How to Create a Search Engine

An Introduction to Information Retrieval Systems

Introduction

Data is often referred to as the new oil, and just like oil, it's rather useless without processing and refinement. Information Retrieval (IR) systems are crucial in this context, and act as a gateway to knowledge and empowering individuals.

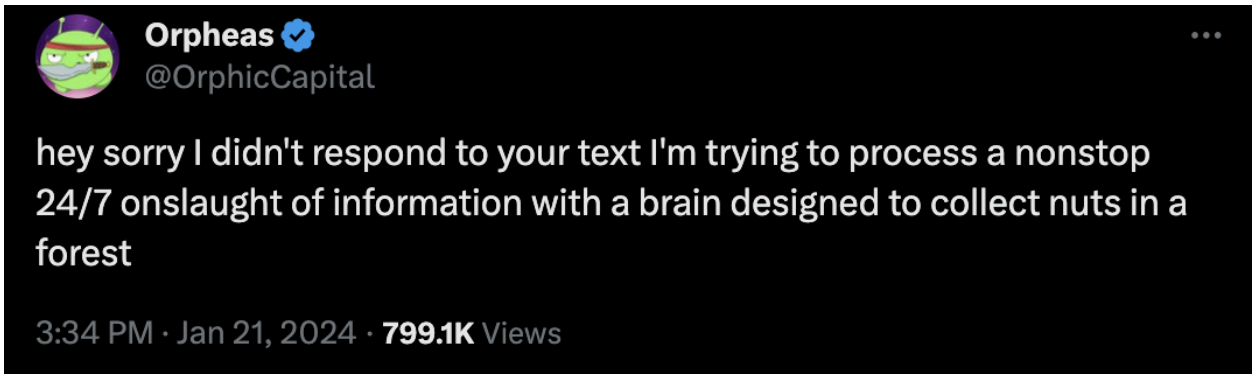
You would be familiar with these systems:

- Google
- ChatGPT
- Document finder
- Keyword search in favorite application

In this article, I will introduce how various IR systems function and demonstrate two possible implementations.

Problem

As a research-aholic, I spend more time than I'd like sifting through useless information.



In the beginning of my Tangible AI internship I came across a corpus of books and transcribed lectures from a particular health niche. Due to the sheer volume, I sought guidance online to help distill the important parts.

Fortunately, I found several communities and individuals willing to share their domain expertise. I was exposed to a myriad of efforts attempting to solve the same problem I faced. These solutions consisted of basic keyword searches.

This sparked an idea for my final internship project – to create a semantic search engine.

Search Models

Before jumping to technicalities, it's important to understand the how's and what's of IR systems.

Models mostly fit into one of these 3 categories:

- Set theoretic (boolean)
- Algebraic (vector space)
- Probabilistic

My project will explore algebraic models.

Set Theoretic (Boolean)

Boolean models are the most basic form of search – often found in keyword search engines.

These models represent documents and queries as mathematical sets of terms or phrases. On these sets, boolean logic (AND, OR, NOT) is used to either include or exclude certain terms from documents.

We can imagine an 'AND' search example where we want our results to include BOTH "machine learning" and "healthcare" in the retrieved documents.

Each model has its tradeoffs:

Pros:

- Simple and effective
- Efficient on large databases
- High precision and recall when exact terms are used

Cons:

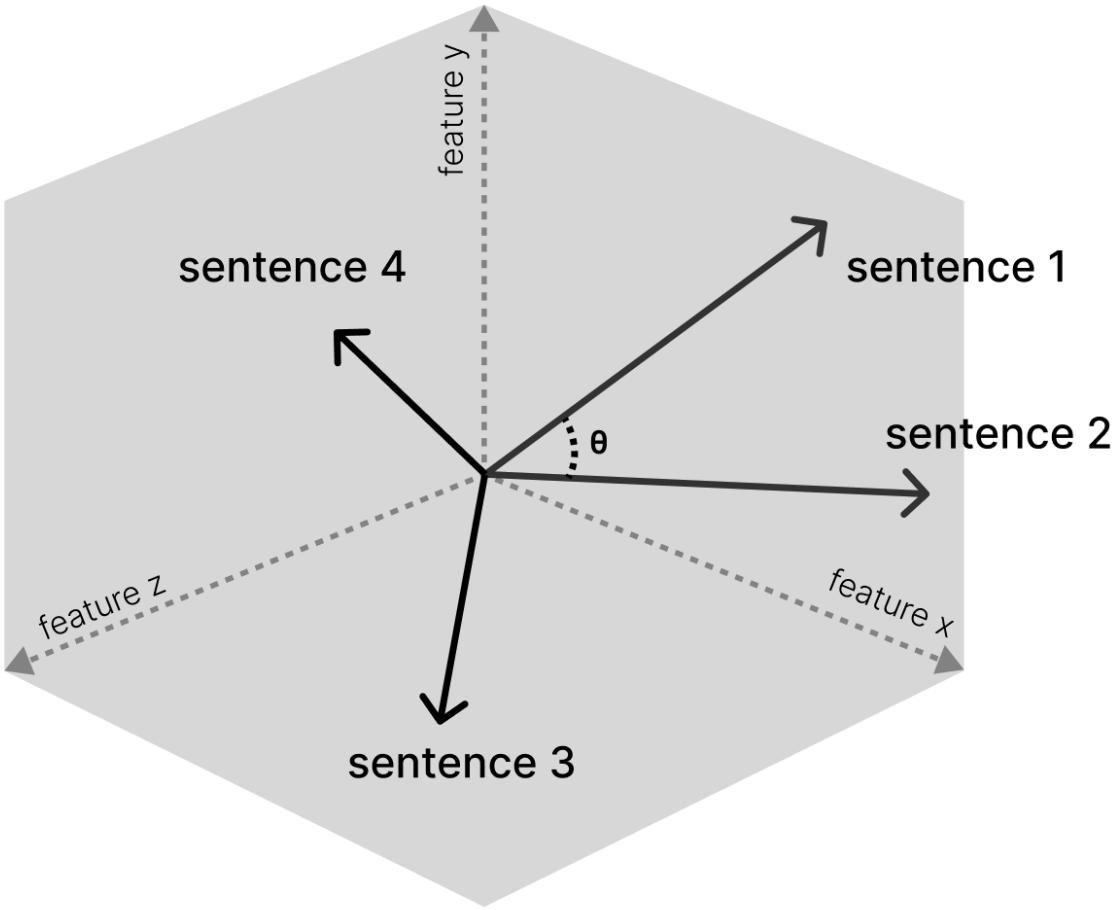
- No ranking by relevance
- Sensitive to misspellings and incapable of handling synonyms
- Incapable of understanding context

Algebraic (VSM)

Vector space models are a step up in complexity and capability – a staple in modern search.

These models represent documents and queries as vectors in multi-dimensional space. We calculate the cosine similarity between document and query vectors to rank them accordingly. Vectors with a higher cosine similarity are likely to be semantically or contextually related.

$$cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$



I experimented with two types of vector models:

- Term Frequency - Inverse Document Frequency
- Embeddings generated with BERT

Pros

- Contextual relevance
- Ranking capability
- Flexibility in handling misspellings and synonyms

Cons

- Computationally more expensive
- Issues with term-weighting and semantic understanding in TF-IDF models

Probabilistic

Probabilistic models offer a more statistically grounded approach – focusing on the likelihood of relevance between document and query.

There are several types of probabilistic models:

- Binary Independence Model
- Probabilistic relevance model
- Language models
- Latent Dirichlet allocation

In simpler models, documents and queries are represented as statistical profiles. While similar to TF-IDF, they capture more complex inter-term relationships.

Human feedback can play an important role in some probabilistic models. As users engage with search results, their interactions can subtly update the model, further improving accuracy and relevance.

Pros

- Adaptability and personalisation
- Capable of delivering highly relevant results (precision)

Cons

- Complexity

TF-IDF

With a basic understanding of IR systems, let's dive into TF-IDF and embedding models.

With the goal of improving the existing keyword searches – I planned my project in two stages. First, to build and evaluate a TF-IDF model and then compare a more sophisticated BERT embedding implementation.

Explanation

In the early stages of development, I chose to define documents as individual sentences. The goal was to provide a similar user experience to the existing keyword searches. By selecting sentences as the unit of search, I'd ensure specific enough results.

Term Frequency - Inverse Document Frequency (TF-IDF) is a statistical measure used to convert documents into a numeric representation. It evaluates how relevant a term is to a document and corpus.

This approach generates sparse high-dimensional vectors to represent each document. The vectors have a positional index for each unique term in the corpus' vocabulary. The vector dimensionality mirrors the size of our vocabulary.

These vectors are sparse because a document is unlikely to capture a large proportion of the vocabulary. For the positions that have terms, we calculate a TF-IDF score.

$$TFIDF = TF(t, d) \times IDF(t)$$

$$TFIDF = (\frac{freq\, of\ t\ in\, doc\, d}{no.\ of\ terms\ in\, doc\, d}) \times \log(\frac{no.\ of\, docs}{no.\ of\, docs\, containing\ t})$$

The math of this model breaks down into the product of two components:

Term Frequency:

The ratio between the frequency of any term “t” and the total terms in the given document (sentence).

Inverse Document Frequency:

A constant that represents the number of documents in a corpus that include the term “t”. Terms that are common across the corpus have a lower IDF value. This emphasizes the corpus specific words.

It’s easiest to understand this concept in the form of an example:

With the following corpus

$$d_1 = the\, cat\, sat\, on\, mat$$

$$d_1 = the\, dog\, sat\, on\, log$$

We take each vector and find the Term Frequencies

$$\begin{matrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ [the, & cat, & sat, & on, & mat] \end{matrix}$$

Then we create a vocabulary and count the occurrence of each term across the corpus

$$\begin{matrix} 2 & 1 & 1 & 2 & 2 & 1 & 1 \\ [the, & cat, & dog, & sat, & on, & mat, & log] \end{matrix}$$

We then plug these values into our equation and create a vector for d_1. This process is repeated for all documents in the corpus.

$$d_1 = [the,\ cat,\ \varnothing,\ sat,\ on,\ mat,\ \varnothing]$$

$$\downarrow$$

$$\vec{d}_1 = \frac{1}{5}\left[\log(\tfrac{2}{2}),\ \log(\tfrac{2}{1}),\ \varnothing,\ \log(\tfrac{2}{2}),\ \log(\tfrac{2}{2}),\ \log(\tfrac{2}{1}),\ \varnothing\right]$$

$$\downarrow$$

$$\vec{d}_1 = \left[0,\ \tfrac{\log(2)}{5},\ 0,\ 0,\ 0,\ \tfrac{\log(2)}{5},\ 0\right]$$

Pipeline

Now that we understand the model, let's implement it with code.

The pipeline comprises of 3 general steps:

- Data processing
- Document vectorisation
- Cosine similarity search

Processing

After obtaining our text files, we can write a small function to open, read and process the corpus.

After instantiating the spaCy “en_core_web_sm” language model, we process the files contents and use list comprehension to separate the content into individual sentences.

Lastly, we iterate over these sentences and append them to a Pandas DataFrame for efficient computation.

```
DATA_DIR = Path.cwd() / "data"
```

```
data = []

    for filename in os.listdir(DATA_DIR):
        if filename.endswith(".txt"):
            print(filename)

            # Create the file path
            file_path = DATA_DIR / filename
            with open(file_path, "r", encoding="utf-8") as file:
                content = file.read()
                # Use spaCy to tokenize the content into sentences
                doc = nlp(content)
                sentences = [sent.text.strip() for sent in doc.sents]
                # Append each sentence to your data list, along with the
filename
                for sentence in sentences:
                    data.append({"filename": filename, "sentence":
sentence})


    df = pd.DataFrame(data)
```

Vectorising

Our next step is to convert the DataFrame of documents into a matrix of TF-IDF vectors.

Sklearn's "TfidfVectorizer" class provides an efficient method for doing this.

The class is instantiated with the following hyper-parameters:

- max_df, min_df: Ignore corpus specific words above or below a specific proportion
- ngram_range: the range of n-grams extracted from the documents
- tokenizer: override the default string tokenization step

Selecting all sentences from the DataFrame, we apply the "fit_transform" function. This will perform the following steps:

- Tokenization
- Vocabulary building – tokens that don't meet the min/max_df criteria are excluded
- Calculate term frequencies of each document
- Compute and apply inverse document frequency
- Generate the TF-IDF matrix

```
from sklearn.feature_extraction.text import TfidfVectorizer
max_df = 0.85
min_df = 0.0
ngram_range = (2, 3)

def custom_tokenizer(text):
    doc = nlp(text)
    return [token.lemma_ for token in doc]

# Initializing the vectorizer with hyperparameters
vectorizer = TfidfVectorizer(tokenizer=custom_tokenizer, max_df=max_df,
min_df=min_df, ngram_range=ngram_range)

# Applying the vectorizer to the dataset
tfidf_matrix = vectorizer.fit_transform(df['sentence'])
```

Search

The search function follows this procedure:

- Vectorize user's search query
- Calculate cosine similarity
- Return list of top-k results

The "transform" method performs a subset of the operations "fit_transform" completes, notably omitting the vocabulary building step.

We then pass our tfidf_matrix and query_vector to Sklearn's "cosine_similarity" function. This calculates the angle between our query and dataset vectors.

```
from sklearn.metrics.pairwise import cosine_similarity

def search(query, vectorizer, tfidf_matrix, df):
    # Creating a vector representation of our search query
    query_vector = vectorizer.transform([query])
    # Using sklearn package to perform cosine_similarity search
    similarities = cosine_similarity(query_vector, tfidf_matrix)
    top_indices = similarities.argsort()[0][-20:]
    # Retrieve the corresponding rows from the DataFrame
    top_docs = df.iloc[top_indices]
    return top_docs
```

Evaluation explanation

A rigorous evaluation process is arguably the most important piece in any Data Science pipeline. Without it, adjusting hyper-parameters won't yield any actionable insight.

There are several metrics available to use when evaluating an IR system. My goal was to find relevant documents, and so recall is what I optimized for.

Recall

Recall measures how many relevant documents your search engine is able to find. For example, if there are 10 relevant documents in your dataset for a specific query, a 50% recall returns 5 of those.

To perform this test we need to compile a dataset mapping queries to relevant sentences.

First, I chose 5 queries to broadly represent corpus data. An example from my project being, "Is salt healthy?".

With the query in mind, I used a keyword search to find representative targets. Due to the size of the dataset, selecting 5-10 examples for each query took considerably longer than I expected.

The pipeline for testing my model follows this procedure:

- Search the dataset using one of the queries
- Tally the results that intersect with the manually selected targets

The hyper-parameters and recall metrics are saved to a table.

Results

max_df, min_df, ngram_range	0.85, 0, (1,1)	0.5, 0.1, (1,1)	0.85, 0, (1,2)	0.85, 0, (1,3)	0.85, 0, (2,3)
Recall	8	0	4	5	5

We can see that increasing n-gram range and restricting the vocabulary had deleterious effects on recall. I found these results suffice, but It would be interesting to expand the results with alternative document frequency (df) ranges.

Embedding

With a strong baseline established, we now shift our focus to evaluating a more modern transformer-based architecture.

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language model that revolutionized NLP after its 2018 debut. It surpassed its predecessors in various benchmarks, crowning itself as the state-of-the-art for its time.

Explanation

The features generated by NN models are far more sophisticated than TF-IDF. In a fashion analogous to the brain, BERT was trained on massive amounts of text data, learning patterns and contextual relationships between words.

When encoding our documents with BERT, it uses the learned knowledge to generate dense vectors, with each dimension carrying rich information.

Pipeline

While most of our TF-IDF pipeline remains unchanged, we need to swap the TF-IDF feature generation with BERT.

Using the SentenceTransformer framework, we instantiate a distilled BERT model optimized for sentence embeddings.

From our DataFrame we extract the “sentence” column and convert it to a list of individual sentence strings. The “encode” method is called with this list, generating a 384 dimension vector for each document.

```
model = SentenceTransformer("all-MiniLM-L6-v2")

def generate_embeddings(df):
    embeddings = model.encode(df["sentence"].tolist(),
show_progress_bar=True)
    return embeddings
```

Results

Without any pre-processing, we see an 88% improvement in recall from 8 – to 15 with BERT embeddings. Quite impressive.

Conclusion

I had a lot of fun developing this project and I'm extremely grateful to Hobson for his mentorship. I encourage everyone to read Natural Language Processing in Action.

You can find my code at https://github.com/earcherc/vector-search