SUP'COM
Higher School of Communication of Tunis

جامعة قرطاج
Université de Carthage

Content and image based image retrieval engine

Realized by:

Ghazi Tounsi
ghazi.tounsi@supcom.tn

Mohamed Karaa
mohamed.karaa@supcom.tn

Supervisor:

Riadh Tebourbi
riadh.tebourbi@supcom.tn

Academic year: 2021 - 2022

Content and image based image retrieval engine

Abstract

This report covers the steps followed to realize a complete text-based and content-based search engine from collecting, cleaning and preparing the data to API and UI/UX development.
The first part of the work is to collect an images dataset and extract features using a pretrained deep learning model (VGG16).
In the second part we compare some of the open source tools used for similarity search and propose a method for text and images indexing using elasticsearch for fast searching.
The last part consists of the development of an API and a user interface using FastAPI and Streamlit and then prepare the work to be deployed using Docker to manage all dependencies and configurations.
The source code and the dataset used are all available at: https://github.com/GhaziXX/content-and-text-based-image-retrieval

Keywords: Approximate Nearest Neighbors; Open Images Dataset; VGG16; Features extraction; Search Engine; ElasticSearch; FastAPI; Streamlit; Docker.

Academic year: 2021 - 2022

Contents

List of Figures

List of Tables

Code snippets

1 Introduction

1.1 Context

Advances in data storage and image acquisition technologies have enabled the creation of large image datasets. These datasets are leveraged to develop different applications using Artificial Intelligence algorithms and Big Data technologies. One of these applications is image search where the aim is to retrieve images that correspond to a certain criteria. To achieve an image search, two different approaches are applied: **text-based image retrieval** and **content-based image retrieval** (CBIR).

In the first approach, images are queried through textual metadata (description, labels, keywords..). This technique requires that all images in a database are annotated, which is not practical for large datasets and cannot provide users with accurate search results corresponding to their exact queries.

For this reason, content-based image retrieval techniques are developed. CBIR engines allow searching for images using other images as queries, returning similar images as a result. This approach offers an automatic searching process without the need of annotating images nor providing keywords as queries. In short, a CBIR algorithm takes in a query image, extracts features from it and then returns images with similar feature vectors.

1.2 Project Overview

This project aims to develop a hybrid image search engine that can use one of the mentioned techniques or combines both. This system takes advantage of Open Images Dataset to search over 2 million images with their corresponding textual metadata. Such large data quantity imposes constraints on the system in terms of result precision and response time.

The developed solution combines different technologies that involve data processing, machine learning, data indexing and querying, software development and deployment. To understand the architecture we can brake it down into two processes: Data processing (from feature extraction to indexing) and image querying. Figure 1 resumes these two processes. The user can interact with the system via a web application, where he can submit and customize his query and display the results. The whole system is containerized using Docker and deployed on a server.

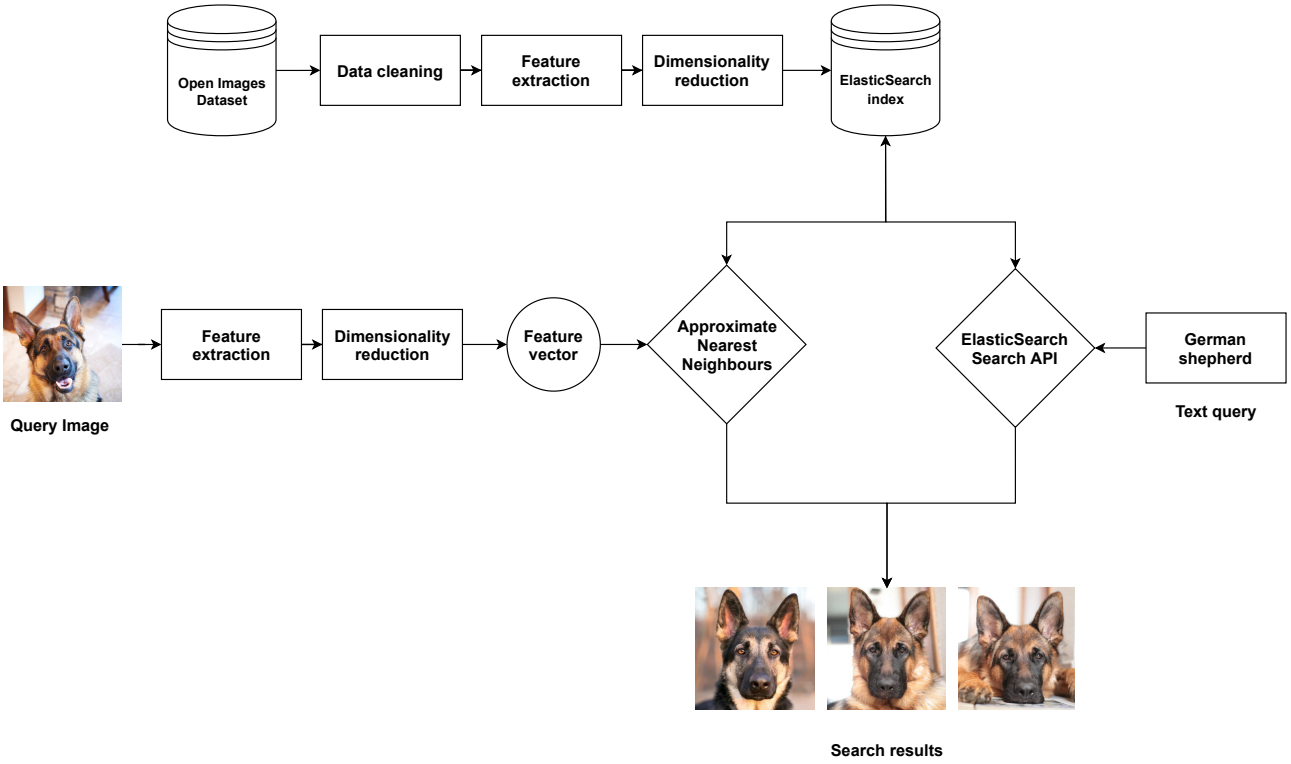


Figure 1: Project overview

2 Dataset

2.1 Data description

Open Images Dataset[1] is an open-source dataset provided by Google. It contains around 9 million very diverse images with rich annotations and often contains complexe scenes.
The dataset is the largest for image classification, object detection, visual relationship detection, instance segmentation, and multimodal image descriptions tasks.
The Open Images Dataset is currently in its sixth version and is split into a training set (9,011,219 images), a validation set (41,620 images), and a test set (125,436 images). The images are annotated with image-level labels, object bounding boxes, object segmentation masks, visual relationships, and localized narratives. In our case, we are only interested in image-level labels which serve the image classification task.
All images have machine-generated image-level labels, which are automatically generated by different computer vision models. Various labels are generated for each image, belonging to different classes. In addition, a part of the training set has human-verified image-level labels. This verification process was mainly to eliminate false positives but not false negatives as some labels might be missing from an image.
Overall, there are 19,958 distinct classes with image-level labels. The trainable classes have at least 100 positive human-verification in the training set. Thus, there are 9,605 trainable classes, and machine-generated labels cover 9,034 of them.

Table 1: Open Images Dataset V6 Image-level labels

	Train	**# Classes**	**# Trainable Classes**
Images	9,011,219	-	-
Machine-Generated Labels	164,819,642	15,387	9,034
Human-Verified Labels	57,524,352	19,957	9,605

The training portion of the dataset is presented as 10 *.tsv* files, each file containing direct images URL from *Flickr*. In addition, classes names and images metadata are also available to download as *.csv* format. In our case, we have used a subset from the training set that contains only 2,400,899 images, and using *Flickr* API, we have extracted each image tags added by users. This will help later in text-based image search.
The dataset can be explored and more inspected at: Open Images Dataset Visualizer

2.2 Data preparation

The dataset is partially clean, but for this project, additional data preparing steps were needed. The following figure presents a general overview of the three main steps and components to prepare the dataset.

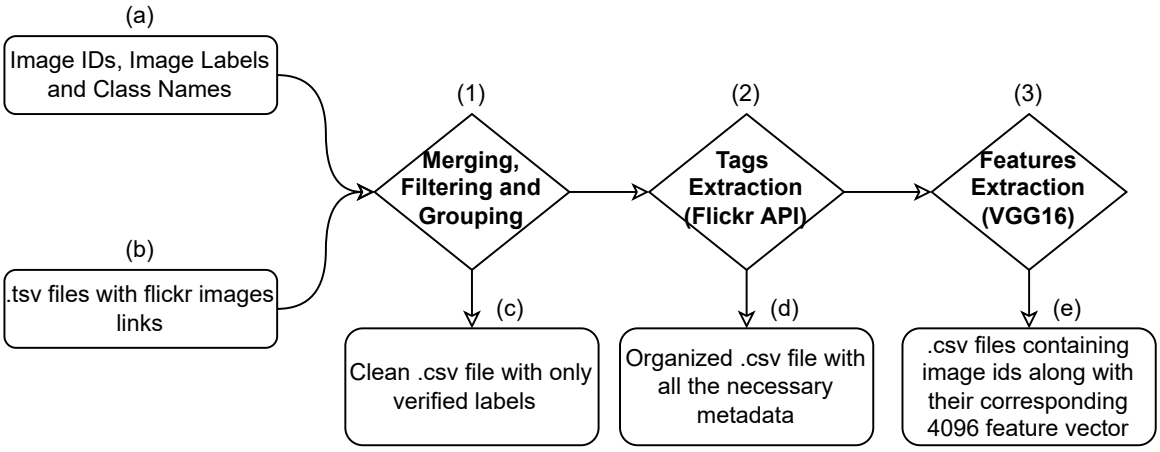


Figure 2: Data preparation pipeline

- **(a):** The first component contains three files that are provided as part of the dataset: Image IDs, Image Labels and Class Names.

  1. The Images IDs file contains image URLs, their OpenImages IDs, the rotation information, titles, authors, and licence information.

```
ImageID,Subset,OriginalURL,OriginalLandingURL,License,AuthorProfileURL,Author,Title,OriginalSize,OriginalMD5,Thumbnail300KURL,Rotation
fe600639,validation,https://farm2.staticfl,https://www.flic,https://cre,https://www.fli,LabHa,_GUT6674,242145,0jBp,https://c8.staticfli,0.0
ba82c70c,validation,https://farm4.staticfl,https://www.flic,https://cre,https://www.fli,Stock,SE_SSA,1658910,qMzF2,https://c7.staticfli,0.0
e3ffa4c8,validation,https://farm6.staticfl,https://www.flic,https://cre,https://www.fli,Ismaïl,ct26,1085575,MXOLn,https://c3.staticfli,0.0
```

Figure 3: Images IDs sample

  2. The Image Labels file contains Human-verified and machine-generated image-level labels. The confidence value differentiates the latter. Labels that are human-verified to be present in an image have confidence = 1.

```
ImageID,Source,LabelName,Confidence
0001eeaf4aed83f9,verification,/m/0cmf2,1
0004886b7d043cfd,verification,/m/01g317,0
0004886b7d043cfd,verification,/m/04hgtk,0
```

Figure 4: Image Labels sample

  3. The Class Names file contains the classes codes and their corresponding names. Ex: */m/0100nhbf -> Sprenger's tulip*

- **(b):** The second component is a set of *.tsv* files where each file contains 1 million image links from Flickr.

```
TsvHttpData-1.0
https://farm8.staticflickr.com/2454/3941286700_4f63e50bc4_o.jpg 507924  MomJ53PDC/u/yKVFxwI+FQ==
https://farm8.staticflickr.com/121/279815194_1aa7e4e72c_o.jpg   28823   QYEPgm1jK5PwDUh9f/0wKA==
https://c6.staticflickr.com/1/163/416261411_26474273df_o.jpg    1083312 83QJQHJeBVe9CBMIArLmRQ==
```

Figure 5: OpenImages Dataset Train sample

- **(c):** The third component results from some cleaning operations performed during **step (1)**. All the previously mentioned files are joined together and then filtered only to keep the Human-Verified labels, and then the files are grouped such that every image has all of its labels.

```
OriginalURL,Label,Author,Title,id,ImageID
https://c1.staticfl,"Plant, Person, Tree, Horse, Animal, Zebra",Steve Jurvetson,Making a Zebra,1028428,ad0d214ecd7cdbf8
https://c1.staticfl,Toy,tracy ducasse,the view,1037372,f25f0f6531740d98
https://c1.staticfl,"Person, Clothing, Microphone, Musical",j bizzie,2004-10G,1049564,540f794272660128
```

Figure 6: Clean file with 2,400,899 row sample

- **(d):** The fourth component is only the previous component but with an additional column that contains users added tags from Flickr. In this **step (2)**, we used six virtual machines from Amazon AWS to perform API calls using the Flickr API. The dataset was divided on all the instances ($\sim$ 400,000 images on each instance), and the total running time was about 30 hours.

```
OriginalURL,Label,Author,Title,id,ImageID
https://c1.staticfl,"Plant, Person, Tree, Horse, Animal, Zebra",Steve Jurvetson,Making a
Zebra,"horse,zebra,spraypaint",1028428,ad0d214ecd7cdbf8
https://c1.staticfl,Toy,tracy ducasse,the view,"the bees,everglade,vacation,clouds,the view",1037372,f25f0f6531740d98
https://c1.staticfl,"Person, Clothing, Microphone, Musical",j bizzie,2004-10G,"organ,horn,performance,electronic
music",1049564,540f794272660128
```

Figure 7: Final Clean dataset ready for indexing

- **(e):** The final component is a file that contains extracted features from images.
  As the images are hosted in Flickr, we developed a simple python script that uses threading to download images locally and extract features parallelly. This method has saved us much time compared to a sequential approach. We initially used an RTX 3080 TI graphics card and local internet connection (100 Mbits), but then we switched to four virtual machines from Amazon AWS with an internet speed of 600 Mbits and NVMe hard drives along with a GTX 1080 TI.
  The process of extracting features from all images took around 50 hours.
  Pretrained *VGG16* (figure 8) and *imagenet* weights were used for feature extraction as almost all the classes of the Open Images Dataset are present in the imagenet dataset. The result of the model is a feature vector with 4096 features. After inspecting the features vectors, we noticed that vectors have many zeros.
  The zeros usually don't add much information when computing similarity, but they occupy storage space, memory, and CPU. We should reduce the dimensionality while preserving most of the data. Thus, we used Principal Component Analysis to reduce dimensionality. And to find the best number of features to keep, we have used some simple cumulative explained variance shown in figure 9. We ended up using 785 components for dimensionality reduction.

  As we have more than 2.4 million images, which means 2.4 million feature vectors, it

was nearly impossible to load all the data into memory to apply PCA.
A workaround for this problem was to load a notably large file with more than 300,000 rows, perform PCA and then save the resulting model. The rest of the data was split into small files, each having 10,000 rows, and then we applied PCA on these files one by one using the previously saved model. By doing this, we ensures that all the feature vectors were reduced using the same PCA base.
This method will be helpful later when trying to search by image similarity. We need to extract features from the query image and then apply PCA using the same model we used to reduce the dimensionality of the data.

Figure 8: VGG16 architecture

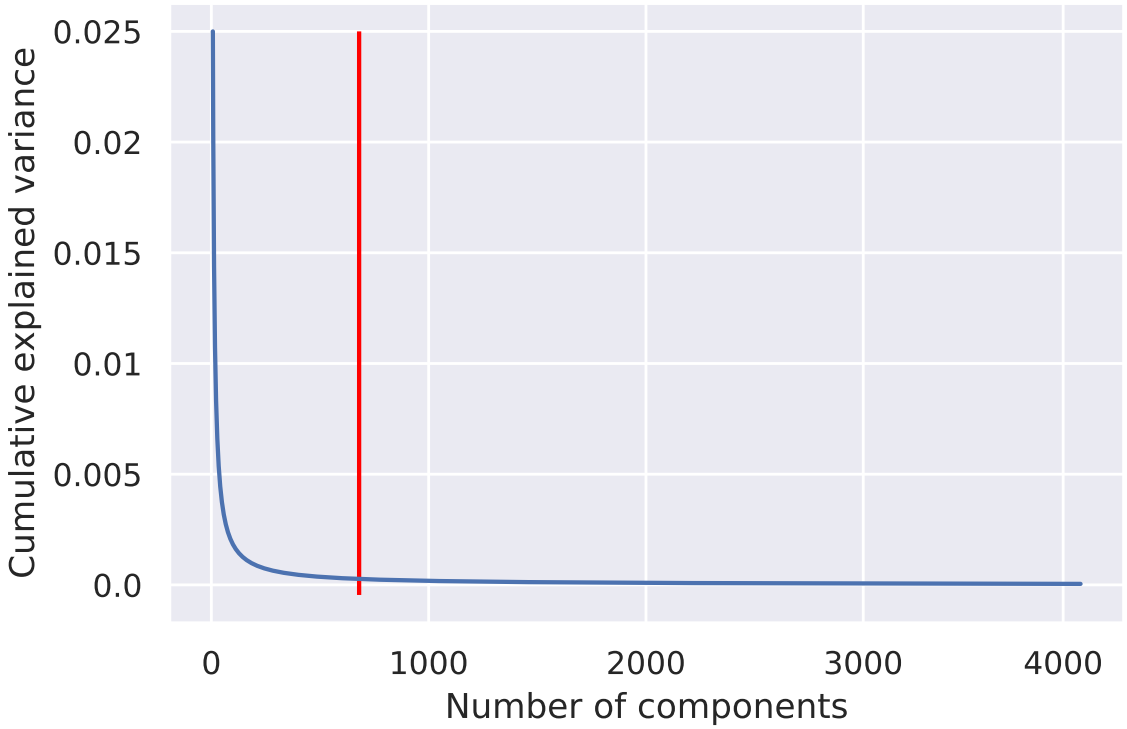


Figure 9: Cumulative explained variance for PCA

3 Data indexing

3.1 Inspiration and Tools

This project provides both content-based image retrieval and text-based image retrieval.
For this, text and images need to be indexed separately. Elasticsearch[2], an open-source RESTful search and analytics engine, is the heart of this work.
Images metadata (Classes, Tags, Author, and Title) are indexed into an elasticsearch index. This enables text search at very high speeds (around 0.1 seconds in our index).

The second step was to index the images for performing a content-based search.
ANN or *Approximate Nearest Neighbours*[3] is the method used in this project.
In large databases of data, Exact Nearest Neighbours is an impractical search method as it requires a long search time; this makes ANN come into the game. A similarity search can be orders of magnitude faster if we're willing to trade some accuracy. Approximate Nearest Neighbor techniques speed up the search by preprocessing the data into an efficient index and are often tackled using these phases:

- **Vector Transformation:** Applied on vectors before they are indexed, there is dimensionality reduction and vector rotation.
- **Vector Encoding:** Applied on vectors in order to construct the actual index for search, amongst these, there are data structure-based techniques like Trees, LSH, and Quantization, a technique to encode the vector to a much more compact form.
- **None Exhaustive Search Component:** Applied on vectors to avoid exhaustive search, there are Inverted Files and Neighborhood Graphs amongst these techniques.

Fortunately, various open-source projects provide easy ANN implementation.
One of the most prominent solutions is *Annoy*[4], which uses trees (forests) to enable Spotify's music recommendations.
PyNNDescent[5], on the other hand, is a Python nearest neighbor descent for approximate nearest neighbors. It provides a python implementation of Nearest Neighbor Descent for k-neighbor-graph construction and approximate nearest neighbor search.
Another library is *NGT*[6], developed by Yahoo. It enables Nearest Neighbor Search with Neighborhood Graph and Tree for High-dimensional Data.
As we are working with Elasticsearch, we have used a plugin called *Elastiknn*[7]. Elasticsearch is indeed a ubiquitous search solution, but its support for vectors is limited. This plugin fills the gap by bringing efficient exact and approximate vector search to Elasticsearch. This enables combining traditional queries with vector search queries for an enhanced search experience. Thus, Elasticsearch + Elastiknn is one of the best and easiest solutions to use.
Table 2 summarises some of the pros and cons of Elastiknn vs. some other libraries like Annoy and NGT.

Table 2: Some Pros. and Cons. of Elastiknn vs. Annoy/NGT

	Elastiknn	Annoy/NGT
Pros	- Datatypes to efficiently store dense and sparse numerical vectors in Elasticsearch documents, including multiple vectors per document. - Approximate queries using Locality Sensitive Hashing for L2, Cosine, Jaccard, and Hamming similarity. - Integration of nearest neighbour queries with standard Elasticsearch queries. - Incremental index updates. Create/ update/delete documents and vectors without ever re-building the entire index.	- Implemented with C/C++ and provides python bindings. - Operate without network I/O. - Operate entirely in memory or using indexes stored on the hard drive. - Use multithreading and parallelization. - Supported distance functions: L1, L2, Cosine similarity, Angular, Hamming, Jaccard, Poincare, and Lorentz. - Implement Neighborhood Graph and Tree for Indexing High-dimensional Data with Quantization. - Share index across processes.
Cons	- Executes entirely in the Elasticsearch JVM and is implemented with existing Elasticsearch and Lucene primitives. - Issues an HTTP request for every query. - Stores vectors on disk and uses zero caching beyond the caching that Lucene already implements. - Query latency typically scales inversely with the number of shards.	- Needs separate indexing and handling from Elasticsearch. - Need for index rebuild when adding new entries.

3.2 Indexing process

The process of indexing the data was mainly split into two tasks:

1. First, we have to index the text data. An index was create with the following fields *'imageId', 'title', 'author', 'tags', 'labels', 'imgUrl'*. The *author* and *imgUrl* fields were not used to index to save space and as it's not crucial for our project (Code snippet 3.1). Then using the preprocessed and clean data and the python client of Elasticsearch, all the data got indexed inside of elasticsearch (Code snippet 3.2). When preparing the index configuration, we had to enable the use of Elastiknn and set its parameters. Doing this will allow the indexing of the feature vectors later on.

Code snippet 3.1: Index settings and creation

```
index = 'open-images'
source_no_vecs = ['imageId', 'title', 'author', 'tags', 'labels', 'imgUrl']
# As we are using a single machine to index the data, and to save storage,
# The number of replicas was set to 0, and the number of shards to 1.
settings = {
  "settings": {
    "elastiknn": True,
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```

```

mapping = {
  "dynamic": False,
  "properties": {
      "imageId": { "type": "keyword" },
      "featureVec": {
          "type": "elastiknn_dense_float_vector",
          "elastiknn": {
            "dims": 785,
            "model": "lsh",
            "similarity": "l2",
            "L": 60,
            "k": 3,
            "w": 2
          }
    },
    "title": { "type": "text" },
    "author": { "type": "text","index":False},
    "tags": { "type": "text" },
    "labels": { "type": "text" },
    "imgUrl":{"type":"text","index":False}
  }
}

if not es.indices.exists(index):
    es.indices.create(index, json.dumps(settings))
    es.indices.put_mapping(json.dumps(mapping), index)
es.indices.get_mapping(index)
```

Code snippet 3.2: Index text data for text-based image search

```
# Load the metadata of the images
meta = pd.read_csv("meta/images_meta.csv",dtype={'id': str})

def text_actions():
  """ Yield a dictionary that maps a row into the index format. """
  for p in tqdm(meta.iterrows()):
    yield {
          "_op_type": "index", "_index": index, "_id": p[1]["ImageID"],
          "imageId": p[1]["ImageID"], "title": p[1]["Title"], "tags":
          ↪  p[1]["Tags"],"labels": p[1]["Label"],
          "imgUrl": p[1]["OriginalURL"]
    }
# Insert the text data into the index
bulk(es, text_actions(), chunk_size=2000, max_retries=2)
# Refresh the index
es.indices.refresh(index=index)
# Merge the data and reindex (Similar to an update function)
# After this step, all the text data will be stored and indexed and text
↪  search can be executed
es.indices.forcemerge(index=index, max_num_segments=1, request_timeout=300)
```

2. Second, we need to index the feature vectors inside elasticsearch index. The process is very similar to indexing text, which is the benefit of using Elastiknn. The index can be

updated without recreating it and reindexing text again.
We used L2 LSH Mapping for mapping the feature vectors, and this is because of the effectiveness of this method when used with feature vectors extracted by deep learning models, which is the case.
L2 LSH Mapping Uses the Stable Distributions method[8] to hash and store dense float vectors such that they support approximate L2 (Euclidean) similarity queries. All the settings using for indexing the feature vectors are detailed in code snippet 3.3. Feature vectors indexing is then as easy as indexing text data. The only thing to consider is the number of vectors to index in each iteration, as dense float vectors are way heavier than strings. In the code snipped 3.4 are detailed the steps to update the elasticsearch index with images feature vectors.

Code snippet 3.3: L2 LSH mapping settings

```
"featureVec": {
          "type": "elastiknn_dense_float_vector", # Vector datatype (dense)
          "elastiknn": {
            "dims": 785, # Vector dimensionality. After appliying PCA, we
            ↪  have reduced vector size from 4096 to 785.
            "model": "lsh", # Model type
            "similarity": "l2", # L2 Similarity
            "L": 60, # Number of hash tables. Generally, increasing this
            ↪  value increases recall. We found that 60 delivers a decent
            ↪  recall at faster speed
            "k": 3, # Number of hash functions combined to form a single
            ↪  hash value. Generally, increasing this value increases
            ↪  precision.
            "w": 2, # Integer bucket width. This determines how close two
            ↪  vectors have to be, when projected onto a third common
            ↪  vector, in order for the two vectors to share a hash value.
          }
```

Code snippet 3.4: Update index with text data with images feature vectors

```
def vector_action(data):
  """ Yield a dictionary that maps a row into the index format.
  We need to convert the feaure vector from the csv file to a numpy array.
  """
  for v in tqdm(data.iterrows()):
    vec = v[1].to_numpy()
    yield { "_op_type": "update", "_index": index, "_id": vec[-1],
        "doc": {
          "featureVec": { "values": vec[:-1].tolist()}
        }}

def vector_actions():
  """ Insert feature vectors into the index"""
  bulk(es, vector_action(data), chunk_size=50, max_retries=10,
   ↪  request_timeout=60)

# Loop over all the files that contains feature vectors and map them one by
↪  one
for i in os.walk("features/"):
  for feature in i[2]:
```

```
    # Read the csv file
    data = pd.read_csv(f"features/{feature}")
    # Remove Unnamed columns if exists
    data = data.loc[:, ~data.columns.str.contains('^Unnamed')]
    # Index the feature vectors
    vector_actions()

# Refresh the index
es.indices.refresh(index=index)
# Merge the data and reindex (Similar to an update function)
# After this step, all the feature vectors will be stored and indexed and
↪  content-based image search using ANN can be executed
es.indices.forcemerge(index=index, max_num_segments=1,
↪  request_timeout=1200)
```

The final index holds both text and feature vectors data and the total size of the index zipped is 24 Gb.

4 API development

In order to define the interactions that a user can perform with the backend of the search engine through the frontend application, we designed a custom API using ElasticSearch python client and FastAPI.

4.1 ElasticSearch client

We used ElasticSearch python client to build function for custom search including multiple features such as fuzzy search, field boosting, search by image link, search by image ID.. First, we start by initializing an ElasticSearch instance and then we use the built-in search function to which we pass the index name and the query body.

4.1.1 Text search

Text search enables users to search for images within the index using multi-field matching (label, title and tags). The following code snippet shows the signature of the text search function.

Code snippet 4.1: Text search function signature

```
def search_by_text_query(self, query: str,
                         fields: Optional[Dict[str,int]] = {"labels":5, "tags":1,
                         ↪  "title":2},
                         use_fuzzy: Optional[bool] = False,
                         size: Optional[int] = 5,
                         from_: Optional[int] = 0)
                      """  Query body  """
 return es.search(index=INDEX_NAME, query=q, size=size, _source=SOURCE_NO_VEC,
  ↪  from_ = from_)
```

This function supports:

- *Fuzzy search:* fuzzy queries enable search for similar expressions as the query, in case the user misspells a word for example.
- *Field boosting:* assign a weight to a field in order to emphasize its importance in the search response.
- *Response document size and beginning index:* to control the number of the returned results and where to start returning them.

4.1.2 Image search

Image search is much more complex, we define 3 different methods for searching:

- *Search by image*: takes in an image query, extracts its features, applies PCA. Next, it performs search using approximate nearest neighbours(ANN), provided by elastiknn plugin to return images having feature vectors similar to the query.
  The query body is shown in the code snippet below, depicting the parameters used for ANN: we use L2 distance (euclidean) and Locality Sensitive Hashing (LSH) to calculate similarity between feature vectors efficiently and quickly.

Code snippet 4.2: Image search query

```
    features = fe.get_from_image(image)
    query = {
                "elastiknn_nearest_neighbors": {
                "vec": {
                    "values": features
                },
                "field": "featureVec",
                "model": "lsh",
                "similarity": "l2",
                "candidates": candidates
                }
            }

```

- *Search by image link*: gets an image from a URL and applies same procedure as normal image search.
- *Search by image ID*: Used for recursive search in case the user wants to show images that are similar to an image present in the search results.

4.1.3 Image and text search

In this part, we use ElasticSearch's "function score" query to combine both types of search. It works as follows: a text search query is executed first, we retrieve the result and then apply the image search on it. This reduces computations and gives more accurate results for the desired query.

Code snippet 4.3: Image and text search query

```
query = {
        "function_score": {
        "query": {
            "bool": {
            "filter": {
                "exists": {
                "field": "featureVec"}},
            "must": {"multi_match": mm}
                    }
                },
        "boost_mode": "replace",
        "functions": [{
            "elastiknn_nearest_neighbors": {
            "field": "featureVec",
            "similarity": "l2",
            "model": "lsh",
            "candidates": condidates,
            "vec": {"values": features}},
            "weight": 2}]
            }
        }
```

4.2 API Endpoints

In order to build a well-defined API schema, we use FastAPI. FastAPI is a web framework for building APIs in python. It offers fast, easy and robust development and it's compatible with open standards.
For each function mentioned in the previous section corresponds an endpoint that grants access to the offered resources. Figure 10 shows each endpoint URL, the request made at this endpoint and the executed operation.

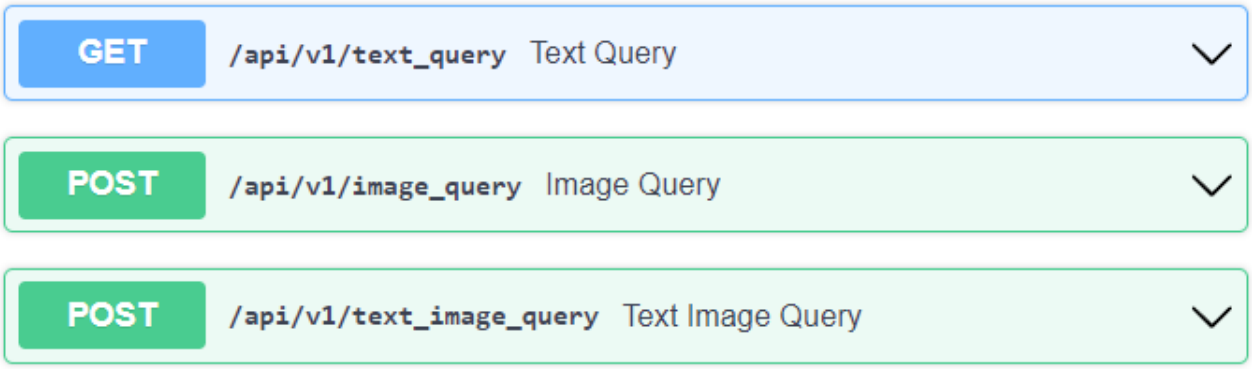


Figure 10: API endpoint swagger documentation

5 Web App development

To enable users to get the best experience out of the search engine, we developed a web application using Streamlit. Streamlit is a python framework that enables fast development of clean and clear user interfaces for data projects.
The welcome page includes a description of the project and how to use it. In the sidebar, the user can write his write and customize his search queries as shown in figure 11.

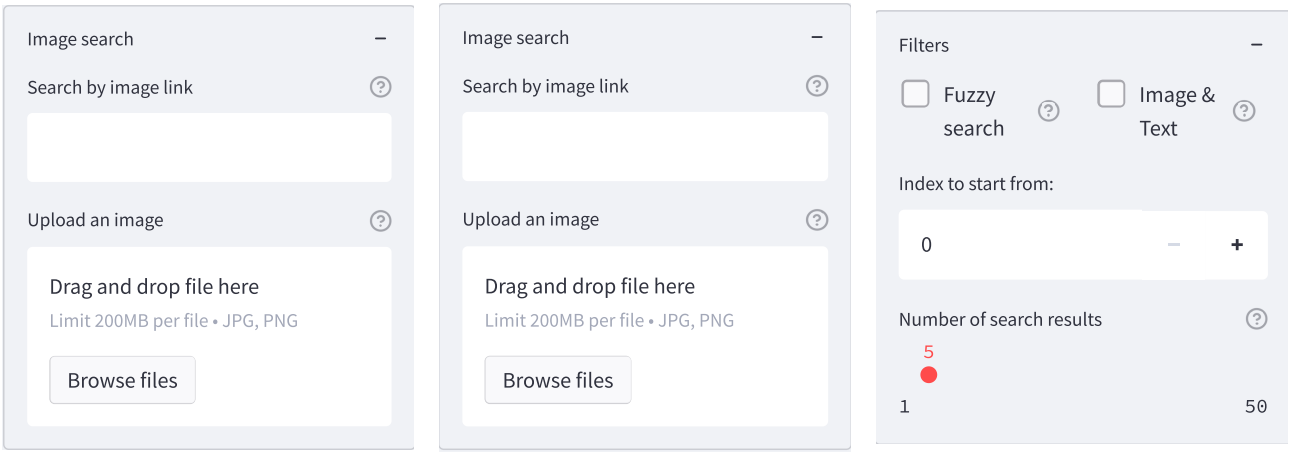
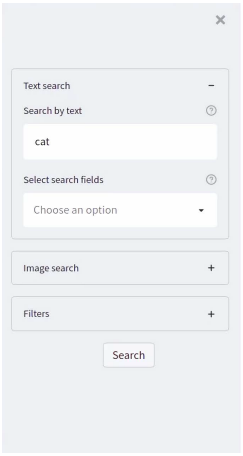


Figure 11: Search tools

The number of search results is displayed above along with the response time. For each hit, we display the image with its title, link, labels and it's score. The results are sorted according to the highest score.



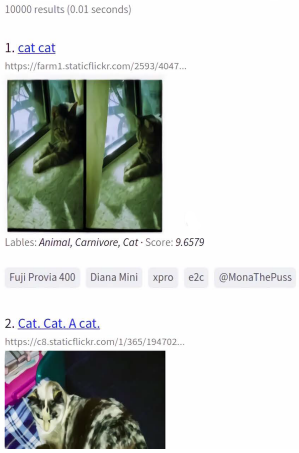
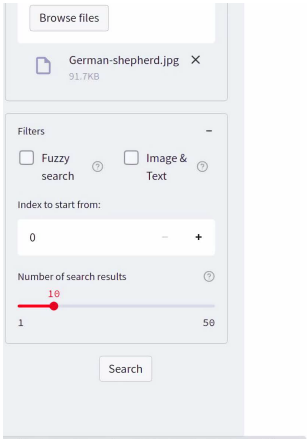


Figure 12: Text search result (cat)



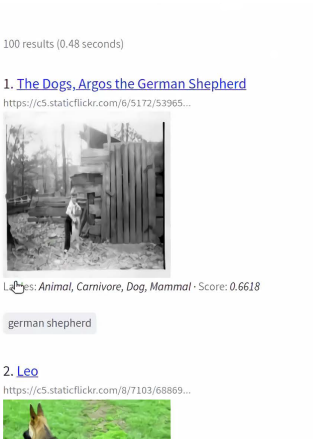


Figure 13: Content-based search result (dog)

6 Deployment

The final phase of the project is deploying it. To do so we needed to:

- Host the ElasticSearch index on a public server to make it accessible for distant users.
- Containerize the application (backend and frontend) using Docker. This makes it easier to automatize running the application on any system : there is no need to install software dependencies, Docker takes care of that.

To dockerize the app, we proceed as follows:

1. Write Dockerfile for the backend. A Dockerfile contains a set of instructions to execute when running a Docker image. For the backend, we use it to install package dependencies (Tensorflow, ElasticSearch client, FastAPI etc..), to expose the port on which the API listens and finally to run the python app.

Code snippet 6.1: Backend Dockerfile

```
FROM python:3.8-slim

WORKDIR /app

RUN apt-get update
RUN apt-get install \
    'ffmpeg'\
    'libsm6'\
    'libxext6'  -y

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

EXPOSE 8080

CMD ["python", "main.py"]

```

2. Write Dockerfile for the frontend (same as the backend)

Code snippet 6.2: Frontend Dockerfile

```
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

EXPOSE 8501

CMD ["streamlit", "run", "main.py"]

```

3. Write docker-compose.yml: needed to run multiple containers simultaneously. In this file we set port mappings, containers dependency over other and storage volumes for containers.

Code snippet 6.3: docker-compose.yml

```
version: '3'

services:
  frontend:
    build: frontend
    ports:
      - 8501:8501
    network_mode: host
    depends_on:
      - api
    volumes:
        - ./storage:/storage
  api:
    build: api
    ports:
      - 8080:8080
    network_mode: host
    volumes:
      - ./storage:/storage

```

4. run docker−compose up −−build to build and start the search engine.

7 Conclusion

This project is very rich in terms of employed technologies and acquired knowledge. It covered multiple aspects of dealing with large datasets and leveraging it to develop an application and getting it into production.

First, we assembled and cleaned the data to fit it to our use case. Then, we transformed the raw images into useful features using VGG16, a trained convolutional neural network. Later, we managed to reduce the dimensionality of the data, this reduces storage size and optimizes search response time.

The next step is to index the available data (features and metadata) using ElasticSearch and leveraging elastiknn plugin. This allows to search using text (ElasticSearch Search API) and using images, i.e features, with Approximate Nearest Neighbours (elastiknn).

To gain full control, we developed an API that gives access to different search functionalities using elastic python client and FastAPI. The API is consumed in a Streamlit web application that enables users to customize their search and display results.

Finally, we deployed the data index on a public server and dockerized the search engine app to make it reproducible and usable on other machines.

References

[1] Open Images Dataset. A dataset of 9 million varied images with rich annotations. https://storage.googleapis.com/openimages/web/factsfigures.html, 2021.

[2] Elasticsearch. What is elasticsearch? https://www.elastic.co/what-is/elasticsearch, 2021.

[3] Eyal Trabelsi. Comprehensive guide to approximate nearest neighbors algorithms. https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6, 2020.

[4] Spotify. Annoy. https://github.com/spotify/annoy, 2021.

[5] Leland McInnes. Pynndescent. https://github.com/lmcinnes/pynndescent, 2021.

[6] yahoojapan. Ngt. https://github.com/yahoojapan/NGT, 2021.

[7] Alex Klibisz. Elastiknn. https://github.com/alexklibisz/elastiknn, 2021.

[8] Wikipedia contributors. Locality-sensitive hashing. https://en.wikipedia.org/wiki/Locality-sensitive_hashing#Stable_distributions, 2021.