Douglas-Quaid - Ex-Carl-Hauser - Documentation

Vincent FALCONIERI

February 2019 - August 2019

Contents

1 Introduction **2**

2 API **3**
2.1 Calls 3
2.2 Design 4
2.3 Ressources 4

3 Components and libraries **5**

4 Parallelism **8**

5 Database operations **10**
5.1 Descriptor storage and serialization 10

6 Core computation **12**

7 Tests and Examples **14**
7.1 Performance and Stats datastructures 14
7.2 Graph evaluation and matching quality 14
7.3 Threshold extraction from quality Graph 18

8 Visualization **20**

Chapter 1

Introduction

Goal The goal of this document is to provide an overview of how the carl-hauser library is built, from API to core computation.

Methodology A State of the Art overview had been performed and is available on the project page at `https://github.com/Vincent-CIRCL/carl-hauser`

In the following, we expose :

- ...

Problem Statement [Cevikalp et al.,] states the Image Retrieval problem as "Given a query image, finding and representing (in an ordered manner) the images depicting the same scene or objects in large unordered image collections"

Please, be sure to consider this document is under construction, and it can contain mistakes, structural errors, missing topics .. feel free to ping me if you find such flaw. (Open a PR/Issue/...)

Chapter 2

API

2.1 Calls

In the first version of carl-hauser, following calls are available. Note that this is the minimal API given the problem.

- **PING** : Allow to quickly check if the API is alive
- **ADD PICTURE** : Store a picture in the database, that could later be fetched if close to a request picture. Returns the ID of the added picture, as stored in the database.
- **WAIT FOR ADD** : Blocking call to wait for an adding to be done
- **REQUEST SIMILAR PICTURE** : Performs a request on the database, to fetch similar pictures. Returns a request id to later fetch results.
- **WAIT FOR REQUEST** : Blocking call to wait for a request to be answered
- **GET RESULTS**: Given a request id, returns a formatted JSON of results (list of similar pictures ids).
- **REQUEST DB**: Request a copy of the database as stored by the Redis storage server

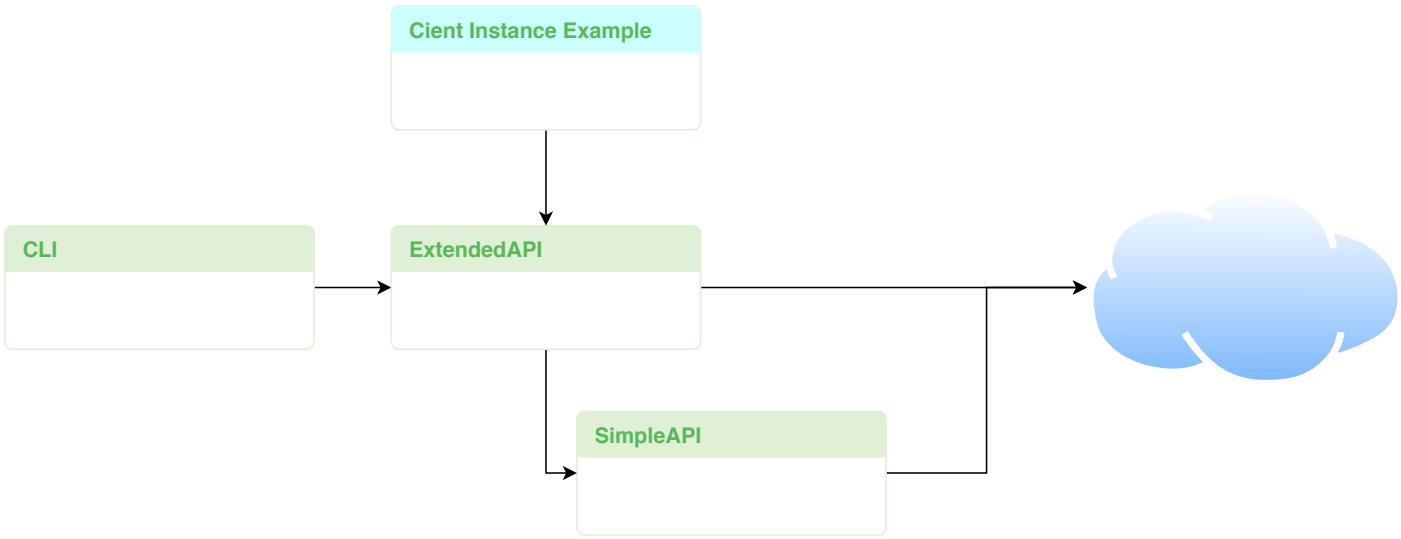


Figure 2.1: CLI and API structure on client side

Two kinds of graphs can be referenced :

- Storage graphs : Graphs and clusters as it is stored in Redis. (Datastructure to improve requests performance)
- Proximity graphs : Graphs computed from many requests, that show which picture is close to which picture. This is not the way it is stored in the database, but a condensed view of all requests that can be made on the database.

2.2 Design

Flask manages the API endpoints listed previously.

2.3 Ressources

- Flask documentation : `https://www.tutorialspoint.com/flask/flask_http_methods.htm` or `http://flask.pocoo.org/docs/0.12/quickstart/` e.g.
- Extensive tutorial to build REST API with Flask : `https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask`

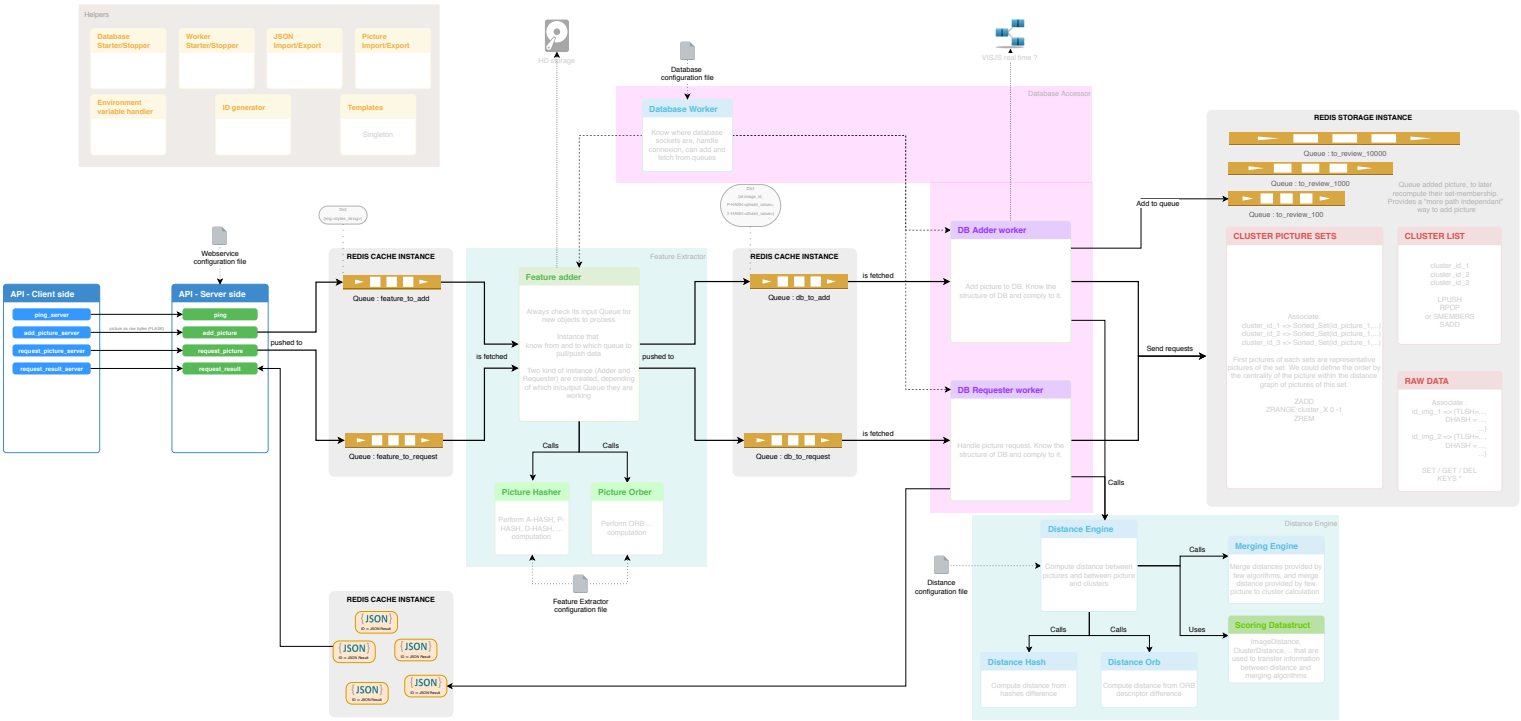
Chapter 3

Components and libraries

Carl-hauser library is split in two : the server side and the client side. The client side is only an accessor of server-side functions.

The application is split in few packages, see Figure **??** :

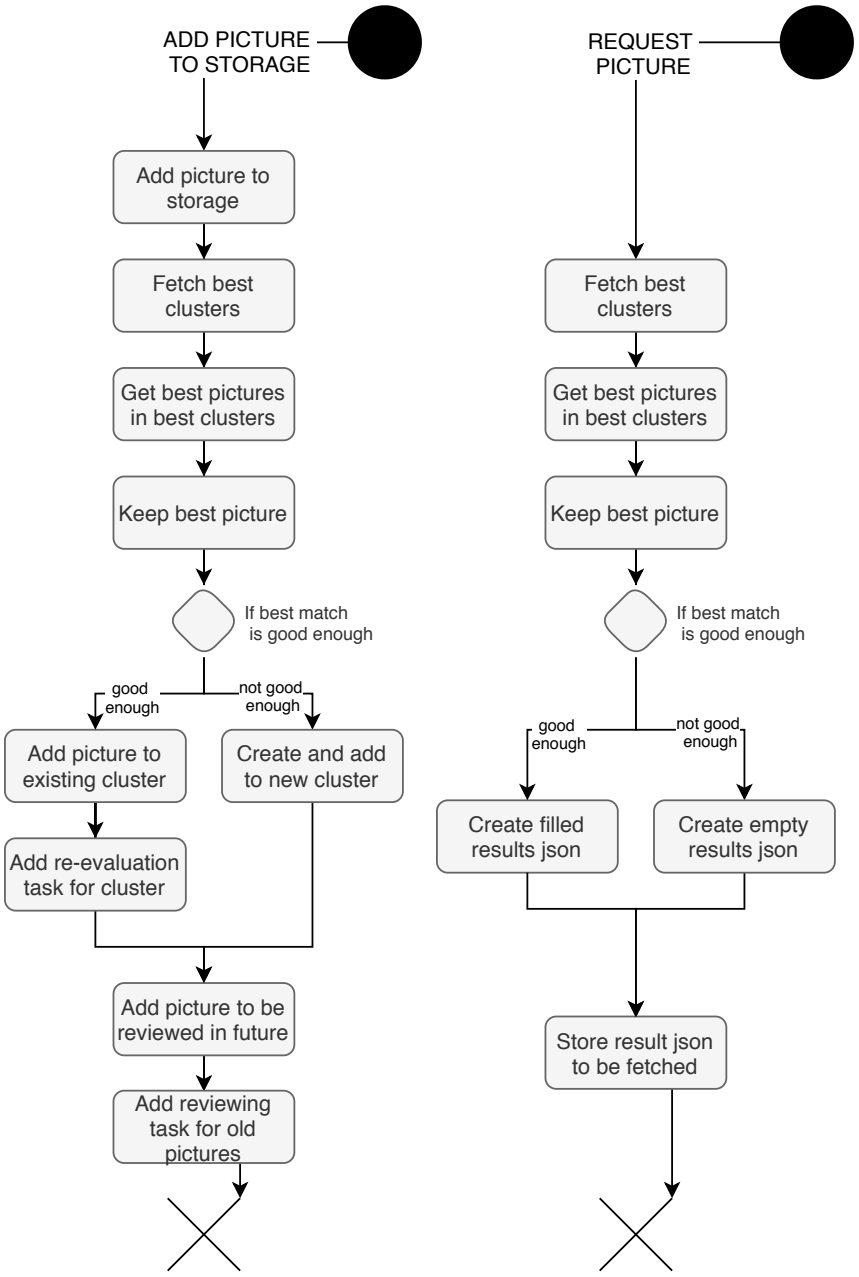
- **API** : The server's endpoint which received client's requests. Mainly running on *Flask*.
- **A Redis cache instance** : it stores current client's requests. For example, a picture to be processed, or a bunch of calculated values to be stored in the redis storage instance.
- **A Redis storage instance** : it stores long-term values, as picture descriptors, or computed datastructure.
- **Feature extractor** : combines feature adder, feature requester, picture hasher, picture orber, ... It computes image representation, as for example, ORB descriptors of a picture, HASHs, ...
- **Database accessor** : combines Database worker, DB Adder, DB Requester, ... It computes and fetch data to and from the redis storage instance. It knows how the database is structured and how to perform request on it.
- **Distance engine** : combines a merging engine, a distance hash, orb distance, scoring datastructure, ... It provides a way to computes the distance between pictures, between picture and clusters, etc. This is where the core computation are performed.



(a) Software architecture

Figure 3.1: Software Components

Let's focus on how a picture is added and how a picture is requested, to the database, see Figure **??**.



(a) Processus to add a picture to database

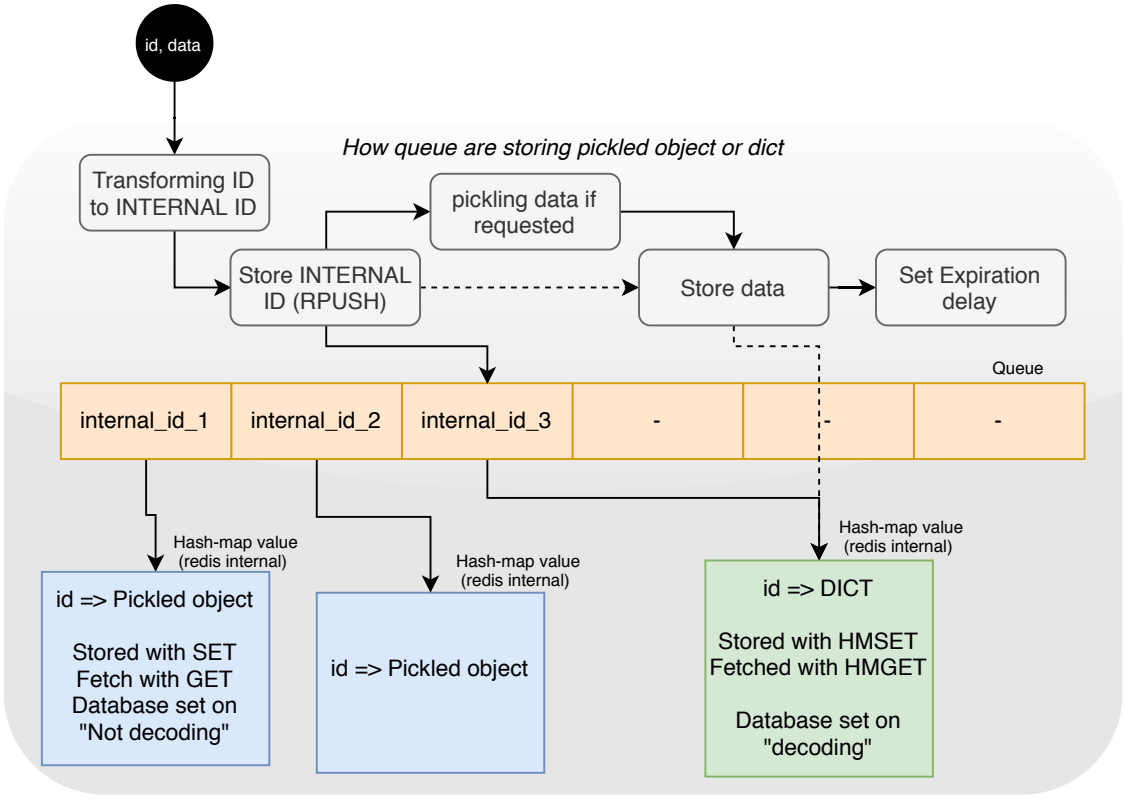
(b) Processus to request a picture's similar pictures

Figure 3.2: Processus

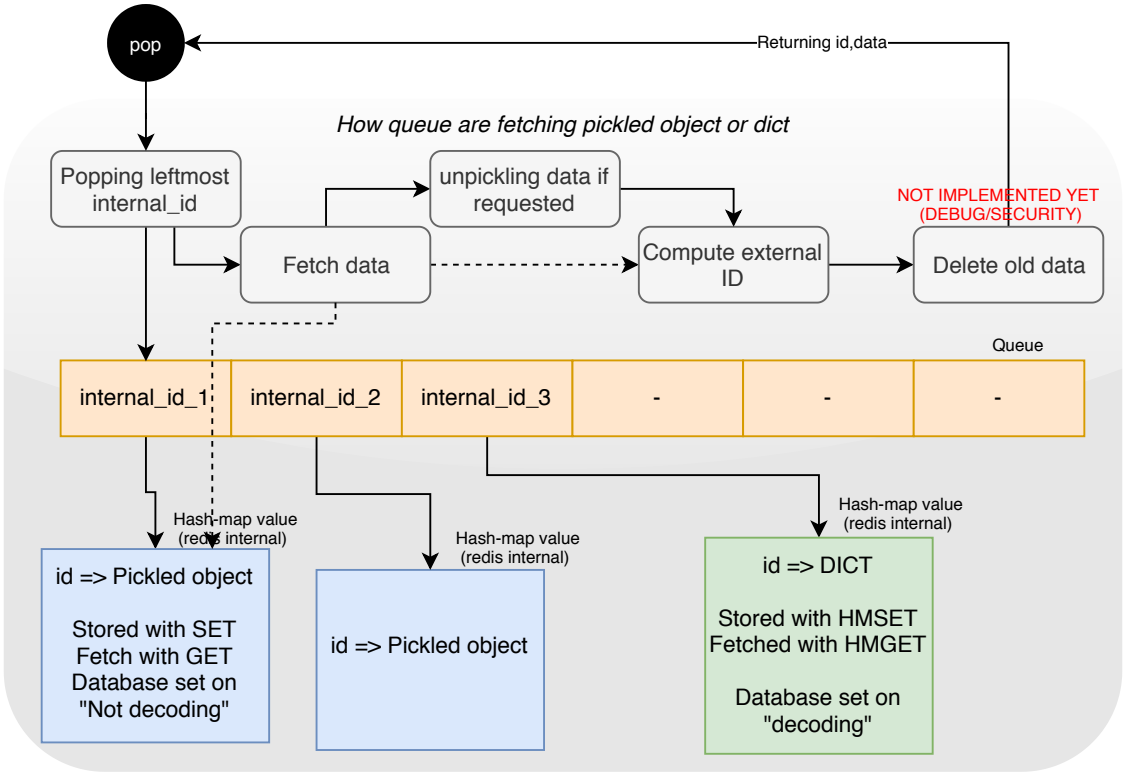
Chapter 4

Parallelism

Queues are heavily used in the application to handle asynchronous tasks. These queues are storing any object in redis cache instance. An easy push/pull interface is implemented, and its internal actions are presented in Figure **??**.



(a) Storing objects in queue



(b) Fetching object from queue

Figure 4.1: Queue management

Chapter 5

Database operations

5.1 Descriptor storage and serialization

We met an important issue to store descriptors of pictures computed by OpenCV, for example. Each interest point in a picture is represented as an object with attributes about this interest point. This data-structure can be called, for example, *cv2.Keypoint*.

When we store string into the database, we could use a HMSET, which is equivalent to storing a python dictionary in redis. HMSET are only handling 1-depth dictionary. However, these objects and the nesting that they imply can't reasonably be stored as a 1-depth dictionary. See Figure 5.1a.

If a HMSET, and so a direct access to any member of the data-structure, is not reasonably possible, then a solution is to "bundle" this data-structure by serializing. We have some choice : JSON, pickle, own datastructure ..

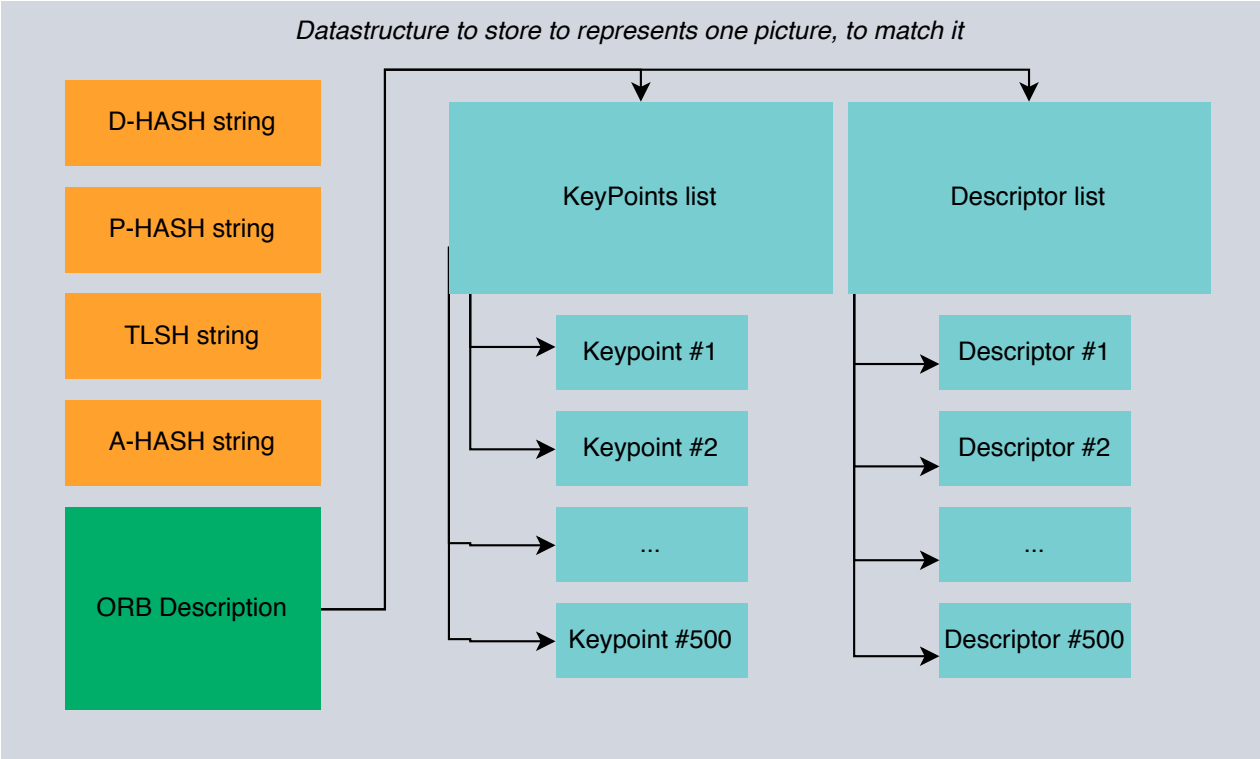
On a performance side, *CPickle* (usable after Python 3.4) seems to provide best results with last version of the protocol. Therefore, we use this implementation. See Figure 5.1b

However, most *OpenCV* objects are not pickle-serializable. More precisely, "pickle stores references to classes and functions, not their definition, because that's way more efficient" [Pyt, a]. Therefore pickle only store data of an object instance, and the type of object it originally was. It does not store class hierarchy nor method definitions. At one point, it "double-checks that it can use that name to load the same class or function back again." [Pyt, a].

For *OpenCV* object, this test fails, because each object name is declared twice : as an object and as a function (due to underlying C++ needs). Therefore, it could not unpickle the data it would pickle, and abort the pickling, leading to an Exception.

A workaround is to register a small method to overwrite the data loading. If such method is set, then Pickle does not perform its sanity-check and so allow the pickling.

Therefore an interesting fix can be built. See Figure 5.1c [Pyt, b]. Then, descriptors "bundle" is pickled and store as a simple key-value pair (*SET/GET* in Redis), retrieved and unpickled. A full bundle for one picture is about *44,5 Ko*.



(a) Representation of one picture

```
pickle          :   0.847938 seconds
cPickle         :   0.810384 seconds
cPickle highest:   0.004283 seconds
json            :   1.769215 seconds
msgpack         :   0.270886 seconds
```

(b) Speed difference for dumping data. Similar results for loading.

```
def patch_Keypoint_pickling(self):
    # Create the bundling between class and arguements to save for Keypoint class
    # See : https://stackoverflow.com/questions/50337569/pickle-exception-for-cv2-boost-when-using-multiprocessing/50394788#50394788
    def _pickle_keypoint(keypoint): # : cv2.KeyPoint
        return cv2.KeyPoint, (
            keypoint.pt[0],
            keypoint.pt[1],
            keypoint.size,
            keypoint.angle,
            keypoint.response,
            keypoint.octave,
            keypoint.class_id,
        )
    # C++ : KeyPoint (float x, float y, float _size, float _angle=-1, float _response=0, int _octave=0, int _class_id=-1)
    # Python: cv2.KeyPoint([x, y, _size[, _angle[, _response[, _octave[, _class_id]]]]]) → <KeyPoint object>

    # Apply the bundling to pickle
    copyreg.pickle(cv2.KeyPoint().__class__, _pickle_keypoint)
```

(c) Speed difference for dumping data. Similar results for loading.

Figure 5.1: Challenge - Datastructure to store in Redis

Chapter 6

Core computation

Matching

Prerequisites : Clusters populated with candidate pictures and an input_picture

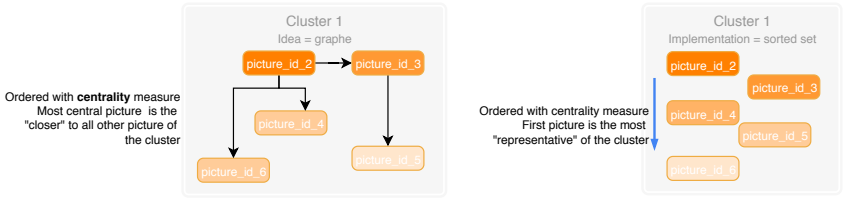
1. Evaluating proximity from input_picture to each cluster
= compute distance between input_picture and *N1* "best representative pictures" of each cluster, and merge the result in one unique "picture_to_cluster" distance.

2. Find best matching clusters
= Keep *N2* best clusters regarding "picture_to_cluster" distance.

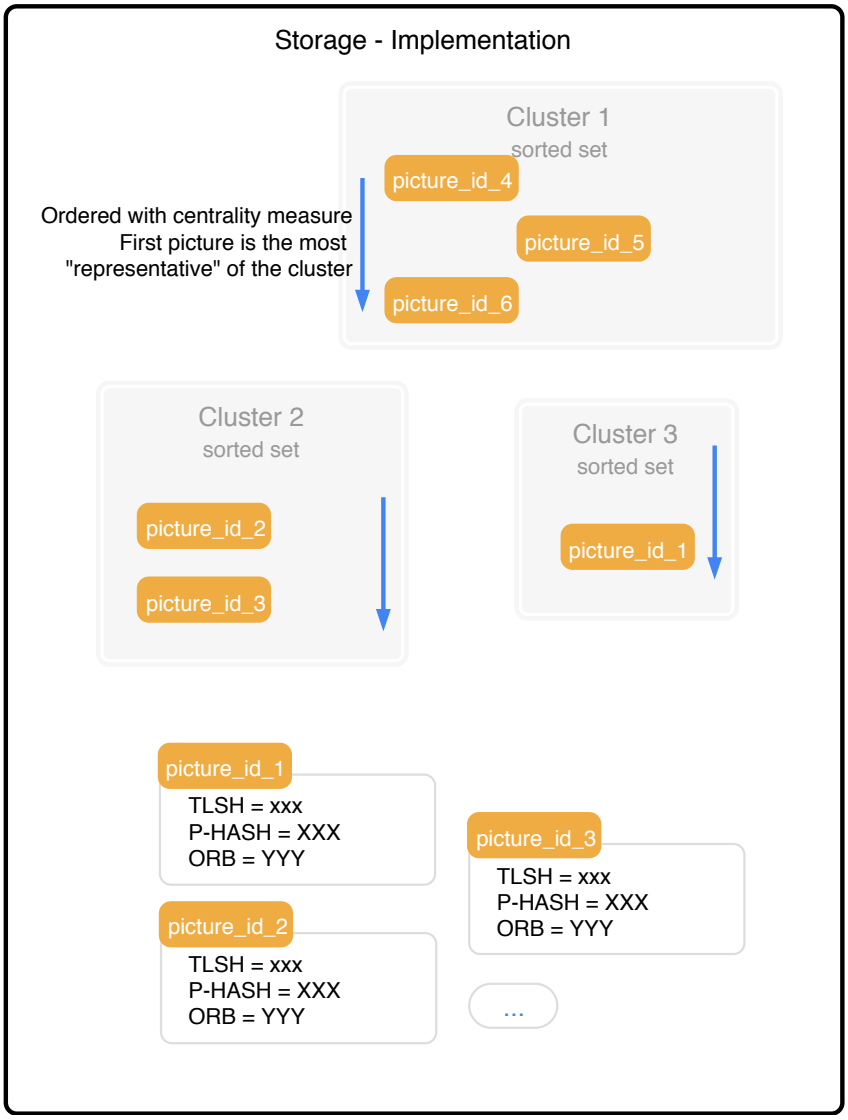
3. Evaluating proximity from input_picture to all candidate picture
= Compute distance between input_picture and each picture of these *N2* clusters.

4. Find best matching pictures
= Keep *N3* best pictures regarding input_picture to candidate picture.

(a) Principle of a similarity search in the redis storage instance



(b) Conceptual view versus Implementation view



(c) Redis storage instance structure

Figure 6.1: Datastructure and search

Chapter 7

Tests and Examples

7.1 Performance and Stats datastructures

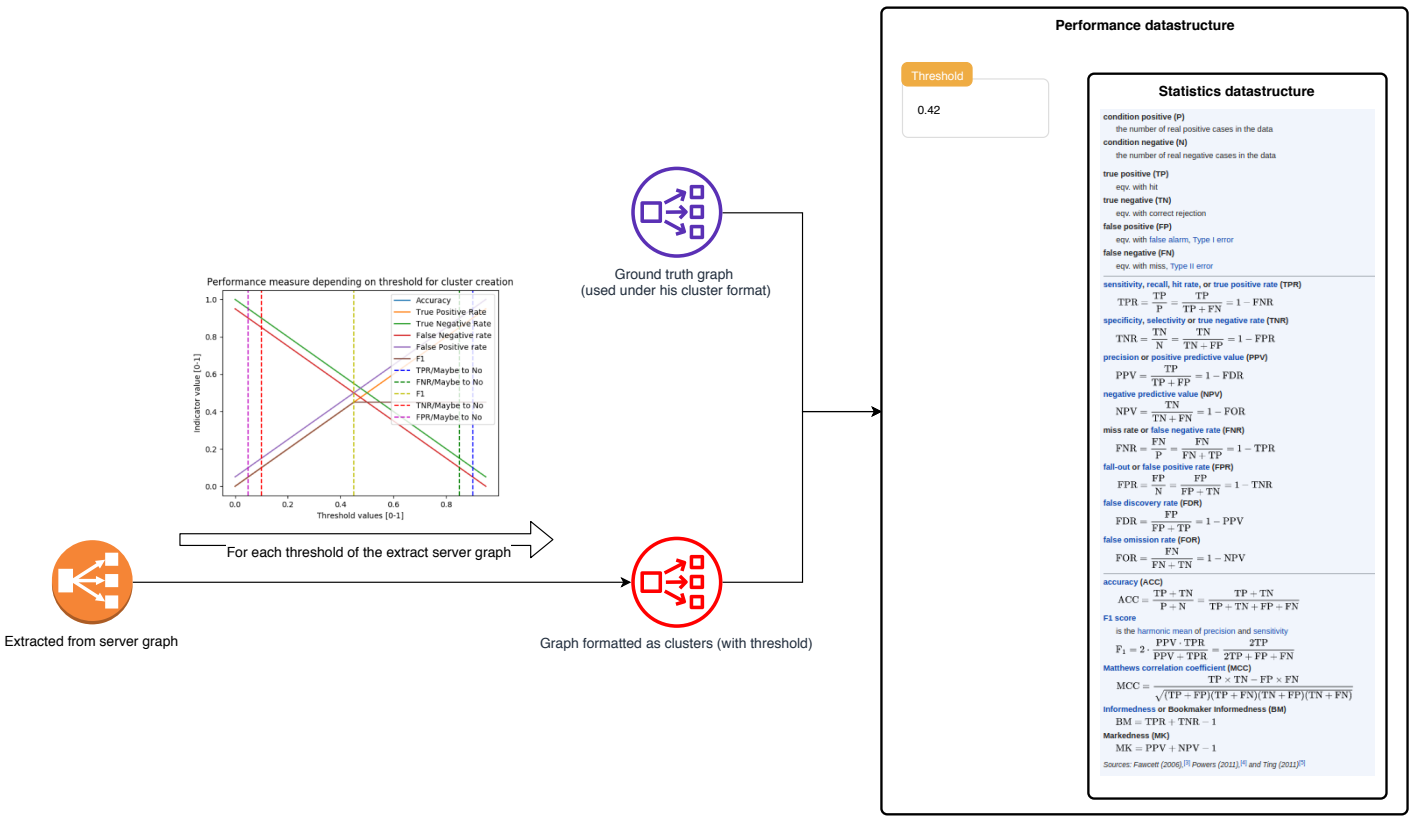


Figure 7.1: Performance and Stats structure, and how it is used to evaluate a full graph for dozens of thresholds

7.2 Graph evaluation and matching quality

We can evaluate the quality of matching made by the library regarding a batch of input files and a ground truth file, that can be constructed with visjs-Classificator, for example.
On one side, we have a clustered version of pictures (output of visjs-Classificator) and on the other

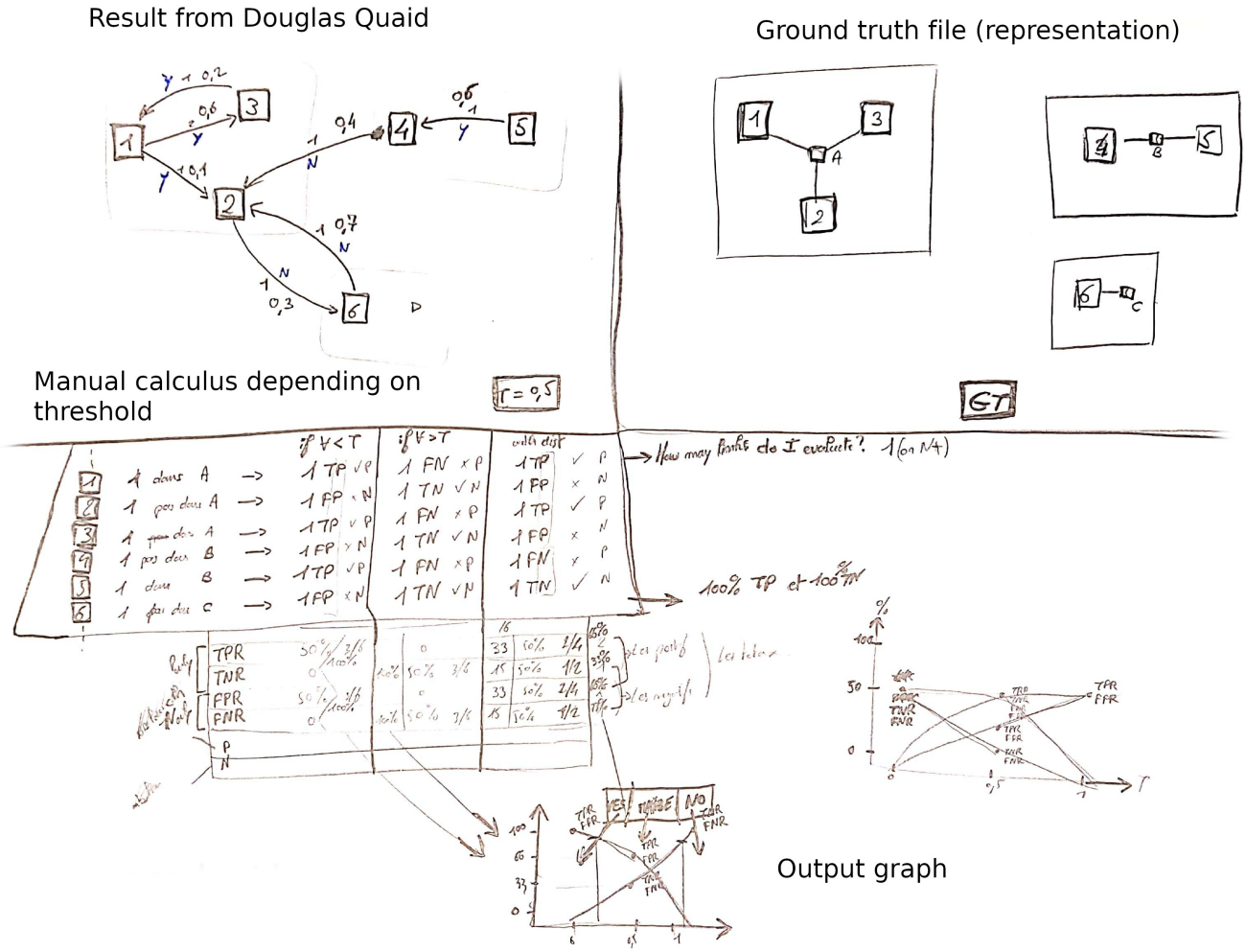
side, we have a a graph representation (output of the client-side graph extractor, which is requesting all pictures of the database and its nearest neighbors).
Quality evaluation of a graph regarding a clustered version of the data is not trivial and so may need some explanation. Starting from the output graph, we evaluate each link (each matches) in an ascending order. So, we go through the best match to the worst match, for each picture of the database. We should evaluate each link as a True Positive, False Positive, True Negative, False Negative, etc.

Essential elements are :

- If the current picture and the matched picture are in the same cluster in the ground truth file, this match is counted as a positive data (P)
- If the current picture and the matched picture are NOT in the same cluster in the ground truth file, this match is counted as a negative data (N)
- If the current match of the current picture is lower than the threshold, this is counted as a positive match (xP) as follow :
  - If then, both pictures were in the same cluster in ground truth file, this is a True Positive match. (TP)
  - Otherwise, if both pictures were NOT in the same cluster in ground truth file, this is a False Positive match. (FP)
- If the current match of the current picture is greater than the threshold, this is counted as a negative match (xN) as follow :
  - If then, both pictures were in the same cluster in ground truth file, this is a False Negative match. (FN)
  - Otherwise, if both pictures were NOT in the same cluster in ground truth file, this is a True Negative match. (TN)

Then other indicators, as True Positive Rate (TPR), True Negative Rate (TNR) etc. can be computed.

Ground truth file format and requests outputs from Douglas-Quaid API are displayed below.



(a) Graph evaluation - Manual verification

Figure 7.2: Test concerning output sgraph evaluation

Listing 7.1: List of requests and result for each image of the database

```
[{' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.1,
                       'image_id': '2'},
                      {' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.6,
                       'image_id': '3'}],
  'request_id': '1',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.3,
                       'image_id': '6'}],
  'request_id': '2',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.2,
                       'image_id': '1'}],
  'request_id': '3',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.4,
                       'image_id': '2'}],
  'request_id': '4',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.6,
                       'image_id': '4'}],
  'request_id': '5',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.7,
                       'image_id': '2'}],
  'request_id': '6',
  'request_time': 0,
  'status': 'matches_found'}]
```

Listing 7.2: VisJS-Classificator output format ground truth file

```
{'clusters': [{'group': '',
               'id': 'A',
               'image': 'A',
               'label': 'A',
               'members': ['1', '2', '3'],
               'shape': 'image'},
              {'group': '',
               'id': 'B',
               'image': 'B',
               'label': 'B',
               'members': ['4', '5'],
               'shape': 'image'},
              {'group': '',
               'id': 'C',
               'image': 'C',
               'label': 'C',
               'members': ['6'],
               'shape': 'image'}],
 'edges': [{'color': 'gray', 'from': '1', 'to': 'A'},
           {'color': 'gray', 'from': '2', 'to': 'A'},
           {'color': 'gray', 'from': '3', 'to': 'A'},
           {'color': 'gray', 'from': '4', 'to': 'B'},
           {'color': 'gray', 'from': '5', 'to': 'B'},
           {'color': 'gray', 'from': '6', 'to': 'C'}],
 'meta': {'source': 'DBDUMP'},
 'nodes': [{'id': '1', 'image': '1', 'label': '1', 'shape': 'image'},
           {'id': '2', 'image': '2', 'label': '2', 'shape': 'image'},
           {'id': '3', 'image': '3', 'label': '3', 'shape': 'image'},
           {'id': '4', 'image': '4', 'label': '4', 'shape': 'image'},
           {'id': '5', 'image': '5', 'label': '5', 'shape': 'image'},
           {'id': '6', 'image': '6', 'label': '6', 'shape': 'image'}]}
```

7.3 Threshold extraction from quality Graph

One ability of the library would be to define its own thresholds from an input dataset provided along with a ground truth file.
We've seen that a scoring mechanism can allow us, for a given threshold, to extract true positive, false positive, true negative, false negative rates and more. This is not enough for the library to work : we need to extract thresholds out of these measures.
A simple mechanism compute these bunch of values for all possible thresholds in some range. For example, we will compute TP,TR,FN,FN for a distance threshold of 0.1, 0.2, 0.3, etc. A threshold at X means that all links with a distance below X are marked as "Good link" thanks to the library and all links with a distance upper than X are marked as "Bad Link" thanks to the library. Therefore, we have more links marked as "good" if the threshold is increased. Therefore, we have more false positive : link that should not have been marked as "good" but were marked as "good".
For each case (True positive, false positive, true negative, false negative) we want to have the "best" threshold. As this definition is related to what is to be expected from the library, we want to configure it by a percentage of acceptable false negative, or acceptable false positive, or minimum true positive, or minimum false negative. We could also want a mean value, if we don't want to have a YES/MAYBE/NO threshold, but only a YES/NO threshold.

Therefore, we defined 4 thresholds obtainable from a score graph. If we setup a 10% acceptable rate of false negative, false positive, etc. we can have for instance :

- A TPR threshold : upper to this threshold, there is more than 90% true positive
- A FNR threshold : upper to this threshold, there is less than 10% false negative
- A TNR threshold : below to this threshold, there is more than 90% true negative
- A FPR threshold : below to this threshold, there is less than 10% false positive

We are then able to know which region of the graph, we label as "YES, it's a match"/"MAYBE, it's a match"/"NO, it's not a match", regarding the current evaluated distance between two pictures .

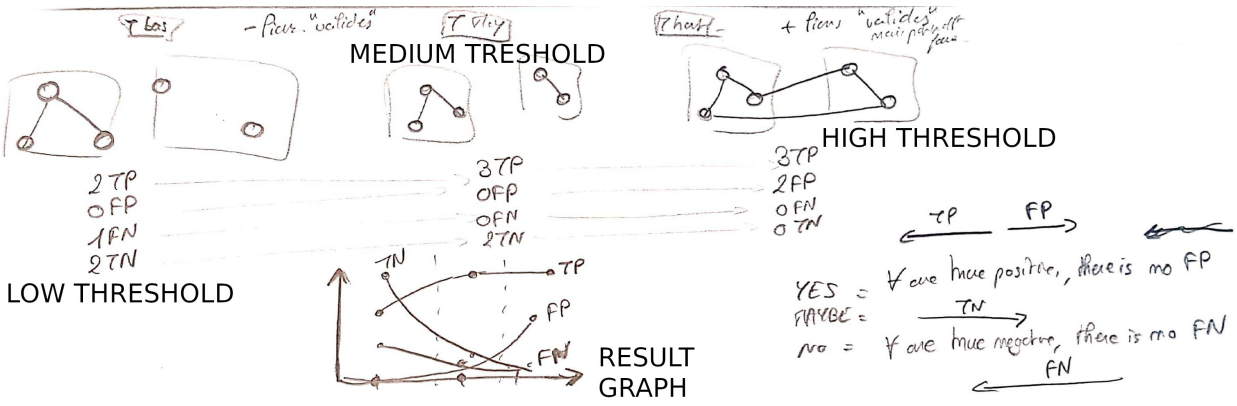
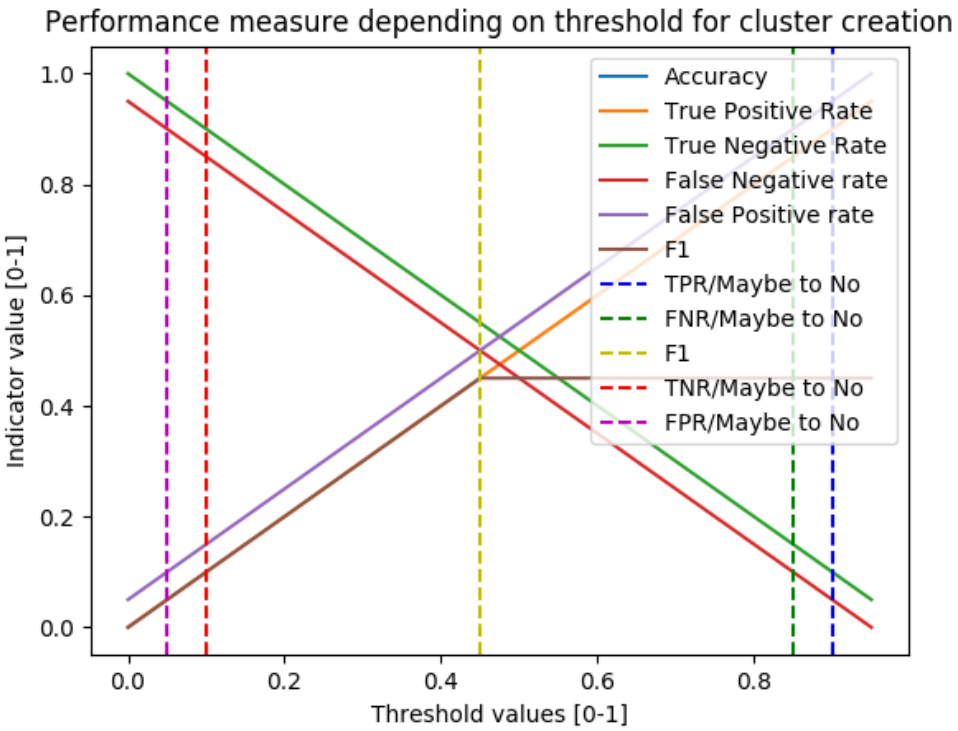
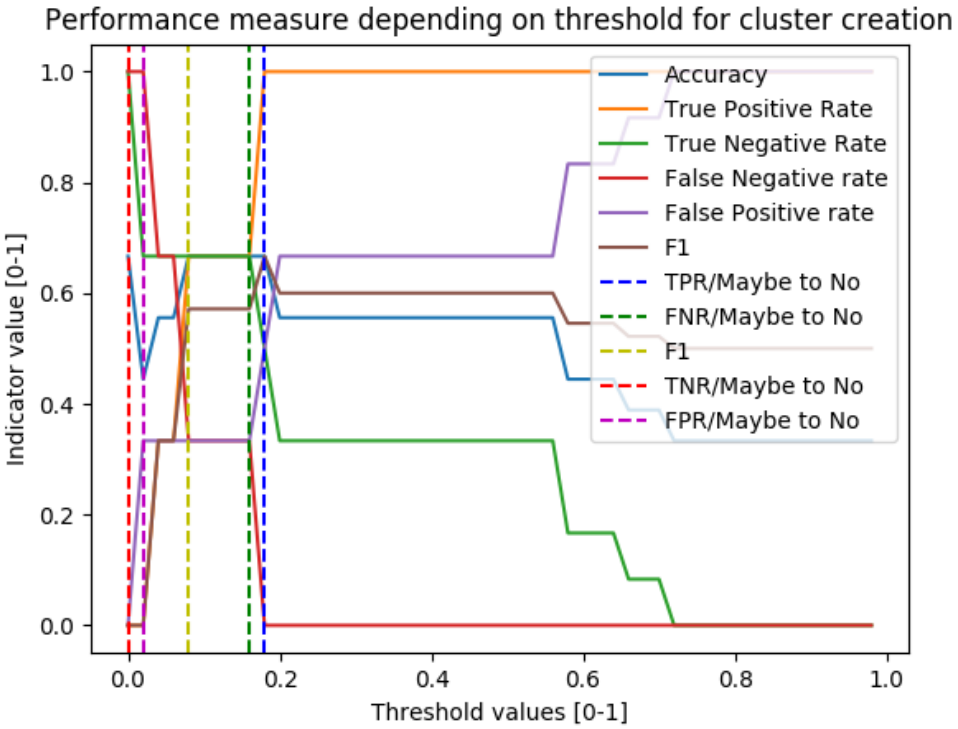


Figure 7.3: Thresholds examples

You can then understand where the areas a placed on 7.4b : the leftmost area (lower than 0.05) is a YES area, between 0.05 and 0.19 a MAYBE area, and upper a NO area.



(a) Simulated values



(b) Real values - ORB

Figure 7.4: Thresholds extraction from a scoring graph

Chapter 8

Visualization

A visualization tool had been built for the occasion and is available at `https://github.com/Vincent-CIRCL/visjs_classificator`

Bibliography

[Pyt, a] Python - Pickle exception for cv2.Boost when using multiprocessing.

[Pyt, b] Python - Pickling cv2.KeyPoint causes PicklingError - Stack Overflow.

[Cevikalp et al.,] Cevikalp, H., Elmas, M., and Ozkan, S. Large-scale image retrieval using transductive support vector machines. 173:2–12.