SUP'COM
Ecole Supérieure des Communications de Tunis

جامعة قرطاج
Université de Carthage

Content and Text Based Image Retrieval Engine

Realized by: Oueslati Wissal

wissal.oueslati@supcom.tn

Mezghani Mariem

mariem.mezghani@supcom.tn

Supervisor: Riadh Tebourbi

riadh.tebourbi@supcom.tn

Academic year: 2023-2024

Abstract

In last few years, the complexity of multimedia contents, especially the images, has grown exponentially, and on daily basis, more than millions of images are uploaded at different archives such as Twitter, Facebook, and Instagram. To search for a relevant image from an archive is a challenging research problem for computer vision research community. Most of the search engines retrieve images based on traditional text-based approaches that rely on captions and metadata. In the last two decades, extensive research is reported for content-based image retrieval (CBIR), image classification, and analysis. In CBIR and image classification-based models, high-level image visuals are represented in the form of feature vectors that consists of numerical values.

In this context, his report outlines the steps taken to create a comprehensive text-based and content-based search engine, encompassing tasks such as data preparation, as well as API and UI/UX development.

The source code and the dataset used are all available at: https://github.com/mariem-mezghani/Search-Engine

Keywords: Approximate Nearest Neighbors; Photo Metadata Dataset; Resnet; Features extraction; Search Engine; Elasticsearch; Streamlit.

Contents

List of figures

Code snippets

1. Introduction

1.1. Context:

Progress in data storage and image capture technologies has opened the door to the creation of extensive image collections. These collections serve as a foundation for various applications that harness the power of Artificial Intelligence algorithms and Big Data technologies. Among these applications, one of the notable ones is image search, which aims to identify images meeting specific criteria. Achieving image search relies on two primary approaches: text-based image retrieval and content-based image retrieval (CBIR).

In the first approach, images are sought through textual metadata, such as descriptions, labels, and keywords. However, this method demands the annotation of all images in a database, a daunting task for large datasets, and often falls short of delivering precise search results aligned with user queries.

To circumvent this challenge, content-based image retrieval techniques have been developed. CBIR engines enable the search for images using other images as queries, returning similar images as results. This approach streamlines the search process by eliminating the need for image annotation or the inclusion of keywords as queries. In a nutshell, a CBIR algorithm takes a query image, extracts its distinctive features, and then returns images with feature vectors resembling those of the query image.

1.2. Project Overview

This project aims to create a hybrid image search engine that utilizes one of the mentioned techniques. It utilizes the Flickerphotos Dataset to search through images alongside their associated textual metadata. Managing such a vast amount of data presents challenges concerning result accuracy and response time.

The developed solution incorporates various technologies, including data processing, machine learning, data indexing and querying, software development, and deployment. The system's architecture can be understood by breaking it down into two core processes: Data processing, which covers everything from feature extraction to indexing, and image querying. Figure 1 provides a summary of these two processes. Users can interact with the system through a web application, enabling them to submit and customize their queries and view the results.

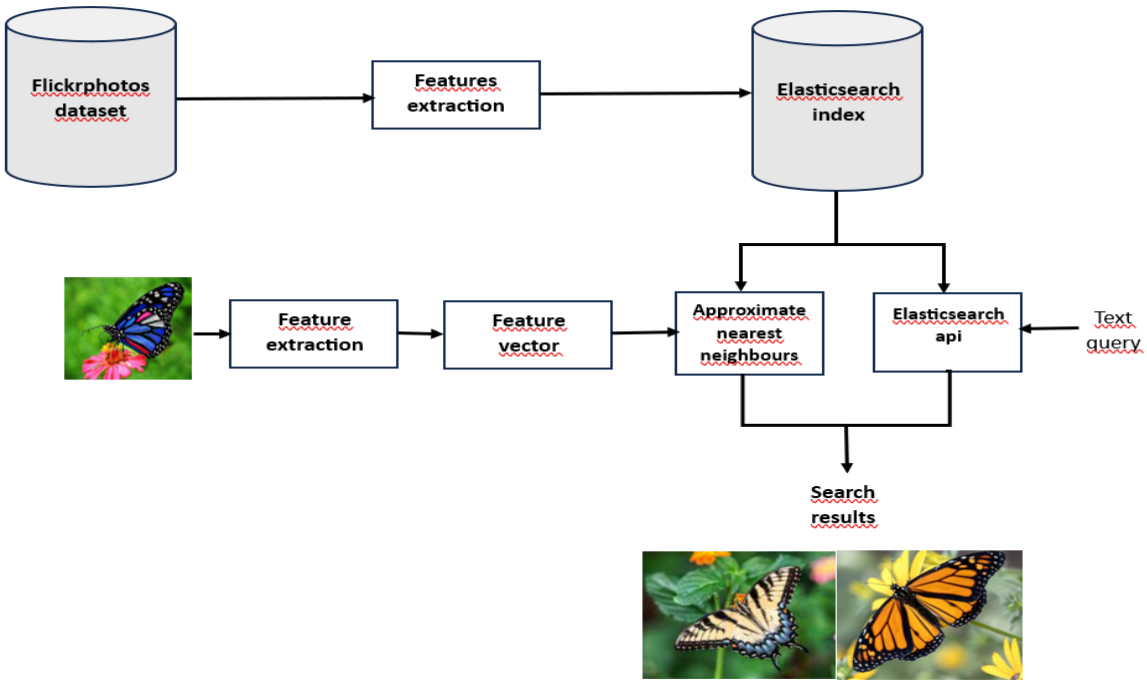

Figure 1: Project overview

2. Dataset

2.1. Data description:

The dataset employed in this project is the expansive Flickrphotos dataset, boasting a staggering 100 million images sourced from Flickr. This dataset, totaling 16 gigabytes, is not only substantial in size but also rich in metadata that enhances the depth and context of each image.

Each image in the dataset comes with a wealth of information, including geographical coordinates (X, Y) through geotagging, user-related details such as "userid," and descriptive attributes like "title" and "tags." This metadata provides a comprehensive understanding of the images, capturing both spatial and semantic aspects.

Furthermore, the dataset is cloud-accessible through Flickr, offering convenient and flexible access for retrieval and analysis. The data is structured in a CSV file named "photo_metadata.csv," featuring essential fields such as "id," "views," "date_taken," "date_uploaded," "accuracy," and various Flickr-specific parameters like "flickr_secret" and "flickr_farm."

This large CSV file, encapsulating the metadata of the images, serves as a robust foundation for our image search engine.

2.2. Data preparation:

In this project, we leverage the Flickrphotos dataset as our primary source of images. Before indexing this dataset, we employ the *tf_efficientnet_b0_ns* model for feature extraction from the image URLs. This process allows us to distill key features from each image, capturing essential visual information.

The extracted features are then organized into a structured format by creating a CSV file. This file contains two essential pieces of information for each image: the unique identifier (ID) and the corresponding feature vector. The ID serves as a reference point for each image, facilitating easy retrieval and identification, while the feature vector encapsulates the distinctive visual characteristics extracted by the tf_efficientnet_b0_ns model.

By adopting this approach, we create an efficient and accessible representation of the image dataset, enabling seamless integration into our hybrid image search engine. The CSV file becomes a valuable resource for quick and targeted image querying, providing a foundation for precise and responsive search results. This process contributes to the overall architecture of our system, optimizing the handling of large datasets and enhancing the user experience in retrieving relevant images.

The tf_efficientnet_b0_ns [2] model is a variant of the EfficientNet architecture, specifically designed for efficient and effective neural network-based image processing tasks. The "tf" prefix indicates that the model is implemented using the TensorFlow framework. The "b0" in its name signifies the baseline architecture, and "ns" typically denotes the "noisy student" training approach.

3. Data indexing

3.1. Tools

This project offers two distinct retrieval methods: content-based image retrieval and text-based image retrieval. To support these functionalities, text and images are indexed separately. Elasticsearch, an open-source RESTful search and analytics engine, serves as the core technology in this project.
The *photo_metadata*, including fields such as "id," "userid," "title," "tags," "latitude," "longitude," and others, is indexed into an Elasticsearch index. This indexing process enables remarkably fast text searches, with query responses typically taking around 0.1 seconds in our specific index setup.

The subsequent phase involved indexing the images to facilitate content-based searches. In this project, the chosen method was KNN, short for *K-Nearest Neighbors*.

In extensive databases, the use of Exact Nearest Neighbors becomes impractical due to its time-consuming nature. This limitation prompts the adoption of KNN, which significantly accelerates similarity searches. K-Nearest Neighbor techniques enhance search efficiency by preprocessing the data into an efficient index [1].

- KNN classifier does not have any specialized training phase as it uses all the training samples for classification and simply stores the results in memory.
- KNN is a non-parametric algorithm because it does not assume anything about the training data. This makes it useful for problems having non-linear data.

3.2. Indexing process

The data indexing process was divided into two primary tasks:

1. Text Data Indexing:
   Initially, we created an index with specific fields including "id," "userid," "title," "tags," "latitude," "longitude," and others as listed in *Code snippet 3.1*. Subsequently, utilizing the preprocessed and refined data along with the Python client of Elasticsearch, all the data was indexed within Elasticsearch *(Code snippet 3.2)*.

```
{ "mappings": {
        "properties": {
          "id": {"type": "text","index" : true},
          "userid": {"type": "text","index" : true},
          "title": {"type": "text","index" : true},
          "tags": {"type": "text","index" : true},
          "latitude": {"type": "double"},
          "longitude": {"type": "double"},
          "views": {"type": "integer"},
          "date_taken": {"type": "date","format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd"},
          "date_uploaded": {"type": "date","format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd"},
          "accuracy": {"type": "short"},
         "flickr_secret": {"type": "keyword","index" : true},
         "flickr_server": {"type": "keyword","index" : true},
         "flickr_farm": {"type": "keyword","index" : true},
         "x": {"type": "keyword","index" : true},
         "y": {"type": "keyword","index" : true},
          "z": {"type": "keyword","index" : true},
      "location" : {"type" : "geo_point"}
        }
    }
}
```

Code snippet 3.1: Text index mapping

```python
def create_image_vector_index():
    # Initialize Elasticsearch client
    es = Elasticsearch(hosts=["http://localhost:9200"])

    # Load the mapping from the JSON file
    with open('mapping.json', 'r') as file:
        mapping = json.load(file)

    # Check if the index already exists
    if not es.indices.exists(index='image_vector_index_final_version'):
        # Create the index with the loaded mapping
        es.indices.create(index='image_vector_index_final_version', body=mapping)
        print("Index 'image_vector_index' created successfully!")
    else:
        print("Index 'image_vector_index' already exists!")
```

Code snippet 3.2: Text index creation

2. Next, it was necessary to index the feature vectors within the Elasticsearch index. This procedure closely mirrors the process of indexing text.
   The content index mapping is presented in *Code Snippet 3.3*, providing an overview of how feature vectors are structured within the index. Further, in *Code Snippet 3.4*, detailed steps are outlined, guiding the creation of the Elasticsearch index specifically for images' feature vectors.

```json
{
  "mappings": {
      "properties": {
          "image_vector": {
              "type": "dense_vector",
              "dims": 2048,
              "index": true,
              "similarity": "l2_norm"
          },
          "image_id": {
              "type": "keyword"
          }
      }
  }
}
```

Code snippet 3.3: Content index mapping

```python
def index_feature_vectors_to_elasticsearch(csv_file_path):
    es = Elasticsearch(hosts=["http://localhost:9200"])

    with open(csv_file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # skip header row

        for row in reader:
            image_id, vector_str = row[0], row[1]
            vector = [float(i) for i in vector_str.split(',')]

            body = {
                "image_id": image_id,
                "image_vector": vector
            }

            es.index(index="image_vector_index_final_version", body=body)

    print("Indexing completed!")
```

Code snippet 3.4: Content index creation

4. API development

To outline the interactions available between the user and the search engine's backend via the frontend application, we created a custom API using the Elasticsearch Python client. This custom API serves as a bridge, enabling seamless communication and interaction between the user interface and the backend functionalities of the search engine.

4.1. Elasticsearch client

We utilized the Elasticsearch Python client to create custom search functions, incorporating advanced features like fuzzy search and the ability to search by image path. To achieve this, we initialized an Elasticsearch instance and employed the built-in search function. This function was utilized by passing the index name and the query body, enabling seamless execution of tailored search operations.

4.2. Text search

Text search functionality enables users to search for images within the index by matching the "tags" field. Below is the *code snippet 4.1* displaying the function for the text search operation.

```
if query:
    result = es.search(
        index='flickrdata',
        body={
            'query': {
                'fuzzy': {
                    'tags': query
                }
            }
        }
    )
    for hit in result['hits']['hits']:
        image_url = f"http://farm{hit['_source']['flickr_farm']}.staticflickr.com/
        {hit['_source']['flickr_server']}/{hit['_source']['id']}_{hit['_source']['flickr_secret']}.jpg"
```

Code snippet 4.1: Text search query

This function offers support for *fuzzy search*, allowing users to search for similar expressions as the query. Fuzzy queries enable the system to find relevant results even if the user misspells a word, enhancing the search accuracy and user experience.

4.3. Image search

Image search involves a more intricate process: it takes an image query, extracts its features, and then utilizes k-nearest neighbors (knn) to search for images with feature vectors similar to the query. The resulting images are returned based on this similarity. The specific query body for this operation is displayed in the *code snippet 4.2* below.

```
def search_by_feature_vector(vector):
    body = {
        "knn": {
            "field": "image_vector",
            "query_vector": vector,
            "k": 10,
            "num_candidates": 100
        },
        "_source": ["image_id"]
    }
    response = es.search(index='image_vector_index_final_version', body=body)
    return response
```

Code snippet 4.2: Image search query

"k": 10

➔This parameter determines the number of nearest neighbors (similar images) to be retrieved from the search. In this case, the query will return the top 10 images that are most similar to the user's query.

"num_candidates": 100

➔This parameter specifies the number of candidates nearest neighbors that the algorithm considers during the search process.

5. Web App development

We created a web application utilizing Streamlit, a Python framework designed for rapid development of user interfaces in data projects. Our goal was to enhance user experience with the search engine. The application's welcome page provides an overview and instructions on how to use it effectively. Users have the option to search for images either by entering text or uploading an image.

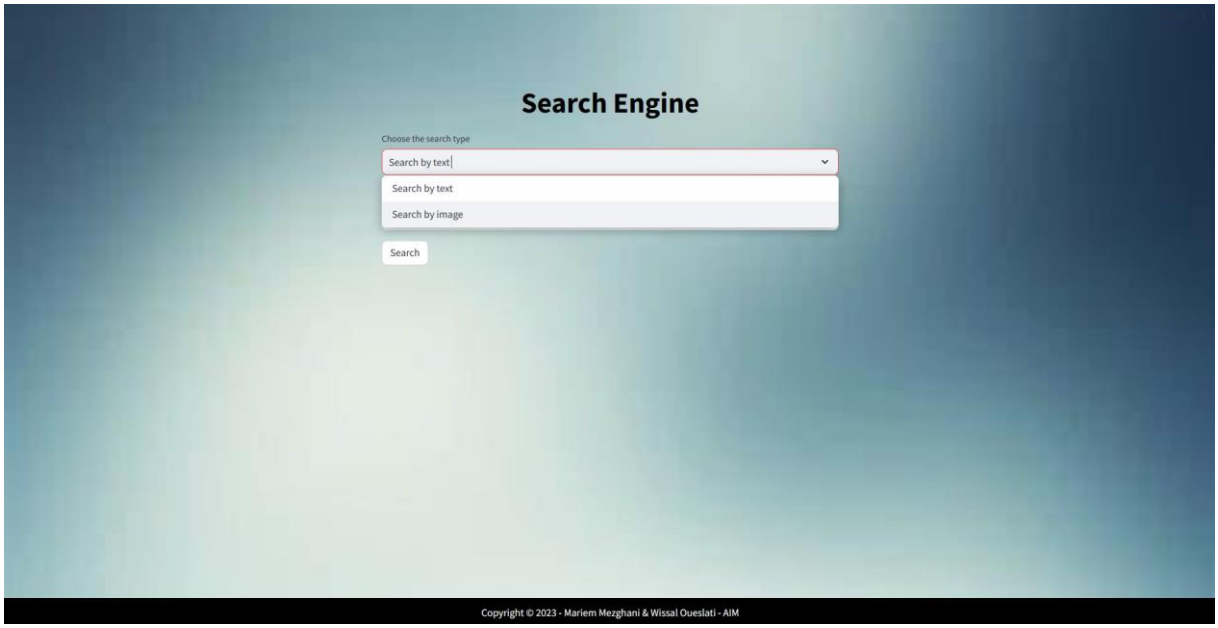

Figure 2: Search tools

Above the search results, users can see the total number of matches and the response time. Each search hit is accompanied by an image and its corresponding title. The results are organized based on their scores, showcasing the highest-rated ones first.

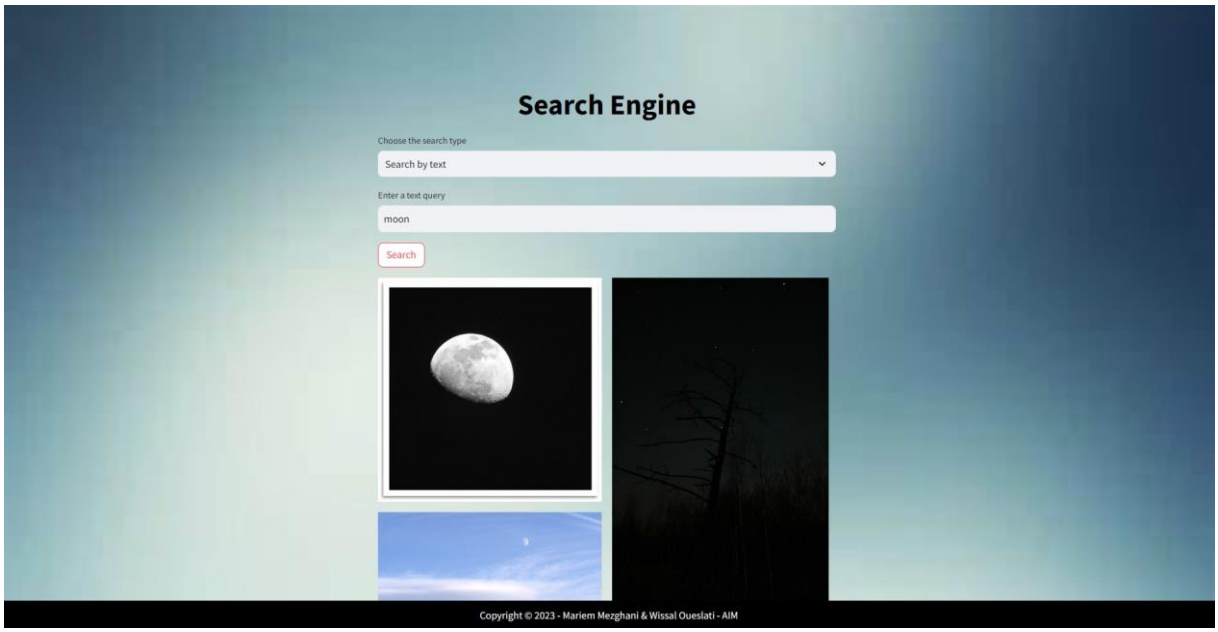

Figure 3: Text search result (moon)

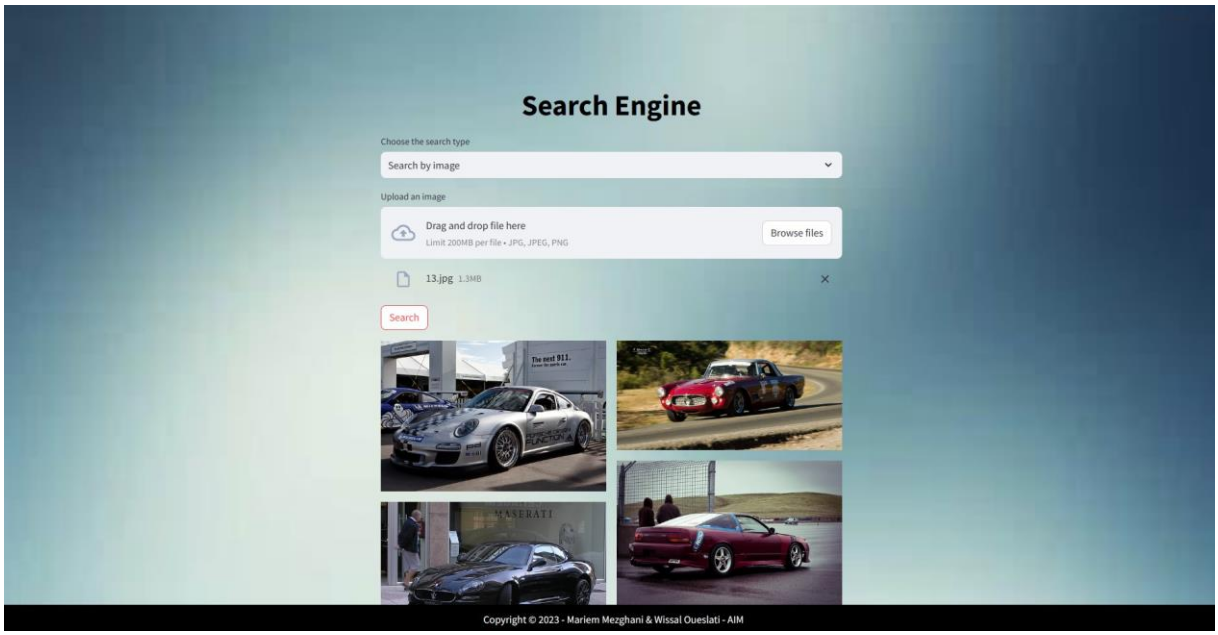

Figure 4: Content-based search result (car)

Conclusion

In summary, this project stands out for its diverse range of employed technologies and the wealth of knowledge gained throughout its development. The endeavor encompassed various aspects of managing extensive datasets and utilizing them to create a functional application.

The process began with meticulous data assembly and cleaning to tailor it to our specific requirements. Subsequently, we harnessed the power of ResNet, a trained convolutional neural network, to convert raw images into valuable features. Moving forward, we indexed the available data, including both features and metadata, utilizing ElasticSearch and Logstash. This indexing enables seamless searching through text via the ElasticSearch Search API, as well as through images, leveraging the extracted features.

In the final stages, we crafted an API that provides users with access to diverse search functionalities. This API was developed using the elastic python client offering a flexible and robust interface. This API is then consumed within a Streamlit web application, empowering users to personalize their search experience and view the results according to their preferences.

References

[1] https://towardsdatascience.com/k-nearest-neighbors

[2] https://paperswithcode.com/method/efficientnet