**Guide d’utilisation du programme de reconnaissance faciale**



Langage utilisé : Python 3.8

Bibliothèque utilisée : OpenCV 4.4.0

IDE utilisé : PyCharm Community Edition 2020.2.1

Fonctionne sur Linux, Raspberry, et Windows

Boudissa Nourredine

**Sommaire**

Arborescence des fichiers……………………………………………………………………………………..3

Utilité du programme………………………………………………………………………………………..….3

Tutoriel d’utilisation du programme……………………………………………………………………..4

Main.py ………………………………………………………………………………………………………………..4

Detection.py ……………………………………………………………………………………………………….5

Face\_detection() ………………………………………………………………………………………………….5

Skin\_detection() …………………………………………………………………………………………………..8

Face\_dataset.py…………………………………………………………………………………………………..11

Face\_dataset()……………………………………………………………………………………………………..11

Face\_traning()……………………………………………………………………………………………………..14

Get\_face\_from\_image.py ……………………………………………………………………………………17

Train\_new\_face() ……………………………………………………………………………………………..…18

Getimages() ………………………………………………………………………………………………………..18

Delete\_user() ……………………………………………………………………………………………………..19

Recognition.py ……………………………………………………………………………………………………19

Face\_recognition\_no\_tracking() …………………………………………………………………………20

Face\_recognition() ………………………………………………………………………………………….....25

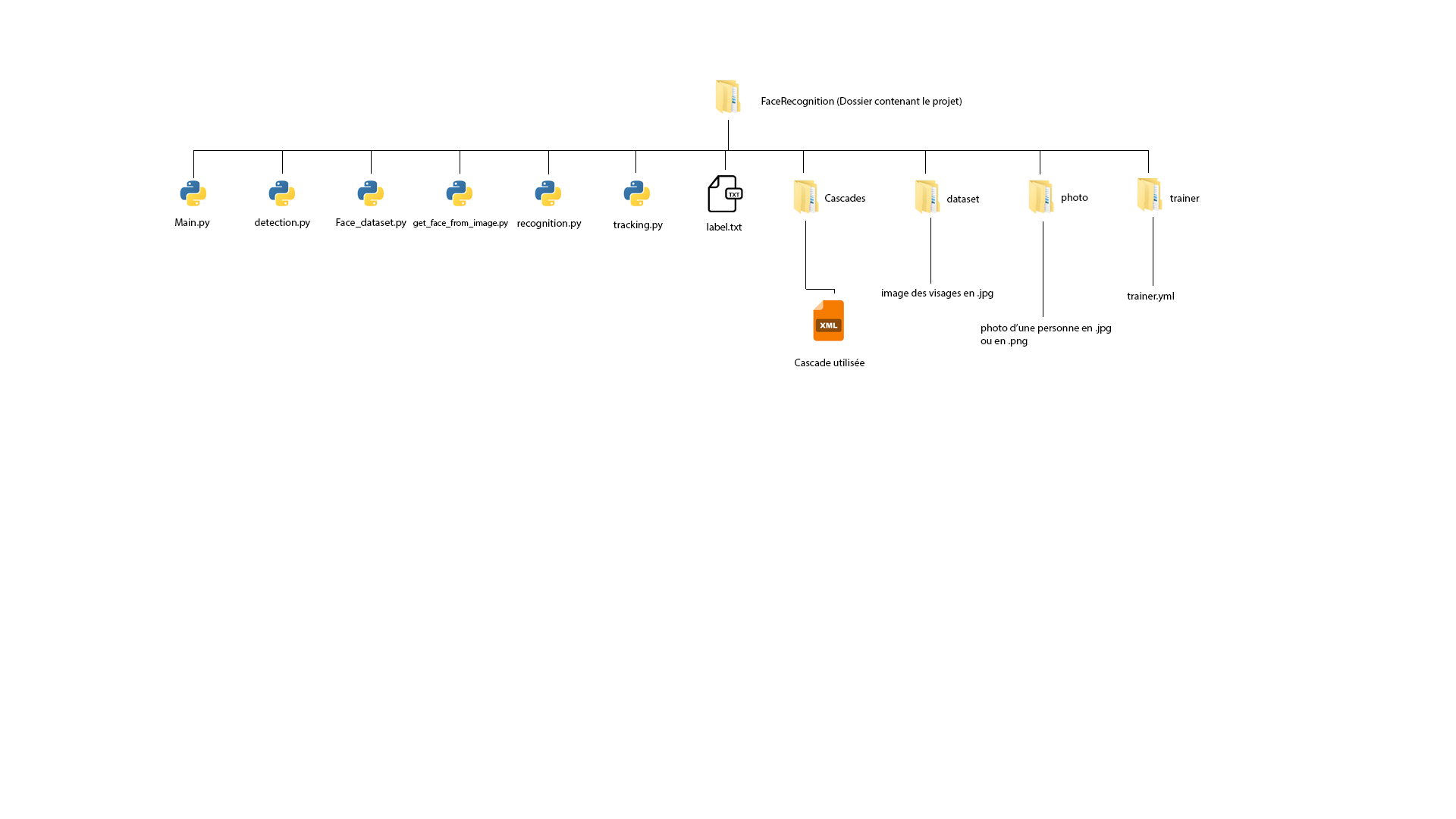
Tracking.py.…………………………………………………………………………………………………………26

Tracking\_zone() ………………………………………………………………………………………………….27

Tracking().…………………………………………………………………………………………………………..30

Bibliographie ……………………………………………………………………………………………………..30

**Arborescence des fichiers**



**Utilité du programme**

Ce programme peut effectuer 8 actions :

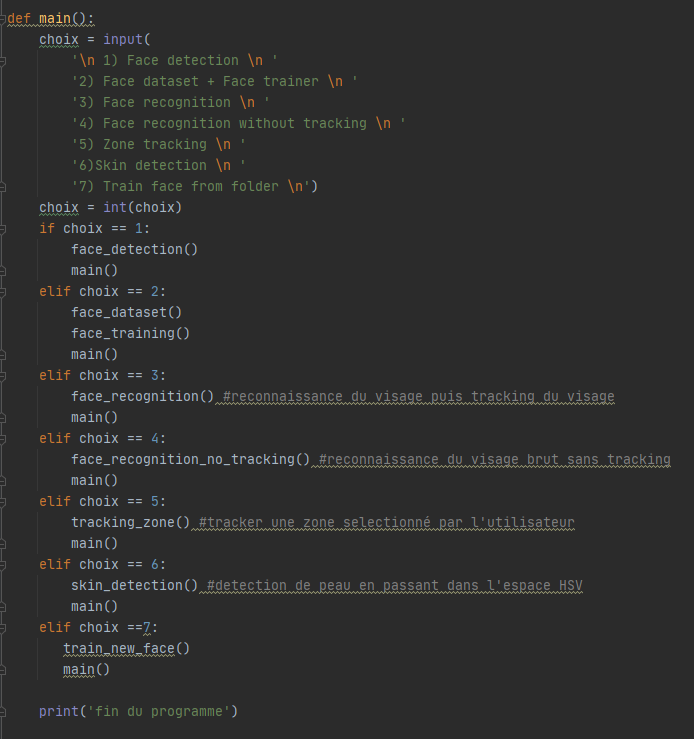
* Détecter les visages d’un flux vidéo
* Enregistrer un visage dans une base de donnée et en tirer les données nécessaires à sa reconnaissance à partir d’un flux vidéo
* Enregistrer un visage à partir de plusieurs photos de lui et en tirer les données nécessaires à sa reconnaissance
* Reconnaître en visage (ayant été enregistré auparavant)
* Détecter la peau humaine
* Détecter un corps humain
* Suivre une zone sélectionnée par l’utilisateur
* Reconnaître un visage puis suivre la personne grâce à un algorithme de tracking

**Tutoriel d’utilisation du programme**

Avant d’utiliser ce programme, il faut avoir installé OpenCV 4.4.0, numpy et imutils.

**Main.py**

Le fichier Main.py permet de lancer le programme, il renvoie vers des fonctions écrites dans les autres fichiers .py  (tracking.py, recognition.py, detection.py, get\_face\_from\_image.py et Face\_dataset.py)



Une fois le programme lancé, on entre la valeur correspondant à l’action que l’on souhaite effectuer. Il y a alors 9 choix correspondants à 8 actions différentes.

**Detection.py**

Ce fichier contient toutes les fonctions effectuant une détection. On retrouve alors 3 fonctions différentes :

* **Face\_detection()**, qui effectue une détection brut de tous les visages dans un flux vidéo.
* **Skin\_detection\_image(Mat image),** qui détecte la peau humaine sur une image entrée en argument
* **Skin\_detection(),** qui effectue une détection de peau humaine sur un flux vidéo en temps réel.

**Face\_detection()**

On commence par sélectionner la cascade que l’on va utiliser pour détecter les visages. Les fichiers utilisés sont sous le format .xml . La cascade utilisé dans l’exemple permet de détecter les visages de face :



On sélectionne ensuite le flux vidéo sur lequel la détection sera faite :

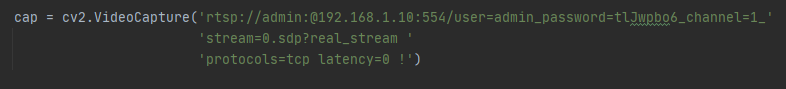


On note que lorsqu’on entre « 0 » comme argument dans **VideoCapture**, alors le flux vidéo choisit sera celui par défaut utilisé comme caméra de l’ordinateur. On peut à la place utiliser le flux vidéo d’une caméra IP en entrant l’adresse avec laquelle on y accède (cette information est trouvable soit dans la documentation de la caméra, ou sur internet)

Exemple de format pour accéder à la caméra IP :

rtsp://[admin:@192.168.1.10:554/user=admin\_password=tlJwpbo6\_channel=1\_stream=0.sdp?real\_stream](http://admin@192.168.1.10:554/user=admin_password=tlJwpbo6_channel=1_stream=0.sdp?real_stream) protocols=tcp latency=0 !

On entre donc cette ligne là pour y accéder :



On effectue ensuite une boucle dans laquelle on commence par lire la dernière image (frame), du flux vidéo :

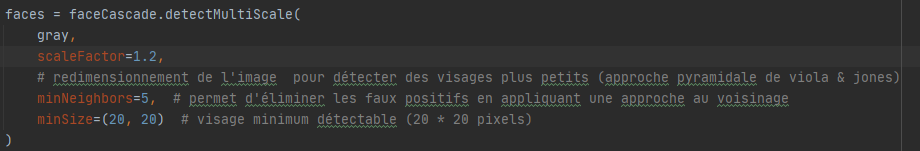


L’image est stocké dans ‘img’

La détection de visage par la méthode de Viola & Jones (en cascade), ne peut s’effectuer que sur une image en niveau de gris. Si notre flux vidéo est initialement en couleur, alors on doit effectuer une conversion de l’image en niveau de gris :



Cette ligne de commande permet d’obtenir une image ‘gray’, qui est la conversion de ‘img’ de l’espace BGR (Blue Green Red), en GRAY (niveau de gris)

Il ne reste plus qu’à détecter sur cette image les visages et de stocker dans une liste les cordonnées de chaque visage détecté :

On détecte ici les visages avec ‘detectMultiScale’ en utilisant la cascade choisit précédemment ‘faceCascade’

On entre en argument en premier l’image sur laquelle s’effectue la détection, c’est donc ici ‘gray’.

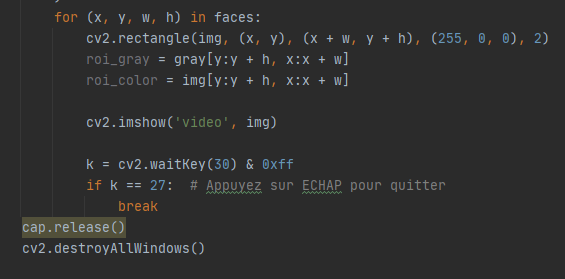
**ScaleFactor** correspond au redimensionnement de l’image que l’on effectue, afin de détecter des visages qui était bien trop petit pour y être détecté comme tel.

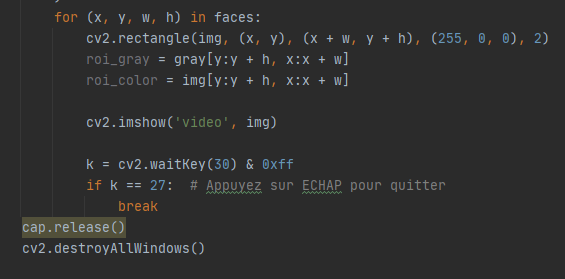
**MinNeighbors** permet d’élminer les faux positifs en appliquant une approche au voisinage. Lorsque l’algorithme est utilisé, le même visage est détecté plusieurs fois avec des cordonnées légèrement différentes. L’algorithme interprète ensuite cela comme un seul visage pour ne pas saturer la détection. Plus **MinNeighbors** sera grand, plus il faudra que l’algorithme détecte le même visage plusieurs fois pour le considérer comme un visage (ici 5 fois). Plus cette valeur est grande, plus il y a donc aussi de chance d’avoir des faux négatifs. Par défaut, on initialise cette valeur à 5.

**MinSize** correspond à la taille en pixel minimale d’un visage détectable. Ici, un visage inférieur à 20 pixels x 20 pixels ne sera pas pris en compte.

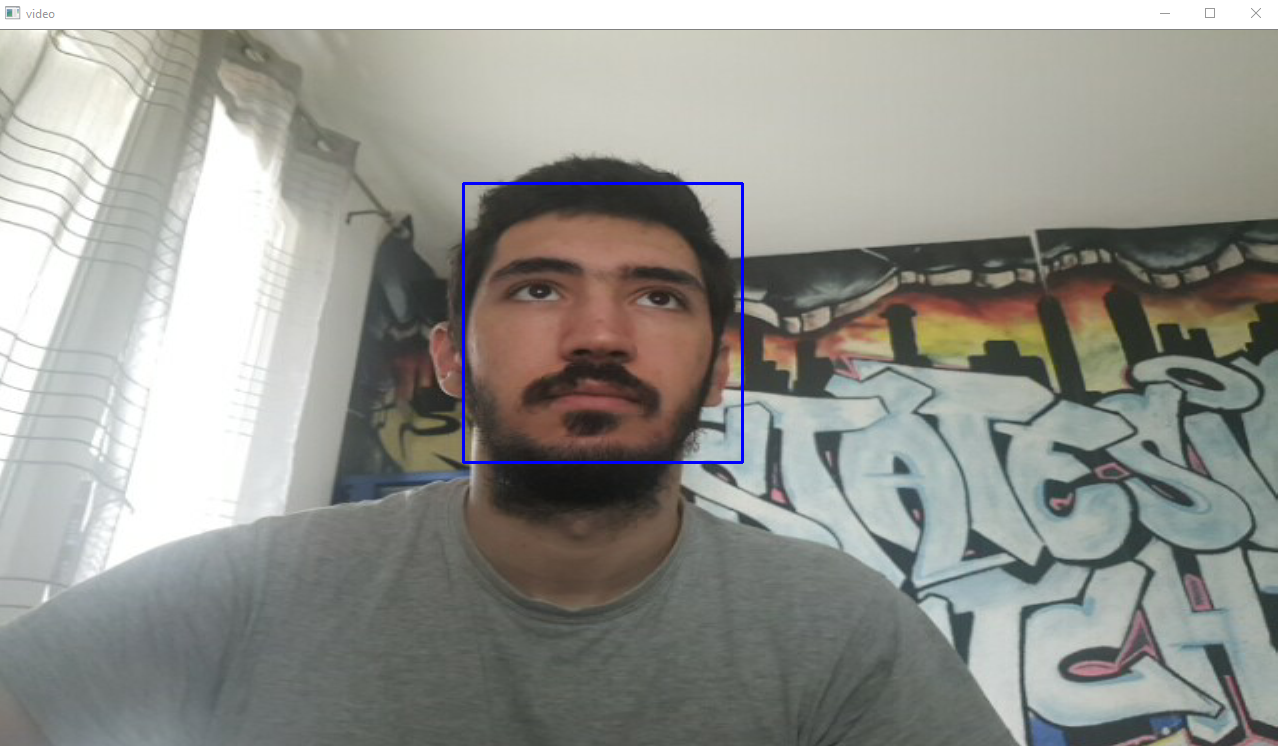
Enfin dans la boucle **While**, après la détection, on effectue une derrière boucle sur le nombre de visage détecté, qui va permettre de tracer sur notre région d’intérêt (les visages), un cadre.

On affiche enfin l’image à la fin de cette boucle.



Ensuite, on peut demander l’appui d’une touche pour mettre fin à la boucle et ensuite arrêter le flux vidéo et fermer la fenêtre sur laquelle le résultat s’affichait :

Résultat de l’algorithme :



Critique de l’algorithme :

La détection des visages s’effectue de façon assez simple et robuste. Les visages sont toujours détectés lorsque la luminosité le permet. En effet, en trop basse luminosité, aucun visage n’est détectable.

Un autre problème vient se poser. L’algorithme peut détecter parfois des visages là ou il n’y en a pas (décor, objet ect…)

Ce problème pourra être comblé par une détection de peau.

**Skin\_detection\_image(Mat frame) et Skin\_detection()**

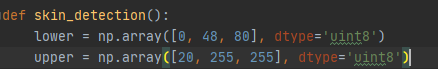
Les 2 fonctions sont les mêmes à une différence près, la première effectue la détection de peau sur une image, et l’autre sur un flux vidéo.

La première est utilisé pour renvoyer vrai ou faux, car on entre en argument un visage détecté par l’algorithme précédemment décrit et on s’attend à ce que l’algorithme nous dise si oui ou non, c’est un visage.

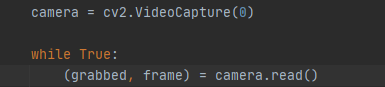
**Skin\_detection()** ne sert qu’à illustrer la détection de peau en temps réel.

La détection de peau se fait dans l’espace HSV (Hue Saturation Value, ou teinte saturation et luminance)

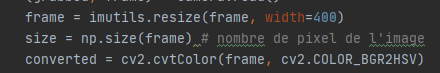
On commence par définir les limites des pixel HSV. Ces valeurs correspondent à l’intensité de la couleur que l’on considère comme étant de la peau. Ces valeurs sont donc purement arbitraire :



Comme pour une détection, on récupère le flux vidéo, et dans une boucle, on récupère la dernière image du flux vidéo :



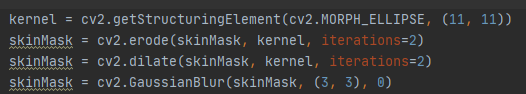
On redimensionne ensuite l’image en la taille voulue, puis on l’a converti en image HSV



On détermine ensuite l’intervalle des pixels qui nous intéresse pour détecter la peau (avec les valeurs précédemment définies, upper et lower) :



Afin d’obtenir une image avec uniquement la peau, on va appliquer plusieurs érosions et dilatations au masque qu’on vient de créer. On effectuera enfin un flou gaussien pour retirer le bruit qui ne nous intéresse pas de l’image :

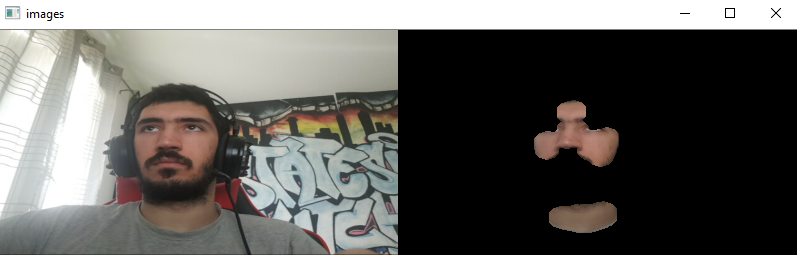


Maintenant que le masque est initialisé avec les transformations souhaités, il suffit de l’appliquer à l’image :



La détection de peau est terminée. Voilà le résultat obtenu si l’on affiche ensuite directement l’image source et l’image avec le masque appliqué :





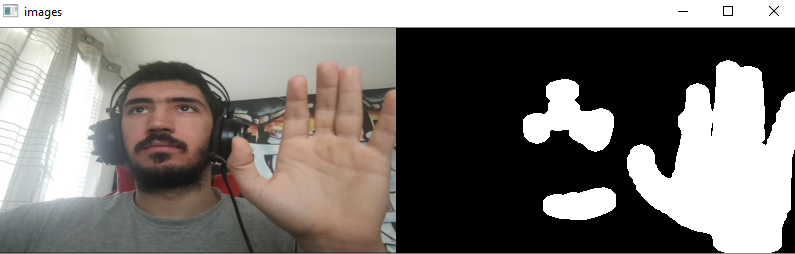
On a à gauche, l’image source, et à droite, l’image n’affichant que la peau.

Cependant, pour pouvoir utiliser cette détection de peau, nous avons besoin d’en sortir des valeurs utilisables. Il est alors intéressant d’effectuer un seuillage binaire afin que les pixels étant détectés comme de la peau s’affiche en blanc. Ainsi, l’image sera en niveau de gris. En comptant le nombre de pixel noir et blanc, on pourra déterminer de façon arbitraire si ce que l’on détecte est un visage ou non .

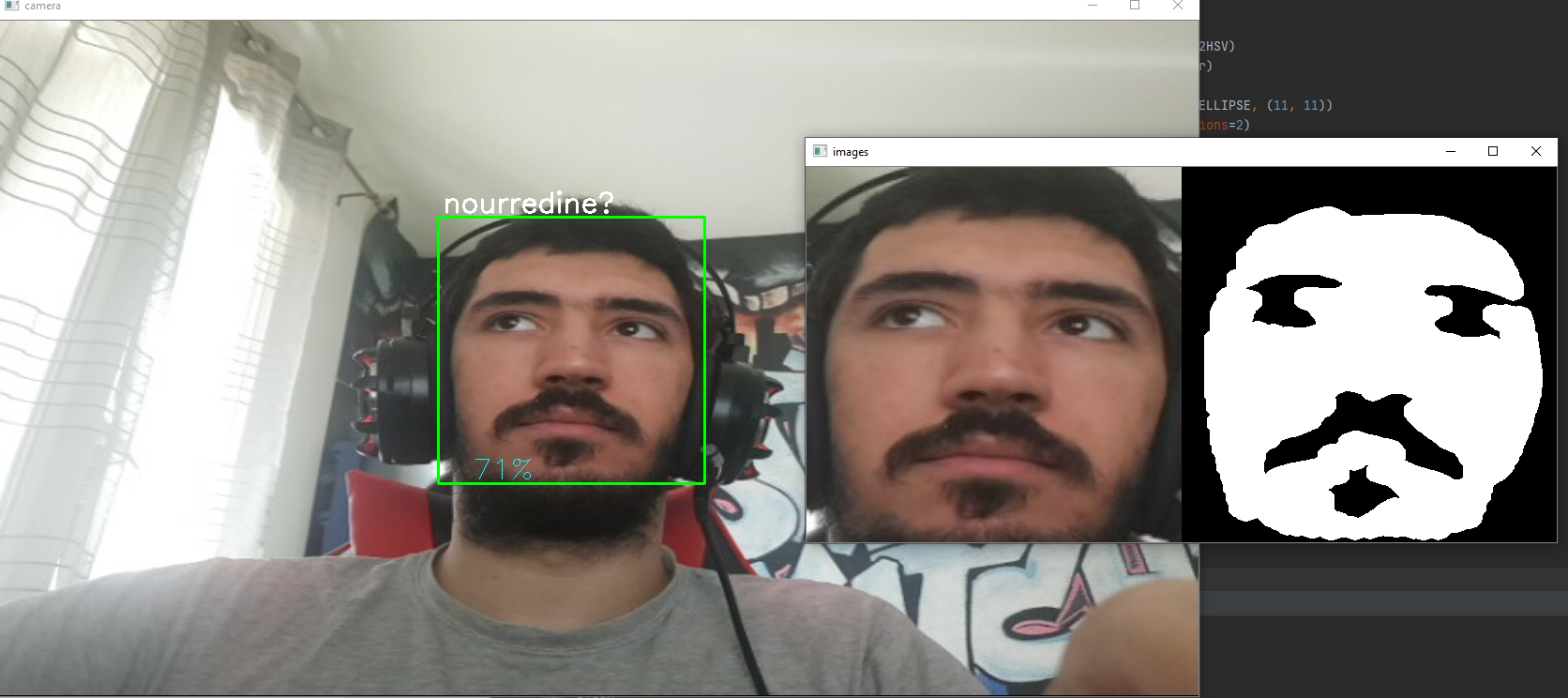
Seuillage binaire :



Il ne reste plus qu’à afficher l’image source et l’image résultat :



Dans la fonction **skin\_detection\_image(frame)** qui sera utilisé pour la reconnaissance faciale que l’on verra par la suite, on peut renvoyer Vrai ou Faux selon le nombre de pixel blanc si l’on considère que l’image entrée en argument est uniquement le visage détecté. Ainsi, pour un certain pourcentage de pixel blanc donc de ‘peau détecté’, on identifie cela comme un visage.



Ici, on voit un exemple de reconnaissance de visage, avec la détection de peau illustré à côté. Comme le visage détecté contient assez de pixel blanc, alors on considère bien cela comme un visage.

**Face\_dataset.py**

Ce fichier contient toutes les fonctions permettant d’enregistrer un visage capté sur un flux vidéo, avec un fichier contenant les informations nécessaires à sa reconnaissance.

En effet, on agit en 2 temps :

-On enregistre les visages avec la fonction « **face\_dataset** », qui leur associera un nom et un ID.

-On entraîne un classifieur pour qu’il puisse reconnaître ces visages-là selon la série d’image associés à chacun avec « **face\_training()** ».

**Face\_dataset()**

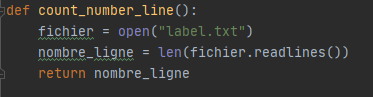
Cette fonction utilise un flux vidéo, on commence donc par en utiliser un :



Puisqu’on va enregistrer les noms des personnes enregistrés dans un fichier texte nommé « label.txt », il est important d’associer un ID à chaque nom (avec par exemple comme ID, le numéro de la ligne sur laquelle il se trouve dans le fichier texte). Initialement, le fichier label est donc composé de x lignes (x étant le nombre de personne enregistrés). L’ID de la personne que l’on souhaite donc enregistrée sera x+1.

On calcule tout simplement x par le nombre de ligne du fichier label (vu que 1 ligne = 1 nom)

Pour cela, on utilise une fonction « **count\_number\_line** », qui s’occupe de compter le nombre de ligne dans un fichier texte :



On obtient donc cela :



On demande ensuite à l’utilisateur le nom de la personne que l’on va enregistrer :



La personne doit se placer face à la caméra pour que son visage soit capturé.

Avant la capture, on écrit dans le fichier « label.txt » dans une nouvelle ligne (la ligne x+1) le nom entré par l’utilisateur :



Capture du visage :

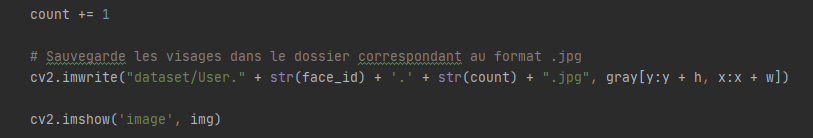
On souhaite capturer 30 fois le visages (cette valeur peut être changé). Ainsi, 30 images du visage seront capturées et enregistrées dans le dossier « **dataset** ».

On initialise donc avant la boucle qui s’itérera 30 fois un compteur à 0.



On détecte ensuite le visage de la même façon que dans la fonction « **Face\_detection** ».

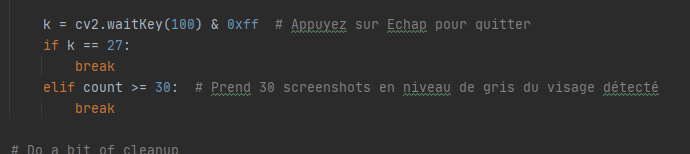
On enregistre uniquement les cordonnées correspondant à celle du visage, et en niveau de gris car cela est nécessaire pour par la suite entraîner le classifieur. On augmente aussi le compteur de 1. On obtient donc cela :



Les images sont enregistrées sous ce nom : User.ID.numéro\_de\_la\_photo.jpg

On affiche ici de façon optionnel le flux vidéo afin que l’utilisateur puisse bien se placer et changer légèrement sa position à chaque prise pour prendre 30 capture du visage légèrement différentes.

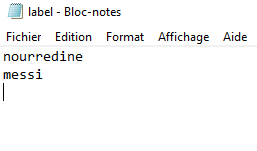
On arrête enfin le programme après 30 itérations de la boucle (ou après l’appui d’une touche) :



Voilà à quoi devrait ressembler le dossier « dataset » :

Les images sont donc bien enregistrées et prêtes à être utilisé pour entraîner le classifieur.

Voilà o quoi devrait ressembler le fichier « label.txt » pour 2 visages enregistrés :



**Face\_training()**

Cette fonction va utiliser les images précédemment enregistrées afin de créer un fichier entrainé nommé « trainer.yml » qui contiendra les données nécessaires à la reconnaissance d’un visage.

Dans cette fonction, on utilisera la méthode d’OpenCV : **LBPHFaceRecognizer\_create()**.

**LBPHFaceRecognizer\_create() :**

Paramètres :

**Int radius** : plus le radius est grand, plus l’image sera lisse mais les informations enregistrées seront plus spatiale.

**int neighbors** : nombre de point nécessaire pour construire une approche circulaire binaire. Par défaut, la valeur est 8. Plus la valeur est grande, plus le temps de calcul sera long.

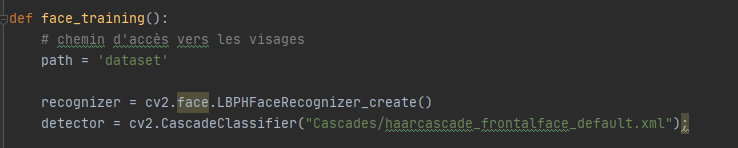
**Int grid\_x** : La valeur par défaut est 8. Elle correspond au nombre de cellule horizontale de la grille. Plus il y a de cellule, plus la grille est fine.

**Int grid\_y** : La valeur par défaut est 8. Elle correspond au nombre de cellule verticale de la grille. Plus il y a de cellule, plus la grille est fine.

**Double treshold** : Seuillage effectué durant la prédiction de visage. Si la distance du pixel le plus proche est plus grand que la valeur du treshold, alors cela renvoie -1.

Si on ne rentre aucun paramètre, alors l’algorithme utilisera des valeurs par défauts, qui ont été pré-calculées comme les plus efficaces dans la plupart des situations.

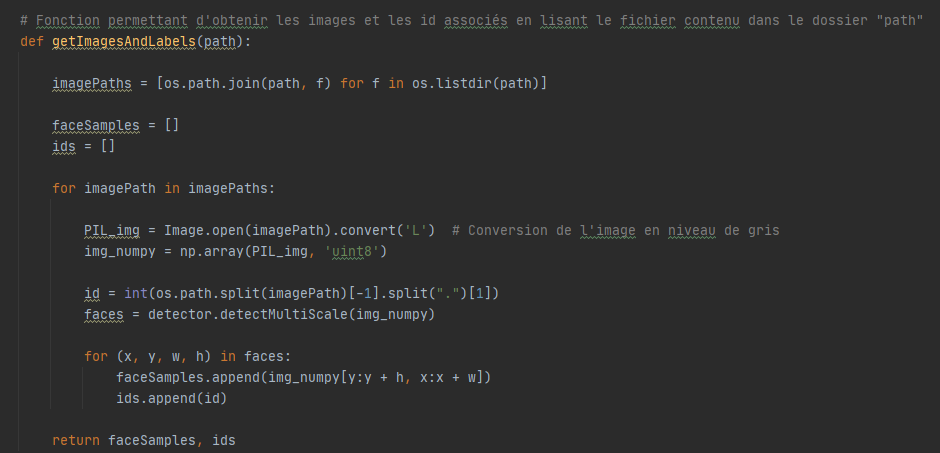
On commence par enregistrer le chemin d’accès vers les visages en .jpg, et à créer l’algorithme de reconnaissance de visage, ainsi que la cascade :



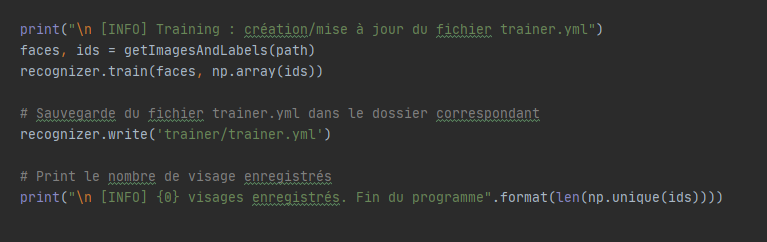
On utilise ensuite une fonction définit dans cette fonction même, « **getImagesAndLabels(string path)**» qui prend en argument le nom d’un dossier, et va retourner une liste d’image de chaque visage contenu dans le dossier, ainsi qu’une liste d’ID associée à chacune des images.

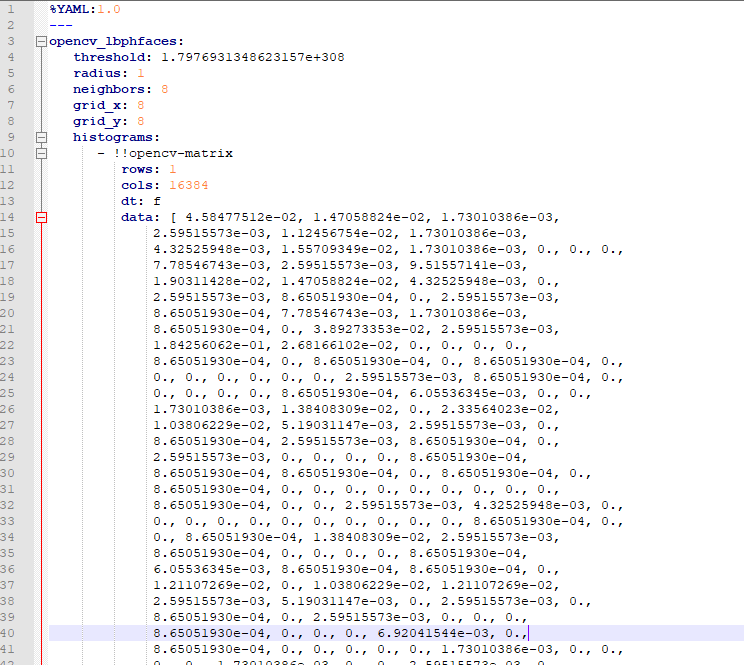
Si 3 visages différents sont par exemple enregistrés dans le dossier « **dataset** », alors elle retournera une liste de 90 visages (30 photos par visage différent), et une liste de 90 ID (les 30 premiers seront la valeur 1, les 30 suivants 2 et les 30 derniers 3, correspondant donc bien aux 90 visages enregistrés)

Voici la fonction « **getImagesAndLabels**» :



Il ne reste plus qu’à entrainer l’algorithme précédemment créé, et d’enregistrer toutes ces informations dans le fichier « **trainer.yml** »**:**

Le fichier « **trainer.yml** »devrait être sous cette forme :



Le fichier est donc entraîné (ou classifié). Il est donc possible de reconnaître un visage. Nous allons voir pour cela en détail le fichier « **recognition.py** », mais avant cela, nous allons voir comment utiliser des photos déjà existantes pour entrainer notre classifieur, sans enregistrer nous même le visage.

**Get\_face\_from\_image.py**

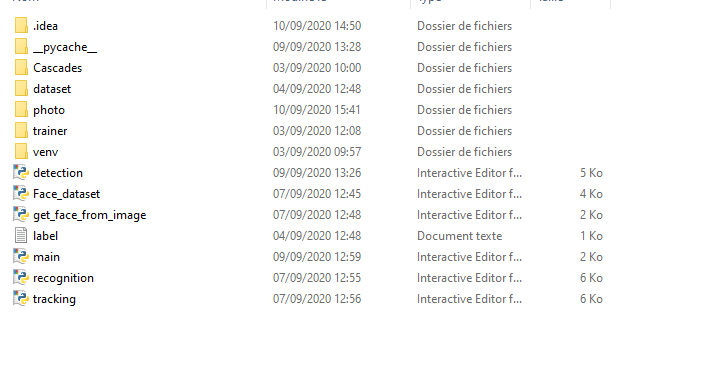
Ce fichier contient 2 fonctions très semblables à celles du fichier «**Face\_dataset.py** ».

Au lieu d’enregistrer nous même les visages à partir d’une caméra, nous allons récupérer des photos déjà existante d’une même personne et les utiliser de la même façon.

Les photos de la personne à enregistrer doivent être stockés dans le dossier « **photo** ».

**ATTENTION !** Uniquement les photos d’une et une seule personne doit être stockés à l’intérieur pour que l’algorithme fonctionne ! Le format des photos doit être en .jpg ou .png, et la taille importe peu. Les photos doivent être en couleur ou en niveau de gris. Elles ne doivent pas forcément suivre un certain format de nom.

Exemple :

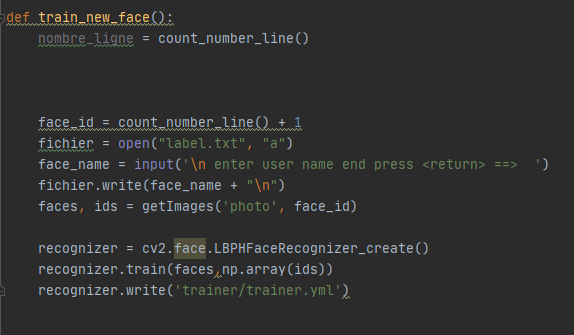




Une fois les images enregistrées dans le bon dossier, il ne reste plus qu’à appeler la fonction « **train\_new\_face()**»qui va mettre à jour le fichier « **label.txt**»avec un nouveau nom entré par l’utilisateur, puis entrainer le fichier classifié avec ces nouvelles images.

**Train\_new\_face()**

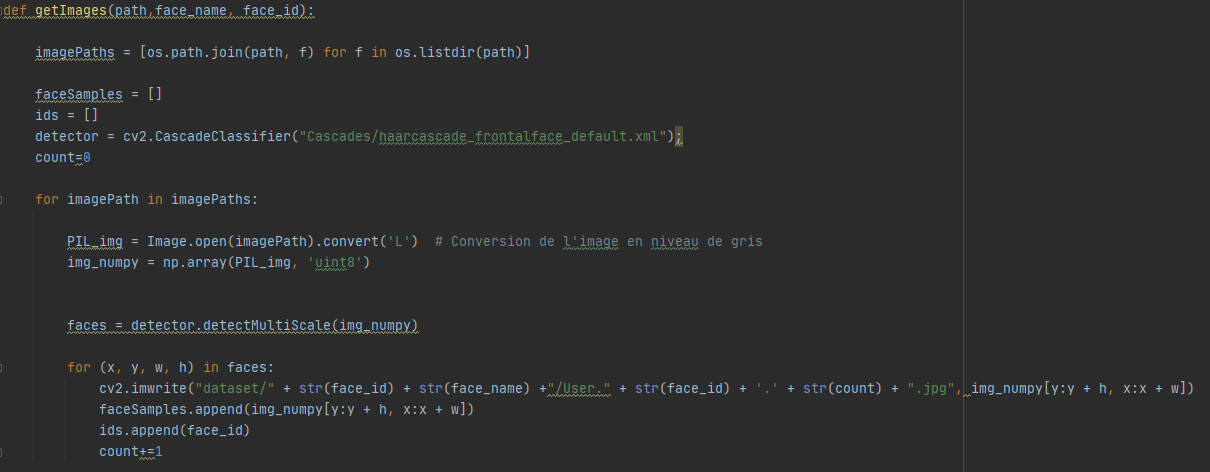
Cette fonction utilise le même principe que la fonction « **Face\_training()**» du fichier « **Face\_dataset.py**»**:**



La seule différence se trouve dans le fait qu’on appelle la fonction « **getImages(path, id)**»à la place de la fonction « **getImagesAndLabels( path)** » du précédent exemple.

**getImages (path, face name, ID)**

Cette fonction va permettre de récupérer les images du dossier « **photo**»**,** d’y détecter les visages, d’enregistrer les visages en niveau de gris dans le dossier « **Dataset**» , et de renvoyer la liste des visages. Son mode de fonctionnement est le même que la fonction « **getImagesAndLabels**»**:**



Une fois les images bien enregistrées et le fichier « **trainer.yml** » bien entraîné, il est possible de reconnaître un visage.

**Delete\_user()**

Cette méthode va lister les utilisateurs enregistrés et va supprimer celui qu’on choisit. Ainsi, on enlève le nom du fichier « **label.txt** » et on supprime le dossier de photos contenant ces photos (incluses dans le dossier « **dataset** » ).

Enfin, on renomme tous les dossiers restant pour qu’ils bénéficient d’un nouvel ID (si on supprime le visage 2, alors le visage 3 devient le visage 2 en terme d’ID).

On ré-entraine ensuite le fichier « **trainer.yml** » sur tous les visages en appelant « **face\_training** »

**recognition.py**

Ce fichier contient 2 fonctions de reconnaissance différente. La première permettra uniquement de reconnaître un visage en y renseignant le taux de pourcentage de ressemblance à la personne mise en correspondance. Elle utilisera le principe de détection de visage, de peau, et les données entraînés vu précédemment. La deuxième fonctionne de la même manière sauf que lorsque le visage est reconnu, un algorithme de suivi est lancé. Ainsi, la reconnaissance n’est effectué qu’une fois, et le suivi de la personne est effectué par une méthode dite de « tracking», ne nécessitant plus le fichier « **trainer.yml** » pour fonctionner.

**Face\_recognition\_no\_tracking() :**

Afin de reconnaître un visage, il est important d’effectuer ces étapes dans l’ordre :

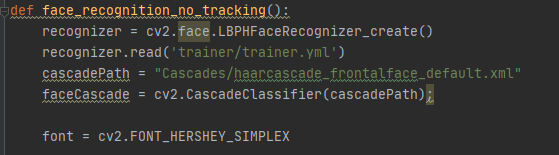
-Détecter les visages d’un flux vidéo

-Détecter sur ces visages la peau

-Déterminer avec nos données si ce visage peut être associé à quelqu’un de connu

-afficher son nom ou non sur le flux vidéo, avec le pourcentage de ressemblance

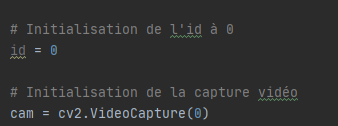
Pour la première étape, il faudra utiliser une cascade. Nous aurons aussi par la suite besoin de lire les données contenues dans « **trainer.yml** », c’est pourquoi nous allons l’ouvrir en lecture (read) seulement :



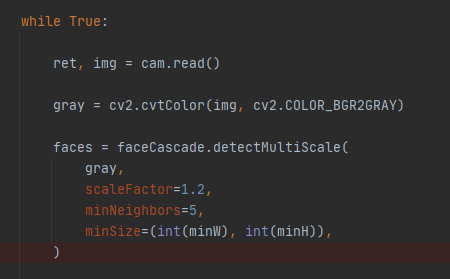
Note : la dernière ligne correspond à la création de la police d’écriture, qui sera utilisé pour afficher le texte sur le flux vidéo. Elle est bien évidemment changeable.

On crée une variable ID qu’on initialise à 0. Selon la personne reconnue, cette valeur changera.

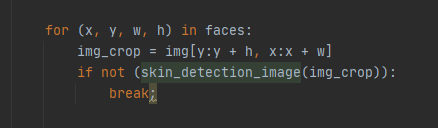
On initialise aussi le flux vidéo :



On crée une boucle dans laquelle on effectue sur ce flux vidéo une détection de visage :



Pour chacun des visages détectés, on effectue une détection de peau. La fonction « **skin\_detection\_image(image)**» renvoie vrai ou faux, selon une certaine valeur de pixel considéré comme étant de la peau. Lorsque la fonction nous renvoie faux, alors on arrête cette itération de boucle, et on passe au prochain visage :

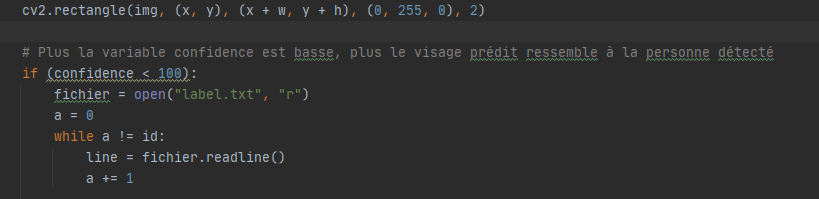


Si le visage est détecté comme étant bien un visage par la détection de peau, alors on continue dans la boucle.

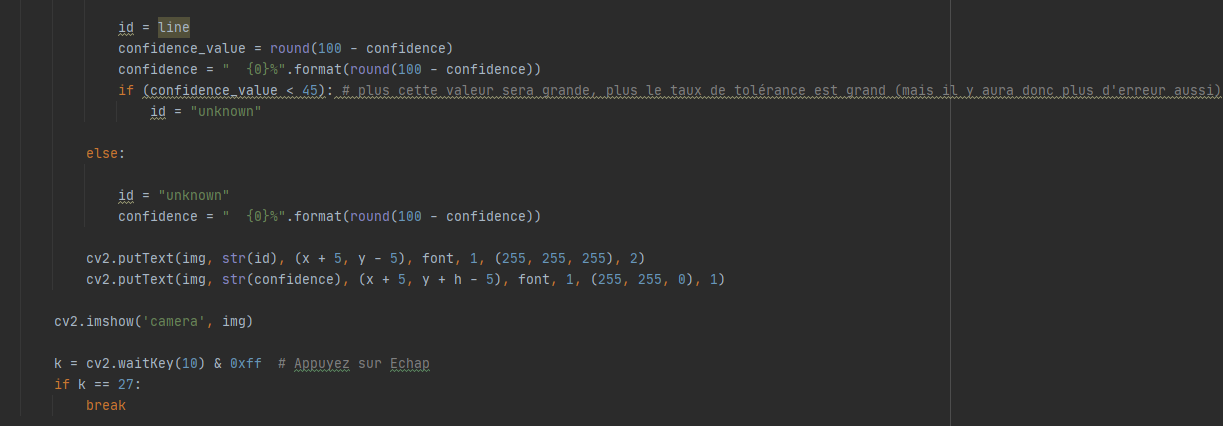
On effectue une prédiction via « **predict()**» , qui prend en entrée une image, et recherche l’ID de l’objet avec le plus de ressemblance enregistrés via nos vecteurs du fichier « **trainer.yml**»**.** Il fait donc une mise en correspondance, et renvoie deux valeurs, l’ID, et la « confidence », qui est une valeur permettant de déterminer le taux de ressemblance (plus cette valeur est petite, plus l’objet ressemble à l’objet enregistré en base de donnée. On considère que si confidence = 0, alors l’objet du flux vidéo ressemble parfaitement à l’objet enregistré, ce qui en pratique impossible)



Si la valeur de **confidence**  renvoyée par « **predict()** » est assez petite (le taux est arbitraire, ici on estime cette valeur à 100), alors on recherche le nom associé à l’ID dans le fichier « **label.txt** » par rapport au numéro de la ligne.



On calcule le taux de ressemblance, si ce taux est trop faible (arbitrairement, confidence < 45), alors on considère que le visage reconnu n’est pas le bon. On affiche donc comme nom « unknown »



En résumé, il y a donc 2 possibilités pour qu’on ne reconnaisse pas un visage :

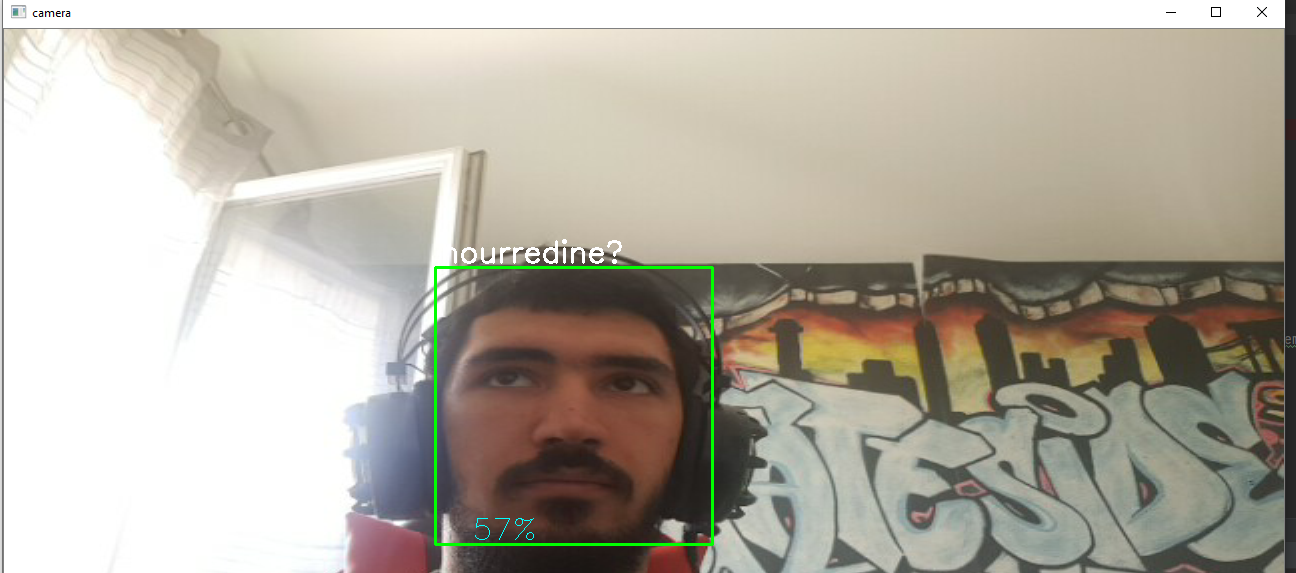
-Le visage a été mis en correspondance avec une personne enregistrée, mais le taux de ressemblance est trop faible, on évite donc de propager une erreur et on préfère afficher « unknown ».

-Le visage n’est pas connu et la fonction « **predict()**» n’a trouvé personne à mettre en correspondance avec le visage, on affiche alors « unknown » .

Plus la valeur de « confidence\_value » sera grande, plus il y aura possibilité d’erreur. En effet, une personne enregistrée sera correctement reconnue, mais à cause du faible taux de ressemblance (qui peut être impacté par plusieurs facteurs tel que la luminosité, ou les vêtements de la personne)

Si le visage est reconnu, alors on affiche son nom au-dessus du cadre englobant dans le flux vidéo, ainsi que le taux de ressemblance en pourcentage.

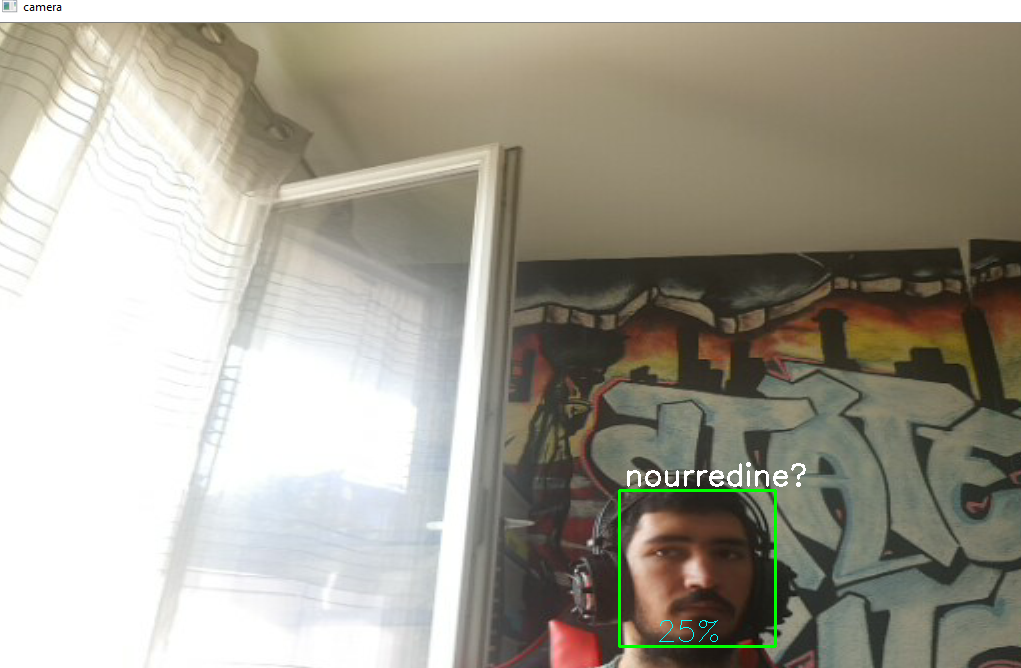
Voici le résultat attendu :



Voici un exemple dans lequel le visage est un peu plus éloigné :



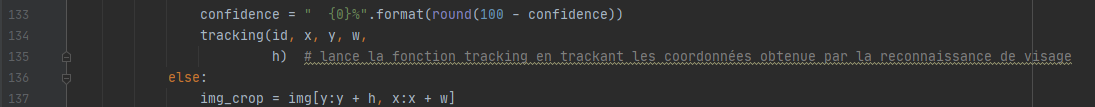
« Unknown » est affiché au-dessus du visage bien que la fonction « **predict()** » l’associe à la bonne personne pour une seule raison : le taux de pourcentage demandé pour la mise en correspondance est trop faible. Si on diminue cette valeur de 45 à 20, la mise en correspondance s’effectue parfaitement :



Il est donc important de noter que cette valeur joue un rôle clé. On se livre plus à des faux positifs, mais à l’inverse, il y aura moins de faux négatifs.

**Face\_recognition():**

Cette fonction est une quasi copie de « **Face\_recognition\_no\_tracking()**»**.** Sauf qu’elle appelle une fois le visage mis en correspondance une fonction « **tracking()**» qui va s’occuper de suivre le visage reconnu. Ainsi, même si on s’éloigne de la caméra ou que l’on tourne la tête (et que notre visage n’est pas visible, ou partiellement visible), les cordonnées de la personne seront connues à l’écran.

Voici l’unique ajout de cette fonction (ligne 134,135 du code source dans le fichier « **recognition.py**» :

Nous allons donc voir en détail comment fonctionne le tracking dans le fichier « **tracking.py** » puis observer le résultat du suivi.

**Tracking.py**

Ce fichier contient 2 fonctions de tracking. Chacun permet d’utiliser jusqu’à 4 algorithmes différents de suivi.

**Méthodes de Tracking :**

**MIL :**

Ce tracker va analyser les pixels compris dans la zone à tracker, et ceux au voisinage. Ainsi, plusieurs « paquets d’images » seront enregistrés. Ce qui est compris autour de la zone à tracker sera utilisé afin de suivre la zone. Cela permet donc de continuer à tracker un objet partiellement caché (en cas de croisement, par exemple). Cependant, si l’objet disparaît du champ de vision puis réapparait, il ne sera plus suivi, car il aura perdu de vue l’objet.

**BOOSTING :**

La méthode Boosting fonctionne de la même manière que le MIL. Cependant, le boosting est un algorithme moins récent, et plus lent. Il est présent dans le code afin de pouvoir le comparer avec le MIL.

**KCF :**

Le KCF (Kernelized Correlation Filters) utilise le même principe que le MIL mais le calcul des données récoltées est différent. Ainsi, il traite les données plus rapidement que le MIL.

**CSRT :**

CSRT (en anglais, Discriminative Correlation Filter), utilise la fiabilité spatiale pour ajuster le filtre à la région d’intérêt et améliorer le tracking. Il est utilisé pour tracker des objets ou régions non rectangulaire. L’algorithme utilise les histogrammes de gradients orientés (HOG) pour fonctionner. Il est plus précis que le KCF mais en contrepartie, est beaucoup plus lent.

**TLD :**

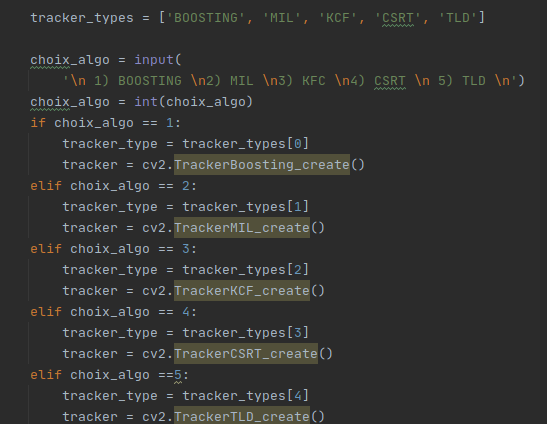
Le TLD décompose le suivi de l’objet en 3 étapes : Le suivi (**T**racking), l’apprentissage (**L**earning)et la détection (**D**etection).

La zone à tracker sera analysée, tout au long du suivi, pour connaître les différentes formes de l’objet au cours du temps. Ainsi, même si l’objet sort du champ de vision, puis y revient, le TLD aura toujours l’objet en vue puisqu’il le détecte.

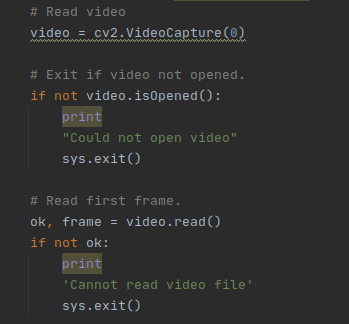
Cet algorithme est celui qui réagit le mieux aux occlusion. Cependant, beaucoup de faux positifs apparaissent dû au fait que si l’objet qui réapparaît n’est pas exactement le même, l’algorithme aura tendance à penser qu’il l’est (comme avec par exemple avec 2 personnes différentes).

**Tracking\_zone():**

Cette fonction permet de suivre une zone sélectionnée par l’utilisateur lui-même sur le flux vidéo. Tout d’abord, on demande à l’utilisateur l’algorithme qu’il souhaite utiliser :



On lit ensuite le flux vidéo :

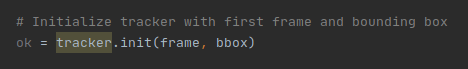


On enregistre la première image de ce flux dans une variable nommé frame.

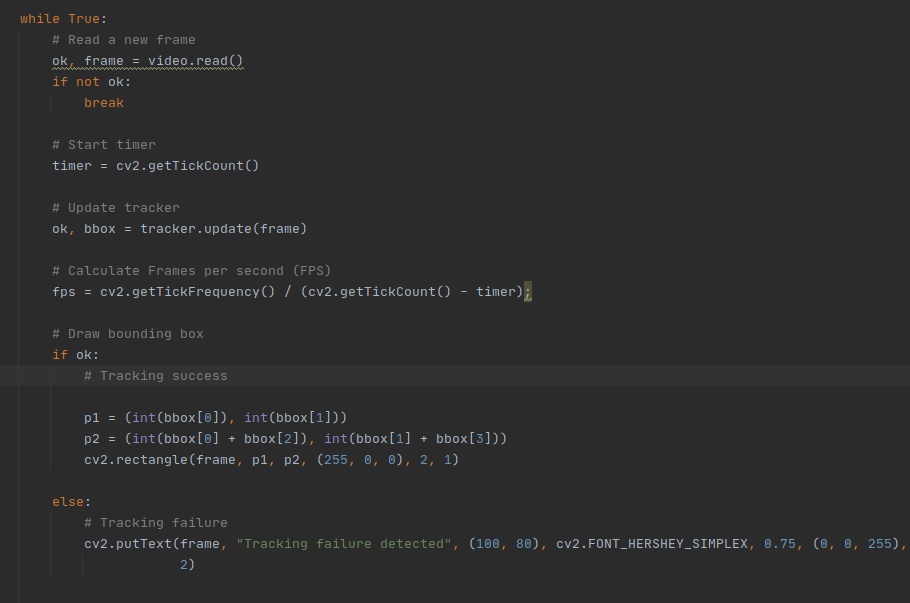
On utilisera ensuite « **selectROI()** » qui va permettre à l’utilisateur de sélectionner une zone sur l’écran :



On initialise ensuite le tracker avec la première image du flux vidéo, et la liste des 4 points, correspondant aux cordonnées de la zone que l’on souhaite suivre :



Dans une boucle, on lit en continu le flux vidéo, puis on met à jour le tracker avec la boîte englobante « bbox » sur la dernière image du flux vidéo. Ainsi, la boîte englobante va avoir des cordonnées différentes à chaque itération de la boucle lors de la mise à jour du tracker.

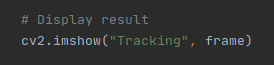


Note : dans cette boucle, le nombre d’image par seconde est affiché sur le flux vidéo grâce à un timer nommé « timer » qui est intialisé au début de la boucle.

Le tracker est mis à jour avec la commande : tracker.update(frame)

Si jamais le tracker ne suit plus rien, alors le suivi est un échec, et il est affiché à l’écran qu’il y a une erreur de suivi (cela arrive en général lorsque l’objet suivi sort du champ de vision de la caméra

On affiche enfin le résultat en affichant l’image :



**Tracking\_(id, a1, a2, a3 ,a4) :**

Cette méthode effectue le même travail que la méthode précédente, sauf que cette fois-ci, elle prend en entrée ce qui sera les cordonnées de la zone à tracker. Cette méthode est alors couplée avec la méthode de reconnaissance, afin d’entrer en argument les cordonnées du visage.

**Bibliographie :**

<https://www.geeksforgeeks.org/opencv-python-tutorial/>

<https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/>

<https://www.pyimagesearch.com/2018/07/30/opencv-object-tracking/>

<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html>

<https://docs.opencv.org/3.4/d7/d8b/tutorial_py_face_detection.html>

<https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>

<https://www.pyimagesearch.com/2014/08/18/skin-detection-step-step-example-using-python-opencv/>

<https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html>

<https://medium.com/dev-genius/face-recognition-based-on-lbph-algorithm-17acd65ca5f7>

<https://www.firediy.fr/article/face-tracking-implementation-de-la-methode-de-viola-jones-en-c>