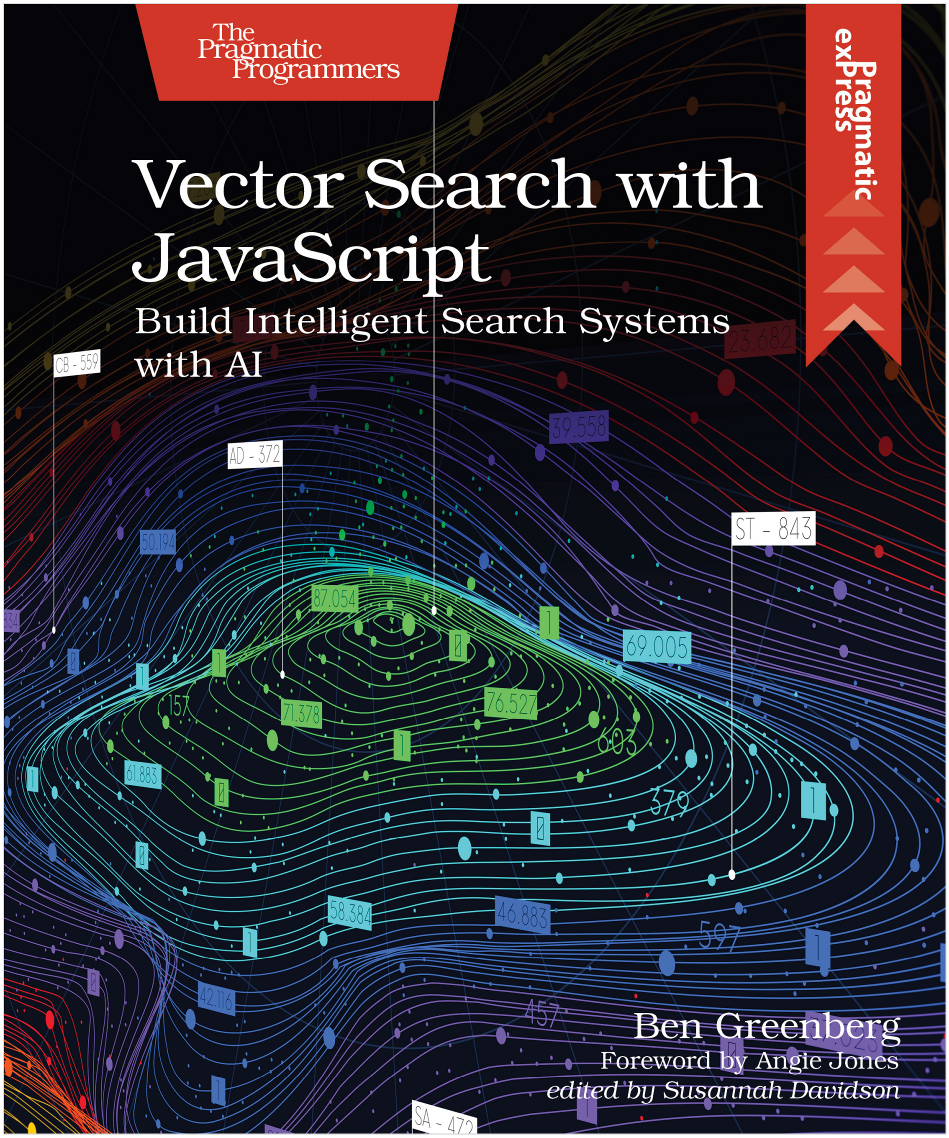
The Pragmatic Programmers

Pragmatic exPress

Vector Search with JavaScript

Build Intelligent Search Systems with AI

Ben Greenberg

Foreword by Angie Jones

edited by Susannah Davidson

CHAPTER 1

Getting Started with Vector Search

Let's get our feet wet with a quick example project to see how useful vector search can be. You'll learn all about the underlying concepts in the following chapters, but for now, we're just going to have some fun and see vector search in action. Later in this book, you'll build a working vector search project from scratch based on a widely adapted open source project. This chapter invites you to work with real data, run real code, and see meaningful results before we introduce any formal definitions or math.

Imagine you're writing this book. Sure, writing the text took a while, but you managed to finish everything… when all of a sudden you get writer's block. You need a title for the book, and you can't figure out what would not only be a great title, but something that fits in with your publisher's style of book titles. That was me, and guess what, I used vector search to help me come up with just the right title for this book.

You now get to follow that same process. Using the OpenAI API, you will generate embeddings from real Pragmatic titles, store them locally, and compare them to a title idea of your own. This exercise gives you a practical and approachable way to engage with vector search before we unpack the deeper concepts behind it. All you need is one file, a small dataset, and a willingness to explore.

Setting Up the Project

Create a new file named name_this_book.js. This file will do everything for you: scrape the website, generate embeddings, compare them, and return the top matches. Embeddings are numerical representations of text that capture its meaning and context. Embeddings are a central concept throughout this book, and we'll explore them in much greater depth as we go.

Before installing the required libraries, make sure you have Node.js installed. You can download it from the official website.[1] Installing Node.js will also install npm, which is the package manager we'll use throughout this book.

Once done installing Node.js, you'll need to install a few libraries:

```
npm install node-fetch dotenv cheerio
```

And you'll need an OpenAI API key in a local .env file:

```
echo "OPENAI_API_KEY=your-key-here" > .env
```

Scraping the Titles

We'll start by scraping all the titles from the Pragmatic Bookshelf catalog. Their site uses pagination, so we'll loop through all pages and extract the title, subtitle, and book URL.

Here's the function to do that:

getting_started_with_vector_search/name_this_book.js

```
async function scrapeTitles() {
  const baseURL = 'https://pragprog.com';
  const maxPages = 27;
  const maxTitles = 50;
  const allBooks = [];

  console.log(
    `Scraping up to ${maxTitles} book titles from Pragmatic Bookshelf...`
  );

  for (let page = 1; page <= maxPages; page++) {
    const pageURL = page === 1
      ? `${baseURL}/titles/`
      : `${baseURL}/titles/page/${page}/`;

    console.log(`  - Fetching page ${page}`);
    const res = await fetch(pageURL);
    const html = await res.text();
    const $ = cheerio.load(html);

    $('.category-title-container').each((_, el) => {
      if (allBooks.length >= maxTitles) return false;

      const anchor = $(el).find('a').first();
      const title = $(el)
        .find('.category-title-title b')
        .text()
        .trim();
      const subtitle = $(el)
        .find('.category-title-subtitle')
```

1. https://www.nodejs.org

```
        .text()
        .trim();
      const url = anchor.attr('href');

      if (title && url) {
        allBooks.push({ title, subtitle, url });
      }
    });

    if (allBooks.length >= maxTitles) break;
  }

  const unique = Array.from(
    new Map(allBooks.map(b => [b.url, b])).values()
  );

  console.log(`
    Finished scraping. ${unique.length} titles captured.`
  );
  return unique;
}
```

This function loops through every results page, uses Cheerio, a JavaScript library, to parse the titles and subtitles, and returns a list of unique book entries.

You are now working with real-world HTML. Scraping is often messy and requires patience. The structure of Pragmatic's catalog is structurally clean, but keep in mind that websites can change. This kind of work is always fragile by nature. If the site layout or class names change, your scraper could break. That means your downstream processes, like generating embeddings or running similarity comparisons, might be working with incomplete or outdated data.

Embedding the Titles

Next, we'll use OpenAI's text-embedding-3-small model to generate an embedding for each full book title.

Let's define a helper function to generate a single embedding:

getting_started_with_vector_search/name_this_book.js

```
async function getEmbedding(text) {
  console.log(`Generating embedding for: "${text}"`);
  const res = await fetch('https://api.openai.com/v1/embeddings', {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${process.env.OPENAI_API_KEY}`,
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      input: text,
      model: 'text-embedding-3-small',
    }),
  });

  const data = await res.json();
  return data.data[0].embedding;
}
```

Then, use that function to embed all the titles:

getting_started_with_vector_search/name_this_book.js

```
async function embedTitles(books) {
  console.log(`Generating embeddings for all titles...`);

  const promises = books.map((book, i) => {
    const fullTitle = book.subtitle
      ? `${book.title}: ${book.subtitle}`
      : book.title;

    console.log(`  (${i + 1}/${books.length}) ${fullTitle}`);
    return getEmbedding(fullTitle).then(embedding => {
      book.embedding = embedding;
    });
  });

  await Promise.all(promises);
  return books;
}
```

This combines the title and subtitle (if present) and generates one embedding per book. All embedding requests are fired off at the same time and processed in parallel. This speeds things up, but if you're working with large datasets or strict API rate limits, you may want to throttle or batch requests instead.

Don't worry if the process still feels mysterious. In the next chapter, we'll unpack exactly how embeddings work and what makes them such a powerful tool for search and recommendation.

Comparing to Your Title Idea

Now you'll embed a new title idea and compare it to the catalog using cosine similarity. Here's how we compute similarity between vectors:

getting_started_with_vector_search/name_this_book.js

```
function cosineSimilarity(a, b) {
  const dot = a.reduce(
    (sum, ai, i) => {
      return sum + ai * b[i];
    },
    0
  );
  const magA = Math.sqrt(
    a.reduce((sum, ai) => {
      return sum + ai * ai;
    }, 0)
  );
  const magB = Math.sqrt(
    b.reduce((sum, bi) => {
      return sum + bi * bi;
    }, 0)
  );
  return dot / (magA * magB);
}
```

And here's how we find and rank the top matches:

getting_started_with_vector_search/name_this_book.js

```
function compareTitles(queryEmbedding, books) {
  console.log('Comparing against catalog...');
  return books
    .map(book => ({
      ...book,
      similarity: cosineSimilarity(
        book.embedding,
        queryEmbedding
      ),
    }))
    .sort((a, b) => b.similarity - a.similarity)
    .slice(0, 5);
}
```

Putting It All Together

Finally, let's connect everything. This main() function runs the entire flow: scrape, embed, store, and compare.

getting_started_with_vector_search/name_this_book.js

```
async function main() {
  const DATA_FILE = 'pragprog_titles.json';
  let books;

  if (fs.existsSync(DATA_FILE)) {
    console.log(`Loading cached data from ${DATA_FILE}...`);
    books = JSON.parse(fs.readFileSync(DATA_FILE, 'utf8'));
  } else {
    console.log('No cached data found. Starting fresh.');

    books = await scrapeTitles();
    books = await embedTitles(books);

    fs.writeFileSync(
      DATA_FILE,
      JSON.stringify(books, null, 2)
    );

    console.log(`Saved ${books.length} books to ${DATA_FILE}`);
  }

  const query = process.argv.slice(2).join(' ');
  if (!query) {
    console.error(
      'Please pass a proposed title as a command-line argument'
    );
    console.error(
      'Example: node name_this_book.js "Mastering Vector Search"'
    );
    process.exit(1);
  }

  console.log(`\nFinding matches for: "${query}"`);
  const queryEmbedding = await getEmbedding(query);
  const topMatches = compareTitles(queryEmbedding, books);

  console.log(`\nTop matches for "${query}":\n`);
  for (const match of topMatches) {
    console.log(
      `- ${match.title}${match.subtitle ? `: ${match.subtitle}` : ''}`
    );
    console.log(
      `  (${(match.similarity * 100).toFixed(2)}% similar)`
    );
    console.log(
      `  ${match.url}\n`
    );
  }
}
```

Run the script like this:

```
node name_this_book.js "Vector Search for Everyone"
```

The script will output the top 5 most similar titles from the catalog along with their similarity scores. A score close to 1 means your title idea is very similar in meaning to a published Pragmatic title, while lower scores indicate less similarity. You can use these results to evaluate whether your idea fits the style and themes of the publisher's catalog. You might also use the output to spark new ideas based on which titles rank highly in similarity.

When you run the script, you'll see the top five matches ranked by similarity. For example, testing the title idea *Learning Vector Search with JavaScript* produced results that leaned into themes of "learning," "programming," and "JavaScript." The closest match, *A Common-Sense Guide to Data Structures and Algorithms in JavaScript*, shared both the language and the educational framing through its use of "learning"-adjacent phrasing. Other top matches included *Machine Learning in Elixir* and *Programming Phoenix LiveView*, which shows that the system focused on action-oriented words like "learning" and "programming" as strong signals. This kind of feedback helps you understand not just which titles are similar, but why. Often, they share verbs, technologies, or a how-to tone. You can use that insight to tweak your phrasing or steer your idea toward a more distinctive niche.

Key Takeaways

In this chapter, you used vector search to compare your own book title idea to real titles from the Pragmatic Bookshelf catalog. Along the way, you wrote a complete program that scraped live data, generated embeddings using OpenAI, and ranked results using cosine similarity, all with just a few lines of code. If you haven't already, take a moment to come up with a title idea of your own and plug it into the script to see how it compares.

To summarize your major wins, you:

- Scraped and parsed real book titles from the web
- Generated vector embeddings with the OpenAI API
- Compared your title idea to the catalog using cosine similarity
- Followed the same approach the author used to name this book

Now that you've seen vector search in action, the next step is to understand how it actually works.