UNIVERSITÀ
DI TRENTO

REPORT

Signal, Image and Video course

A.Y. 2021/2022

The Finder

Alberto Casagrande Alessio Belli Joy Battocchio

Idea

Fetching similar images in (near) real time is an important use case of information retrieval systems.
The reasons why this kind of system is needed are many:

- *Copyright issues*. After images are posted on the internet, someone might modify them and repost them claiming that those images are his.
- *Performance*. The presence of near-duplicates affects the performance of the search engines critically.
- *Criminal investigation*.
- *Medical diagnosis*.
- *Storage optimization*.
- *Privacy*.

The problem of finding near-duplicate images is very well known, therefore there are already plenty of algorithms for this purpose.
All the algorithms we studied for the implementation of our system were somehow incomplete: some of them required too much computational power, some others were simply not accurate enough to be used alone, and others had some problems in particular cases (see ORB with blurred images).

Our idea is to combine different methods in order to fill the weakness of one algorithm with the accuracy of another. We choose to use three algorithms:

1. *Histograms comparisons*
2. *Features comparisons*
3. *ORB*

The proposed result will be the weighted sum of the three methods. The three weights are part of the algorithm, but they could be changed if necessary.
Finally, whether two images are near-duplicates or not is decided using a threshold.
Therefore, the algorithm is regulated by 4 parameters, whose values are already proposed by our implementation considering the best performance in our dataset.

Then we implemented two different search algorithms based on what situation we are in. The first one consists in searching in a large dataset all the images that are near-duplicates of another external image (useful for detection), the second one selects all the groups of near-duplicate images within a folder (useful for storage optimization).

Where did we start from?

The project called *'The Finder'* was born while attending a course called *'Comunicazioni Multimediali'* by Professor *Nicola Conci* in the 2020/2021 academic year.
Initially, the idea was to create a programme capable of searching, within a dataset of photographs, for images similar to one inserted by the user. All this was to be done without the use of image metadata.

At the end of the first version of *'The Finder'*, the programme was able to search for images similar to the input image using two metrics: histogram similarity and features similarity. Through a weighted average of these two similarity scores, the most similar

images were extracted (the number of similar images to be displayed was chosen by the user when uploading the image).

The programme was designed with a *client-server* architecture. When the server was initialised, the features and histograms of the dataset were calculated and saved within it, so that the search algorithms could be executed more efficiently.

The project ended with a positive result as, in most situations, the most similar images to the image entered by the user were extracted.
Among the aspects to be improved was the absence of other functionalities and also the slow search for similar images.

How did we expand the project?

Now the idea is not only to search for the most similar images, but also to search the dataset for near-duplicates. In addition, a new feature has been added whereby images uploaded by a user can be divided into groups of similar images.

The project therefore no longer focused on searching for similar images, but rather near-duplicates. This change of direction was made possible through the adoption of a new metric in addition to those already present. This metric is the similarity between the descriptors of the images through *ORB*.

ORB had been considered in the realisation of the first version of the project, but was discarded as it did not produce satisfactory results for the idea of the previous programme.

Programming languages and libraries

To implement the project, we have decided to use python, which has a large amount of supported libraries. Among them, those useful for the purposes of the project are definitely *Numpy*, *OpenCV*, *Flask*, *TensorFlow* & *Keras*.
Numpy is an open source Python library that presents mathematical functions for the manipulation of large matrices and multidimensional arrays. **OpenCV** is a software library in the field of computer vision. When integrated with other libraries, such as Numpy, it makes possible the processing of the structure of arrays for analysis, identifying the pattern of the image and its various features through the vector space. **Flask** is a web framework, it is a Python module that allows you to easily develop web applications. A Web Framework represents a collection of libraries and modules that allowed us to develop a web application without worrying about low-level details such as protocol, thread management, and so on. Through this framework, it is possible to manage in a simple and effective way the interaction between web pages and python code executed on the server side.
By means of simple functions such as `render_templete("index.html", data=data)` it is possible to render client-side web pages by passing data dynamically, which are managed by the *Jinja2* module of *Flask*. **TensorFlow** is an open source end-to-end platform for machine learning. TensorFlow uses dataflow graphs, which are

structures that describe how data moves along a graph, which is composed of an array of processing nodes. Each node in the graph represents a mathematical operation, and each relationship or connection between nodes is a multidimensional array, also called a tensor. In our project, such a platform is used as a low-level framework for the high-level Keras interface. **Keras** is a library written in Python (released under MIT license), for machine learning and neural networks and supports TensorFlow as a back-end. In our case, models made available already trained have been used. Among the different models available, it was decided to use ResNet50, which is a convolutional neural network model that uses convolution and pooling filters.

Matching Criteria

The application, as mentioned above, has two main functions:

1. searching for the most similar images in the dataset with respect to the query image uploaded by the user.
2. splitting near-duplicate images within the folder uploaded by the user into groups.

For both functions, 3 different matching criteria were used: comparison by histograms, comparison by features, and comparison by ORB descriptors. We used 3 different matching criteria in order to try to exploit their different characteristics and properties while minimizing their weaknesses. Therefore, based on the purpose of each function, different weights have been given to the various algorithms.

Histogram comparison

As for comparing images using histograms, this is done by dividing each image into four windows. Initially, for each image in the dataset, the histogram is calculated for each of the 4 windows in which the images are divided, using the `calcHist()` function of OpenCV. Each histogram is saved to a file (`histograms.yml`) using OpenCV's FileStorage class.
Finally, in the query phase the same process is used to extract 4 histograms from the input image, one for each image window. The comparison of the input image with each image in the dataset is done by comparing the various corresponding windows and averaging them.

This technique has allowed us to overcome the drawbacks of positioning. To extend this concept it is useful to give an example: let's assume we have an image containing mountains with a blue sky and another with the sea of the same color. Without using the segmentation technique, the similarity of the histograms would be high even though the

images have different contents since it is done on the whole image, while evaluating 4 windows would show differences since the sky of the mountains and the sea are located in different windows.

The actual comparison of the histograms is done through the function `compareHist()` of OpenCV, which requires as input the two histograms to be compared and a third attribute that specifies the metric to be used. In our case we used the correlation method `cv2.HISTCMP_CORREL`. For the correlation method, the higher the metric, the more accurate the match.

```
result_histograms = []
image_names = os.listdir("gallery/")
for filename in image_names:
    nomeFile = os.path.splitext(filename)[0]
    compare=0
    for r in range(0,2):
        for c in range(0,2):
            hist_file=fsRead.getNode(f'histogram_{r}_{c}_{nomeFile}').mat()
            compare+=cv2.compareHist(hists[f'histoquery_{r}_{c}'],hist_file, cv2.HISTCMP_CORREL)
            countc+= 1
        countr+=1
        countc=0

    media=compare/4
    result_histograms.append((nomeFile, media))
return result_histograms
```

Regarding the search for near-duplicate images within the folder uploaded by the user and their division into groups, in that case the images are compared in pairs, where for each pair the 4 respective histograms of the two images are compared and averaged.

Feature comparison

Keras allows you to create new models or use existing ones, in our case ResNet50.
In particular, it has not been used the model in its entirety because we are not interested in the prediction and classification functions (which are performed by the last levels of the model) but only in the feature extraction function. For this reason, the last level used is the one called "*AVG_POOL*".

The code, after having instantiated a model, passes an image (resized to *224x224* to match the specifications required by ResNet) and makes it a Numpy matrix. The model then uses predefined functions to extract the features.
We then proceed to save these multidimensional arrays containing the features in different *.npy* files for each image via functions of the Numpy library.
This happens when the *Flask* server is started for the first time; once these files have been saved *.npy* and keeping the same dataset it will be possible to skip this process, which is the most onerous at the computational level. Each time the dataset is modified within the server, it will instead be necessary to call the function and recalculate all the features.

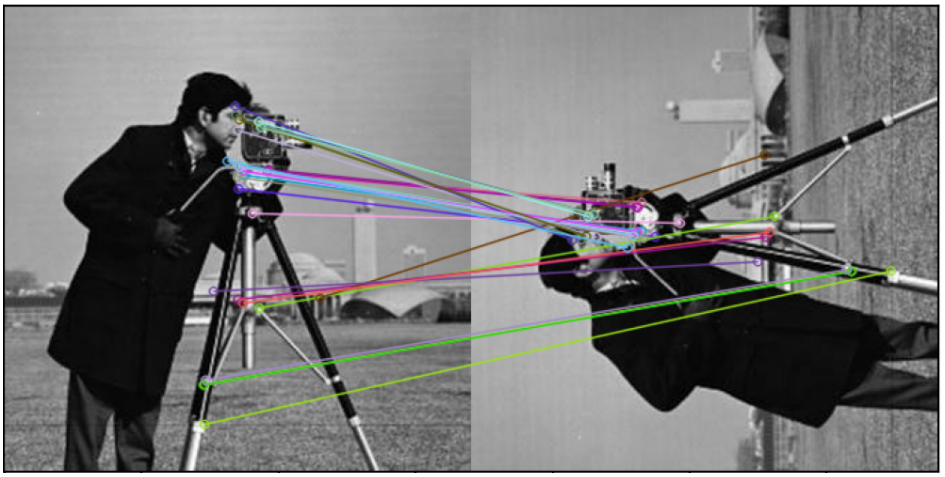
When you insert a query image, all the features of the individual files are added to a single list.

The characteristics of the query image are then extracted and at that point cosine similarity between all the saved characteristics and the characteristics of the query is calculated. This similarity reflects the similarity between the images.

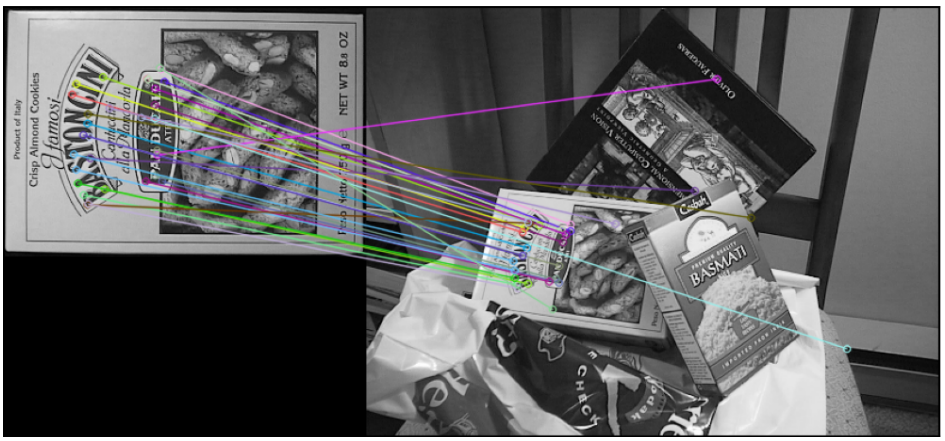
ORB

In order to more accurately detect the so-called near-duplicates, it was necessary to implement an additional algorithm. After a careful web search, we noticed that the main techniques in the literature extract features based on key points. The main key point-based feature extraction methods are definitely SIFT, SURF, ORB, FAST, BRISK and KAZE. After testing them, we came to the conclusion that the best in performance and accuracy was ORB (Oriented FAST and Rotated BRIEF). ORB features are invariant to scale, rotation and limited affine changes.

To implement the algorithm,it is enough to use the OpenCV library, instantiating the orb object using the `cv2.ORB_create()` function. To extract descriptors from an image, it is necessary to call the `detectAnCompute(image, None)` function on the orb object and pass the image as a parameter. This function returns two elements: key points and descriptors. In our case, only the descriptors were taken into account.
The comparison between two images is performed by using a *Matcher* (`cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)`) which uses the Hamming distance as measurement. The *Matcher* object has an important method, which is `BFMatcher.match(desc_a, desc_b)` that takes as input the descriptors of the two images and returns the matches between them.
Here are some examples of matching between pairs of images:



Original image and rotated image



Object in the first image contained in the second image

Original image and noise image

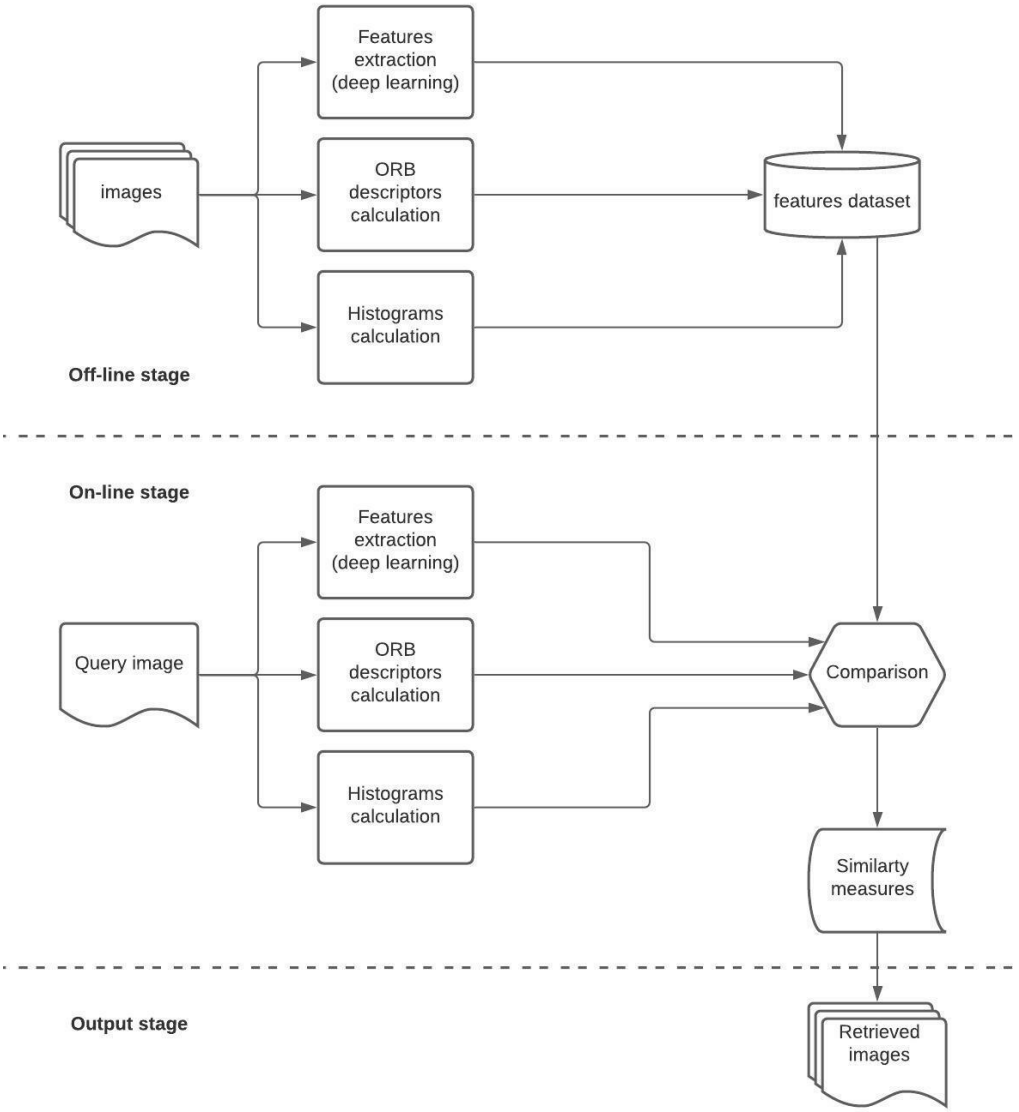
In detail, once the server is started for the first time, the system calls the `creaDescriptor()` function defined in the `descriptor.py` file, whose task is to calculate all the descriptors of the images in the dataset and save them to files. In this way, when the user upload the query image, it is sufficient to detect and compute the descriptors of that image, and compare them with those saved to files, without having to compute all the descriptors of the dataset images in the online stage, and, as a result, this allows us to reduce the processing time.

```
for filename in image_names:
    nomeFile = os.path.splitext(filename)[0]
    image=cv2.imread(f"gallery/{filename}")
    orb = cv2.ORB_create()
    kp_a, desc_a = orb.detectAndCompute(image,None)
    feature_path = f"descriptor/{nomeFile}.npy"
    np.save(feature_path, desc_a)
```

On the other hand, in the case of searching for near-duplicate images within the folder uploaded by the user and their division into groups, descriptors are calculated once the user has uploaded the images and afterwards they are compared in pairs. In this case, the orb algorithm is even more important, as it is very effective in detecting near-duplicates.

Application functionality

Upload single image



The first functionality of the web app deals with the search for similar and near-duplicate images within our image dataset compared to the image entered by the user.

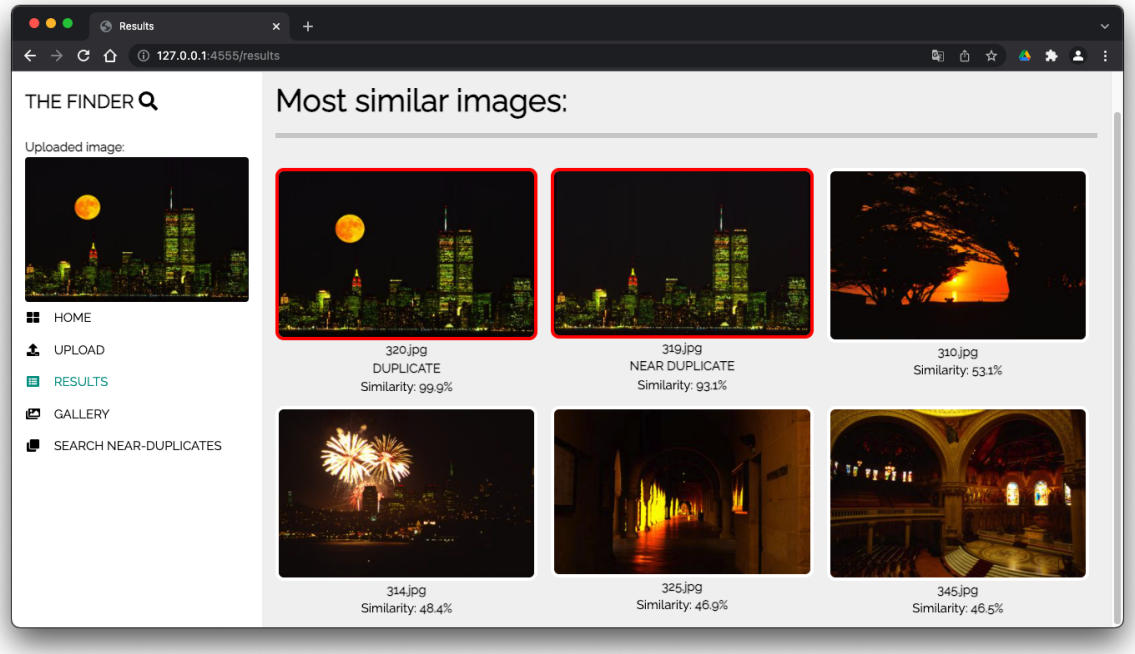
The user, having gone to the `upload.html` page of the site, will be able to upload his own image simply with the help of a form. The user, before carrying out all the necessary computation to search for similar images, is also asked for the number of similar images to be displayed (from a minimum of one to a maximum of ten).
Once the *'Upload'* button has been pressed, the functions required to search for similar images are invoked.
These functions are based on the matching criteria listed above.

For each matching criterion, lists of objects are created containing the name of the image and its similarity score with the input image entered by the user. Once all the matching criteria have correctly calculated the various similarities, a weighted average is

performed. The weights that have been assigned to the various criteria are: *histograms 20%*, *features 40%* and *ORB 40%*.
These weights were defined following several tests that allowed us to see what the optimal situation might be.

At this point, the program created a list containing for each image its average similarity to the one entered by the user. Once this is done, the n similar images are extracted (where n is defined by the user) and passed via a *json* file to a new html page called `results.html` where these images can be displayed. This page displays the similar images with their similarity percentages, also divided by their matching criteria.

Great attention has also been paid to near duplicates. If the similarity through *ORB* is greater than *70%*, these images are displayed with a red frame. In addition, if the *ORB* similarity is greater than *70%* but less than *100%*, "*Near Duplicate*" is displayed, or if the similarity is equal to *100%*, "*Duplicate*" is displayed.



Upload folder

The second application we implemented is the detection of groups of near-duplicate images within an uploaded folder.
The user can upload an entire folder of images from the page `duplicate.html` by using a simple form.

Now the system computes all the pairwise comparisons putting the results in a N^2 matrix (N is the total number of images in the uploaded folder) where the element in position `[i][j]` corresponds to the similarity (expressed in percentage) between the i^{th} and j^{th}

images. For optimization only the top half triangle of the matrix is filled since it is symmetrical.
The similarity is computed with the following weights: *histograms 20%*, *features 30%* and *ORB 50%* (ORB is weighted more in this method because it aims only to find near-duplicates and not simply similar images).

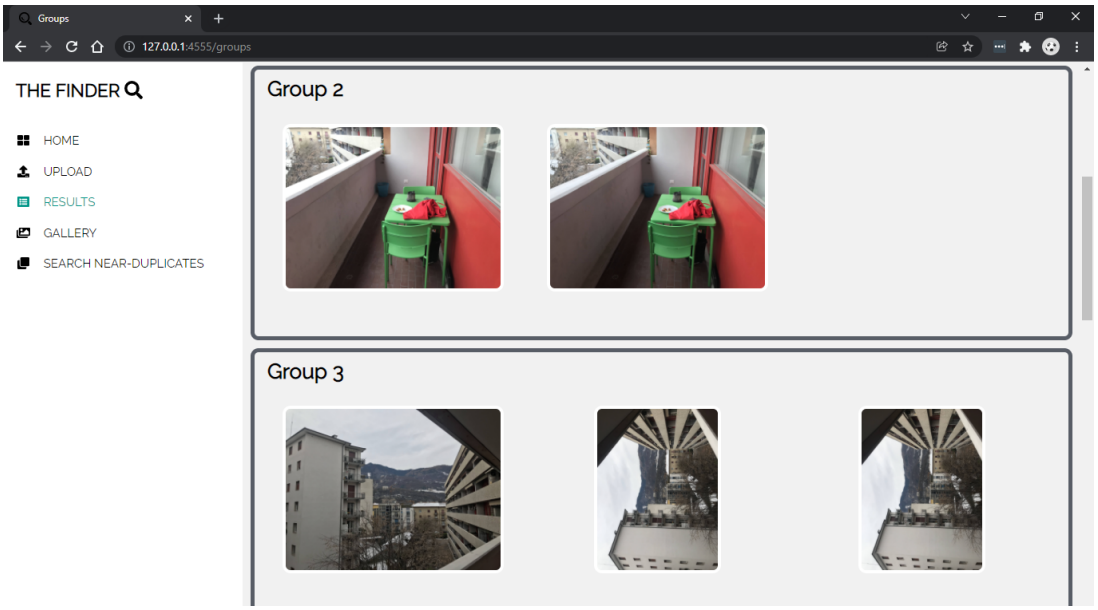
After that the matrix is transformed changing all the values above a given *threshold* to `1`, and all the others to `0`.
This new matrix is used to detect all the groups of near-duplicate images, using a simple algorithm that loops through half of the matrix and saves the coordinates of the `1` elements in sets.

```python
#funzione di normalizzazione della matrice
#input: matrice con i valori di similarità, soglia
#output: matrice di 0 (no near-duplicate) e 1 (near-duplicate)
def normalizeMatrix(pairwiseMatch,threshold):
    matchMatrix = np.zeros((len(pairwiseMatch),len(pairwiseMatch)))
    for i in range (len(pairwiseMatch)):
        for j in range (i+1,len(pairwiseMatch)):
            if(pairwiseMatch[i][j] > threshold):
                matchMatrix[i][j] = 1
            else:
                matchMatrix[i][j] = 0

    return matchMatrix

#funzione di ricerca dei gruppi di near-duplicate
#input: matrice di similarità
#output: lista di gruppi
def multimatch(matchMatrix,threshold):
    matchGroups = []
    normalizedMatchMatrix = normalizeMatrix(matchMatrix,threshold)
    n = len(normalizedMatchMatrix)
    for i in range (n):
        for j in range (i+1,n):
            if(normalizedMatchMatrix[i][j]==1):
                tempSet = {i,j}
                foundFlag = -1
                for k in range (len(matchGroups)):
                    if (tempSet.intersection(matchGroups[k]) != set() and foundFlag == -1):
                        foundFlag = k
                        matchGroups[k] = matchGroups[k].union(tempSet)
                    elif (tempSet.intersection(matchGroups[k]) != set() and foundFlag != -1):
                        matchGroups[foundFlag] = matchGroups[foundFlag].union(matchGroups[k])
                        matchGroups[k] = set()
                if(foundFlag==-1):
                    matchGroups.append(tempSet)
    return matchGroups
```

The resulting groups of images are sent to the client in the same way already described above, and displayed in the `groups.html` page.
Note: only the images that are found to be near-duplicate with some other images are displayed in this page.



Conclusion

After many tests the system has shown good results, both in detecting near-duplicate images from the dataset and dividing a folder in groups.
The main drawback of our solution is that it has four parameters that need to be tuned, however we performed many adjustments to find the best values in most situations, so the algorithms are already proposed with what we think to be the best values for those parameters.

One necessary remark is about the reliability of ORB: of the three methods we use it has shown to be the best in finding near-duplicates in case of cropped or rotated images, but it fails completely in some particular cases of images that have been through a low-pass filter. Therefore our proposal to combine the three methods described above solves this issue and within our tests was capable of finding all the near-duplicate images.

Another issue that needs to be taken into account is that the complete process requires a lot of computational power and therefore could take a few minutes to run in a normal pc; that's why we build the system as a client-server application, so to leave all the complex computations to the server that has more computational power at its disposal.

Our application is developed within a university project, so there is space for some adjustments: for example it should be given to the user the possibility to select his own dataset, and a dashboard that allows him to tune all the parameters.
In conclusion we think that with these and a few other upgrades this could be a good platform for real life applications. It does not aim to be the best solution available, but it shows good quality that other systems do not.