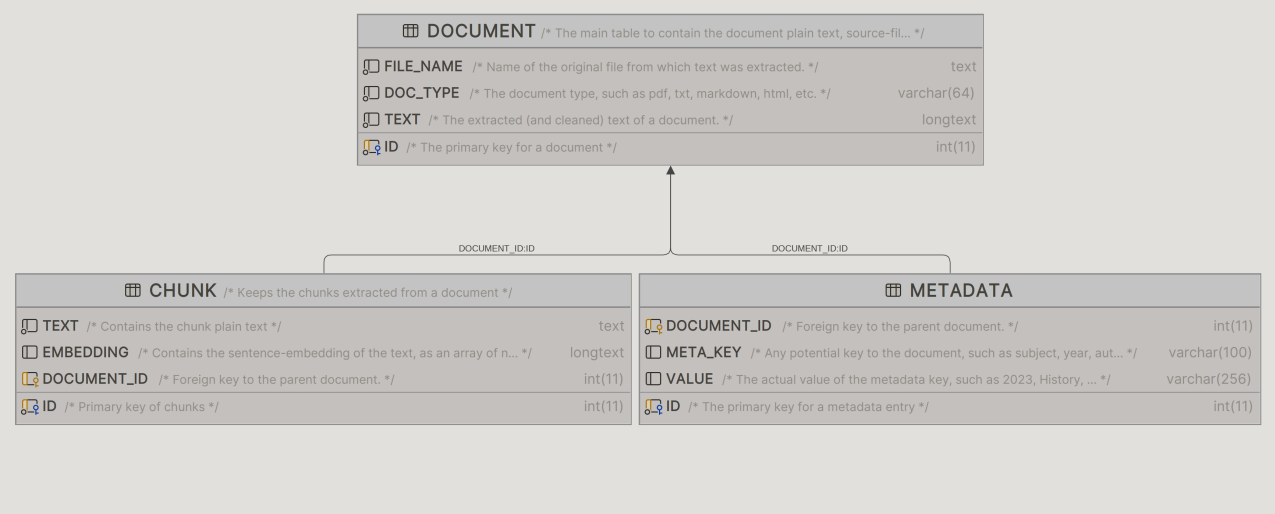


Figure 3.13: Relational schema for storing the extracted plain-text and metadata of a document, along with the chunks it partitions into.

DOCUMENT It contain the name of file, its document-type, and extracted plain-text content.

CHUNK A child table to DOCUMENT, containing all the chunks that the plain-text of a document partitions into.

METADATA Another child table of the DOCUMENT, containing an extensible set of metadata as key-value pairs.

3.5.2 Object-relational mapping

Object-oriented design and relational databases have different reification of the same data: the former in terms of classes and objects, while the latter in terms of tables, columns, rows and foreign-key relationship. To bridge over

the impedance mismatch between the two frameworks, each language has support for mature object-relational mapping libraries.

For Java and its derivative languages, Hibernate[10] has been a popular approach, with other libraries simplifying it further. (Spring[11], and Quarkus-Panache[12]).

For Python, an equivalent one that has emerged is *SqlModel*.[13]. It provides a seamless integration between SQLAlchemy[14] and Pydantic[15]. Recall that SqlAlchemy provides a convenient interface to engage with relational databases, which Pydantic provides, among other things, data-beans, with data-validation support.

[10] hibernate.org/

[11] spring.io/
[12] quarkus.io/guides/hibernate-orm-panache

[13] sqlmodel.tiangolo.com/

[14] www.sqlalchemy.org/
[15] docs.pydantic.dev/

3.5.3 *Directly using SQL templates*

Some engineers eschew object-relational mappings for the additional overhead it brings. Instead, they prefer directly using SQL-templates for the inserts and retrievals. As such, it represents an alternate implementation choice.

3.6 The Semantic Search Service

Create a micro-service that follows the below sequence of events:

- on receiving a REST-api call, it extracts the query text
- uses the Sentence-Encoder you used in the previous section to convert this text into a query vector $\mathbf{X}_q$
- search for the chunk-identifiers in the vector index
- then reach out o the relational database to retrieve the specific chunks and the parent documents
- also run the query through a keywords-based search engine (here, Elastic-Search, Solr, etc – whichever you used)

- aggregate the search results from both the sources (Vector Index, as well as traditional search)
- re-rank the search results using a cross-encoder transformer
- serve the results back as the REST-api response

Ensure that your micro-service is meant to scale well, and can handle a thousand requests per minute.

3.6.1 Phased approach

Take a three-phase approach, where in the first phase, you do the essentials. For example, in the first phase, build the micro-service without traditional search results being added to the results, and without an cross-encoder.

In the second phase, show the results from keywords-search and the vector-search separately in the UI, so you can see the difference.

Finally, in the third phase, implement the re-ranking of the aggregated results from the two sources, and show a single list of results.

3.6.2 Model metrics

How would you evaluate the performance of your search engine? Propose relevant measures, and produce empirical data to show the effectiveness or goodness of your search results.

3.6.3 Performance and scalability

Run experiments to establish the performance and scalability profile of your AI-system, and report the results.

3.6.4 Web client

Create a simple web-client to engage with your AI-search system, and display the results. Functionally, the page should be approximately as depicted in the figure below:

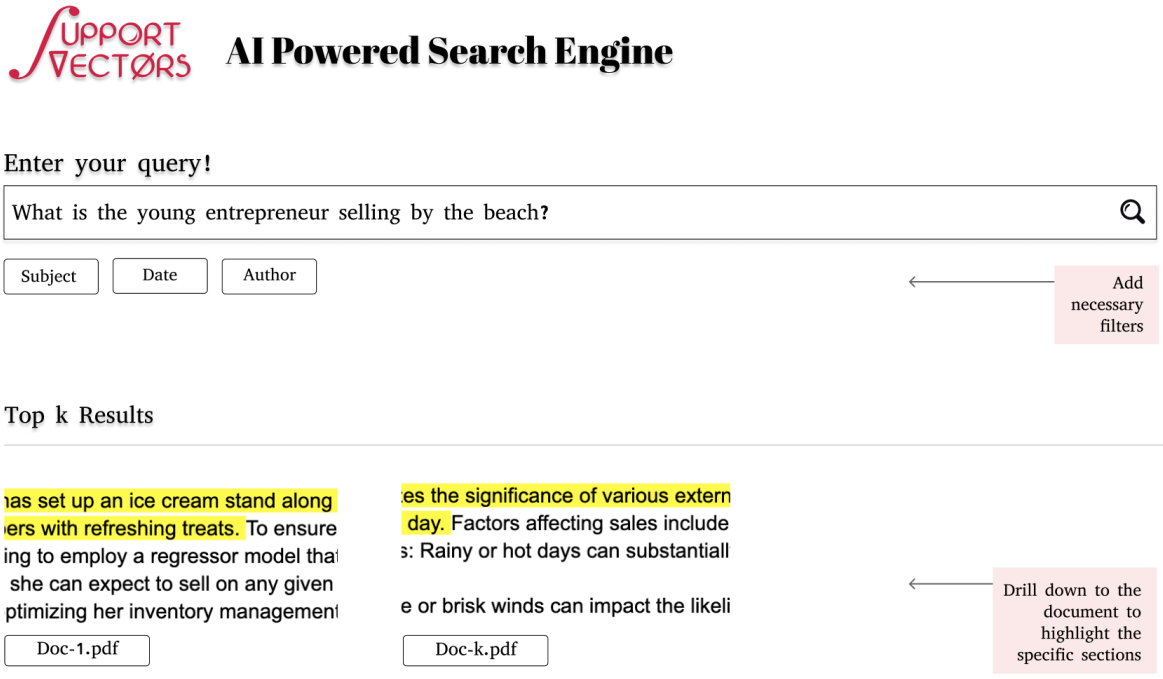


Figure 3.14: A web user interface for the AI-search

"When I have a little money, I buy books; and if I have any left, I buy food and clothes."

Erasmus

4

When the books speak eloquently

Contents

This chapter delves into the second project.

4.1 Introduction

We will release the second project next week – for now, this chapter is a teaser of things to come!

"When I have a little money, I buy books; and if I have any left, I buy food and clothes."

Erasmus

A

Neural activation functions

Contents

This is a quick summary of the various neural activation functions.

A.1 Introduction

[1] **lecun1998efficient**

Sigmoid (Logistic) Activation Function [1]

- Formula: $f(x) = \frac{1}{1+e^{-x}}$
- Strengths: Smooth gradient, output values between 0 and 1, simple mathematical form.
- Weaknesses: Vanishing gradient problem, not zero-centered.
- Uses: Binary classification problems, output layer.

Tanh Activation Function

- Formula: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Strengths: Outputs between -1 and 1, zero-centered.
- Weaknesses: Vanishing gradient problem.
- Uses: Hidden layers.
- Reference: Not explicitly covered by the provided references.

[2] **nair2010rectified**

ReLU Activation Function [2]

- Formula: $f(x) = \max(0, x)$
- Strengths: Mitigates vanishing gradient, computationally efficient.
- Weaknesses: Dying ReLU problem.
- Uses: Convolutional neural networks, deep learning.

[3] **maas2013rectifier**

Leaky ReLU Activation Function [3]

- Formula: $f(x) = x$ for $x > 0$ and $f(x) = \alpha x$ for $x \leq 0$, where α is a small constant.
- Strengths: Addresses dying ReLU issue, introduces small negative slope.
- Weaknesses: Hyperparameter α may need tuning.
- Uses: Convolutional neural networks, deep learning.

[4] **he2015delving**

Parametric ReLU (PReLU) Activation Function [4]

- Formula: Similar to Leaky ReLU, but α is learned from data.
- Strengths: More flexible than Leaky ReLU.
- Weaknesses: Risk of overfitting on small datasets.

- Uses: Convolutional neural networks.

Exponential Linear Unit (ELU) Activation Function [5]

[5] **clevert2015fast**

- Formula: $f(x) = x$ for $x > 0$ and $f(x) = \alpha(e^x - 1)$ for $x \leq 0$.
- Strengths: Addresses dying ReLU issue, saturates for negative inputs.
- Weaknesses: Slower computation than ReLU.
- Uses: Convolutional neural networks, deep learning.

Swish Activation Function [6]

[6] **ramachandran2017searching**

- Formula: $f(x) = x \cdot \text{sigmoid}(\beta x)$, where β is a learnable parameter.
- Strengths: Smooth, non-monotonic function.
- Weaknesses: Computational complexity due to sigmoid.
- Uses: Deep neural networks.

Maxout Activation Function [7]

[7] **goodfellow2013maxout**

- Formula: $f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$
- Strengths: Generalizes ReLU and its variants.
- Weaknesses: Increases the number of parameters.
- Uses: Neural networks to replace traditional activations.

Softplus Activation Function [8]

[8] **dugas2001incorporating**

- Formula: $f(x) = \log(1 + e^x)$
- Strengths: Smooth approximation to ReLU.
- Weaknesses: Can suffer from saturation.
- Uses: Neural networks as an alternative to ReLU.

Gaussian Error Linear Unit (GELU) Activation Function [9]

[9] **hendrycks2016gaussian**

- Formula: Approximated as $f(x) = 0.5x(1 + \tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$
- Strengths: Offers improved accuracy.
- Weaknesses: Computationally more intensive.
- Uses: Neural networks for improved performance.

Sinusoid Activation Function [10]

[10] **agostinelli2014learning**

- Formula: $f(x) = \sin(x)$
- Strengths: Oscillatory, captures periodic patterns.
- Weaknesses: Not commonly used, can cause instability.
- Uses: Experimental neural network architectures.

SoftExponential Activation Function

- Formula: Varies based on parameter α, for example $f(x) = -\frac{\log(1-\alpha(x+\alpha))}{\alpha}$ for $\alpha < 0$.
- Strengths: Generalizes several activation functions.
- Weaknesses: Complexity due to different forms.
- Uses: Neural networks for varied activations.
- Reference: Not explicitly covered by the provided references, but mentioned in various sources.

neural activation

There were 8 pages in this document.

FEEDBACK

Please send any feedback on this document, or report errors to the author at asif@supportvectors.com