Multimedia Retrieval and Cloud Computing

Build and Cloud Deployment of a Multimedia Search Engine

Faculté Polytechnique de l'UMONS

António Pedro Dantas, Bernardo Costa Moreira

January 16, 2022

Abstract

The goal of this project is to develop and host a multimedia indexing and search application on Cloud resources. It is divided into two main parts, one from the Multimedia Retrieval course and other from the Cloud and Edge Computing course.

Contents

1 Introduction **3**

2 Multimedia Retrieval **4**

2.1 Context 4

2.2 Implementation 4

2.2.1 Feature Extraction 5

2.2.2 Search Engine Application 6

2.3 Result Analysis 8

3 Cloud and Edge Computing **11**

3.1 Context 11

3.2 Implementation 11

3.3 Problems encountered 14

4 Conclusion **15**

1 Introduction

The goal of this project is to develop and host a multimedia indexing and search application on Cloud resources. It is divided into two main parts, one from the Multimedia Retrieval course and other from the Cloud and Edge Computing course.

On the first part we should develop a search engine that exploits the descriptors lectured in the Multimedia Retrieval course. We must create a script that indexes a vehicles database with the descriptors of our choosing giving the possibility to combine them. It should then be able to use those descriptors to retrieve similar images to the ones we provide, using different similarity calculation functions. Afterwards, we should analyse the search results regarding their recall and precision.

The second part aims to host our multimedia search application on a Cloud or Edge resource on the form of Software as a Service, *SaaS*.

2 Multimedia Retrieval

2.1 Context

The objective of this part is to develop a search engine that exploits the descriptors lectured in the Multimedia Retrieval course. We must create a script that indexes a vehicles database with the descriptors of our choosing giving the possibility to combine them. It should then be able to use those descriptors to retrieve similar images to the ones we provide, using different similarity calculation functions. Afterwards, we should analyse the search results regarding their recall and precision.

2.2 Implementation

We've decided to develop a **Flask** app to take on this part of the project. Flask is a web framework that allows us to develop web applications using Python code. This comes in handy not only to later deploy our application easily but also to treat our resources using Python code. Our app's root folder holds the script that extracts the image's features, the search engine application, the vehicles database that will be used to test our work and a descriptors directory that saves all information regarding every image's features for every chosen descriptor.

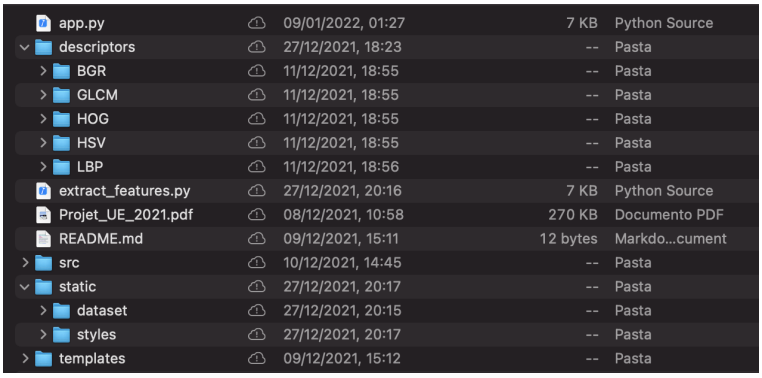

Figure 1: Application's root folder.

2.2.1 Feature Extraction

For the feature extraction part we have developed a Python script that looks into the **dataset** of all vehicle images provided and extracts their features according to the descriptors we chose for this project. The descriptors chosen were **RGB**, **HSV**, **HOG**, **GLCM** and **LBP**. The goal of these descriptors is to generate text files for every image that hold values of the features they are analysing. These values can then be compared between different images to evaluate their similarity degree.

RGB and **HSV** are two descriptors that are related to color. **RGB** creates a histogram of the image in question that holds information on the red, green and blue pigmentation of every pixel. On the other hand, **HSV** is an alternative model for color analysis on an image that describes colors (Hue) in terms of their shade (Saturation) and their brightness (Value).

HOG is a feature descriptor used for the purpose of object detection. The main concept is that by using the description of intensity gradients and edge directions we will be able to extract local shape's information.

Finally, we use **GLCM** and **LBP** for describing textures. **GLCM**, or Gray Level Co-Occurrence Matrix, characterizes the texture of an image by calculating how often pairs of pixel with specific values and in a specified spatial relationship occur in a given image. On the other hand, **LBP** describes the spatial behavior of intensity values in any given neighborhood. It first divides the picture into cells and then, for each pixel in a cell, compares the pixel to each of its neighbors. The pixels are followed along a circle, having their values updated according to the original pixel and an applied filter. The LBP value of a section is given by the sum of its pixel values.

We believe that with these choices we can clearly identify every relevant feature of an image given that we have a good combination of color, shape and texture descriptors.

The script for this part is on the *extract_features.py* file and can easily be run using the command **python3 extract_features.py**.

2.2.2 Search Engine Application

The search engine application is comprised of three main pages, the **search**, **results** and **about** page.

The **search** page lets the user choose what is the image that he wants to input and consequently find similar ones to. It gives him the option to pick the number of results retrieved and choose the combination of descriptors that he desires. We also have a section that allows him to select the similarity calculation function. For that, we have two options. One of them is the **Euclidean** distance, which compares each image feature description using a line segment between them to calculate a length. The other one is the **Bhattacharyya** distance, which measures the similarity of the descriptors using probability distributions.

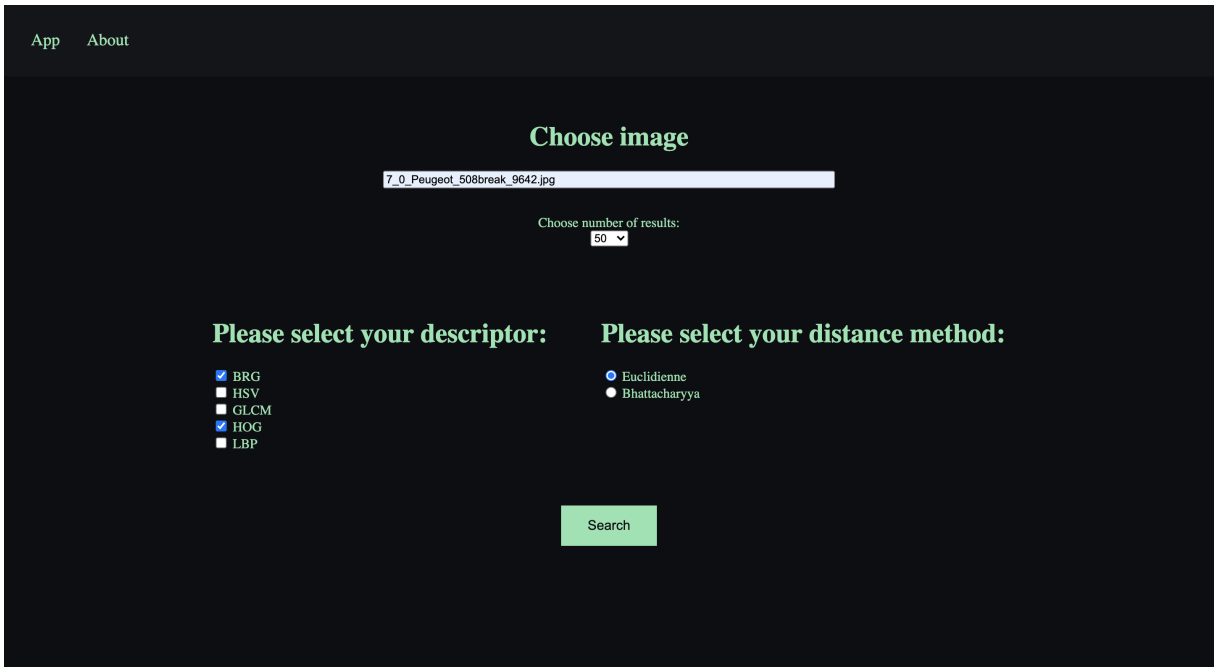

Figure 2: Search Page.

The **results** page displays in a user friendly fashion the image that the user requested along with a graph that let's us analyse the recall and precision of the algorithm as the results are retrieved. Below, we can see all the images that were considered to be the most similar in descending order of similarity.

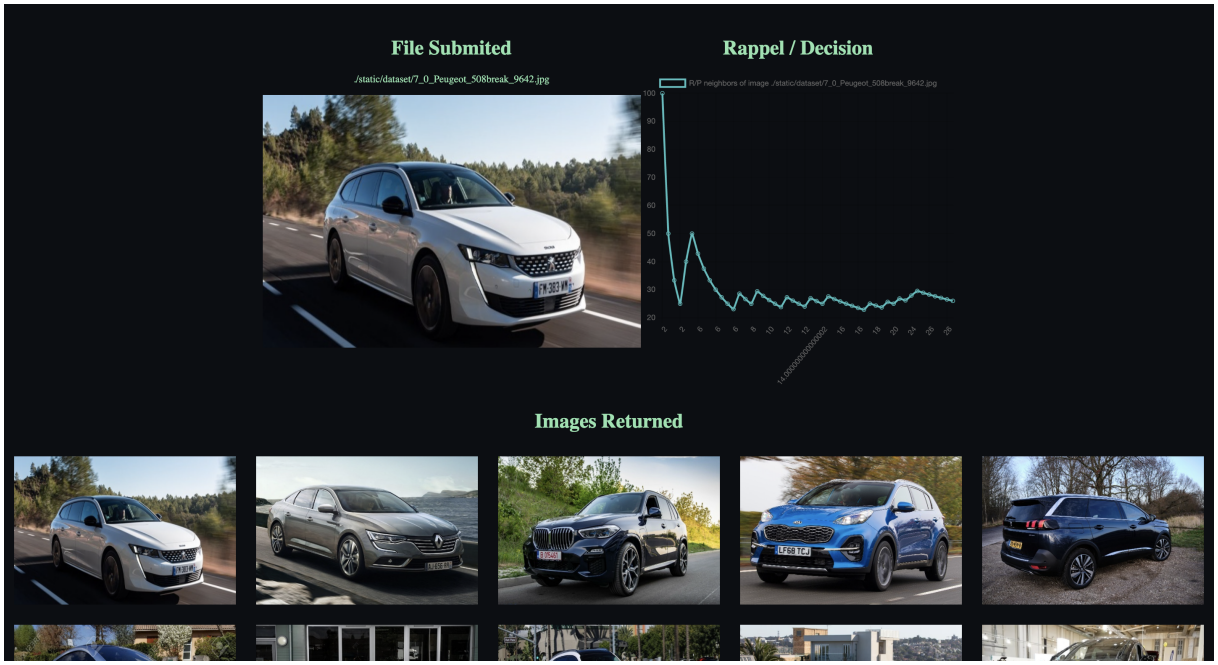

Figure 3: Results Page.

Finally, the **about** page displays summarized info on the apps objective and their creators.

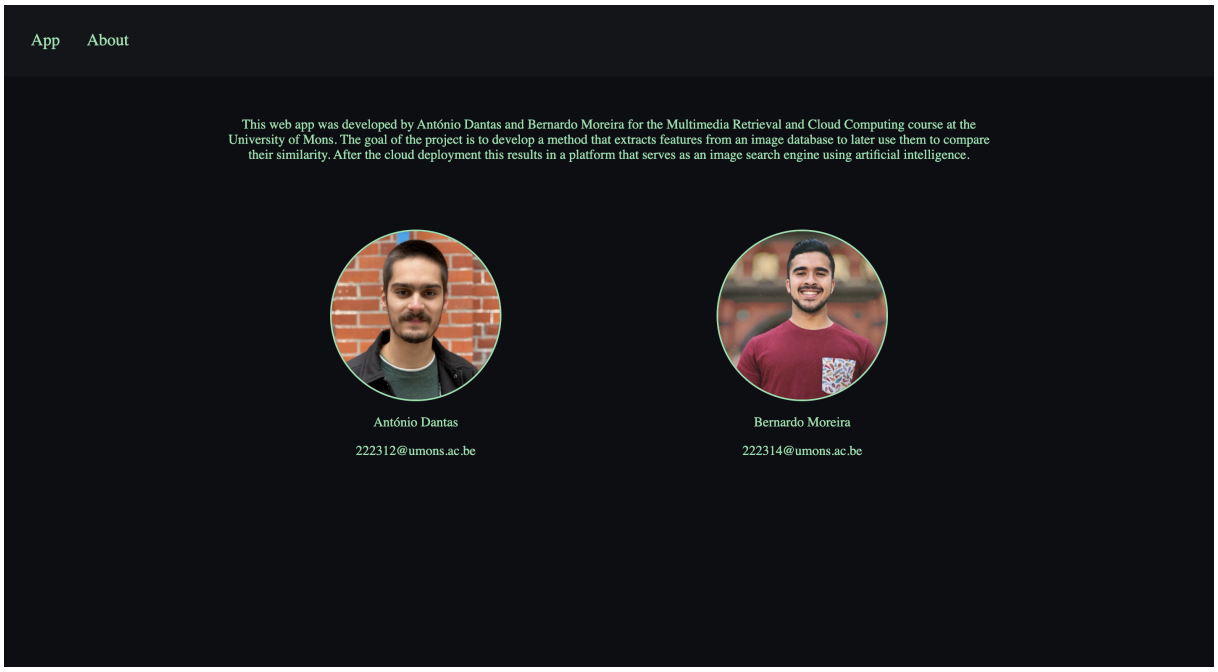

Figure 4: About Page.

The script for this part is on the *app.py* file and can easily be run using the command **python3 app.py**.

2.3 Result Analysis

To test our application and analyze our search results, we've gathered information about the **recall** and **precision** of various requests. Since our image database is based on vehicles, we considered that an item is set to be relevant if it is of the same brand as the requested one.

As we know, recall is the fraction of relevant retrieved items. Since the car's dataset is very big and has around 1000 vehicles in each brand, we only pointed out the number of relevant returned cars, not to end up numbers with a high decimal point set. On the other hand, **precision** acknowledges the fraction of relevant items in the retrieved instances, so this result is pointed out as a percentage.

We believe that to test our database the way to attain the best results would be to concatenate a color and a shape descriptor. We didn't test out texture because all of the vehicles surface and finishes look very similar (very polished and shiny) and the background for each image changes, which could lead to the algorithm giving images with different cars but similar backgrounds a high similarity coefficient. Having this in mind, we used the RGB and HOG descriptors to test our vehicles dataset with an Euclidean distance function method. Besides the recall and precision, the results table also holds an AP column, that accounts for the average precision between the Euclidean and Bhattacharyya methods, and a MaP column, that accounts for the mean precision of all requests. The requests were made both by retrieving 50 and 100 results. Below, we can see which requests account for which vehicles as well as the table holding our findings.

Indice requête	Classe	Images
R1, R2, R3	1	1_4_Kia_stinger_1944, 1_2_Kia_sorento_1675, 1_9_Kia_stonic_2629
R4, R5, R6	3	3_1_Renault_Twingo_4491, 3_0_Renault_grandscenic_4372, 3_5_Renault_clio_5101
R7, R8, R9	5	5_0_Mercedes_ClasseCLS_7059, 5_3_Mercedes_classeC_7403, 5_8_Mercedes_CLA_7992
R10, R11, R12	7	7_0_Peugeot_508break_9642, 7_3_Peugeot_Rifter_10091, 7_6_Peugeot_3008_10530
R13, R14, R15	9	9_0_Audi_A6_12288, 9_3_Audi_Q7_12722, 9_4_Audi_A1_12833

Figure 5: Request list.

Request	R		P		AP		MaP	
	Top50	Top100	Top50	Top100	Top50	Top100	Top50	Top100
R1	6	9	12%	9%	10%	10,5%		
R2	7	11	14%	11%	12%	11,5%		
R3	6	11	12%	11%	12%	9,5%		
R4	8	16	16%	16%	14%	14,5%		
R5	7	11	14%	11%	15%	12%		
R6	7	17	14%	17%	15%	15,5%		
R7	11	17	22%	17%	23%	16,5%	14,8%	12,9%
R8	7	11	14%	11%	16%	12%		
R9	5	7	10%	7%	7%	5,5%		
R10	13	18	26%	18%	24%	18,5%		
R11	8	20	16%	20%	18%	19,5%		
R12	4	10	8%	10%	6%	8%		
R13	8	14	16%	14%	17%	12,5%		
R14	8	13	16%	13%	18%	13,5%		
R15	8	14	16%	14%	16%	14%		

Figure 6: Request results.

As we can see, unfortunately, our results weren't very good. We can, however, find a reason for that to happen. The truth is that the descriptors we provided would be very prone to errors with the data we are dealing with. Firstly, RGB and any other color descriptor would fail on this task because not only different cars within the same brand but also the same model of car can be presented with different colors. This would throw off these types of descriptors and provide incorrect similarity results. The HOG descriptor can detect shapes, which at first glance may seem good, but not to the level of detail that separates most of the car models. It is difficult to assume which brand of car we are dealing with since most of them tend to have standard, sports or truck models which will deceive our algorithm. Also, the same model can be presented in different photo angles which makes it hard to compare their shapes across all of the images database. If we didn't need to compare something so specific we are very confident that our algorithm would have retrieved great results.

3 Cloud and Edge Computing

3.1 Context

After finishing the Flask app, we need a way to deploy it, so that everyone can use it. That's where Cloud enters. In order to deploy it to the Cloud we'll use **Docker**.

Docker is an open platform for **developing, shipping and running** applications. It's based on containerization technology that enables the creation and use of Linux containers. With this software we can handle containers like lightweight virtual machines, where we can create, deploy and move them from environment to other environment, which optimizes our app for the cloud.

In order to do this we'll create a Docker image. A **Docker image** is lightweight, executable package of software that includes everything needed to run an application, for example, code, system tools, system libraries etc...

3.2 Implementation

First, we need to check the requirements that our project needs. This will allow Docker to install all the dependencies before running the project, and for this, we need to create a *requirements.txt* file. In our case, we already knew the dependencies therefore we created and wrote the dependencies by hand; if we didn't knew, or there were too many dependencies to write by hand, we could've executed the following code :

pip freeze > requirements.txt

This will create a *txt* file with the name of *requirements*, and will write inside this file, the name of the dependencies and their respective versions installed in our Flask app.

Now, one can install the required dependencies by simply running, instead of doing *pip install* for every package:

pip install -r requirements.txt

The next step is to generate the Dockerfile. We'll show an example, which was the one we used in our project.

```
FROM python:3.7-slim
WORKDIR /app
COPY . /app
RUN apt-get update && apt-get install -y python3-opencv
RUN pip install -r requirements.txt
EXPOSE 88
ENV NOM Antonio_Bernardo
CMD ["python3", "app.py"]
~
~
~
```

Figure 7: Dockerfile example

FROM : we will install our Docker image from this existing Python image.

WORKDIR : define the root directory of our application in the container.

COPY: Copy every file to inside of our app.

RUN : First we need to update our packages and the opencv library, and after that we can proceed to install all the requirements.

EXPOSE: Informs Docker that the container listens on the specified network ports at runtime.

ENV: Define the environment variable

CMD: The command to run our application

After creating the Dockerfile inside our app, we need to build the Docker image : **docker image build -t antonio_bernardo_flask_app .**

Once the image creation is completed, we can now run the docker image, and specify the desired ports : **docker run -p *88:88* -d antonio_bernardo_flask_app**

88:88 : this is just an example, the first number represents the number of the **port** to which we allocated the container in our machine. The second is the **port** where the application will run on the container.

In order to access our web application one just need to use our machine's IP and add the port 88, like this *http://195.154.54.150:88/*.

After inserting that URL in the browser, now we have access to the app, which looks like the following image:

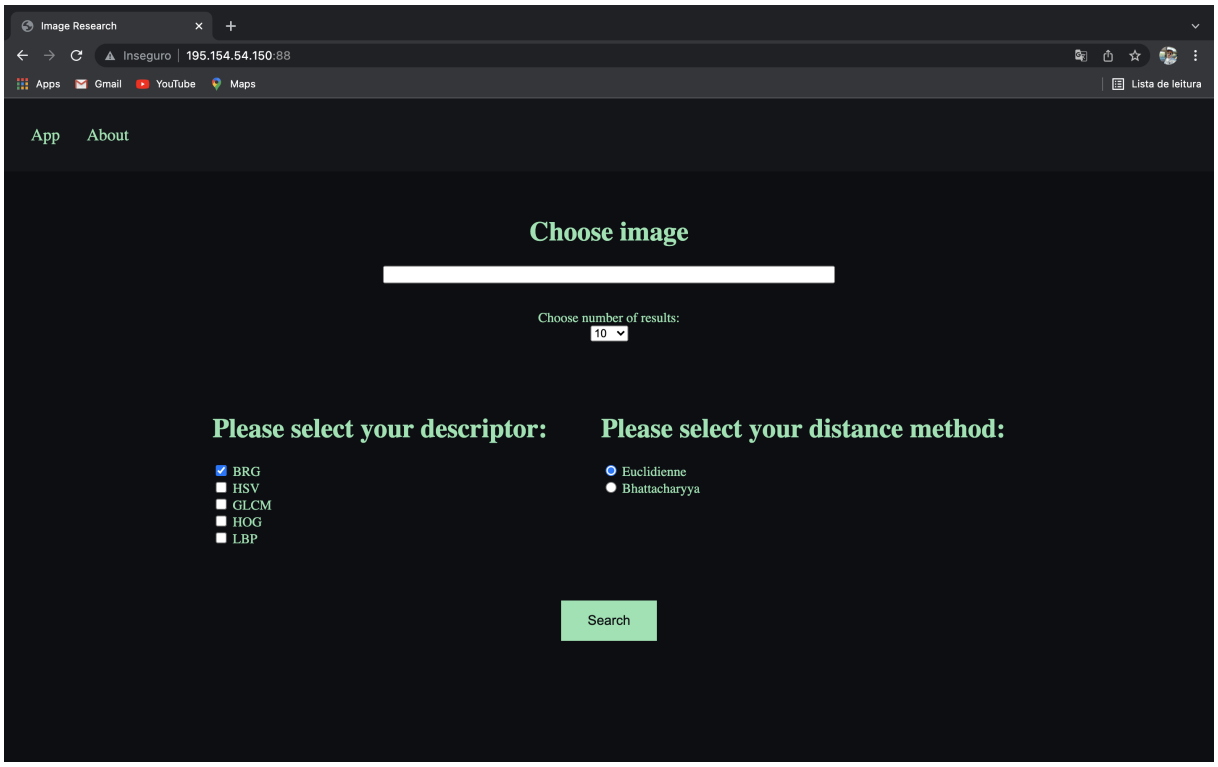

Figure 8: Our App Website

3.3 Problems encountered

Initially, we tried to generate the Docker image, but our project was too big, and our VM didn't have enough memory for that image. This problem occurred because of our *descriptors* folder, which occupied **5GB** of memory. We moved the folder to the outside of our app to solve this issue. That solved the memory problem, but now we still need to have access to that folder, otherwise our app will not work. This can be done by running the docker with one more argument : *docker run –rm -v* ***/home/antonio-bernardo/descriptors:/app/descriptors*** *-p 88:88 antonio_bernardo_flask_app*

This command will run our docker image, and provide the descriptors folder to our app. We can also see that we'll be running our app on port 88.

The last change that we had to do was the *app.run* command in our python app, so that our app can be accessed remotely, and not only on localhost.

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=88, debug=True)
```

Figure 9: App Rub

4 Conclusion

With this project, we got the chance to learn about some subjects that were either previously unknown to us or still a little bit unclear.

The Multimedia Retrieval part made us understand how image feature extraction and comparison work. We now have a clear vision of what needs to be done to process any type of multimedia format and extract information out of it. It was an excellent introduction to the area of computer vision and, although we didn't get the best results when testing our application, we are proud of everything we accomplished during this course.

The Cloud and Edge Computing part were also very interesting. We were already familiar with Docker and machine virtualization, but this course helped us get more comfortable with these technologies. We understood how important it is to replicate the working environment before the project's deployment, to avoid compatibility errors and improve the overall Software as a Service experience.

References

[1] Damilare Jolayemi. *Build and deploy a Flask app using Docker*. URL: https://blog.logrocket.com/build-deploy-flask-app-using-docker/. (accessed: 12.01.2022).

[2] MathWorks. *Texture Analysis Using the Gray-Level Co-Occurrence Matrix (GLCM)*. URL: https://www.mathworks.com/help/images/texture-analysis-using-the-gray-level-co-occurrence-matrix-glcm.html. (accessed: 12.01.2022).

[3] pythonbasics. *What is Flask Python*. URL: https://pythonbasics.org/what-is-flask-python/. (accessed: 11.01.2022).

[4] Scholarpedia. *Local Binary Patterns*. URL: http://www.scholarpedia.org/article/Local_Binary_Patterns. (accessed: 12.01.2022).

[5] Wikipedia. *Bhattacharyya Distance*. URL: https://en.wikipedia.org/wiki/Bhattacharyya_distance. (accessed: 12.01.2022).

[6] Wikipedia. *Euclidean Distance*. URL: https://en.wikipedia.org/wiki/Euclidean_distance. (accessed: 12.01.2022).

[7] Wikipedia. *Precision and. Recall*. URL: https://en.wikipedia.org/wiki/Precision_and_recall. (accessed: 12.01.2022).