alexmolas.com

A search engine in 80 lines of Python

17-22 minutes

February 05, 2024 · 26 mins · 4718 words

Last September I hopped on board with Wallapop as a Search Data Scientist and since then part of my work has been working with Solr, an open source search engine based on Lucene. I've got the basics of how a search engine works, but I had this itch to understand it even better. So, I rolled up my sleeves and decided to build one from scratch.

Now, let's talk goals. Ever heard of the "Small Website Discoverability Crisis"? The problem it's basically that small websites, ones like this one, are impossible to be found using Google or any other search engine. My mission? Making those tiny websites great again. I believe in bringing back the glory of the little guys, away from the Google SEO frenzy.

In this post I will walk you through the journey of buliding a search engine from scratch using Python. As usual, all the code I've written can be found on my GitHub (microsearch repo). This implementation doesn't pretend to be a production-ready search engine, just a usable toy example showing how a search engine works under the hood.

Also, let me be sincere and admit I've exaggerated a little bit in the post title. Indeed, the search engine I've implemented is around 80 lines of Python, but I've also written some complementary code (data crawler, API, HTML templates, etc.) that's over makes the whole project a bit bigger. However, I think the interesting part of this project is the search engine which has less than 80 lines.

PS. After writing this post and `microsearch` I realized that Bart de Goede did something similar a couple of years ago. My implementation is very similar to Bart's, but in my case, I think I did some things better, in particular (1) my crawler is async, which makes things much faster, and (2) I've implemented a user interface that allows to interact with the search engine.

microsearch

Now, let's delve into the components that make up `microsearch` and explore how I crafted each element: (1) the crawler, (2) the inverted index, (3) the ranker, and (4) the interface. In the following sections, I'll provide both theoretical descriptions and practical details on how each concept was implemented in my project.

Crawler

The first step to building a search engine is to have data to search. Depending on your use case you can crawl existing data (as Google does) or you

can use your own data (as Wallapop or any other e-commerce/marketplace does).

Since one of my intentions was to build a “local Google” I decided to use data from the blogs I follow to build the search engine. In this case, crawling consists of downloading and cleaning all the posts of a certain list of blogs. To make it easier I’ve only crawled posts of blogs with RSS [1]. And to make it faster, I’ve used the `asyncio` Python library. Using asynchronous code has sped up the crawling time from 20 minutes to 20 seconds.

In my case, I’ve used a list of 642 RSS feeds. Of these feeds around 100 are the ones I usually read (blogs about ML, data science, math, etc.), and I scrapped the other 500 from surprisetalk blogs.hn project.

▼ Crawler code

```
import argparse
import aiohttp
import asyncio
import feedparser
import pandas as pd
from bs4 import BeautifulSoup
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def parse_feed(feed_url):
    try:
        feed = feedparser.parse(feed_url)
        return [entry.link for entry in feed.entries]
    except Exception as e:
        print(f"Error parsing feed {feed_url}: {e}")
        return []


async def fetch_content(session, url):
    async with session.get(url) as response:
        return await response.text()
```

```
async def process_feed(feed_url, session, loop):
    try:
        post_urls = await loop.run_in_executor(None, parse_feed, feed_url)
        tasks = [fetch_content(session, post_url) for post_url in post_urls]
        post_contents = await asyncio.gather(*tasks)
        cleaned_contents = [clean_content(content) for content in post_contents]
        return list(zip(post_urls, cleaned_contents))
    except Exception as e:
        print(f"Error processing feed {feed_url}: {e}")
        return []


def clean_content(html_content):
    soup = BeautifulSoup(html_content, "html.parser")
    for script in soup(["script", "style"]):
        script.extract()
    text = soup.get_text()
    lines = (line.strip() for line in text.splitlines())
    chunks = (phrase.strip() for line in lines for phrase in line.split("  "))
    cleaned_text = " ".join(chunk for chunk in chunks if chunk)
    return cleaned_text


def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--feed-path")
    return parser.parse_args()


async def main(feed_file):
    async with aiohttp.ClientSession() as session:
        loop = asyncio.get_event_loop()
        with open(feed_file, "r") as file:
            feed_urls = [line.strip() for line in file]
```

```
        tasks = [process_feed(feed_url, session, loop) for feed_url in feed_urls]
        results = await asyncio.gather(*tasks)

    flattened_results = [item for sublist in results for item in sublist]
    df = pd.DataFrame(flattened_results, columns=["URL", "content"])
    df.to_parquet("output.parquet", index=False)


if __name__ == "__main__":
    args = parse_args()
    asyncio.run(main(args.feed_path))
```

Inverted index

An inverted index is a data structure that maps keywords to documents. This data structure makes it trivial to find documents where a certain word appears. When a user searches for some query the inverted index is used to retrieve all the documents that match with the keywords in the query.

To implement the inverted index I've used a `defaultdict` with the signature `dict[str, dict[str, int]]`. This is, a mapping that given a word (a `str`) returns another mapping from URL (a `str`) to the number of times that word appears in the URL (a `int`). The default value of the mapping is a mapping from URL to `0`, so if we try to get the value of a keyword that doesn't exist in a URL we get a zero.

The logic of the inverted index is defined within a class called `SearchEngine`. We initialize it with two private dicts.

```
class SearchEngine:
    def __init__(self):
        self._index: dict[str, dict[str, int]] = defaultdict(lambda: defaultdict(int))
        self._documents: dict[str, str] = {}
```

Then we implement the `index` methods, which receives an URL and its content, normalizes the content (ie: remove punctuation, everything to lowercase, etc.), and then add it to the index.

```
def index(self, url: str, content: str) -> None:
    self._documents[url] = content
    words = normalize_string(content).split(" ")
    for word in words:
        self._index[word][url] += 1
```

To make the indexing process more usable we can implement a bulk index option, which receives a list of URLs and documents and index them.

```
def bulk_index(self, documents: list[tuple[str, str]]):
```

```
    for url, content in documents:
        self.index(url, content)
```

Finally, we can read the index using the `get_url` method, which receives a keyword and returns the URLs that contain the keyword.

```
def get_urls(self, keyword: str) -> dict[str, int]:
    keyword = normalize_string(keyword)
    return self._index[keyword]
```

For example, to index the document `Foo` with the text `Hello, World! My name is Foo!`, and the document `Bar` with the text `Hello, World! My name is Bar, I'm not Foo!` and then search for the word `Foo`, we can do it as

```
>>> from microsearch.engine import engine
>>> engine.index("Foo", "Hello, World! My name is Foo!")
>>> engine.index("Bar", "Hello, World! My name is Bar, I'm not Foo!")
>>> engine.get_urls("foo")
defaultdict(<class 'int'>, {'Foo': 1, 'Bar': 1})
>>> engine.get_urls("Foo")
defaultdict(<class 'int'>, {'Foo': 1, 'Bar': 1})
```

Ranker

Once you have a set of matching documents for a given query, you need a way to sort them. The most famous ranker is Google's PageRank, which ranks documents based on the links. However, other options to rank the documents exist, such as BM25, which ranks documents based on the content. In my case, I decided to use the standard BM25. The score between a query and a document is computed as

where the query contains the keywords , , …, , the document has length , the average length of a document is defined as , and are free parameters, is the number of times that keyword appears in the document , and finally is the inverse document frequency, computed as

where is the number of documents and is the number of documents containing . There are other ways to compute the IDF, but apparently, you can justify this option from theoretical grounds.

With all this math we are ready to implement the missing part of our `SearchEngine` class. Firstly we add the constants and as parameters of our class

```
class SearchEngine:
    def __init__(self, k1: float = 1.5, b: float = 0.75):
        self._index: dict[str, dict[str, int]] = defaultdict(lambda: defaultdict(int))
        self._documents: dict[str, str] = {}
        self.k1 = k1
        self.b = b
```

now we expose some useful properties that we'll use later

```
@property
def posts(self) -> list[str]:
    return list(self._documents.keys())

@property
def number_of_documents(self) -> int:
    return len(self._documents)

@property
def avdl(self) -> float:
    return sum(len(d) for d in self._documents.values()) / len(self._documents)
```

With this information, we are ready to implement our BM25 scorer. The first thing we need to implement is the inverse document frequency method

```
def idf(self, kw: str) -> float:
    N = self.number_of_documents
    n_kw = len(self.get_urls(kw))
    return log((N - n_kw + 0.5) / (n_kw + 0.5) + 1)
```

and with this method, we can finally implement the BM scorer. This method receives a keyword and returns a mapping from all the URLs that contain that keyword to their score.

```
def bm25(self, kw: str) -> dict[str, float]:
    result = {}
    idf_score = self.idf(kw)
    avdl = self.avdl
    for url, freq in self.get_urls(kw).items():
        numerator = freq * (self.k1 + 1)
        denominator = freq + self.k1 * (1 - self.b + self.b * len(self._documents[url]) / avdl)
        result[url] = idf_score * numerator / denominator
    return result
```

This method receives a keyword, and for all the indexed documents it computes the BM25 score for that keyword. With this method we can finally implement the `search` method, which will be the one we'll use to make queries to our search engine.

The `search` method receives a query, normalizes it, extracts its keywords (ie: splits it by space), computes the BM25 scores for each keyword, and returns a dictionary of URLs with their total score.

```
def search(self, query: str) -> dict[str, float]:
    keywords = normalize_string(query).split(" ")
    url_scores: dict[str, float] = {}
    for kw in keywords:
        kw_urls_score = self.bm25(kw)
        url_scores = update_url_scores(url_scores, kw_urls_score)
    return url_scores
```

Following the same example as before, we can use it to search as

```
>>> from microsearch.engine import engine
>>> engine.index("Foo", "Hello, World! My name is Foo!")
>>> engine.index("Bar", "Hello, World! My name is Bar, I'm not Foo!")
>>> engine.search("foo")
{'Foo': 0.19869271730423296, 'Bar': 0.16844281759753774}
>>> engine.search("foo bar")
{'Foo': 0.19869271730423296, 'Bar': 0.8088260293054897}
```

Putting everything together, we have a search engine class that implements the functionalities to index and search documents in less than 80 lines of code.

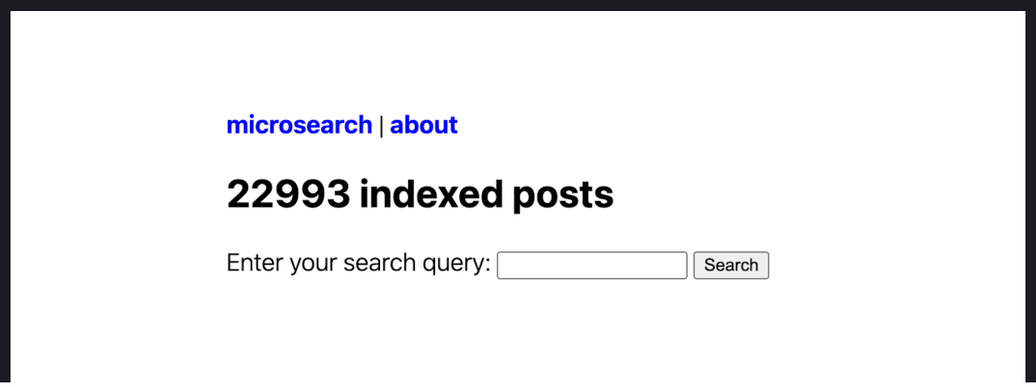
▶ Complete code to have a search engine in 80 lines of Python

Interface

Finally, once we have a search engine, we want to expose it somehow. In my case, I decided to build a small FastAPI app that exposes an endpoint with the search engine, and then it also renders a simple webpage that allows you to search. To make the output easier to read I decided to just select the top-N URLs.

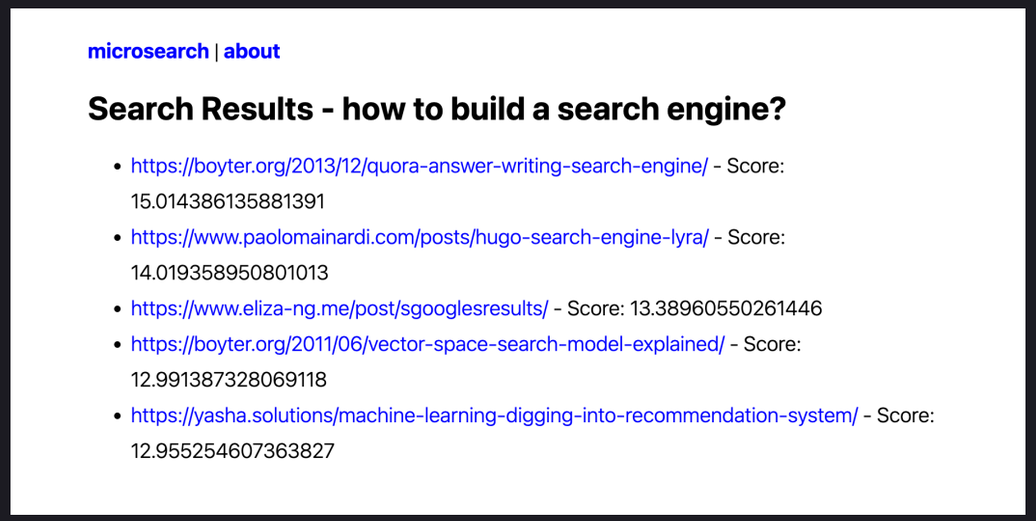
▶ FastAPI search engine app

If you run it you'll see something like



Search Engine interface.

then you can introduce your queries using the search box and search the indexed documents. For example, if I search for `how to build a search engine?`



Search results for the query `how to build a search engine`.

I'm aware this is not the nicest UI ever, and the UX can be improved a lot. However, it works fast, the results aren't so bad, and most importantly, I've built it myself from scratch.

Missing features

For the readers who usually work with search engines, it's obvious that there are a lot of missing features in my implementation. This is a non-exhaustive list of what it's missing.

- This implementation doesn't have **query operators**, ie operators that allow you to tune how the query behaves. For example, if you google `how to build a search engine -solr` you'll get results that don't contain the word `solr` in them.
- The reverse index works by indexing single keywords, and there's no option to **index n-grams**. Google allows you to search for `"search engine"` (notice the double quotes) and it'll only show you results where the two words appear in this specific order.
- Also, this implementation doesn't have **query or document expansion**, so if you search `engine` you won't get documents with the word `engines`.
- The **crawling and indexing parts are independent** but we could implement it in such a way that the indexing happens during the crawling, ie: as soon as we have a document we index it. This could be done asynchronously as well.

Conclusions

I’ve enjoyed a lot working on this project. It has helped me to understand better how Solr works under the hood, and while I still have a lot to learn I think I have a better intuition now.

As a side effect, I’ve also learned how amazing is writing asynchronous code for IO-bounded operations. My first implementation of the crawler took ages to finish, and with the async implementation, it took a moment to finish.

My next step on my journey to build a personal search engine is to implement semantic search capabilities in the search engine. I’ve been playing a bit with embedding models and ANN, so my next step is to add this functionality to microsearch. Keep tuned for more!
