A Vector-Based Search Engine for Scientific Figures
CS 734: Intro to Information Retrieval
Semester Project Report

Caleb Bradford

December 7, 2023

1 Motivation

In this implementation based project, I was tasked with utilizing information retrieval methods to build a vector-based search engine for a large dataset of scientific figures. This included building a web based interface capable of handling queries through text as well as queries through reverse image search (searching for images most similar to a given image query). The interface also needed to be able to handle a faceted search, where the user could specify what type of figures they wanted to see returned. The dataset provided was the SciFig dataset, a dataset of over 264,000 scientific figures gathered from the ACL Anthology. The figure types for the faceted search are defined in the SciFig Pilot dataset, a labelled subset of the data.

2 System Design

2.1 Architecture Diagrams

My methods for developing this vector search engine mainly consisted of two parts: generating vector embeddings and ingesting them to an ElasticSearch instance (Figure 1), and building a simple interface and backend to handle search queries to the ElasticSearch instance (Figure 2). Further details about the specific implementations of these tasks can be found in section 3.

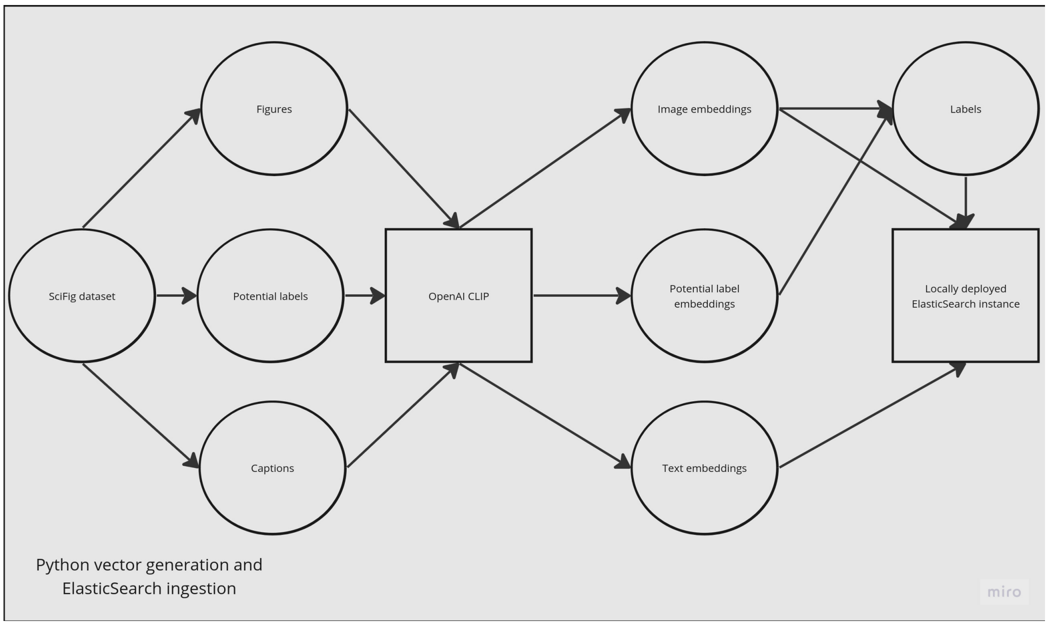

Figures
SciFig dataset
Potential labels
Captions
OpenAI CLIP
Image embeddings
Potential label embeddings
Text embeddings
Labels
Locally deployed ElasticSearch instance
Python vector generation and ElasticSearch ingestion
miro

Figure 1: My pipeline for vector embedding generation and ElasticSearch index ingestion

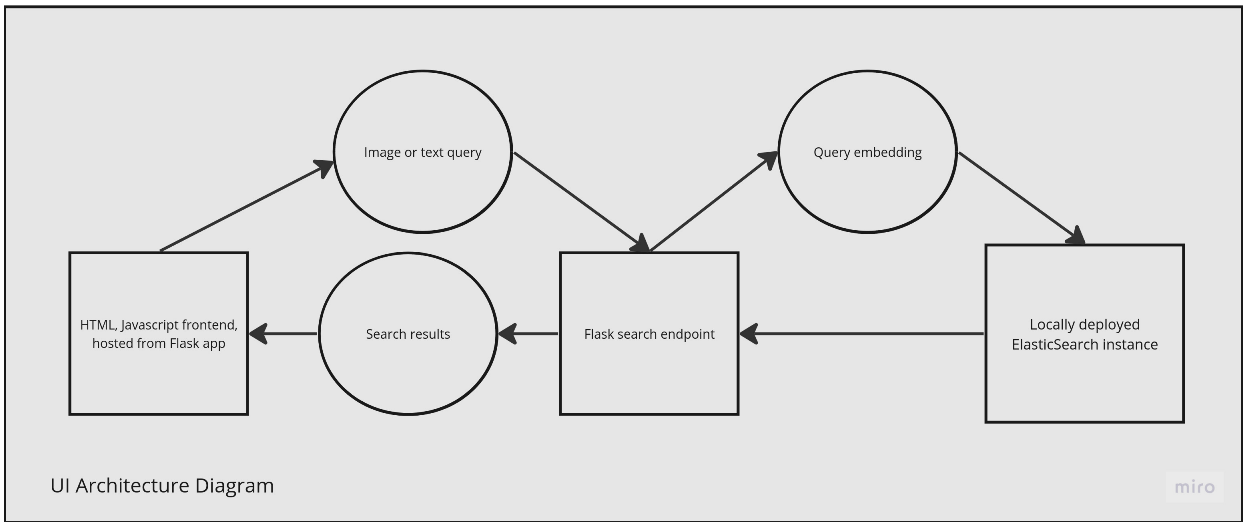
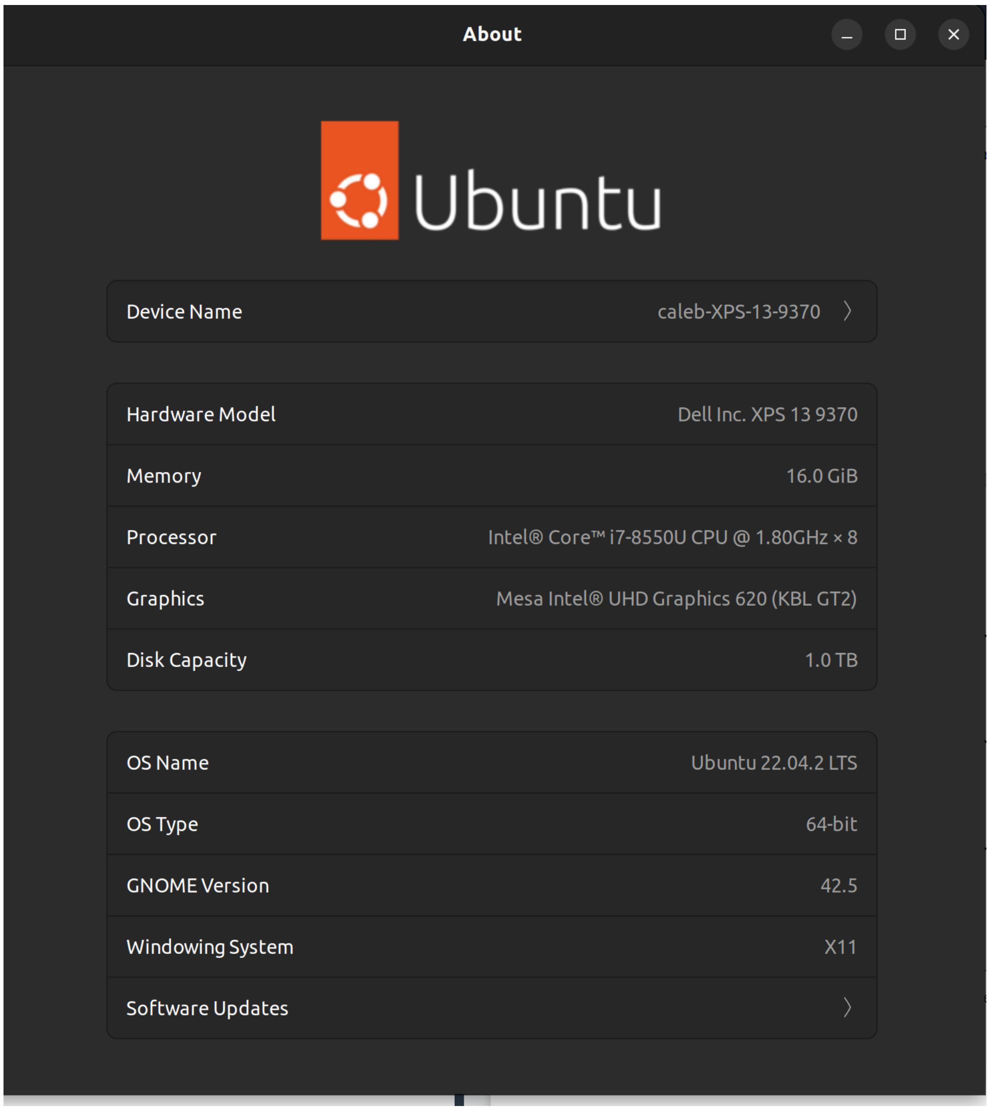


Figure 2: My architecture diagram for the search engine

2.2 Infrastructure

2.2.1 Hardware

I opted to use my own personal laptop for all coding, computation, and hosting for the ElasticSearch instance and the search interface. The specifications for this hardware is shown in Figure 3


About

Ubuntu

Device Name	caleb-XPS-13-9370
Hardware Model	Dell Inc. XPS 13 9370
Memory	16.0 GiB
Processor	Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
Graphics	Mesa Intel® UHD Graphics 620 (KBL GT2)
Disk Capacity	1.0 TB
OS Name	Ubuntu 22.04.2 LTS
OS Type	64-bit
GNOME Version	42.5
Windowing System	X11
Software Updates	

Figure 3: The specifications for the machine used in this project

2.2.2 Software

I used a variety of software tools to complete this project. Arguably the most important of which was OpenAI's CLIP model, a neural network trained to generate vector embeddings for both images and text in an aligned vector space. The multimodal search of this project would not be possible without it. The implementation of CLIP is accomplished through a variety of Python scripts. An ElasticSearch instance deployed locally through Docker was utilized to store and utilize the search index used for this project. A Python Flask application is used to interface with the ElasticSearch instance. The Flask app also hosts the simple frontend written using HTML and Javascript.

3 Implementation

3.1 Data Preprocessing

3.1.1 Image embedding generation

In order to generate the image embeddings I ultimately utilized for my reverse image search, I utilized the clip-retrieval Python wrapper available at `https://github.com/rom1504/clip-retrieval/tree/main`. This wrapper provided a command that could, when provided a directory consisting of just images, generate embeddings of all images in the directory. I ran the command below (Listing 1) to accomplish this process on the entire SciFig dataset (stored in png) and stores it in a numpy array saved in a directory called all_embeddings.

Listing 1: Command run to utilize the clip-retrieval inference functionality to generate image embeddings for all figures in SciFig dataset

```
clip-retrieval inference --input_dataset png --output_folder all_embeddings
```

In addition to the embeddings, the inference command also produced a metadata file that contained important information tied to each embedding, including the image path for each figure. It also generated a statistics file containing the amount of time it took to generate all of the embeddings. According to that file, the process took about 14,120 seconds for all of the figures in the SciFig dataset.

3.1.2 Text embedding generation

While I originally planned to only use the image embeddings for both the text and reverse image search, I ultimately decided to embed the captions provided for each scientific figure and utilized those embeddings for my text search. I made this decision because I figured the embeddings of the text captions for each figure would be more semantically meaningful to the image than their image embeddings.

So, I wrote a Python script called create_text_embeddings.py that utilized the clip and torch libraries to get the caption text and image paths for each figure from the provided csv file. It then generated the text embedding of the figure caption. It also leveraged the metadata file generated from the image embedding generation and the image path to ensure that the resulting text embedding array would align with the image embedding array. I saved the final array as a numpy array called text_embeddings.npy. This process took 13,116 seconds, or about 3 and a half hours.

3.2 Labelling the figures with CLIP

In order to facilitate the required faceted search, I needed to label all of the figures. To do this, I further utilized the CLIP model to perform zero-shot prediction on the entire dataset. This was a use case of CLIP mentioned on the CLIP GitHub page. Basically, given an embedding and a list of potential labels, CLIP can attempt to predict the most likely label for that embedding by comparing it to the embeddings of all of the labels.

To implement this, I wrote a Python script called predict.py, which was largely inspired by the example on the CLIP GitHub page. It basically computed the vector embeddings for all of the potential labels from the SciFig Pilot dataset, and then used those to perform zero shot prediction on all of the image embeddings. I wrote the labels to text files, stored in a directory called labels. Each label file is named the index of the embedding that the label was predicted from.

3.3 ElasticSearch Deployment/Configuration

I chose to locally host my ElasticSearch instance through Docker using the latest Docker image for ElasticSearch. I made no changes to the configuration of the instance itself.

For the mappings I used for the index, I made the below mappings.json (Listing 2) file when creating my index.

Listing 2: The mappings.json file used to configure the ElasticSearch index

```
{
    "settings": {
      "index.refresh_interval": "5s",
      "number_of_shards": 1
    },
    "mappings": {
      "properties": {
        "image_embedding": {
          "type": "dense_vector",
          "dims": 512,
          "index": true,
          "similarity": "cosine"
        },
        "text_embedding": {
          "type": "dense_vector",
          "dims": 512,
          "index": true,
          "similarity": "cosine"
        },
        "image_path" : {
          "type" : "keyword"
        },
        "label": {
          "type": "keyword"
        },
        "caption": {
          "type": "keyword"
        }
      }
    }
  }
```

3.4 Indexing/Data Ingestion

In order to ingest everything I have generated thus far into my ElasticSearch instance, I wrote a Python script called populate_elastic.py that utilized the elasticsearch Python library. Since I kept the text embeddings, the image embeddings, and the labels aligned, I just needed to build a Python list where the elements are dictionaries to represent each figure. Each dictionary contained the image embedding, the text embedding, the label, the plaintext caption, and the image path. Each of these is stored in a field in each index entry. Once that list is built, it is ingested to the ElasticSearch index.

3.5 Backend for Search

With my ElasticSearch instance fully populated, I needed to write an API endpoint capable of accepting either a text or an image as a query, using that to query the ElasticSearch instance, and then return the relevant parts of the results. It needed to handle the faceted search aspect, filtering out results that did not have the specified label if any were specified.

To do this, I wrote a Flask app running on localhost capable of accepting HTTP POST requests with either a text or image query. It also optionally accepts figure types to filter by. The endpoint encodes the query using CLIP, and then uses that embedding to query the appropriate field in ElasticSearch (text or image) for the top 10 search results. Once ElasticSearch returns the results, a dictionary containing only the necessary details of the results is built. This is also when results are filtered if this is a faceted search. Finally, this dictionary is returned as a JSON response.

3.6 Frontend Interface

From here, the only task left is to build an interface capable of accepting user input for text or image queries, and sending the queries to the backend for processing. It also needed to allow for selection of figure types for faceted search.

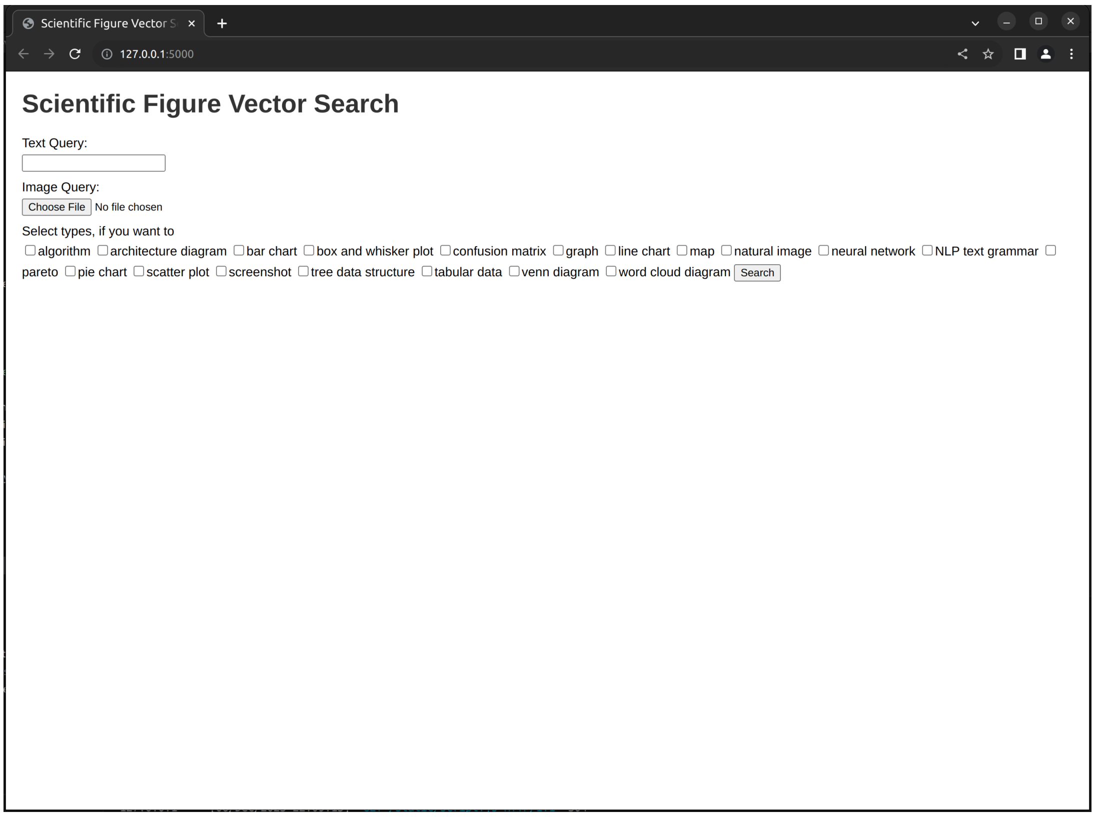
To accomplish this, I wrote a simple UI using HTML and Javascript and hosted it on the same Flask app as the backend through localhost. The HTML page had a form with a field for a text query and a field for an image query. The form also had check boxes for each figure type to facilitate the faceted search functionality. When the search button is pushed, the query is created and sent to the backend through Javascript. Once it receives the response, the search results are appended to the DOM of the page. The server is also setup to handle the necessary requests for the images the site makes when rendering the results. For each result, the image, caption, and label are shown.

3.7 Source Code and Version Control

All of the source code for this project is available to view at on GitHub at `https://github.com/calebkbrad/Vector-Search-Engine`.

4 Final effect

This section includes several screenshots of the working search engine.



Scientific Figure Vector Search

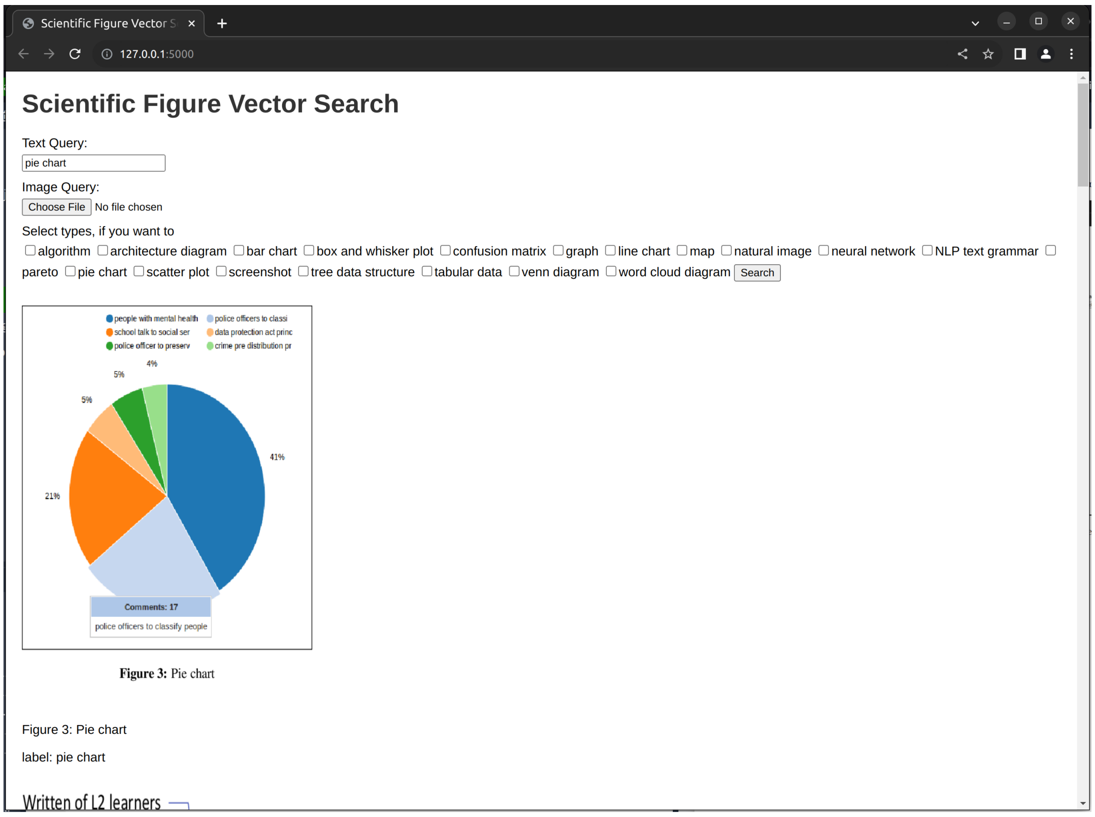
Text Query:

Image Query:

Select types, if you want to

☐ algorithm ☐ architecture diagram ☐ bar chart ☐ box and whisker plot ☐ confusion matrix ☐ graph ☐ line chart ☐ map ☐ natural image ☐ neural network ☐ NLP text grammar ☐ pareto ☐ pie chart ☐ scatter plot ☐ screenshot ☐ tree data structure ☐ tabular data ☐ venn diagram ☐ word cloud diagram

Figure 4: The interface when initially accessed



Scientific Figure Vector S ×

127.0.0.1:5000

Scientific Figure Vector Search

Text Query: pie chart

Image Query: Choose File No file chosen

Select types, if you want to

☐ algorithm ☐ architecture diagram ☐ bar chart ☐ box and whisker plot ☐ confusion matrix ☐ graph ☐ line chart ☐ map ☐ natural image ☐ neural network ☐ NLP text grammar ☐ pareto ☐ pie chart ☐ scatter plot ☐ screenshot ☐ tree data structure ☐ tabular data ☐ venn diagram ☐ word cloud diagram Search

people with mental health
police officers to classi
school talk to social ser
data protection act princ
police officer to preserv
crime pre distribution pr
4%
5%
5%
41%
21%
Comments: 17
police officers to classify people

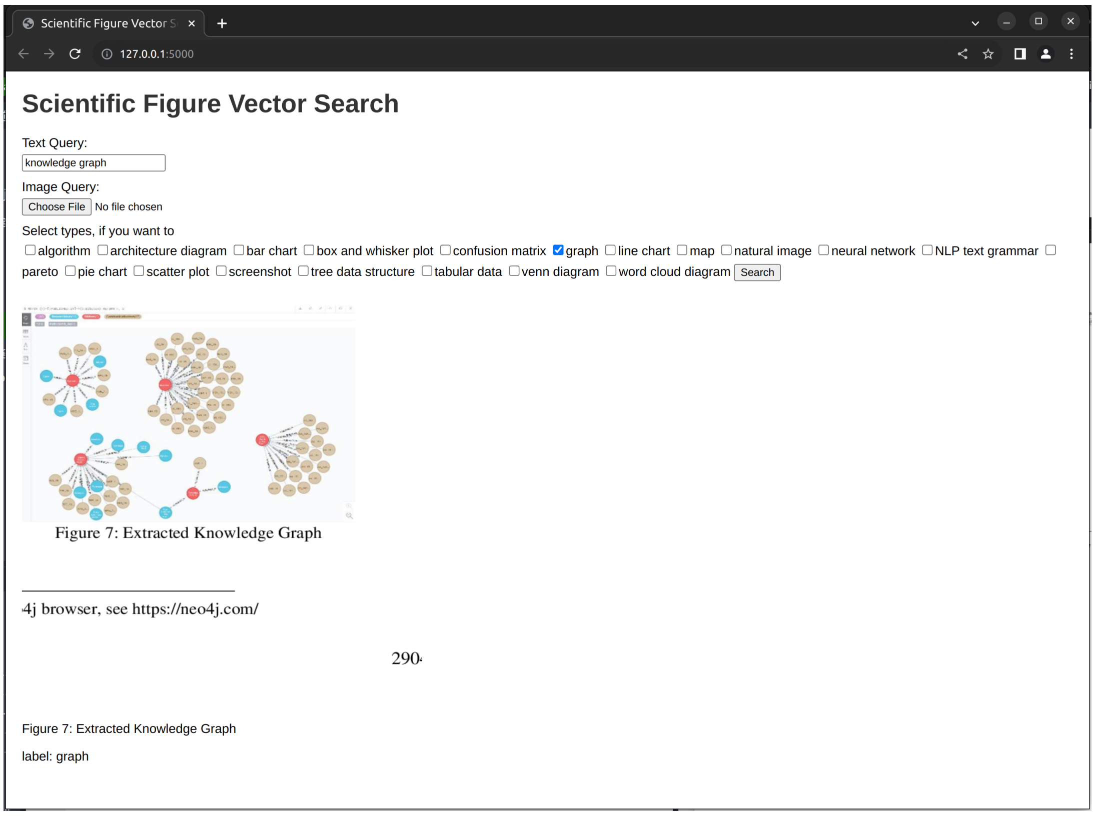
Figure 3: Pie chart

Figure 3: Pie chart

label: pie chart

Written of L2 learners

Figure 5: A text query for "pie chart"



Scientific Figure Vector S ×

127.0.0.1:5000

Scientific Figure Vector Search

Text Query: knowledge graph

Image Query: Choose File No file chosen

Select types, if you want to

☐ algorithm ☐ architecture diagram ☐ bar chart ☐ box and whisker plot ☐ confusion matrix [✓ graph] ☐ line chart ☐ map ☐ natural image ☐ neural network ☐ NLP text grammar ☐ pareto ☐ pie chart ☐ scatter plot ☐ screenshot ☐ tree data structure ☐ tabular data ☐ venn diagram ☐ word cloud diagram Search

Figure 7: Extracted Knowledge Graph

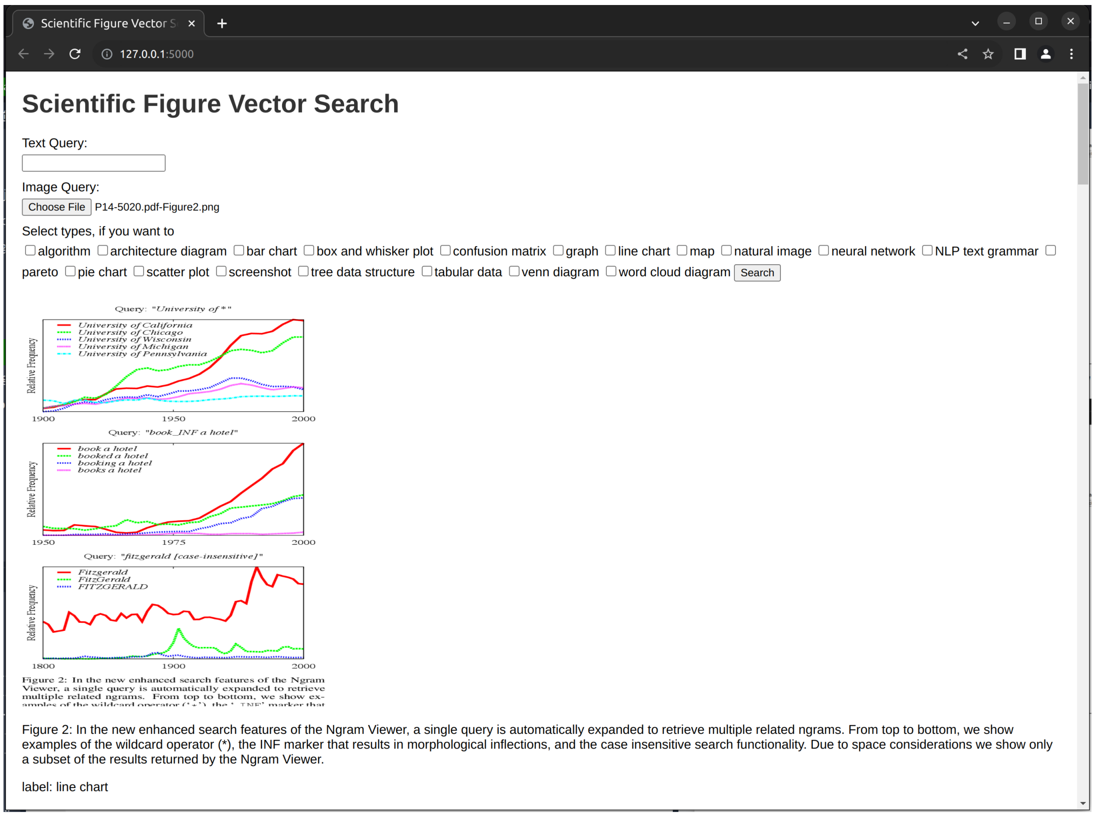
4j browser, see https://neo4j.com/

290

Figure 7: Extracted Knowledge Graph

label: graph

Figure 6: A text query for "knowledge graph", with graph chosen as the filter type (only one result is left)



Scientific Figure Vector Search

Text Query:

Image Query:

Choose File P14-5020.pdf-Figure2.png

Select types, if you want to

☐algorithm ☐architecture diagram ☐bar chart ☐box and whisker plot ☐confusion matrix ☐graph ☐line chart ☐map ☐natural image ☐neural network ☐NLP text grammar ☐ pareto ☐pie chart ☐scatter plot ☐screenshot ☐tree data structure ☐tabular data ☐venn diagram ☐word cloud diagram Search

Figure 2: In the new enhanced search features of the Ngram Viewer, a single query is automatically expanded to retrieve multiple related ngrams. From top to bottom, we show examples of the wildcard operator (*), the INF marker that results in morphological inflections, and the case insensitive search functionality. Due to space considerations we show only a subset of the results returned by the Ngram Viewer.

label: line chart

Figure 7: An image query for one of the figures in the dataset (it shows as the first result)

5 What I learned from this project

In completion of this project, I learned a lot about the vector space model for search engines and how to implement a vector search engine with ElasticSearch. In doing so, I also learned how to utilize OpenAI's CLIP model. This was the first time I've worked with a model like this, so it was an extremely valuable experience. I also had to learn various web development skills, like writing a Flask application.

6 Conclusions

In summary, I built a vector based search engine for the SciFig dataset utilizing ElasticSearch. Doing so required the use of OpenAI CLIP to generate vector representations of the figures, ingesting them to an ElasticSearch instance, building an API to handle queries, and building an interface to make the queries.

While I am happy with what I accomplished, my search engine is far from perfect and there were several aspects that I would improve on if I did this again. For one, I could have used much better computing resources for all aspects of the process. While my laptop was able to handle everything, vector generation would likely have been much faster if I used better hardware. I also may have gotten better search performance if I hosted my ElasticSearch instance on another device with higher memory limits. Another aspect I could have improved was having a more sophisticated and reproducible pipeline for deployment of the search engine. As for classification of the figures in the dataset, I could have used methods to fine tune the CLIP model for classification of the figures by using the data from the SciFig Pilot dataset. This could have improved the often inaccurate classification made by CLIP out of the box. Lastly, I would have liked to learn to use a more sophisticated frontend framework than just plain HTML and Javascript to improve the UI.