MEERKAT

Facial Recognition

Technical Manual

MEERKAT - FACIAL RECOGNITION

Welcome!

Hello, this document will describe the technical details of the facial recognition product from Meerkat. It also includes a brief description of the system architecture, and show the most efficient way to incorporate our technology according to your application needs.

DELIVERY METHODS

1. API Rest on cloud

This is the most effortless and straight way of usage. The API was make to ease the client access, providing an easy and quick integration. In a few minutes it is possible to create the first person base to run your first facial recognition tests.

ACCESS

The facial recognition is available as an Rest API and its access is made through HTTP requests. This service is running on the cloud on the Amazon servers. The access is done by using an unique hexadecimal key, avoiding the use of login and passwords. The client is responsible for this key security since the addition and exclusion of your database is done through this key.

SECURITY

The access can be made through HTTP or HTTPS for the increase in security. Furthermore, Meerkat does **not** keep the training images on your data base, only the facial descriptors used on the recognition process.

QUOTAS AND PLANS

The payment model is done by a monthly subscription without any cancelation fee. It is also available a free version which may help on the first tests. Besides that, we also offer two more plans that limit the number of persons, points and HTTP calls to the API. The number of calls is accounted on each month, starting from the subscription day. For example, if the user subscribed on November 10, on every 10th of each month the HTTP calls quota is cleared.

The number of persons is increased at each new label (i.e. person), and the number of points is the total number of faces that the user provided for training. For example, if the user trained 4 people, with 5 images for each one, the total number of points is 20.

The available plans are the following:

	Calls/month	No. points	No. people	Details
Free	1000	100	10	e-mail support
Standard	10000	1000	100	Phone and e-mail support
Pro	100000	10000	1000	Phone and e-mail support

2. API on premise

We also provide an on premise solution of the facial recognition software, joining the simple API usage with the high performance of a dedicated server. This model is also the ideal one for facial recognition *hard-users*, when the faces database should be stored locally and/or the connection with the API on cloud is not viable — one perfect example occurs when the client need to use the facial recognition on a surveillance video camera. This approach is not feasible using an API on cloud since the communication limitations through the internet (jittering, bandwidth, etc.) makes the real-time image analysis of a camera stream impractical.

ACCESS

In order to facilitate the facial recognition integration with your software we provide a modified version of the API that will run in one (or several) client servers. This API version supports the usage of web socket, which allows the user to send a video stream with high performance. Besides this feature, all the routes from the cloud API are present on this version. Also the user can easily and quickly set up a server with the on premise API since it is delivered as a Docker container (compatible with Windows/Linux/Mac). More details on the technical section.

If you need to use facial recognition on a video stream we offer two different approaches. The first one is for the user to develop a simple wrapper for the client to open an web socket connection with the on premise server (this connection is done in a different port from the API HTTP). The other alternative is for the user to utilize a *daemon* provided by Meerkat which delivers the camera stream to the server and save the results on a JSON. On this second approach it is also possible for the client to set up an Web Hook, and the server automatically send the results to any HTTP server provided by the client.

PERSONALISATION

This model is the ideal one if the client needs a higher performance or if it is interested in features that are exclusive for this version. Also we are open to get in touch with our clients and really understand their needs. For example, we can personalize our machine learning algorithms to better suit the characteristics of your problem, boosting the performance and accuracy of our facial recognition system.

CLUSTERING

This an exclusive feature of the on premise version, which process a video with multiple people and, automatically, group similar faces and associate each group to a label. This feature can be used to ease the people training process. For example, using the camera of a receptionist desk, it is possible to extract some seconds of the video of this video, use the clustering to automatically separate each person faces, and use those faces for the facial recognition training process.

One big advantage of this feature is that multiple people can be present on the video, so it is possible to create a non-invasive and passive people training process, avoiding the need for a specific training setup.

3. SDK

If the client already has a robust system that use and process the camera streams it is desirable to directly integrate our facial recognition system on the client software. To attend those specific needs, Meerkat provides an SDK (Software Development Kit) of the facial recognition according to the clients system and programming language. In order to provide a high performance software, the SDK is a wrapper of our main facial recognition library which is made entirely in C++.

ACCESS AND DOCUMENTATION

The access is done through the provided SDK library. We provide the binary of the client OS library with a documentation, that includes tutorials and examples.

PLAN

If the client chooses this version, it should get in touch with us through sales@meerkat.com.br.

SYSTEM

The facial recognition system is divided in two main steps, called training and recognition. In order to allow the API to recognise some one, first it must be "taught" how it looks like, which is done on the training stage. Once the training step for a given person is finished, the user may send images to the API, and if this person is present and facing the camera, it will be recognised.

In the following sections we will get in more details on how the Training and Recognition process works.

1. Training

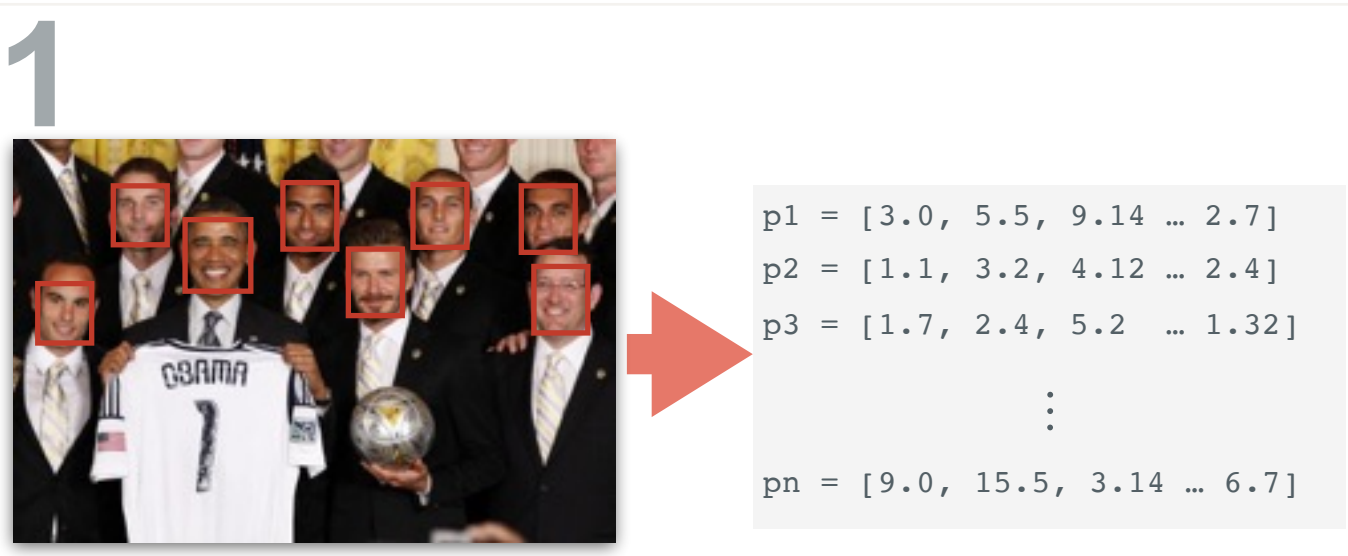
The training step is essential to the recognition process, since it is how the system will learn the

appearance of someone. This process is shown in the figure below. The user must give a set of images with associated labels - same labels represent the same person. Then it is extracted the descriptor of each face and stored at the user database in conjunction with their respective labels. In general it is a recommended to have at least 5 pictures of every people in different environments and facial expressions.

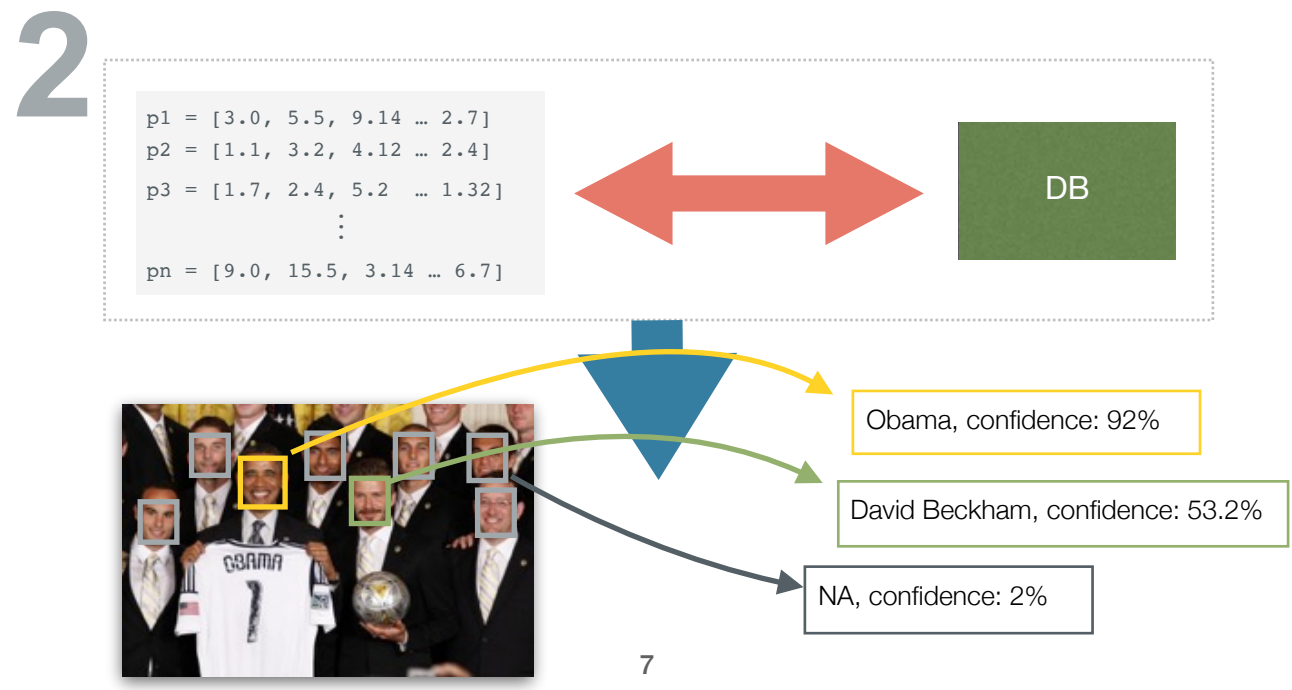
If the training step is done by extracting frames of a video, the person must (and shouldn't) be static during the video shooting. Another advice is to use training images that will be similar with the images of the recognition process, i.e. similar environment and/or lighting. More details regarding the training process on the Best Practices chapter.

2. Recognition

In the recognition, the user provides an image to the system which detects tall the faces. It is then generated a face descriptor for each one, and they are compared with the ones on the user database,



Feature extraction for each detected face.



Comparison with the database for extraction of recognition labels and associated confidence levels. Small confidence levels usually indicates people that are not in the database.

which associates a prediction *label* with a confidence value for each one. This is illustrated in the previous figure. Faces that are not present on the database will have small confidence values, usually around 0%. This informs the user that the system did not found any good match with the trained faces.

3. Clustering and Verification

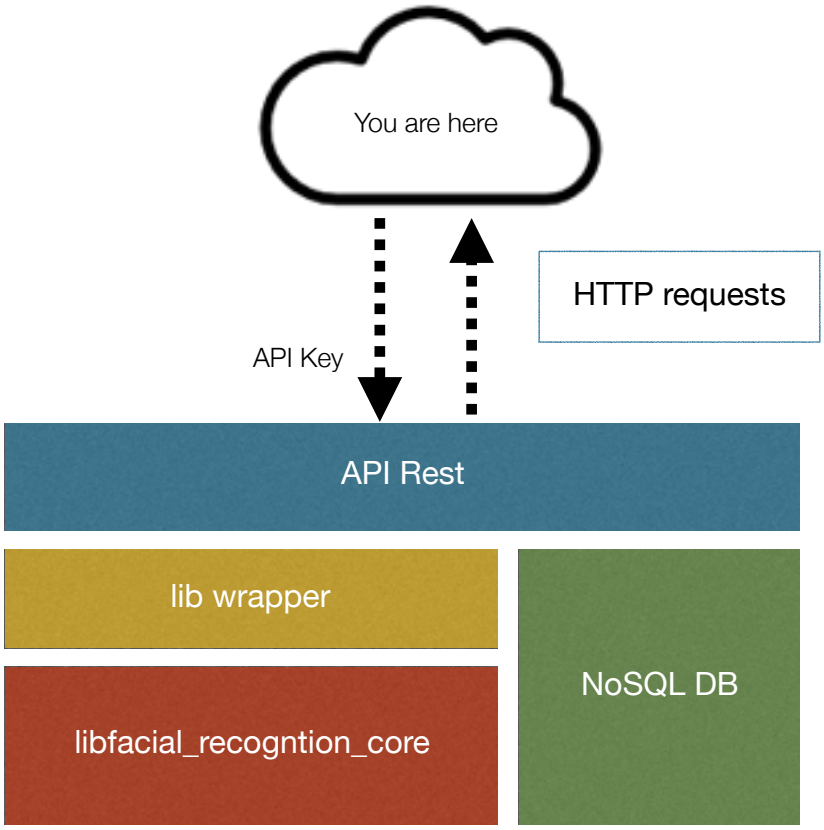
There are also the face **clustering** and **verification** processes that are based on the same principles previously shown. The clustering process is only available on the on premise version, and the verification is already available on the cloud API.

The clustering process uses a probabilistic distance metric to find groups of similar faces. For every set of similar faces that the algorithm finds, it associates an unique id. This process can be quite useful on the training stage.

The verification process is somewhat similar to the recognition. The difference is that instead of comparing a given face with the whole trained dataset of faces, it is done only with the faces of a given person (either already trained or by passing another picture of him). This provides a much more reliable method of verification. The most common usage is to have a register photo of a given person and to pass an image of someone to validate if they are indeed the same person.

SYSTEM ARCHITECTURE

On all of our facial recognition systems (cloud API, on premise and SDK) the core implementation is done in C++ by Meerkat (patent pending). The API architecture can be seen on the figure below:

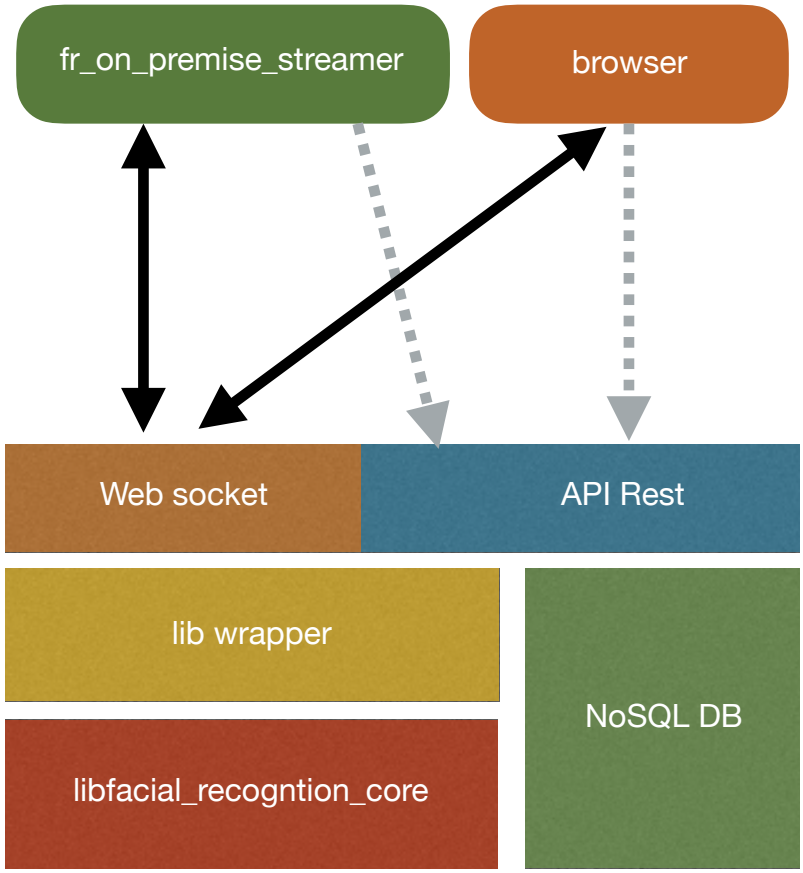


Cloud system architecture

In all the system architecture we prioritise the recognition performance. As mentioned, the *libfacial_recognition_core* was implemented with optimisations to speed up the recognition. We also use a modern data base (NoSQL) where the points and labels of the client database stays pre-indexed. Above the basic recognition lib, there is a Python wrapper to be used on the API. In a similar manner, if the client wishes a SDK, we develop a wrapper on top of the C++ code to provide an easy integration with the client system without loosing the recognition performance.

A significative alteration can be seen on the on premise software version. Since the facial recognition can be used on camera streams, use individual HTTP layers would produce a prohibitive performance hit on

the system. To surpass this problem we provide the support of websocket connections, which create a full duplex (on both ways) communication between the client and the server:



On premise version employs websockets for client/server connections (browsers also)

As seen in the figure above, the websocket connection allows the client to use our desktop client (called *fr_on_premise_streamer*), and also allows a browser implementation through JavaScript. This allows the facial recognition to be run on the browser, in real time! As will be shown within this technical manual, there is a connection address that is created to send the frames and receive the recognition results. It is important to notice that the training step is not done using camera streams, but through HTTP calls to the on premise API.

BEST PRACTICES

In order to have good results when using computer vision methods, it is important to note several factors such as characteristics of the camera, its positioning and the environment lighting. We gather in here a brief set of best practices when using our facial recognition in order for the client to achieve the best results.

1. Camera Configuration

A correct camera configuration is quite important, especially due to the scene lighting, the people positioning and the camera distance.

ENVIRONMENT LIGHTING

Our method generate face descriptors that are robust to different illumination changes on the face, however due to the limited luminance dynamic range of the cameras they should not be positioned pointing to strong lights, the sun or areas with strong illumination.

POSITIONING

The camera must be positioned in such way that the people faces must be vertical. And as it can be seen on the figure above, the faces must be frontal to the camera, with a maximum yaw of 25º.

DISTANCE

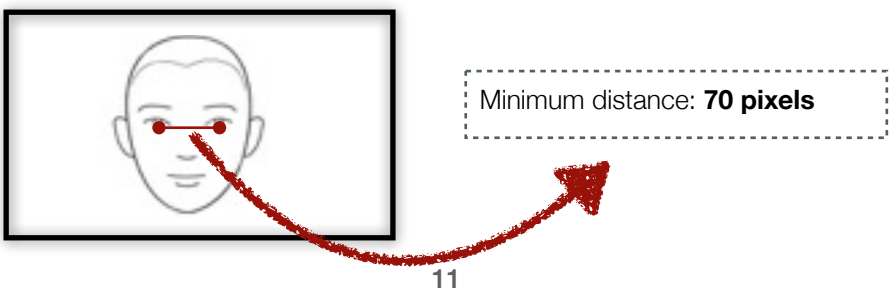
The camera distance can vary greatly from camera to camera, according to the lens and sensor characteristics of each one. Independent of the camera distance, the captured faces must have a minimum size, which will be reviewed as follows, on Section 2.

IMAGE QUALITY

On the current facial recognition software we do not evaluate the image quality before the recognition. Therefore if the camera present a noisy image or if the faces are blurred due to focus problems, the recognition results are not going to be very reliable.

2. Face Dimensions

The facial recognition quality depends on the size (resolution) of the faces, either for the training as for the recognition. In order to have good results, the minimum distance between the eyes should be 70 pixels, as it can be seen on the following figure. For faces with eyes distance above 300 pixels there will no longer be any significant performance increase.



3. Use of the Recognition Confidence

It is possible to be more or less strict on the recognition process by using the resulting confidence value. For example, to block or allow the access of some one on a restricted area, the client system can only accept people with recognition with confidence of 80% or above. This way you can have a big certain that the people was correctly recognised, avoiding to incorrectly let someone pass.

However this will let the recognition process more selective, since correct recognition that present a confidence lower than the 80% threshold, such as 70%, will be denied access. It is up to the system user to find a good value that better suit their needs.

Another utility for the confidence is to discriminate between trained and not trained people. A good model is to assume that recognitions with confidence below 20% are probably from non-trained persons. With this information it is possible to recognise the trained people as well as identifying which ones are not.

REST API

The Meerkat offers the facial recognition service as a Rest API, and the access is done through HTTP requests. This service is available on cloud through the Amazon servers, and can be accessed using an hexadecimal unique key, which avoids the use of login and/or passwords.

All the Swagger documentation can be accessed in https://api.meerkat.com.br/frapi/docs/ .

1. General Overview

The API interface is quite simple, you just need to know how some of the most important routes works to start using our facial recognition system. The most important routes are **/train** and **/recognize**:

- **/train/person:** This route is used to include a labeled person in your database. If more than one person is present in the image, it will take the largest face detection of the image. It is as simple as giving an image and a label to the system.
- **/recognize/person:** This will detect all the faces present in the image and associate labels to them. The data will contain the predicted label and a value of confidence ranging from 0 to 100. The value of confidence is ideal to attest the accuracy of the recognition and to prone people that are not in the database (a threshold of < 20 is usually enough for the latter).

Another route that might be useful, depending on the client needs, is the **/verify**:

- **/verify/person:** This route receives an image and a label, detects the largest face in the image and verifies if it is the same person given by label. Besides this information, it is returned a confidence value referent to the result.

Your API calls will always be identified by the API key, which must be present in the HTTP request headers. This key can be obtained here: http://www.meerkat.com.br/solution_fr.html

There are two ways to pass an image to the API, depending on the content type of the request:

- **multipart/form-data:** Give this Content-Type, the API will look for a parameter in the form called image. See in the example bellow how to perform this, is quite simple. For obvious reasons, this only works with POST requests.
- **application/JSON:** In this case, you should provide an url of the image inside a json. The name of the parameter is imageUrl and should be available for direct access.

These couple of points together with the swagger reference should be enough to perform the first steps on facial recognition, so lets do it!

2. *Getting Started*

TRAINING

Imagine that you have a local path with a couple of subdirectories which contain the train images for you base. For instance, a directory called ‘train’ with subdirectories ‘Arnold_Schwarzenegger’, ‘Julianne_Moore’, ‘Luke_Skywalker, ‘Pele’, etc. The following snippet of code in python can be used to train this dataset:

```
import requests
from requests_toolbelt import MultipartEncoder
train_path = './train'
HOST = 'http://api.meerkat.com.br/frapi/'
api_key = YOUR_API_KEY

for label_name in os.listdir(train_path):
    for image_name in os.listdir(train_path+'/'+label_name):
        filename = train_path+'/'+label_name+'/'+image_name
        m = MultipartEncoder(fields={'image': ('filename', open(filename, 'rb')
),
                                     'label': label_name})
        res = requests.post(HOST+'/train/person', data=m,
                    headers={'Content-Type': m.content_type, 'api_key': api_key
})
```

This code uses the Python package requests and requests_toolbelt, but to use your favorite language/ package is quite straight-forward. You will know that it worked if the returned status code is 200 and you get back a JSON similar to this:

```
{
    personSamples: 3,
    selectedFace: {
        bottom_right: {
            x: 156,
            y: 123
         },
         top_left: {
            x: 196,
            y: 162
         }
    },
    trainId: 06752ebc7286633fcc1f31dc29e04037
}
```

You might want to keep the trainId for later if you are interested in removing this training sample.

RECOGNITION

Recognition is as simple as it can get, you just need to make a request (GET or POST) to the **/recognize/people** route:

```
filename = 'some_image_to_recognize.png'
m = MultipartEncoder(fields={'image': ('image', open(filename, 'rb'))})

res = requests.post(HOST+'/recognize/people', data=m, headers={'Content-Type':
m.content_type, 'api_key': api_key})
```

You should get a JSON like this:

```
{
     people: [
          {
               bottom_right: {
                    x: 277,
                    y: 315
               },
               top_left: {
                    x: 107,
                    y: 145
               },
               recognition: {
                    confidence: 70.6480930725795,
                    predictedLabel: Schwarzenegger
               }
          }
     ]
}
```

Notice that together with the face rectangle and the recognized label, we also provide a confidence value from the face recognition algorithm.

That is it. This (together with the referenced document) should be enough for you to start using the API.

API ON PREMISE

The on premise version of our facial recognition software joins the ease of the API usage with the high performance of a dedicated local server. This model is recommended for facial recognition hard-users, when the faces database must bus stored locally and/or the connection with the cloud API is not viable.

On the next sections it will be reviewed the installation, configuration and usage of the facial recognition API on premise.

1. Installation

The installation is quite simple and fast, since it is distributed through a Docker container which guarantees an easy integration with Windows, Linux and Mac. The user just need to run the script ‘install_server.sh' provided by us, given that the docker client is already installed. During the installation, if you are using an Unix system, the docker commands will require 'sudo' privilege to proceed.

Within the installation the Docker image will be downloaded from the internet, and it has a size of approximately 1.3GB. At the end of the script the Docker should be already running the container called 'meerkat', and the user can confirm that the API is running by typing 'http://localhost/frapi/version' on the browser. Notice that the API initialization may take around 40 seconds, depending on the characteristics

2. Configurations

During the installation, Docker will bind some ports. Those are the ports and their usage:

- **80:** nginx
- **9001:** supervisord
- **8091:** couchbase
- **4444:** frAPI

At the moment there is no way to alter the bind of those ports during the installation. If the user can't use some of those ports, it is possible to change the lines 40 or 45 of the installation script (depending if your OS is Mac or Unix) and choose different ports to be mapped.

To use the API the computers sending the HTTP requests must use the server ip on port **80** and with the prefix **/frapi/**. If the user wishes to do the recognition directly on the server, it may be done on port **4444** and without any prefix. The usage of this on premise API is identical to the cloud model, it is also necessary to use an API key. This key is generated automatically in the installation moment, and can be obtained by seeing the container logs. This can be done by first obtaining the container ID (using 'sudo docker ps -l' and searching for the ID of the 'meerkat' container). Then you just type the following commando: 'sudo docker logs id_meerkat | grep api_key', using the correct ID in the place of ‘id_meerkat’.

3. Usage

The API can be used either with images or video, and provides a different interface for each one. To use images, the interface is identical to the cloud API, so the user may refer to the previous Section. To use the video features that are only present on the on premise version just read the following sections..

CLUSTERING

The clustering process uses a probabilistic distance to group similar faces within a video, and the people within the video don't need to be trained on the system. This feature can be very useful on the training step, since the faces of each person can be automatically extracted without requiring a more rigorous setup.

The clustering process can be accessed through the following routes:

- **/cluster/process (POST):** this route receives a video url, given on the parameter *videoUrl*. It is also necessary to provide the number of people that appears on the video (*numPersons*) and the number of faces that you wish for each group/cluster (*numFaces*). In case of uncertainty on the number of persons, it is better to over estimate the value of *numPersons.* This route returns a *clusterId* which the user must keep to retrieve the result on the future, since this process is not instantaneous. It is also possible to use an webhook by passing an url to were the results will be send once the clustering is done.
  - Arguments (Json format):
    - **videoUrl (string):** video url of the video to be processed.
    - **numPersons (number):** number of people on the video. n case of uncertainty on the number of persons, it is better to over estimate this value.
    - **numFaces (number):** number of faces to be returned for each cluster (people).
    - **stepFrame (number):** number of frames that the system will ignore for each processed frame. This forces the algorithm to not return faces on consecutive frames since they are mostly redundant. Default: 25;
    - **webhook (object, optional):** the user can set an webhook so that the API will automatically send the cluster results as soon as they are ready.
      - **postUrl (string):** url to where the result will be sent (through POST).
      - **postImages (bool):** indicates if the user wants the image of the faces to be sent on the post. **Important:** all the images (encoded in jpeg) are sent on a single post whose size is proportional to the total number of faces.
  - Response:
    - **clusterId (string):** id identifying this clustering process. May be used to retrieve the result if the user is not using a webhook.
    - **videoUrl (string):** video url.

- **/cluster/result (GET):** this rout receives a *clusterId* and can have two different responses: a JSON indicating that this video is still being processed **or** the result of the clustering process. Once the result of a given clustering process is retrieved, it is removed from the database. The result consist of *numPersons* clusters, each one with *numFaces* faces. Those faces are returned in order of confidence, i.e. the first faces are the ones with higher chance of being part of their cluster.
- **/cluster/result (DELETE):** this route receives a *clusterId* and remove their respective result from the database.
- **/cluster/list (GET):** returns a list of all clustering processes, either still being processed or the ones that already ended (but those are not removed from the database, such as **/cluster/result** route does). It is a good way to know all the clustering process that are still being processed.

To do the clustering of faces on a video, the user must first call the **/cluster/process** route with the video url. It will receive a *clusterId* that must be used to retrieve the results. Since this process may take some minutes (it will greatly vary according to video resolution, length, number of faces, the server hardware, etc.), so the user must periodically consult the api (through the **/cluster/result** route) to check if this process is over. Another alternative is to use a webhook, so that the results are automatically send to the user once they are available.

The result consists of several clusters (defined by *numPersons*), and each cluster contains several faces, which are indicated by a rectangle (pointing the image position) and their respective frame (on which frame this given face appears). With all those informations the user may retrieve the faces of each cluster that, ideally, should represent distinct people. And if the user wants, it is possible to use a webhook with the parameter *postImages*, so that the server also return the images of the faces. Since those images will be transmitted through a single POST request the user must be aware of limitations on the maximum size of POST that your system/network may have.

USER MANAGEMENT

It is possible to add new users to the system, each one with an unique api key. This is a simple and quick way to have multiple training bases on the same system. For example, if you use the facial recognition on several different stores, it is possible to have one training base for each one, but using a single appliance.

Those are the routes to manage users:

- **/user/create (POST):** this route create a new user on the system, with a new unique api key and training bases. It receives a JSON with the fields *email*, *accessKey* (that must be ***6869fab6ae6e276e7f6e1c3fcf5253ca***) and *subscription*, that might be *free*, *standard*, *pro* (just as mentioned here: http://www.meerkat.com.br/solution_fr.html) or *unlimited*.
- **/user/list (GET):** returns a list of all the users.
- **/user (GET):** returns the information of the user of the api key present on the header.

VIDEO STREAM

It is possible to use the facial recognition on a video by sending the stream to the server throgh websockets. The use of websockets allows a much more practical and fast communication to process video streams, which would be prohibitive doing through HTTP interface.

The user must connect a websocket on the route **/recognize**, providing the api_key as a query string, like this: *ws://serverr:port/recognize?api_key=yourapikey*. Then you should send each frame of the video encoded as jpeg (to reduce the bandwidth), and it will receive a JSON with the recognition results on the same format as the one from **/recognize/people** route.

We also provide an easier and practical option to the user: we made a small server that manages multiple video stream in an efficient manner. If the user wishes to use it, more details about it will be seen next.

STREAMS SERVER

This server is located at the folder '**fr_on_premise_streamer'** and can be initialized executing the script 'run_streamer.sh'. After its initialization, the user can configure it by using the route **(POST) /config/apply** and passing a configuration JSON. As default, the server always tries to load the configuration 'config/config.json'.

An example of configuration can be seen next:

```
{
    "frapi":
    {
        "api_key": "31a5d10dd5725b32f58d1347278ea335",
        "ip": "localhost",
        "port": 4444,
        "output": {
            "json": {
                "node_frames": 5,
                "dir": "./out_json"
            },
            "http_post": {
                "url": "http://localhost:5000/test_premise_post",
                "post_image": true
            }
        }
    }
    "testSequences": [
        {
            "camera_url": "0", "label": "Webcam", "plotStream": true,
            "tempCoherence":
            {
                "tempWindow": 15,
                "threshold": 0.6
            }
        },
        {
            "video_file": "/home/meerkat/test_video.avi",
            "label": "TestVideo", "plotStream": true
        }
    ],

}
```

It has two main configuration fields: 'frapi', that configures the connection with the on premise server, and 'testSequences', that defines the video streams.

The configurations of **frapi** are the following:

- **api_key (string):** access api key.

- **ip (string):** ip from the server.
- **port (int):** connection port with the server.
- **output (optional):** defines how the results should be presented.
  - **JSON (optional):** saves the results on a folder, with the stream label and timestamp.
    - **node_frame (int):** number of frames to be grouped on each JSON.
    - **dir (string):** folder where those results are going to be saved.
  - **http_post (optional):** configures an webhook, which POST the recognition result.
    - **url (string):** url to send the POST.
    - **post_image (bool):** defines if the processed frame should be returned on the post, in jpeg. **CAREFUL:** this option may impact the system performance.

The video streams are defined on **testSequences**, and are configured as following:

- **label (string):** stream label. It must be unique (i.e. no other stream should present the same name).
- **plotStream (bool):** defines if the stream result should be shown on a window.
- **tempCoherence (optional):** uses temporal coherence to lower the number of false positives. However, with this option the user will loose the information of confidence and position of the people, it will only receive as result a list of people that are present on a single frame.
  - **tempWindow (int):** number of frames to use on the temporal window. The bigger this value, lower the chance of false positives to be detected, however the time (in frames) for a person to detected is larger.
  - **threshold (float):** confidence threshold for someone to be recognized. The higher this value, the more strict the recognition system will be. Value
- **camera_url / video_file (string):** input video. It may be from an ip camera (**camera_url** receives an url to connect) or a video file (**video_file**). When using the option **camera_url**, it will always be used the most recent frame (i.e., some frames may be ignored if the server is not able to keep up with the frame rate). Thi characteristic is important so that we always have the most recent result possible.

If the server is is already sending some video streams to the facial recognition server, it is possible to add another stream jusy by sending a configuration JSON with this new stream present on the **testSequences** field. This new video stream will be send to the facial recognition server (through a new websocket connection), and the previous streams will remain untouched.

For example, if we started the streamer server with the previous configuration, and later on we send the configuration below, we would close the stream "Webcam" since it is no longer present, and open a new stream called "NewStream". This operation will not have any effect on the "TestVideo" stream, which will be keep transmitting and receiving the recognition results from the facial recognition server.

```
{
    "testSequences": [
        {
            "video_file": "/home/meerkat/test_video.avi",
            "label": "TestVideo", "plotStream": true
        },
        {
            "video_file": "/home/meerkat/new_stream.avi",
            "label": "NewStream", "plotStream": true
        }
    ],

    "frapi":
    {
        "api_key": "31a5d10dd5725b32f58d1347278ea335",
        "ip": "localhost",
        "port": 4444,
        "minConfidence": 0.8,
        "output": {
            "json": {
                "node_frames": 5,
                "dir": "./out_json"
            },
            "http_post": {
                "url": "http://localhost:5000/test_premise_post",
                "post_image": true
            }
        }
    }
}
```