UNIVERSITÀ DI PISA
SCUOLA DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Industrial Application

Film reproduction prototype documentation

Dessì Matteo
Ritorti Fabiana

CONTENTS

1.PROTOTYPE OVERVIEW

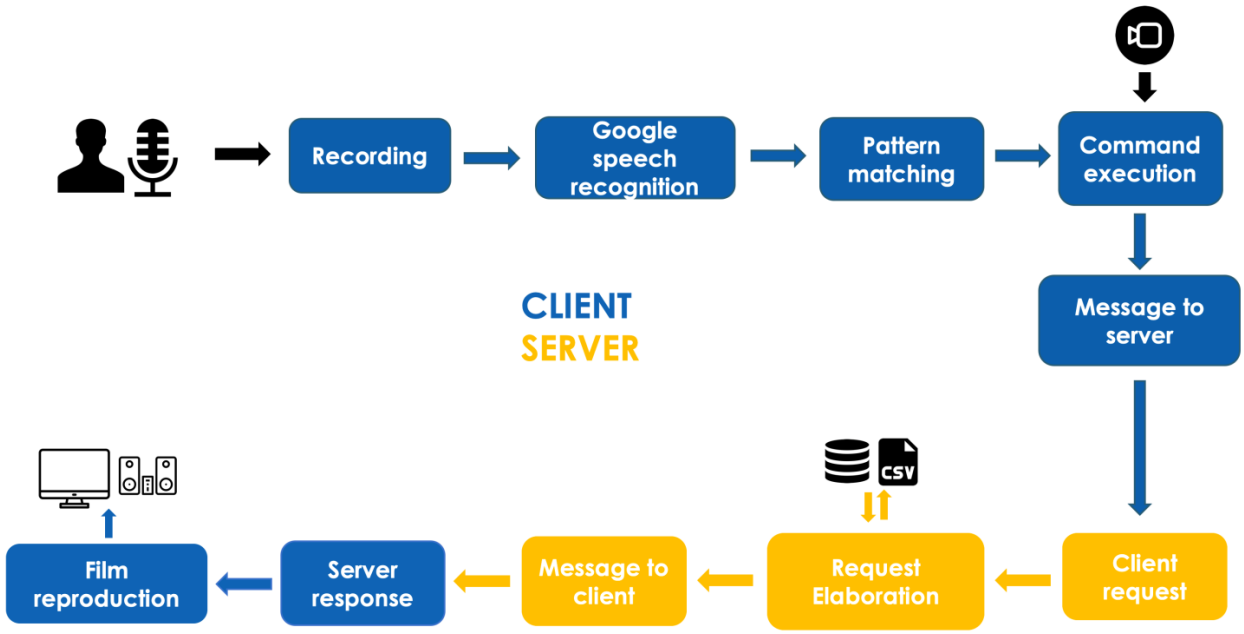
The goal of our prototype is to reproduce a movie trailer that satisfies the interests of users interacting with it.
Following a login phase, the system accesses to the interests of the users and, based on them, it provides a list of "candidates" movies from which they can choose the one they are most interested in through the microphone.

1.1 HARDWARE PHERIPHERALS

The realization of this prototype is carried out using Raspberry Pi 3 B+ to which have been connected the following hardware peripherals:

- *Raspberry pi camera,* used for the face recognition phase;
- *USB mini microphone mi305,* used by the users to give commands
- *HDMI monitor,* to visualize the interaction with the system
- *Bluetooth stereo,* for the audio reproduction.

1.2 PROTOTYPE FLOW OF EXECUTION

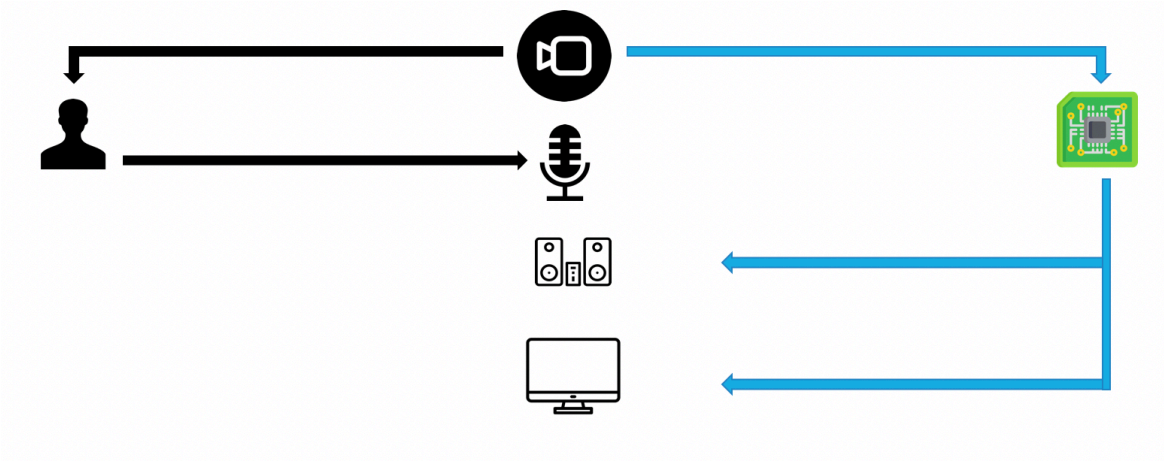


The typical flow of execution of our prototype is the following:

1. The user will give a command through the microphone.
2. The spoken is recorded and passed through a noise smoothing system.
3. On the preprocessed recording, the client calls the API offered by google to perform speech recognition.
4. The client controls which was the interaction requested by the user.

5. If the interaction required is implemented, the client executes the actions necessary to satisfy the request; in many cases (for example login and registration) this requires to retrieve frames of the users through the video camera.
6. The client creates a request and sends it to the server.
7. The server receives a client request.
8. The server elaborates the request; in order to do so, the server has to access the database of user interests, the csv file containing the user credentials or the flat file of database of film reviews.
9. The server elaborates the answer and send it to the client.
10. The client receives the answer from the server and act according to it.
11. Finally, if the user issued a film reproduction request, the client will reproduce the trailer of the film through the monitor and stereo.

2.USER-MACHINE INTERACTION



The user-machine interaction adopted in our prototype consists:

- On the user side:
  - the use of the microphone to provide commands to the system;
  - the use of the video camera to let the system record his/her and confirm the identify.
- On the machine side:
  - the reproduction of the audio which is the most adapt to respond to the user request, including the audio of a film trailer;
  - display on the monitor the film trailer;
  - take the frames of the users face through the video camera.

2.1 MICROPHONE IMPLEMENTATION

```
import speech_recognition as sr

r = sr.Recognizer()
        microphone_index = search_microphone_index()
        if(microphone_index==-1):
            print("No microphone,try again!")
            exit

        with sr.Microphone(device_index=microphone_index) as m:
            r.adjust_for_ambient_noise(m,duration = 5)
```

```
import pyaudio

def search_microphone_index():
    myPyAudio = pyaudio.PyAudio()
    devices = myPyAudio.get_device_count()
    for i in range(0,devices):
        name = myPyAudio.get_device_info_by_index(i)['name'].split(':')[0]
        if(name=='USB PnP Sound Device'):
            return i
    return -1
```

To do the microphone implementation, first an instance of **Recognizer ()** is created from the **speech_recognition** library, which is a python library that uses the recognizer class to convert the user's speech to text format; then, through this instance, an instance of the Microphone class is created with the *sr.Microphone()* function to which is given as a parameter the device index that corresponds to the USB PnP Sound Device peripheral.

The device index is provided by the *search_microphone_index()* , function in which we search for the index that corresponds to the USB PnP Sound device peripheral in a list returned by *get_device_info_by_index(),* a function of **pyaudio** (module that provides Python bindings for Port Audio) that returns the device parameters as a dictionary.

In order for the microphone to properly work we decide to adopt a noise smoothing mechanism through the execution of the *adjust_for_ambient_noise()* method.
The *adjust_for_ambient_noise()* arguments are the microphone previously implemented and the duration in seconds that corresponds to the time in which the system listens.

This function reads for the “duration” of the file stream the audio recorded through to the microphone and calibrates the recognizer to the noise level of the audio.

2.2 SPEECH RECOGNITION AND PATTERN MATCHING

```
from playsound import playsound
import speech_recognition as sr


def listen_phrase(r,m):
    text = ""
    audio = ""
    firstTime = True
    with m as source:
        while(audio==""):
            if (firstTime == False):
                playsound('./audio/unknown_response.mp3')
            else:
                audio = r.listen(source)
                firstTime = False

        try:
            text = r.recognize_google(audio,language="it-IT")
        except Exception as e:
            print(e)
    print(text.lower())
    return text.lower()
```

After the microphone implementation, what is spoken by the user is understood by the system through the **speech recognition** implemented in the *listen_phrase()* function. Two methods of the **speech_recognition** library are used: the first is *r.listen()*, in which what is said is heard is converted into audio data; the second is the *recognize_google()* method the google speech recognition API,and through which, by means of the audio parameter, what was said is converted into text; the recognition language is determined by the language parameter given to the *recognize_google()* method, our case Italian.

```
if(text== "login"):
```

What is understood by Google's speech recognition system is used in a pattern matching stage is which we check whether if the users have said any of the available interactions with the system.

```
from playsound import playsound

playsound('./audio/command.mp3')
```

From the machine side, the reproduction of the audio most adapt to respond to the user request is executed through the *playsound()* function, function of **playsound** library which requires as argument the path to the film audio to be played.

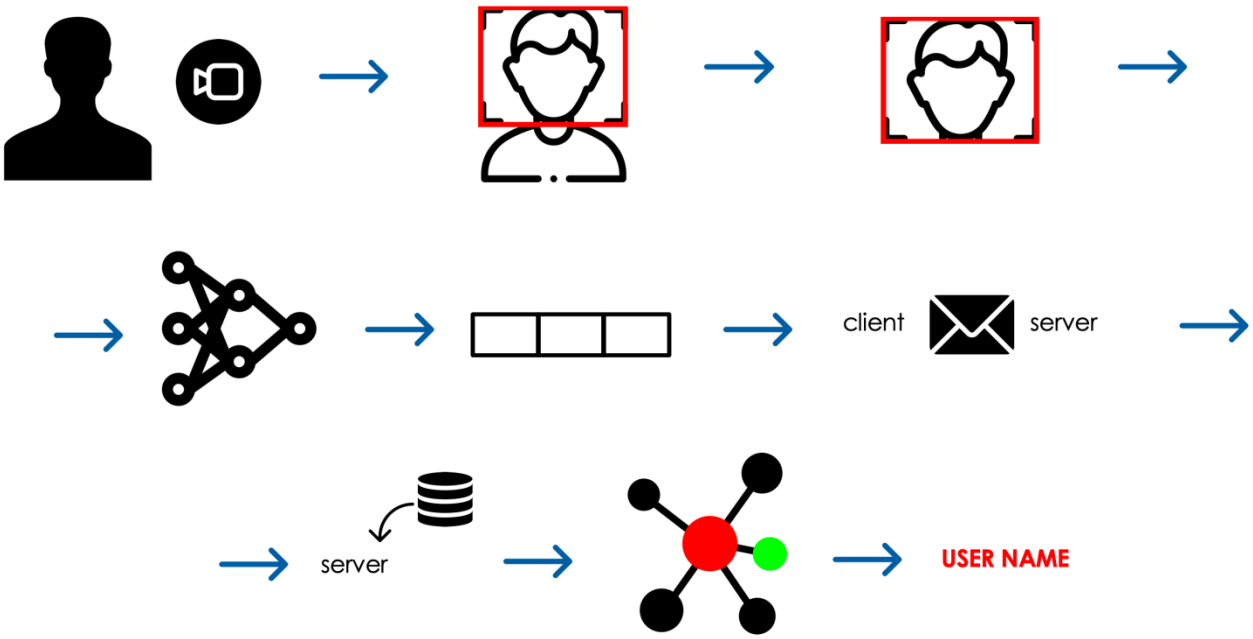
3. APPLICATION FUNCTIONALITIES

At the start of the application the system offers the following functionalities to the users:

- **Registration:** users can register themselves with first name, last name and facial identity. If a registered user tries to register again, the system will inform him/her that the user has already registered;
- **Login**: the users can log into the application through his/her face identity and access the movie playback; this functionality can be valid only if the users has already registered, otherwise the system will inform that the login has failed. After logging in, the users can choose which application between film, music and news they are interested in executing (our prototype is limited to only the films);
- **Delete**: users can delete their profiles, after confirming its identity through face identification;
- **Exit:** users can exit from the application.

4. FACE RECOGNITION

Both during the registration phase and login phase a face recognition will be carried out: in the first case it will serve the user to register his/her facial identity, and in the second case it will serve him to be recognized by the system; in both cases the user must stand in front of the Raspberry pi camera for 10 seconds.
Face recognition is done through the use of the VGGFace2 pretrained model based on Resnet-50 network that is a CNN that is 50 layers deep.



First through the *VideoCapture()* method from the **OpenCV** library, we get a video capture object for the camera, used later to recall the *read()* function to capture the video frame. Since the camera takes frames (we capture more than one frame for consistency reason), which may not exclusively contain the user's face, a detector is used to extract the ROI (in this case, the face) from all the frames, through the *getImageROI()* function.
After that, we extract through the function *extract_average_features()* all the features related to the ROIs previously saved; from this features we then obtain an average, normalized descriptor.

```
import cv2
import os
import glob
import numpy as np
from PIL import Image
from IPython.display import clear_output, display
from imutils.video import VideoStream, FileVideoStream
from tqdm.notebook import tqdm
import time



def get_average_features(function_type):

    vid = cv2.VideoCapture(0)
    if not vid.isOpened():
        print("Error opening video")
    count=0
    all_features = []
    frame = ""
    directory = ""
    detected_faces = 0
    if(function_type=='registration'):
        directory = './registration'
    else:
        directory = './login'
    while(True):
        # Capture the video frame
        # by frame
        ret, frame = vid.read()
        detected_faces = detect(frame)
        if (detected_faces == []):
            vid.release()
            cv2.destroyAllWindows()
            return frame,[],[]
        for detected_face in detected_faces:
            imageROI = getImageROI(frame , detected_face)
            cv2.imwrite(directory+'/img'+str(count)+'.jpg', imageROI,
[int(cv2.IMWRITE_JPEG_QUALITY), 85])
        count +=1

        if cv2.waitKey(1) & 0xFF == ord('q') or count==10:
            break
        time.sleep(1)
    # After the loop release the cap object
    vid.release()
    # Destroy all the windows
    cv2.destroyAllWindows()
    average_desc = extract_average_features(directory)
    return frame,detected_faces,average_desc
```

```
def extract_average_features(directory_path):
    average_descs = []
    img_paths = glob.glob(directory_path + "/*.jpg")
    descs = [fe.extract(cv2.imread(path), FEAT_LAYER, normalize=True) for path
in tqdm(img_paths)]
    desc_dim = len(descs[0])
    sum_vector = np.zeros(desc_dim)
    count_vector = np.zeros(desc_dim)
    average_vector = np.zeros(desc_dim)
    for desc in descs:
        for i in range(0,desc_dim):
            sum_vector[i] += desc[i]
            count_vector[i] += 1
    for i in range(0,desc_dim):
        average_vector[i] = sum_vector[i] / count_vector[i]
    average_descs.append(average_vector/(np.linalg.norm(average_vector)))
    return np.array(average_descs)
```

To verify if an user is already present in the system we use a 1NN classifier to search for him/her.
After the creation of the *OneNNClassifier()* in which are contained all the ids and descriptors of users already registered, the *predict()* function, having as arguments the average_feat of the user we are interested in searching, is called.
In the *predict()* function the dot product between the query (the average_feat) and each descriptor is computed; if the closest distance between the searched features and any descriptor in the system is < 0.80 we assume that nobody already registered corresponds to the search, otherwise the user is already registered and has been recognized.

```
classifier = on.OneNNClassifier(total_ids, total_descs)
closest = classifier.predict(average_feat,username)
```

```
import numpy as np


def predict(self, queryF,username):
        if (self.descs == []):
            return None
        max_score = 0.0
        max_id = ''
        for i in range(0,len(self.descs)):
            if (self.ids[i] == username):
                return None
            score = np.dot(self.descs[i],queryF)
            if (score > max_score):
                max_score = score
                max_id = self.ids[i]
        if(max_score < 0.80):
            return None
        else:
            return max_id
```

4.1 COMPUTATION TIME COMPARISON

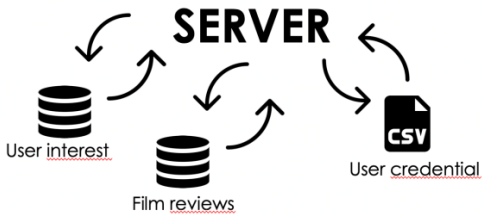
The table below shows the times taken to perform feature extraction locally and, on the raspberry, respectively.
To calculate the average time in both cases 10 tests were carried out; the time of each tries and the average time are the following:

TEST	LOCAL TIME(s)	RASPBERRY TIME(s)
1	11.2	66.3
2	11.1	63.4
3	11.2	65.5
4	11.2	67.3
5	11.5	65.3
6	11.1	68.2
7	11.1	66.1
8	10.9	66.7
9	11.2	66.5
10	11.1	67.2

The average time to do the feature extraction **locally** is: **11.2 s**
The average time to do the feature extraction on **raspberry** is: **66.3 s**

5. SERVER

The server is the component of our prototype which is in charge of elaborating the client requests; in order to do so the server has to access the user interest db, implemented as a mongoDB, the film review db, which is a flat file, and a csv, containing all the credentials of the users (the users credential are left stored into a dedicated csv since, in possible future development of the prototype, the idea is to put such file in an Hadoop framework and, using the information contained on it, execute map-reduce job whose aim is to identify the user trying to register/login; this implementation is sure to have better performance than the actual one, however we don't have the machines to implement it)



The server has been implemented using the **socketserver** library in the following way:

```
import socketserver
import socket


def server_program():
    HOST = socket.gethostbyname(socket.gethostname())
    print("host",HOST)
    PORT = 5000
    server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)
    server.serve_forever()
```

First, we get the host ip of the pc on which the program is running and then, by specifying a dedicated port (in our case,5000) we create a TCPServer that will *serve_forever()* all the tcp connection incoming from the client using the class **MyTCPHandler**, which is purposely developed to answer all the client requests.

```
import pickle
from re import search
import socket
import sys
import numpy as np
import time
from csvConn import csvConn
import socketserver
from mongoConn import Connection_DB
from film_sugg import Film_Suggestor
from utilities import
get_from_string_to_original,get_users_from_string,get_string_from_array

class MyTCPHandler(socketserver.BaseRequestHandler):
```

```
    def get_entire_message(self,connection):
        message = ""
        counter = 0
        while True:
          message_piece = connection.recv(2000)
          if not message_piece:
              break
          else:
              message_part = message_piece.decode()
              message_part_splitted = message_part.split('_fine_messaggio')
              if(len(message_part_splitted)>1):
                  message += message_part_splitted[0]

                  break
              else:
                  message += message_part_splitted[0]
        return message.split('new_part_')

    def handle(self):
        received_message = self.get_entire_message(self.request)
        print(received_message[1])
        if(received_message[1]=='login'):
            string_array = received_message[2]
            features = get_from_string_to_original(string_array.split('/'))
            username = csvConn().searchUser(features,'')
            print(username)
            if (username == "none"):
              self.request.sendall(('new_part_fallimento_fine_messaggio').encode())
            else:
                self.request.sendall(('new_part_' + username + '_fine_messaggio').encode())

        elif(received_message[1]=='registration'):
            username = received_message[2]
            string_array = received_message[3]
            features = get_from_string_to_original(string_array.split('/'))
            username_old = csvConn().searchUser(features,username)
            if (username_old == 'none'):
                csvConn().regUser(username,features)
                Connection_DB().insertUser(username)
                self.request.sendall(('new_part_successo_fine_messaggio').encode())
            else:
                self.request.sendall(('new_part_fallimento_fine_messaggio').encode())

        elif(received_message[1]=='delete'):
            string_array = received_message[2]
            features = get_from_string_to_original(string_array.split('/'))
            username = csvConn().searchUser(features,'')
            print(username)
            if (username == "none"):
                self.request.sendall(('new_part_fallimento_fine_messaggio').encode())
            else:
                csvConn().deleteUser(username)
                Connection_DB().deleteUser(username)
                self.request.sendall(('new_part_successo_fine_messaggio').encode())

        else:
            users = get_users_from_string(received_message[2])
            cd = Connection_DB()
            interest = [cd.get_random_best_common_interests(users,'film_interest')]
            cd.close()
            fs = Film_Suggestor()
            films = fs.provide_suggestions(interest)
            string_suggestions = get_string_from_array(films)
            self.request.sendall(('new_part_'+string_suggestions+'_fine_messaggio').encode())
```

The class **MyTCPHandle**r has two function:

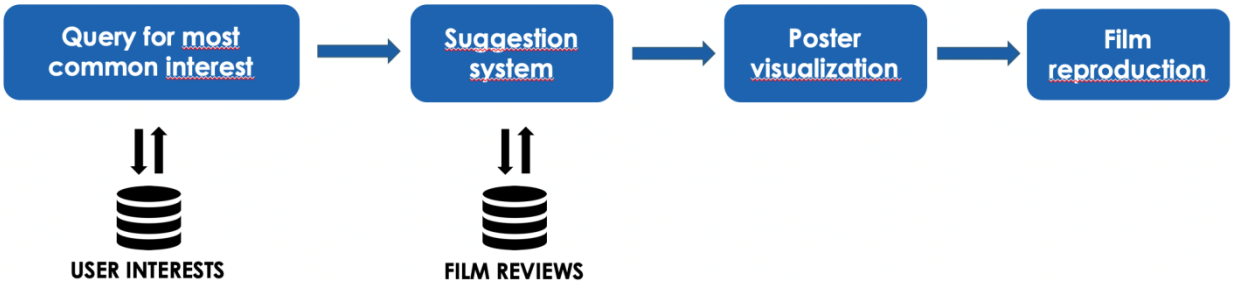
- **get_entire_message(self,connection)**: the function is called passing as argument the tcp connection automatically created between the client and server. The purpose is to reconstruct the message that the client sent through the connection: since it's impossible to know beforehand which will be the dimension of the

encoded message incoming from the client, we had to specify a standard number of bytes (in our case 2000) in which will the message will be split (as the library is implemented it's necessary to specify such dimension, and what happens is that if the message is bigger than such dimension, it will be automatically splitted into multiple pieces); for each split we apply the *decode()* function which allows to obtain the text in clear, and such text is incrementally collected until we have reconstructed the original message sent by the client.

- **handle(self)**: this function is responsible for handling the user requests. Initially, it will call the *get_entire_message()* to get the client message and, once the message is returned, it checks which one it was the command issued by the client and answer according to it, specifically:
  - if the request is a *'registration'* request, part of the decoded message also contains the user_credential and facial features. The server will access to a csv, locally stored, containing the information of all the users registered up until this moment, and check if the user requesting to register it's already present or not; if it's the first case, the server will send a failure message to the client, otherwise it will add the user credentials to the csv, add a new document into the mongoDB storing for each user its detected interests (in this case, since we don't know anything about the user, we assign him/her random interests) and send a success message to the client.
  - if the request is a *'login'* request, part of the decoded message also contains the facial features. The server will access to a csv, locally stored, containing the information of all of all the users registered up until this moment, and check if the user requesting to register it's already present or not; if it's the first case, the server will send a success message also containing the name and surname of the now logged user, otherwise it will send a failure message.
  - if the request is a *'delete'* request, part of the decoded message also contains the facial features. The server will access to a csv, locally stored, containing the information of all the users registered up until this moment, and check if the user requesting to register it's already present or not; if it's the first case, the server will remove the user from both the csv and mongoDB and send a success message, otherwise it will send a failure message.
  - if the request is a *'film_request'* request, part of the decoded message also contains the credentials of the users which desire to see a film. The server will read from the mongoDB all the users interests and select one of those most common; using such interest as base, it will call the *provide_suggestions()* function which will select (with modalities explained later in the documentation) the films candidates to suggest; finally, these film candidates are sent with the reply message from the server to the client.

6.FILM TRAILER REPRODUCTION

The film trailer reproduction is the core of our prototype since its final aim is to, from the users already logged in, first display the posters of those films selected by the system between the ones that could be of interest for the users, and then reproduce the trailer of the film which was the choice.



In order to implement this functionality, we have adopted the following paradigm:

- send a request for film suggestion to the server
- access the mongoDB and get the interests of all the users logged in interested in watching a film; then these interests are intersected in order to identify which isthe most common (if there are more than one, it is randomly selected between the most common)
- identify 15 film candidates through a selector which requires as the only parameter the common interest;
- send the film candidates to the client;
- download and display the posters of the film candidates;
- reproduce the trailer of the film chosen by the users.

6.1 ACCESS TO MONGODB AND INTEREST

The first thing our system has to do is to identify an interest which is suitable for most of the users already logged in.
In our to do so, we have previously created a database on mongo structured in the following way:

```
{ "_id" : ObjectId("61b5d8d4507ac2ed6383b2b3"), "name" : "ciccio_pasticcio", "film_interest" : [ "War","Musical", "Romance" ], "music_interest" : [ "Dance", "Classical", "Jazz" ], "news_interest" : [ "Fashion","Medical", "Online" ] }
{ "_id" : ObjectId("61b5da7ef60d90c8b1121f8e"), "name" : "fabiana_ritorti", "film_interest" : ["Documentary", "War", "Mystery" ], "music_interest" : [ "Disney", "Reggae", "Opera" ], "news_interest" : ["Online", "Sport", "Western" ] }
```

Each entry of the db corresponds to a single registered user and, in addition to its username credential, it contains its actual identified interests regarding film, music and

news (when an user is freshly registered we will assign him/her three random interests, but over time, through possible future development of the project and the integration of the mood detection, those interests will be dynamically modified in order to better represent the actual real interests of the users). The server, which has also received through the *film_request* message the credentials of the users which desire to see a film together, will execute the following function:

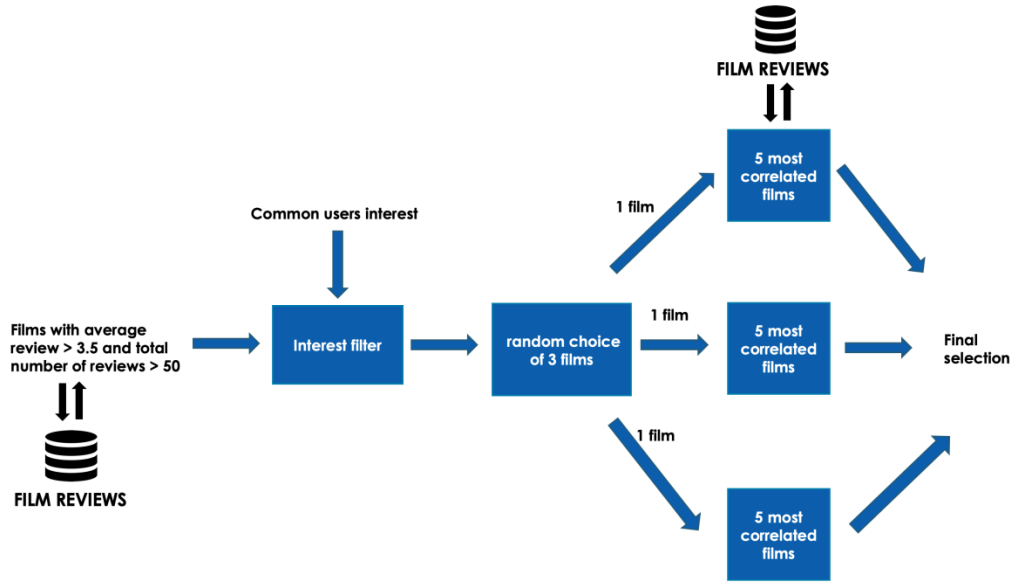
```
import pymongo
from datetime import datetime
from getpass import getpass
from pprint import pprint
import numpy as np
import os
import re
import random


def get_random_best_common_interests(self,names,interest_type):
        string = self.interest_string(interest_type)
        interests =
list(self.db.Data.aggregate([{"$match":{"name":{"$in":names}}},{"$unwind":string
},{"$group":{"_id":string,"count":{"$sum":1}}},{"$sort":{"count":-1}}]))
        if not interests:
            return self.get_random_interest(interest_type)
        else:
            max_pop = interests[0]["count"]
            selected_interests = []
            for interest in interests:
                if(interest["count"]<max_pop):
                    break
                else:
                    selected_interests.append(interest["_id"])
                return random.choice(selected_interests)
```

Inside this function the server will access to the previously mentioned mongoDB and perform a query using as arguments the usernames and the type of interests we want to identify (film, music, news). The query returns a list of couple formed by the interest and its count, namely the number of users to which the interest belongs, ordered by the count. If for whatever reason no interest has been identified the function returns one of the random interests of the specified type, otherwise it will return one of those interests which have the max count, which translated to returning one of those interests that belong to the most users as possible.

6.2 SUGGESTION SYSTEM

The suggestion system goal is to identify films that could be of interest for the users, using as parameter for the search the most common interest previously identified.



The adopted paradigm, implemented in the following function

```
import pandas as pd
import chardet
import random

def provide_suggestions(self,interests):
      candidates = self.cold_start(interests)
      titles = random.sample(list(candidates['title']),3)
      suggestion = []
      for title_name in titles:
        suggestion_of_film = list(self.predict_movies(title_name).index.values)
        for single_suggestion in suggestion_of_film:
          suggestion.append(self.reconstruct_title(single_suggestion))
      mylist = list(dict.fromkeys(suggestion))
      return mylist
```

consists in 3 steps:

- identification, through the *cold_start()* function, of those films in the database which have as genre the one passed as argument to the function, a total number of reviews gathered of at least 50, and an average review vote of 3.5;

```
import pandas as pd

def cold_start(self,interests):
      candidates = self.movies
      for interest in interests:
        candidates = candidates.loc[candidates[interest]>0]
      candidates=pd.merge(self.ratings,candidates,on="title")
      candidates.sort_values(by='rating', ascending=False)
      candidates = candidates[candidates['rating']>3.5]
      return candidates[candidates['number of
ratings']>50].sort_values('rating',ascending=False)
```

- random sampling to select three films from the ones returned from the previous function, and on each one of them it will be called the function in the next step;

- identification, through the *predict_movies()* function, of the 5 most correlated films to the film name passed as argument to the function. The correlation of one film to another is computed through the use of a correlation matrix obtained by the intersecting the reviews of the users collected in the db; the idea behind the use of the correlation is that if the majority of those users which have positively reviewed the film passed as argument to the function have also positively reviewed another film ,their correlation will be very high, and is therefore very likely that the correlated film could be of interest to the users. In addition, a partial filter to the possible proposals is applied by selecting only those films which have an average rating of 3.0 and a number of total reviews > 50, thus guarantying a certain standard quality for the selection.

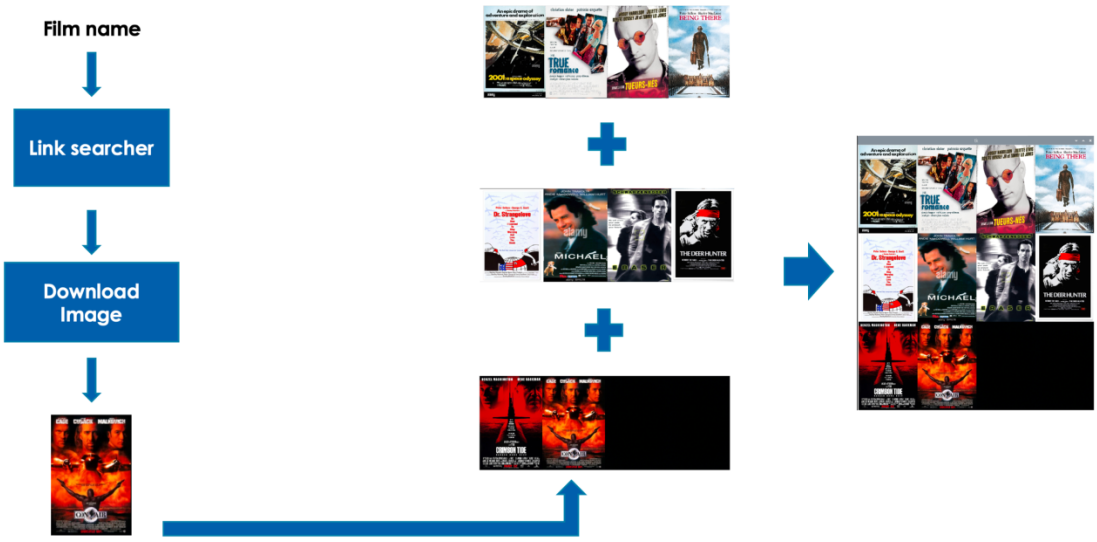
```
import pandas as pd


def predict_movies(self,movie_name):
        movie_user_ratings=self.moviematrix[movie_name]
        similar_to_movie=self.moviematrix.corrwith(movie_user_ratings)
        corr_movie=pd.DataFrame(similar_to_movie,columns=['correlation'])
        corr_movie.dropna(inplace=True)
        corr_movie=pd.merge(self.ratings,corr_movie,on="title")
        corr_movie=corr_movie[corr_movie['rating']>3.0]
        predictions=corr_movie[corr_movie['number of
ratings']>50].sort_values('correlation',ascending=False)
        return predictions.head(5)
```

We could have limited to just select 15 random films from the ones belonging to the requested genre satisfying the specified quality criteria, however this paradigm has the defect that the number of possible selections is possibly very limited, therefore what could happen is the fact that the same films could be proposed to the users multiple time in a short period. In addition, by following our paradigm we offer to the users the possibility to see films which could belong to different genre to the one specified but of which we are confident it will be of their interest anyway; (it is equivalent to a friend saying that, since I liked 'Batman' of Nolan I should also see 'Inception' because I will probably like it even though the two films have nothing in common).

6.3 POSTER VISUALIZATION

The client will receive the suggestions from the server and, at this point, has to visually display the posters of the choices in order to allow to the users to express, through the microphone, the desire to see a specific film.



The poster visualization has been implemented in the following function:

```
from simple_image_download import simple_image_download as simp #pip install
simple_image_download
import requests
import cv2
import matplotlib
import tkinter
from PIL import ImageTk, Image
import numpy as np
from IPython.display import display
import speech_recognition as sr
import numpy
from utilities import listen_phrase,get_num
from playsound import playsound
from utilities import control_number
import time
import os,glob

def show_movie_posters(self,film_names,r,m):
        self.clean_directory()
        response = simp.simple_image_download
        links = []
        first_correct_films = []
        for film_name in film_names:
            film_urls = response().urls(('movie poster ' + film_name), 10)
            links.append(self.get_correct_film_link(film_urls))
        correctLinks_counter = 0
        film_counter = 0
        for url in links:
            if(url!="Not found"):
                try:
                    img_data = requests.get(url).content
```

```
                    with
open('./movie_posters/image_'+str(correctLinks_counter)+'.jpg', 'wb') as
handler:
                        handler.write(img_data)
                        first_correct_films.append(film_names[film_counter])
                        correctLinks_counter+=1
                except:
                    pass
            film_counter+=1
        first_num_films = len(first_correct_films)
        num_films,correct_films =
self.delete_wrong_images(first_num_films,first_correct_films)
        posters = self.vertical_stack(num_films)
        root = tkinter.Tk()
        img = ImageTk.PhotoImage(Image.fromarray(cv2.cvtColor(posters,
cv2.COLOR_BGR2RGB)))
        panel = tkinter.Label(root, image = img)
        panel.pack(side = "bottom", fill = "both",
                   expand = "yes")
        root.update()
        playsound('./audio/choose_film.mp3')
        numero_scelto = control_number(r,m)-1
        root.destroy()
        return correct_films[numero_scelto]
```

in which the following paradigm has been adopted:

- through the use of the library **simple_image_download** the system is able to retrieve some links from which it could be possible to download the poster images;

```
film_urls = response().urls(('movie poster ' + film_name), 10)
```

- for each link the system tries to obtain the poster image by downloading the content of the request sent to the specified url;

```
import requests

img_data = requests.get(url).content
with open('./movie_posters/image_'+str(correctLinks_counter)+'.jpg', 'wb') as
handler:
      handler.write(img_data)
```

- the downloaded images are checked in order to control if each one of them is displayable; if it's not the case, the corrupted images are removed;
- inside the *vertical stack()* function the available posters are divided into blocks; each block of images is used in *horizonal stack()* function, which takes care of tiling the images and to fill the last row of a number of black images in order to obtain an uniform number of pics per row. Finally, the rows are incrementally vertically stacked in order to create the final collage that will be displayed to the users.

```
import numpy

def horizontal_stack(self,images):
        dim = len(images)
        horizontal_images = images[0]
```

```
        if(dim==1):
            return horizontal_images
        else:
            for i in range(1,dim):
                horizontal_images = numpy.hstack((horizontal_images,images[i]))
            return horizontal_images
```

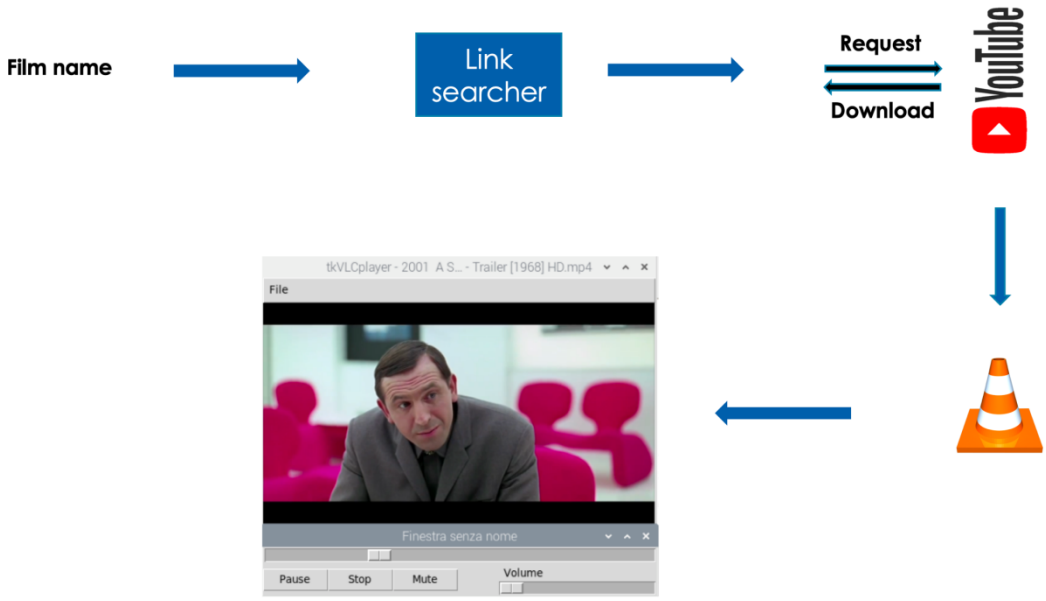
```
import cv2


def vertical_stack(self,num_films):
        images = []
        for j in range(0,num_films):
            image = cv2.imread('./movie_posters/image_'+str(j)+'.jpg')
            dsize = (200, 300)
            resized_image = cv2.resize(image, dsize)
            images.append(resized_image)
        remaining = num_films%4
        if(remaining==0):
            num_rows = num_films//4
        else:
            num_rows = (num_films//4)+1
        if(num_rows==1):
            first_row = self.horizontal_stack(images[0:num_films])
            return first_row
        else:
            missing_images = ((num_rows-1)*4)-num_films
            for j in range(num_films,num_rows*4):
                image = cv2.imread('./movie_posters/black.jpg')
                image = resized_image = cv2.resize(image, dsize)
                images.append(resized_image)
            vertical_images = self.horizontal_stack(images[0:4])
            for i in range(1,num_rows):
                low_extr = (i*4)
                high_extr = (i*4)+4
                horizontal_images =
self.horizontal_stack(images[low_extr:high_extr])
                vertical_images =
numpy.vstack((vertical_images,horizontal_images))
            return vertical_images
```

6.4 SHOW TRAILER

After the system has obtained the film choice the users are interested in watching, expressed through the microphone, the movie should be reproduced through a streaming created between the client and a server which has access to all the movies available on the platform.

In our case, however, we don’t have any multimedia content, therefore as an alternative we decided to download the trailer of the film from YouTube and display it.



In order to do that, the following code is executed:

```
import vlc
# import standard libraries
import sys
import os
import urllib.request
import pytube
import re

    root = Tk.Tk()
    player = Player(root, video=_video)
    player.Download_Trailer(name_film)
    video_path = ('./film/'+ (os.listdir('./film/')[0]))
    player._Play(video_path)
    root.protocol("WM_DELETE_WINDOW", player.OnClose)
    root.mainloop()
```

First, we create an instance of a wrapper, the class **Player,** which is used to contain the actual player through which the film will be reproduced.
It was necessary to implement such class because in the environment on the dispositive in which the project actually runs, the libraries normally used are not yet integrated, therefore their normal reproduction either not work or are not responsive to input from the system (for example, closing the reproduction window only ends the visual display of the film, however the process isn't killed and consequentially the audio continues until the trailer finishes).
Through the **Player** instance the trailer of the film is download, using its internal function *Download_Trailer()* which has as argument the name of the film.

```
import vlc
# import standard libraries
import sys
import os
import urllib.request
import pytube
import re


def Download_Trailer(self,name_film):
        try:
            os.remove('./film/'+ (os.listdir('./film/')[0]))
        except:
            pass
        search_url =
"https://www.youtube.com/results?search_query=english+cinema+trailer+"+name_film
        html = urllib.request.urlopen(search_url)
        video_ids = re.findall(r"watch\?v=(\S{11})",html.read().decode())
        first_trailer = "https://www.youtube.com/watch?v="+video_ids[0]
        yt = pytube.YouTube(first_trailer)
        yt.streams.filter(progressive=True,
file_extension='mp4').order_by('resolution')[-1].download('./film/')
```

The *Download trailer* function works adopting the following schema:

- Through some keywords added to a standard paradigm that the YouTube links usually follows, we create a YouTube search URL;
- A request, through the **urllib** library, is sent to the created URL
- The answer to our requests is an html containing all the possible trailer link available on YouTube, consequentially, the answer is first read in its html form and then decoded;
- Through the **pytube** library a request of download is sent to YouTube, using as argument the first of the possible link we have collected in the previous step and additional parameters that guarantees the quality of the downloaded video.

The trailer, now locally stored, is finally reproduced inside the wrapper until either it finishes or the x button, which has a dedicated function to end the reproduction, is clicked.

References

***RESNET 50 for face classification**:*

- Download link: https://www.kaggle.com/keras/resnet50
- Article for the use: https://towardsdatascience.com/beginners-guide-on-image-classification-vgg-19-resnet-50-and-inceptionresnetv2-with-tensorflow-4909c6478941

***RESNET 10_300x300 for ROI detection**:*

- Download link: https://github.com/opencv/opencv_3rdparty/raw/dnn_samples_face_detector_20170830/res10_300x300_ssd_iter_140000.caffemodel and https://github.com/opencv/opencv/raw/3.4.0/samples/dnn/face_detector/deploy.prototxt
- Article for the use: https://sefiks.com/2020/08/25/deep-face-detection-with-opencv-in-python/

SUGGESTION SYSTEM:

- DB Film Reviews download link: https://www.kaggle.com/arushikhokharr/a-beginner-s-guide-to-recommendation-systems/data
- Article for the use: https://www.kaggle.com/arushikhokharr/a-beginner-s-guide-to-recommendation-systems

PYTHON LIBRARIES:

- Speech Recognition: https://pypi.org/project/SpeechRecognition/
- Playsound: https://pypi.org/project/playsound/
- Simple_Image_Download: https://pypi.org/project/simple-image-download/
- Pytube: https://pypi.org/project/pytube/
- Tkinter: https://pypi.org/project/tkinter-page/
- URLLib: https://docs.python.org/3/library/urllib.html
- PyMongo: https://pypi.org/project/pymongo/
- Vlc: https://pypi.org/project/python-vlc/