

Day 3: Add Search

Welcome to Day 3 of our [7-Day AI Agents Email Crash-Course](#).

In the first part of the course, we focus on data preparation. Before we can use data for AI agents, we need to prepare it properly.

We have already downloaded the data from a GitHub repository. Yesterday (Day 2), we processed it by chunking it where necessary.

Now it's time to use this data. We will index this data by putting it inside a search engine. This allows us to quickly find relevant information when users ask questions.

In particular, we will:

- Build a lexical search for exact matches and keywords
- Implement semantic search using embeddings
- Combine them with a hybrid search

At the end of this lesson, you'll have a working search system you can query about your project. This search engine can be used later by the AI agent to look up user questions in the database.

1. Text search

The simplest type of search is a text search. Suppose we build a Q&A system for courses (using the FAQ dataset). We want to find the answer to this question:

"What should be in a test dataset for AI evaluation?"

Text search works by finding all documents that contain at least one word from the query. The more words from the query that appear in a document, the more relevant that document is.

This is how modern search systems like Apache Solr or Elasticsearch work. They use indexes to efficiently search through millions of documents without having to scan each one individually.

In this lesson, we'll start with a simple in-memory text search. The engine we will use is called [minsearch](#).

Note: This search engine was implemented as part of a workshop I held some time ago. You can find details [here](#) if you want to know how it works.

Let's install it:

```
Shell
uv add minsearch
```

We will use it for chunked Evidently docs.

To remind you, this is how we prepared the docs:

```
Python
evidently_docs = read_repo_data('evidentlyai', 'docs')

evidently_chunks = []

for doc in evidently_docs:
    doc_copy = doc.copy()
    doc_content = doc_copy.pop('content')
    chunks = sliding_window(doc_content, 2000, 1000)
    for chunk in chunks:
        chunk.update(doc_copy)
    evidently_chunks.extend(chunks)
```

You can find `read_repo_data` in the first lesson and `sliding_window` in the second lesson.

Let's now index this data with minsearch:

```
Python
from minsearch import Index

index = Index(
    text_fields=["chunk", "title", "description", "filename"],
    keyword_fields=[]
)

index.fit(evidently_chunks)
```

Here we create an index that will search through four text fields: chunk content, title, description, and filename. The `keyword_fields` parameter is for exact matches (we don't need it for now).

We can now use it for search:

```
Python
query = 'What should be in a test dataset for AI evaluation?'
results = index.search(query)
```

For DataTalksClub FAQ, it's similar, except we don't need to chunk the data. For the data engineering course, it'll look like this:

```
Python
dtc_faq = read_repo_data('DataTalksClub', 'faq')

de_dtc_faq = [d for d in dtc_faq if 'data-engineering' in d['filename']]

faq_index = Index(
    text_fields=["question", "content"],
    keyword_fields=[]
)

faq_index.fit(de_dtc_faq)
```

Here we filter for only data engineering FAQ entries. We search through the question and content (answer) text fields.

The result will look like this:

```
Python
[{'id': '3f1424af17',
  'question': 'Course: Can I still join the course after the start date?',
  'content': "Yes, even if you don't register, you're still eligible ...",
},
 {'id': '068529125b',
  'question': 'Course - Can I follow the course after it finishes?',
  'content': 'Yes, we will keep all the materials available, so you can ...',
}
...
]
```

This is text search, also known as "lexical search". We look for exact matches between our query and the documents.

2. Vector search

Text search has limitations. Consider these two queries:

- "I just discovered the program, can I still enroll?"
- "I just found out about the course, can I still join?"

These ask the same question but share no common words (among important ones). Text search would fail to find relevant matches.

This is where embeddings help. Embeddings are numerical representations of text that capture semantic meaning. Words and phrases with similar meanings have similar embeddings, even if they use different words.

Vector search uses these embeddings to identify semantically similar documents, rather than just exact word matches.

For vector search, we need to turn our documents into vectors (embeddings).

We will use the sentence-transformers library for this purpose.

Install it:

```
Shell
uv add sentence-transformers
```

Let's use it:

```
Python
from sentence_transformers import SentenceTransformer
embedding_model = SentenceTransformer('multi-qa-distilbert-cos-v1')
```

The `multi-qa-distilbert-cos-v1` model is trained explicitly for question-answering tasks. It creates embeddings optimized for finding answers to questions.

Other popular models include:

- `all-MiniLM-L6-v2` - General-purpose, fast, and efficient
- `all-mpnet-base-v2` - Higher quality, slower

Check [Sentence Transformers documentation](#) for more options.

This is how we use it:

```
Python
record = de_dtc_faq[2]
text = record['question'] + ' ' + record['content']
v_doc = embedding_model.encode(text)
```

We combine the question and answer text, then convert it to an embedding vector.

Let's do the same for the query:

```
Python
query = 'I just found out about the course. Can I enroll now?'
v_query = embedding_model.encode(query)
```

This is how we compute similarity between the query and document vectors:

```
Python
similarity = v_query.dot(v_doc)
```

The dot product measures similarity between vectors.

Values closer to 1 indicate higher similarity, closer to 0 means lower similarity. This works because the model creates normalized embeddings where cosine similarity equals the dot product.

So we can create embeddings for all documents, then compute similarity between the query and each document to find the most similar ones.

This is what `VectorSearch` from `minsearch` does. Let's use it.

First, we turn our docs into embeddings. This process takes time, so we'll monitor progress with tqdm:

```
Python
from tqdm.auto import tqdm
import numpy as np

faq_embeddings = []

for d in tqdm(de_dtc_faq):
    text = d['question'] + ' ' + d['content']
    v = embedding_model.encode(text)
    faq_embeddings.append(v)

faq_embeddings = np.array(faq_embeddings)
```

We combine question and answer text for each FAQ entry. We convert the list to a NumPy array for efficient similarity computations.

Now let's use **VectorSearch**:

```
Python
from minsearch import VectorSearch

faq_vindex = VectorSearch()
faq_vindex.fit(faq_embeddings, de_dtc_faq)
```

This creates a vector search index using our embeddings and original documents.

Let's use it now:

```
Python
query = 'Can I join the course now?'
q = embedding_model.encode(query)
results = faq_vindex.search(q)
```

We first create an embedding for our query (**q**), then search for similar document embeddings.

You can easily do the same with the Evidently docs (but only use the `chunk` field for embeddings):

```
Python
evidently_embeddings = []

for d in tqdm(evidently_chunks):
    v = embedding_model.encode(d['chunk'])
    evidently_embeddings.append(v)

evidently_embeddings = np.array(evidently_embeddings)

evidently_vindex = VectorSearch()
evidently_vindex.fit(evidently_embeddings, evidently_chunks)
```

3. Hybrid search

Text search is fast and efficient. It works well for exact matches and specific terms, and requires no model inference. However, it misses semantically similar but differently worded queries and struggles to handle synonyms effectively.

Vector search captures semantic meaning and handles paraphrased questions. It works with synonyms and related concepts. But it may miss exact keyword matches.

Combining both approaches gives us the best of both worlds. This is known as "hybrid search."

The code is quite simple:

```
Python
query = 'Can I join the course now?'

text_results = faq_index.search(query, num_results=5)

q = embedding_model.encode(query)
vector_results = faq_vindex.search(q, num_results=5)

final_results = text_results + vector_results
```

4. Putting this together

Our search is implemented!

But before we can use it in our agent, we need to organize the code. Let's put all the code into different functions:

```
Python
def text_search(query):
    return faq_index.search(query, num_results=5)

def vector_search(query):
    q = embedding_model.encode(query)
    return faq_vindex.search(q, num_results=5)

def hybrid_search(query):
    text_results = text_search(query)
    vector_results = vector_search(query)

    # Combine and deduplicate results
    seen_ids = set()
    combined_results = []

    for result in text_results + vector_results:
        if result['filename'] not in seen_ids:
            seen_ids.add(result['filename'])
            combined_results.append(result)

    return combined_results
```

5. Selecting the best approach

We have seen 3 approaches: text search, vector search, and hybrid search. You may wonder, how do I select the best one? We will discuss evaluation methods later in the course.

But like with chunking, you should always start with the simplest approach. For search, that's text search. It's faster, easier to debug, and works well for many use cases. Only add complexity when a simple text search isn't sufficient.

But let's first build our agent! Our data is ready. Tomorrow, we will build a conversational agent that can answer questions based on the data we collected.

If you have suggestions about the course content or want to improve something, let me know!

Homework


- For the project you selected, index the data
- Experiment with text and vector search
- Which approach makes sense for your application? Manually inspect the results and analyze what works best
- Make a post on social media about what you're building

Learning in Public

Instead of keeping your progress private, share assignments, reflections, and projects online.

Explaining what you've learned helps you understand it better, builds confidence, and creates visible proof of your skills.

Example post for LinkedIn

 Day 3 of building my AI agent: implementing search!


Today I built three types of search for my AI system:

- Text search for exact keyword matches
- Vector search for semantic similarity
- Hybrid search combining both approaches

We should start with simple text search, and add semantic search only when text search is not enough.

My project focuses on [YOUR PROJECT DESCRIPTION]. [Text/Vector/Hybrid] search works best because [YOUR REASONING].

Here's my repo: [YOUR_REPO_LINK]

Next up: Building the actual conversational agent! 




Following along with this amazing course - who else is building AI agents?

You can sign up here: <https://alexeygrigorev.com/aihero/>


Example post for Twitter/X

Day 3: Search engine for my AI agent 

3 search methods implemented:

-  Text search (exact matches)
-  Vector search (semantic similarity)
-  Hybrid search (best of both)

Here's my repo: [YOUR_REPO_LINK]

Tomorrow: Building the conversational agent 

Join me: <https://alexeygrigorev.com/aihero/>

Community

Have questions about this lesson or suggestions for improvement? You can find me and other learners in **DataTalks.Club Slack**:

- [Join DataTalks.Club](#)
- Find us in the [#course-ai-bootcamp channel](#)

In the community channel, you can:

- Ask questions about the course content
- Share your implementation and get feedback
- Show off your GitHub repositories
- Suggest improvements to the course materials
- Connect with other course participants

Don't hesitate to reach out - the community is here to help each other succeed!

