Published in Better Programming

Tyler Hawkins Follow
Aug 19, 2021 · 6 min read ★ · Listen

Build an Article Recommendation Engine With AI/ML

A Python app to get better content suggestions

translates to increased ad revenue for the company.

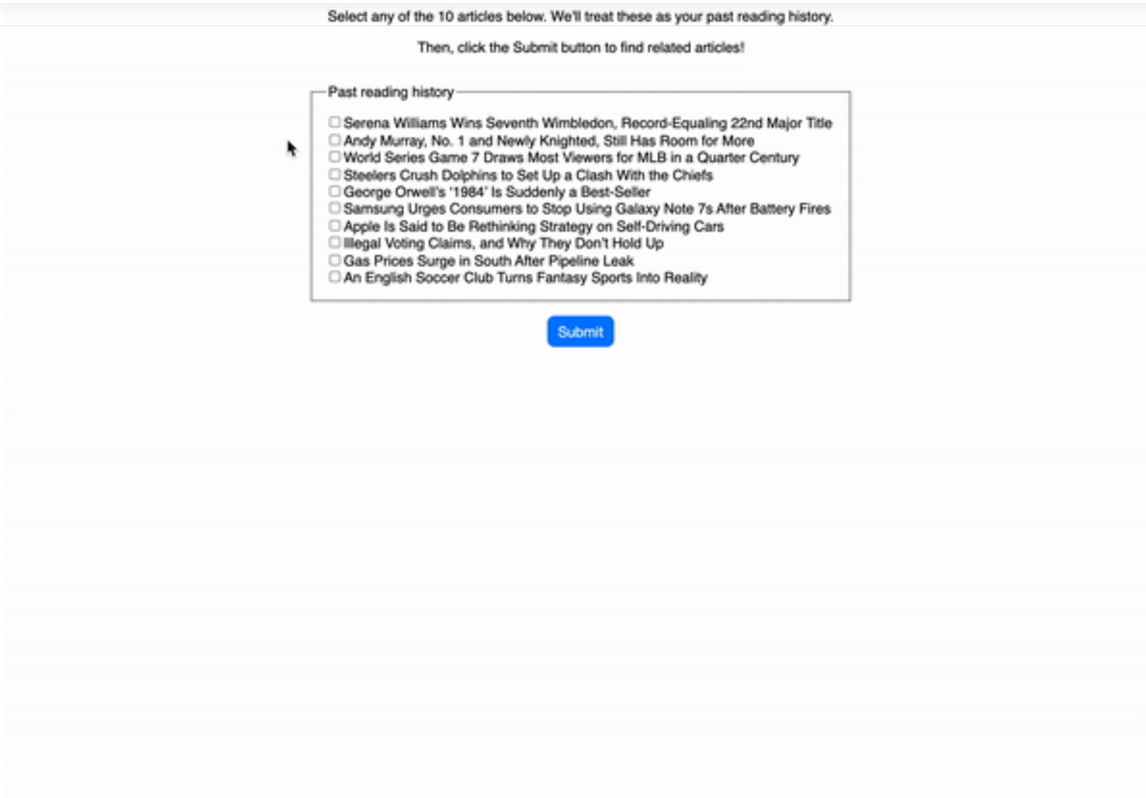
If you've ever visited a news website, online publication, or blogging platform, you've likely been exposed to a recommendation engine. Each of these takes input based on your reading history and then suggests more content you might like.

As a simple solution, a platform might implement a tag-based recommendation engine — you read a "Business" article, so here are five more articles tagged "Business." However, an even better approach to building a recommendation engine is to use **similarity search and a machine learning algorithm**.

In this article, we'll build a Python Flask app that uses Pinecone — a similarity search service — to create our very own article recommendation engine.

Demo App Overview

Below, you can see a brief animation of how our demo app works. Ten articles are initially displayed on the page. The user can choose any combination of those ten articles to represent their reading history. When the user clicks the Submit button, the reading history is used as input to query the article database, and then ten more related articles are displayed to the user.



Demo app — article recommendation engine

As you can see, the related articles returned are exceptionally accurate! There are 1,024 possible combinations of reading history that can be used as input in this example, and every combination produces meaningful results.

So, how did we do it?

In building the app, we first found a dataset of news articles from Kaggle. This dataset contains 143,000 news articles from 15 major publications, but we're just using the first 20,000. (The full dataset that this one is derived from contains over two million articles!)

We then cleaned up the dataset by renaming a couple of columns and dropping those that are unnecessary. Next, we ran the articles through an embedding model to create vector embeddings — that's metadata for machine learning algorithms to determine similarities between various inputs. We used the Average Word Embeddings Model. We then inserted these vector embeddings into a vector index managed by Pinecone.

similar news articles and displays them in the app's UI. That's it! Simple enough, right?

If you'd like to try it out for yourself, you can find the code for this app on GitHub. The `README` contains instructions for how to run the app locally on your own machine.

Demo App Code Walkthrough

We've gone through the inner workings of the app, but how did we actually build it? As noted earlier, this is a Python Flask app that utilizes the Pinecone SDK. The HTML uses a template file, and the rest of the frontend is built using static CSS and JS assets. To keep things simple, all of the backend code is found in the `app.py` file, which we've reproduced in full below:

```
from flask import render_template
from flask import request
from flask import url_for
import json
import os
import pandas as pd
import pinecone
import re
import requests
from sentence_transformers import SentenceTransformer
from statistics import mean
import swifter

app = Flask(__name__)

PINECONE_INDEX_NAME = "article-recommendation-service"
DATA_FILE = "articles.csv"
NROWS = 20000

def initialize_pinecone():
    load_dotenv()
    PINECONE_API_KEY = os.environ["PINECONE_API_KEY"]
    pinecone.init(api_key=PINECONE_API_KEY)

def delete_existing_pinecone_index():
    if PINECONE_INDEX_NAME in pinecone.list_indexes():
        pinecone.delete_index(PINECONE_INDEX_NAME)

def create_pinecone_index():
    pinecone.create_index(name=PINECONE_INDEX_NAME, metric="cosine", shards=1)
    pinecone_index = pinecone.Index(name=PINECONE_INDEX_NAME)

    return pinecone_index

def create_model():
    model = SentenceTransformer('average_word_embeddings_komninos')

    return model

def prepare_data(data):
    # rename id column and remove unnecessary columns
```

Open in app Get started

```python
    data['content'] = data.content.swifter.apply(lambda x: ' '.join(re.split(r'(?<=[.:
    data['title_and_content'] = data['title'] + ' ' + data['content']

    # create a vector embedding based on title and article columns
    encoded_articles = model.encode(data['title_and_content'], show_progress_bar=True)
    data['article_vector'] = pd.Series(encoded_articles.tolist())

    return data

def upload_items(data):
    items_to_upload = [(row.id, row.article_vector) for i, row in data.iterrows()]
    pinecone_index.upsert(items=items_to_upload)

def process_file(filename):
    data = pd.read_csv(filename, nrows=NROWS)
    data = prepare_data(data)
    upload_items(data)
    pinecone_index.info()

    return data

def map_titles(data):
    return dict(zip(uploaded_data.id, uploaded_data.title))

def map_publications(data):
    return dict(zip(uploaded_data.id, uploaded_data.publication))

def query_pinecone(reading_history_ids):
    reading_history_ids_list = list(map(int, reading_history_ids.split(',')))
    reading_history_articles = uploaded_data.loc[uploaded_data['id'].isin(reading_hist

    article_vectors = reading_history_articles['article_vector']
    reading_history_vector = [*map(mean, zip(*article_vectors))]

    query_results = pinecone_index.query(queries=[reading_history_vector], top_k=10)
    res = query_results[0]

    results_list = []

    for idx, _id in enumerate(res.ids):
```

```
94            })
95
96        return json.dumps(results_list)
97
98    initialize_pinecone()
99    delete_existing_pinecone_index()
100   pinecone_index = create_pinecone_index()
101   model = create_model()
102   uploaded_data = process_file(filename=DATA_FILE)
103   titles_mapped = map_titles(uploaded_data)
104   publications_mapped = map_publications(uploaded_data)
105
106   @app.route("/")
107   def index():
108       return render_template("index.html")
109
110   @app.route("/api/search", methods=["POST", "GET"])
111   def search():
112       if request.method == "POST":
113           return query_pinecone(request.form.history)
114       if request.method == "GET":
```

Let’s go over the important parts of the `app.py` file so that we understand it.

On lines 1–14, we import our app’s dependencies. Our app relies on the following:

- `dotenv` for reading environment variables from the `.env` file
- `flask` for the web application setup
- `json` for working with JSON
- `os` also for getting environment variables
- `pandas` for working with the dataset

- `requests` for making API requests to download our dataset
- `statistics` for some handy stats methods
- `sentence_transformers` for our embedding model
- `swifter` for working with the pandas dataframe

On line 16, we provide some boilerplate code to tell Flask the name of our app.

On lines 18–20, we define some constants that will be used in the app. These include the name of our Pinecone index, the file name of the dataset, and the number of rows to read from the CSV file.

On lines 22–25, our `initialize_pinecone` method gets our API key from the `.env` file and uses it to initialize Pinecone.

On lines 27–29, our `delete_existing_pinecone_index` method searches our Pinecone instance for indexes with the same name as the one we're using ("article-recommendation-service"). If an existing index is found, we delete it.

On lines 31–35, our `create_pinecone_index` method creates a new index using the name we chose ("article-recommendation-service"), the "cosine" proximity metric, and only one shard.

On lines 37–40, our `create_model` method uses the `sentence_transformers` library to work with the Average Word Embeddings Model. We'll encode our vector embeddings using this model later.

On lines 62–68, our `process_file` method reads the CSV file and then calls the `prepare_data` and `upload_items` methods on it. Those two methods are described next.

On lines 42–56, our `prepare_data` method adjusts the dataset by renaming the first "id" column and dropping the "date" column. It then grabs the first four lines of each article and combines them with the article title to create a new field that serves as the data to

On lines 58–60, our `upload_items` method creates a vector embedding for each article by encoding it using our model. The vector embeddings are then inserted into the Pinecone index.

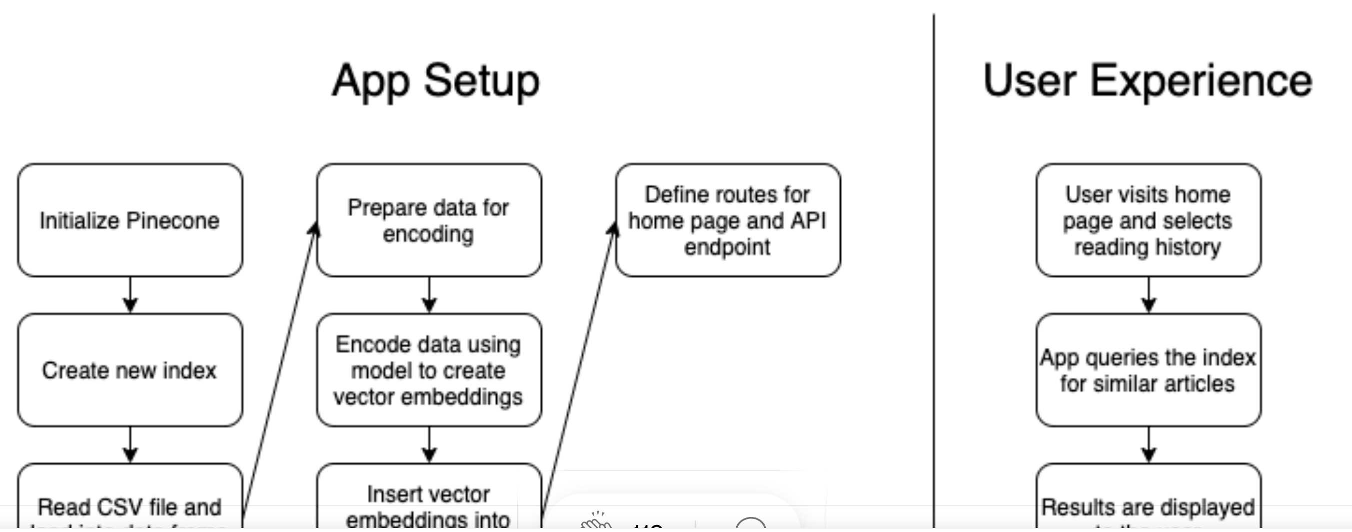
On lines 70–74, our `map_titles` and `map_publications` methods create some dictionaries of the titles and publication names to make it easier to find articles by their IDs later.

Each of the methods we've described so far is called on lines 98–104 when the backend app is started. This work prepares us for the final step of actually querying the Pinecone index based on user input.

On lines 106–116, we define two routes for our app: one for the home page and one for the API endpoint. The home page serves up the `index.html` template file along with the JS and CSS assets, and the API endpoint provides the search functionality for querying the Pinecone index.

Finally, on lines 76–96, our `query_pinecone` method takes the user's reading history input, converts it into a vector embedding, and then queries the Pinecone index to find similar articles. This method is called when the `/api/search` endpoint is hit, which occurs any time the user submits a new search query.

For the visual learners out there, here's a diagram outlining how the app works:



Example Scenarios

So, putting this all together, what does the user experience look like? Let's look at three scenarios: a user interested in sports, a user interested in technology, and a user interested in politics.

The sports user selects the first two articles about Serena Williams and Andy Murray, two famo tennis players, to use as their reading history. After they submit their choices, the app responds with articles about Wimbledon, the US Open, Roger Federer, and Rafael Nadal. Spot on!

The technology user selects articles about Samsung and Apple. After they submit their ces, the app responds with articles about Samsung, Apple, Google, Intel, and iPhones. Great recommendations again!

The politics user selects a single article about voter fraud. After they submit their ch e ap responds with articles about voter ID, the US 2020 election, voter turnout, and claim egal voting (and why they don't hold up).

Three for three! Our recommendation engine is proving to be incredibly useful.

Conclusion

We've now created a simple Python app to solve a real-world problem. If content sites can recommend relevant content to their users, users will enjoy the content more and will spend more time on the site, resulting in more revenue for the company. Everyone wins!

Similarity search helps provide better suggestions for your users. And Pinecone, as a ity service, makes it easy for you to provide recommendations to your users so that you can focus on what you do best — building an engaging platform filled with content worth reading.

Thanks to Anupam Chugh

Recommended from Medium

Aaron George

Solve Sudoku from an Image

Banjodayo

AI at the Edge: Model Optimizer, Inference Engine, and MQTT

Alahira Jeffrey Calvin

Hyperparameter Tuning for Beginners—Part One

Manish Kumar Lal

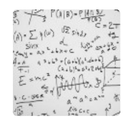
Dog Breed Classification Project using Convolution Neural Networks (CNN)

Dr. Shahin Rostami

Machine Learning with Kaggle Kernels — Part 4

Tiago ... in Artificial Intelligence in...

Are CNNs much different from MLPs?



Magda Stenius

Difference as Distance

Bharath K in Towards Data Science

OpenCV: Complete Beginners Guide To Master the Basics Of Computer Vision With Code!