Université de Technologie de Compiègne
SY32 : Analyse et synthèse d'images

Projet : Détection de visages

Marie Simatic & Guillaume Jounel

Printemps 2017

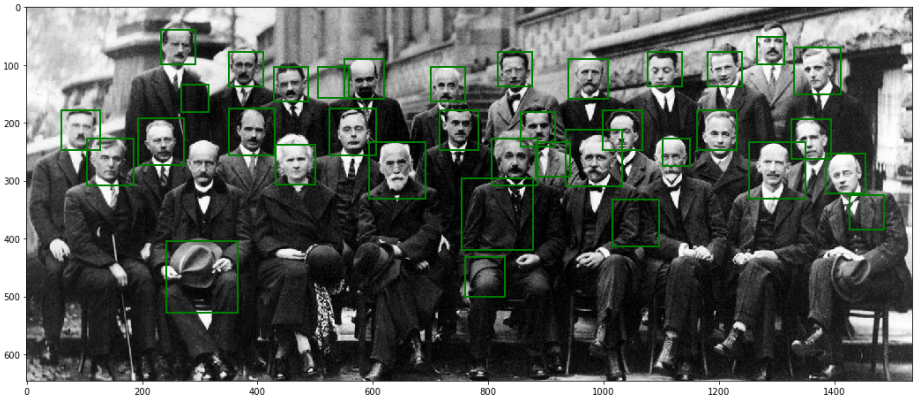

Figure 1: Détection de visages sur la célèbre photo du 5ème Congrès Solvay (1927)

Résumé Projet réalisé dans le cadre de l'UV SY32 (Analyse et synthèse d'images) à l'Université de Technologie de Compiègne. À partir de 1000 images d'entraînement contenant chacune un visage dont les coordonnées sont fournies, un détecteur de visage a été entraîné, basé sur l'usage d'un classifieur simple couplé à une fenêtre glissante. Les technologies utilisées sont les librairies Scikit-image et Scikit-learn sous Python 3.4.3.

Table des matières

Introduction

Ce rapport rend compte d'un projet de détection de visages réalisé dans le cadre de l'UV SY32 (Analyse et synthèse d'images) à l'Université de Technologie de Compiègne. 1000 images d'entraînement nous ont été fournies, contenant chacune un visage dont les coordonnées sont renseignées. L'objectif est alors de construire un détecteur de visage en transformant le problème de détection en multiples problèmes de classification à l'aide d'une fenêtre glissante. L'ensemble de notre code est disponible à l'adresse suivante : `https://github.com/guillaumejounel/Face-detection`. L'ensemble du projet fonctionne sous Python 3.4.3 à l'aide des libraires Numpy, Scikit-image et Scikit-learn, sous Spyder 3.1.2. À noter que le fichier `data.spydata` peut être chargé directement sous Spyder dans l'explorateur de variables et qu'il contient un grand nombre de données, notamment l'ensemble des faux positifs qui demande un temps considérable de calcul. Nous remercions M. Xu de nous avoir permis de réaliser ce projet passionnant sous Python et nous vous souhaitons une agréable lecture.

1 Données d'apprentissage

Les données d'apprentissage ont été données sous la forme suivante :

— Un set de 1000 images comportant chacune un visage
— Un fichier `label.txt` comportant l'emplacement, pour chaque image, des coordonnées du rectangle contenant le visage.

La récupération des données de `label.txt` se fait directement via la fonction `loadtxt()` de la bibliothèque Numpy, comme montré par le code 1.1.

Code Source 1.1: Chargement des données issues du fichier `label.txt`

```
pathFile = "projetface/label.txt"
data = np.loadtxt(pathFile)
```

La première étape à faire afin de commencer tout traitement de données est d'uniformiser les informations relatives à chaque image, c'est à dire que chaque image traitée devra avoir la même taille et contenir sensiblement la même partie de visage.

Pour résoudre le problème de taille, nous avons dans un premier temps choisi d'utiliser un carré déterminé de la façon suivante, pour chaque image :

— La taille de son côté est le plus petit côté du rectangle contenant le visage
— Son centre est le centre du rectangle.
— Le carré est ensuite redimensionné à une taille commune à tous les carrés extraits des données.

La taille commune est défini par la petite taille de rectangle trouvée dans les données. C'est le rôle de la fonction `minFace(data)` que de déterminer cette taille minimum qui sera appelée tout au long de notre code `newSize`. Cette taille était de 30 pixels.

Dans la suite du projet, nous avons remarqué que nous détections souvent des bouches et des yeux mais rarement des visages (voir figure 2). Nous avons alors compris que notre classifieur ne pouvait pas correctement détecter le contour des visages si ceux-ci étaient régulièrement coupés, et qu'il fallait donc plutôt agrandir le rectangle pour en faire un carré plutôt que de réduire le plus grand côté. La fonction `minFace(data)` a été modifiée en conséquence et la taille utilisée est alors de 46 pixels. Nous avons fait le choix de ne pas considérer les images pour lesquelles le carré redimensionné "à l'extérieur" dépassait de l'image. Ainsi, nous n'utilisons que 904 images sur les 1000.

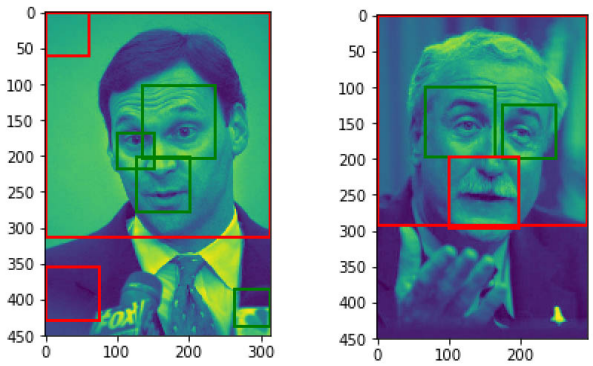

FIGURE 2: Exemple de détections réalisées suite à un recadrage intérieur

La fonction `dataSquare(data)` ci-dessous permet, à partir des données extraites de `label.txt` originelles, d'en extraire les carrés associés à chaque image. La fonction `cropImage` quant à elle permet d'extraire la zone définie dans la variable de type `data` et d'éventuellement la redimensionner à une une taille `newSize * newSize`.

Le résultat de l'usage de ces premières fonctions est illustré par la figure 3.

Code Source 1.2: Définitions des fonctions `dataSquare()` et `cropImage()`

```
# Transforme les rectangles des datas en carrés
# data = [[NbImage, x, y, width, height]
#         ...]
def dataSquare(data):
    # Récupération des "rectangles"
    newData = np.array(data)
    # Récupération du minimum entre la largeur et la hauteur
    minwh = np.minimum(data[:, 3], data[:, 4])
    # Modification de la largeur
    newData[:, 1] += (data[:, 3]-minwh)//2
    # Modification de la hauteur
    newData[:, 2] += (data[:, 4]-minwh)//2
    # Transformation des rectangles en carrés
    newData[:, 3:] = np.transpose(np.array([minwh, minwh]))
    return newData

# retourne l'image croppé selon le rectancle
# si l'argument newsize vaut 1 / True croppe selon un carré
```

```
def cropImage(n, data, pathTrain, newsize=0):
    img = np.array(io.imread(pathTrain + "%04d" %(data[n, 0]) + ".jpg"),
                   dtype=np.uint8)
    x, y, w, h = map(int, data[n][1:])
    img = img[y:y+h, x:x+w]
    if newsize:
        img = resize(img, (newsize, newsize), mode='reflect')
    return img
```

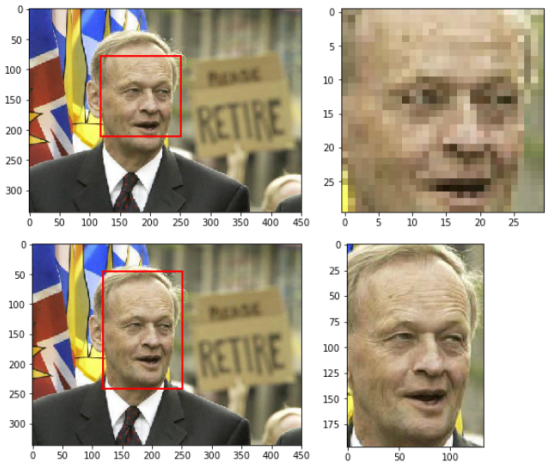

FIGURE 3: Résultat des fonctions `dataSquare()` et `cropImage()` (avec recadrage intérieur)

Afin d'obtenir plus d'exemples positifs, pour chaque exemple positif nous utiliserons en plus son symétrique par rapport à l'axe horizontal.

1.1 Exemples négatifs

Il nous faut à présent obtenir des exemples négatifs. Pour ce faire, nous choisissons des parties d'images qui ne recoupe pas (ou suffisamment peu) la partie contenant le visage.

Le recouvrement d'une partie d'une image par une autre nous est donné par la fonction `recouvrement()` définie par le code 1.3.

Code Source 1.3: Définitions de la fonction `recouvrement()`

```
# revoie le recouvrement de deux carrés
def recouvrement(x1, y1, w1, h1, x2, y2, w2, h2):
    xinter = max(0, min(x1 + w1, x2 + w2) - max(x1, x2))
    yinter = max(0, min(y1 + h1, y2 + h2) - max(y1, y2))
    ainter = xinter * yinter
    aunion = (w1*h1) + (w2*h2) - ainter
    return ainter/aunion
```

```
# exemples d'utilisation :
print("Recouvrement total:", recouvrement(10,10,30,30,10,10,30,30))
print("Recouvrement nul:", recouvrement(10,10,30,30,40,40,30,30))
print("Recouvrement partiel:", recouvrement(10,10,30,30,20,20,30,30))
```

Nous avons considéré qu'un carré ne contenait pas de visage s'il avait moins de 30% de recouvrement avec le carré contenant le visage. Afin d'extraire un certain nombre d'exemple négatif d'une même image, nous avons créé deux fonctions

`negatifRandom()` cette fonction renvoie un exemple négatif. Elle détermine des carrés aléatoires dans l'image et retourne le premier qui ne soit pas considéré comme un visage.

`exemplesNegatifs()` renvoie n exemples négatifs à partir d'une même image. Nous n'avons pas jugé nécessaire de tester l'inter-recouvrement entre ces exemples négatifs, en partant du principe qu'ils seront assez différents entre eux, au vu du caractère aléatoire de leur génération.

Nous avons choisi de prendre 10 exemples négatifs pour chaque image fournie, compromis nécessaire entre performance de l'apprentissage et coût en temps de calcul. À noter que nous prenons également le symétrique par rapport à l'axe horizontal comme négatif. Bien entendu, il s'agit d'images de taille $newSize * newSize$ dont la position (x, y) et l'échelle sont aléatoires.

2 Apprentissage

2.1 Vecteur descripteur

Le premier paramètre à déterminer pour la création de notre classifieur est le vecteur descripteur, caractéristiques types de chaque image.

Intensité des pixels Le choix le plus simple serait d'utiliser l'intensité en nuance de gris des images comme vecteur descripteur. Néanmoins, en plus d'être très peu rapide (pour une image de taille $30 * 30$, ce vecteur est de taille 900 et est donc très grand et rend les calculs très lents), ce vecteur n'est pas très performant : sensible à la couleur, aux changements d'intensité, des réglages caméras...

Gradient des images Une autre solution envisagée est d'utiliser le gradient de l'image, insensible aux couleurs et peu sensible à la luminosité puisqu'il mesure les variations. Néanmoins cette méthode partage encore un gros inconvénient avec la première méthode : la taille du vecteur descripteur et donc le temps bien trop grand demandé par son utilisation.

Histogramme de Gradient Orienté (HOG) Les features de HOG sont programmées par défaut dans la libraire `skimage`. Les paramètres que nous pouvons utiliser sont :

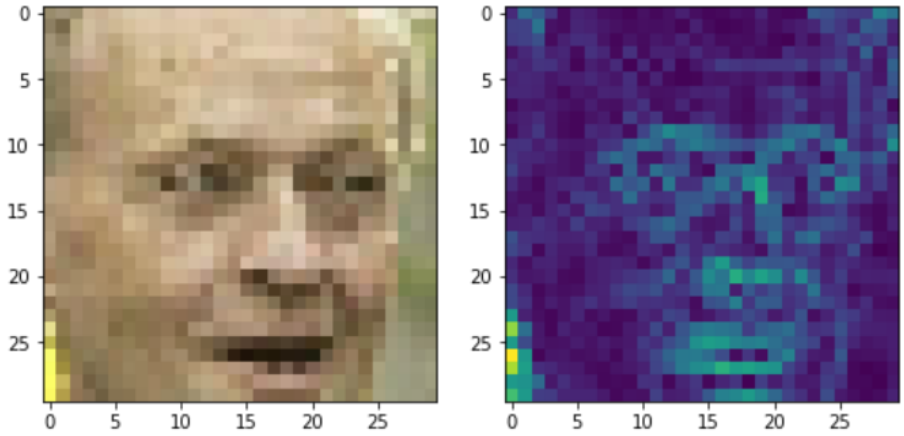

FIGURE 4: Comparaison entre une image normale et son gradient

Le nombre d'orientations : 8
Le nombre de pixels par cellule : 9
Le nombre de cellule par blocs : 1
La méthode de normalisation de blocs : L1

Ces choix représentent un compromis vis-à-vis du coût que cela implique : un descripteur avec plus de cellules et de blocs aurait sûrement été plus efficace, mais celui-ci a une longueur de 200 et il est suffisant pour garantir des résultats acceptables, notre puissance de calcul étant limitée, tandis qu'un descripteur trop petit rendrait difficile un apprentissage efficace. La figure 5 montre une représentation de HOG d'une image, côte à côte avec l'image réelle.

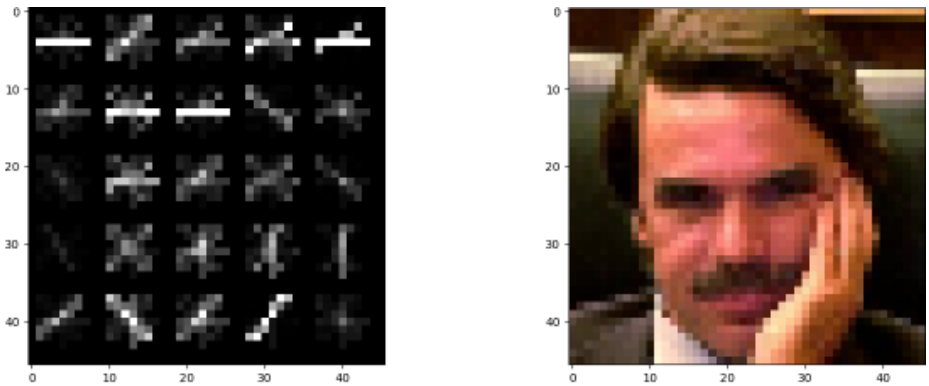

FIGURE 5: Comparaison entre une image normale et sa représentation de HOG

2.2 Classifieur

Adaboost Avec un classifieur Adaboost (paramètres par défaut), les résultats que nous avons obtenu n'étant pas exceptionnels et demandant beaucoup de temps, nous avons décidé d'essayer un autre classifieur.

Random Forest De même, le classifieur Random Forest (paramètres par défaut), nous a donné trop peu de résultats intéressants (très strict). En revanche, celui-ci avait l'avantage d'être entraîné très rapidement.

SVM Notre décision finale s'est portée sur le choix de ce classifieur avec un noyau linéaire, ses résultats nous ayant paru les plus probants. Celui-ci a l'avantage d'être facilement paramétrable selon C.

2.3 Mesure de la performance de notre classifieur

Validation Croisée Pour ajuster le paramètre C, une méthode usuelle consiste à effectuer de multiples validations croisées et de tracer une courbe du taux d'erreur en fonction de C. On peut ensuite réitérer l'opération avec plus de précision afin d'obtenir un C optimal. C'est ce qui a été réalisé grâce à la fonction `graphValidationCroisee()`, comme le montrent les figures 6, 7 et 8.

Le choix de C représente un compromis entre la minimisation d'erreur dans l'apprentissage et la capacité de généralisation d'un modèle. Ainsi, un trop grand C donnera un bon résultat mais ne sera pas généralisable. De plus, un grand C augmente le temps nécessaire à l'apprentissage.

En revanche, comme nous l'avons vu en cours, la quantité de négatifs étant largement supérieure aux positifs, un C optimal pourrait bien correspondre à un modèle qui ne détecte rien... C'est pourquoi cette méthode n'est pas suffisante pour établir l'efficacité de notre classifieur et que l'on introduit les notions de précision et de rappel dans la partie suivante.

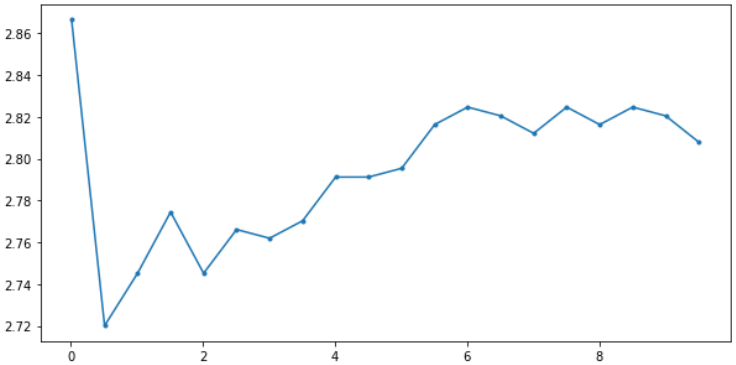

FIGURE 6: Courbe du taux d'erreur en fonction de C (de 0.01 à 10, pas de 0.5)

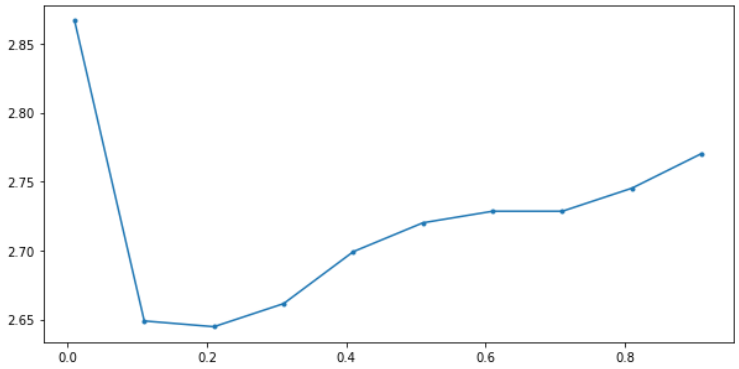

FIGURE 7: Courbe du taux d'erreur en fonction de C (de 0.01 à 1, pas de 0.1)

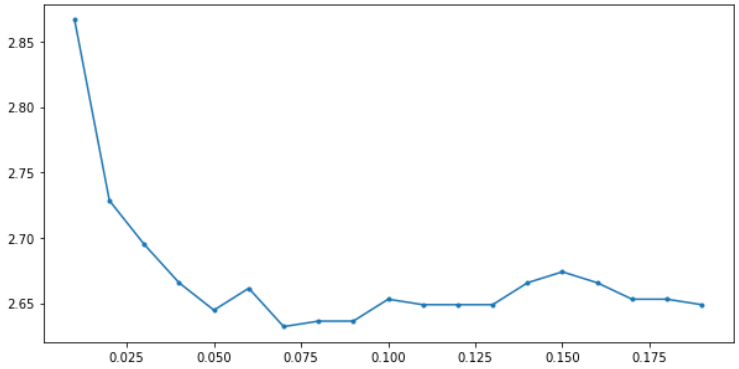

FIGURE 8: Courbe du taux d'erreur en fonction de C (de 0.01 à 0.2, pas de 0.01)

Rappel, Précision Nous mesurerons la performance de notre classifieur en le testant sur nos données d'entrainement, ce qui nous permet d'obtenir les taux de rappel et de précision, c'est à dire :

$$rappel = \frac{N(visages\_correctement\_detectes)}{N(visages\_a\_detecter)} \tag{1}$$

$$precision = \frac{N(visages\_correctement\_detectes)}{N(visages\_detectes)} \tag{2}$$

Ces deux valeurs sont obtenues en parsant les données détectées et les données d'entrainement via la fonction `affichAnalyseResultat()`.

Influence du C sur le rappel et sur la précision Nous avons cherché à déterminer l'influence du C sur les taux de rappels et de précision de notre classifieur. Manquant de temps pour réaliser cette étude sur l'intégralité des 1000 échantillons fournis, nous n'avons utilisé notre classifieur que sur 100 images, divisant le temps de calcul par 10.

Cette échelle est à prendre en compte dans la comparaison des taux de rappels (cf. figure 10). Le taux "réel" que l'on aurait obtenu étant à multiplier par 10.

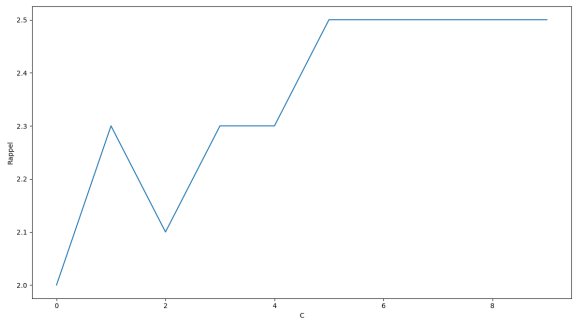

FIGURE 9: Comparaison des différents taux de rappel par rapport au C (sur un échantillon de 100 images

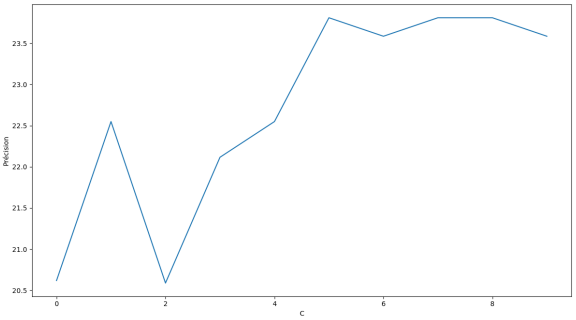

FIGURE 10: Comparaison des différents taux de précision par rapport au C (sur un échantillon de 100 images

Dans les deux cas, l'impact du C sur le rappel et sur la précision est de l'ordre de 5%, ce qui est finalement assez négligeable.

Le script `./tests/rappel_precision_c.py` a été utilisé pour produire ces courbes.

3 Détection

Maintenant que nous avons un classifieur entraîné, nous pouvons désormais l'utiliser sur les données de test, dont nous n'avons aucune autre information qu'elles contiennent parfois un ou plusieurs visages.

3.1 Fenêtre glissante

La première étape est de déterminer où se situerait un éventuel visage dans l'image. Pour cela, nous utilisons le principe de fenêtre glissante. Pour une image nous la parcourons avec un carré de taille `newSize*newSize` analysé par notre classifieur de telle sorte que nous puissions déterminer s'il est probable qu'il y ait un visage à cet endroit ou non.

La fonction `fenetreGlissante` parcourt l'intégralité d'une image grâce à la fenêtre glissante et le classifieur détermine le score de chaque fenêtre. L'argument `animated`, lorsqu'il vaut 1, permet d'observer la fenêtre se déplacer sur l'image et réaliser les détections (attention, l'affichage augmente considérablement les temps de calcul).

Néanmoins l'utilisation de cette seule fonction ne serait pas très utile : en effet, il doit pouvoir détecter les visages à toute échelle possible. C'est rendu possible par la fonction `fenetre_glissante_multiechelle`, qui va redimensionner l'image selon différents facteurs et appeler `fenetreGlissante` pour chacune de ces nouvelles images, avec un argument comprenant le ratio utilisé, afin que `fenetreGlissante` renvoie les coordonnées de fenêtres adaptées à l'image réelle.

Dans un premier temps, nous redimensionnons l'image en la minimisant d'un ratio de telle sorte qu'elle mesure *newSize* pixels (`newSize/min(img.shape)`), puis à chaque niveau, nous modifions ce ratio relativement en l'augmentant (`ratio *= 1.3`) jusqu'à ce qu'il atteigne 1 (imagine originale). Le pas, quant à lui, est relatif à la taille de la fenêtre (`newSize//4`).

3.2 Suppression des non maximas

La dernière étape du processus de détection d'image est la suppression des non maximas. Cette dernière consiste à éliminer les redondances de détection comme illustré par la figure 11.

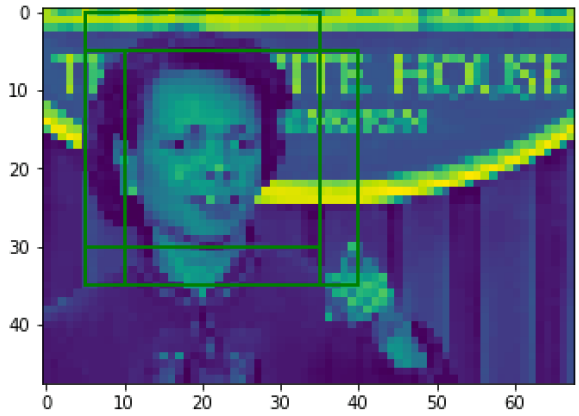

FIGURE 11: Exemple d'un même visage détecté plusieurs fois par l'algorithme de fênetre glissante

L'idée est donc de ne conserver que les carrés détectés avec les plus hauts scores et de supprimer les carrés redondants. Nous avons implémenté l'algorithme de la façon suivante :

Code Source 1.4: Implémentation de l'algorithme des nons maximas

```
# supprime les boites "non maximales" (facteur de recouvrement)
def suppressionNonMaximas(data, facteur=0.2):
    # Tri des boîtes par ordre décroissant de score
    data.view('i8,i8,i8,i8,i8')[::-1].sort(order=['f0'], axis=0)
    i = 0
    # Pour chaque boite dans la liste
    while (i < len(data)):
        scorei, xi, yi, wi, hi = data[i]
        # On compare aux boites suivantes
        for j in range(i+1, len(data)):
            if data[j][0] != 0:
                xj, yj, wj, hj = data[j][1:]
                # Si leur recouvrement est supérieur à 50% on considère que
                # c'est une boite redondante et on la supprime donc
                if recouvrement(xi,yi,wi,hi,xj,yj,wj,hj) > facteur:
                    data[j][0] = 0
        # On passe à la boîte suivante
        i+=1
    # Retourne les boites qui n'ont pas été éliminées (score != 0)
    return data[data[:,0] != 0]
```

3.3 Calcul et analyse des résultats

La fonction `calculResultats()` permet de mesurer notre classifieur aux les données de test. Son fonctionnement est simple : elle appelle notre algorithme de détection sur chacune des images d'un répertoire.

Cette fonction renvoie un array sous la forme spécifiée par le sujet du projet c'est à dire : `Numéro_image X Y Width Height Score`

Il est possible de spécifier l'intervalle des images à calculer via les paramètres `debut` et `fin` de la fonction. Cette fonctionnalité permet :

— de lancer plusieurs consoles IPython en même temps sur un même calcul (pour 1000 images, lancer une console sur les 500 premières, une autre sur les 500 d'après) et ainsi permettre à plusieurs cores CPU de travailler.
— de se prévenir d'éventuels bugs de Python, intervenant à la deuxième heure d'un calcul de sept heures, par exemple, en nous permettant d'obtenir des variables de résultats intermédiaires, véritable "checkpoints" de nos calculs.

La fonction `calculResultatsTrain()` permet quant à elle de confronter les résultats obtenus avec les données d'entraînement afin de déterminer, pour chaque détection, si cette dernière est erronée ou non (cf. figure 12.

La fonction `affichAnalyseResultat()` retourne les valeurs de précision et de rappel d'une détection sur les données d'entraînement.

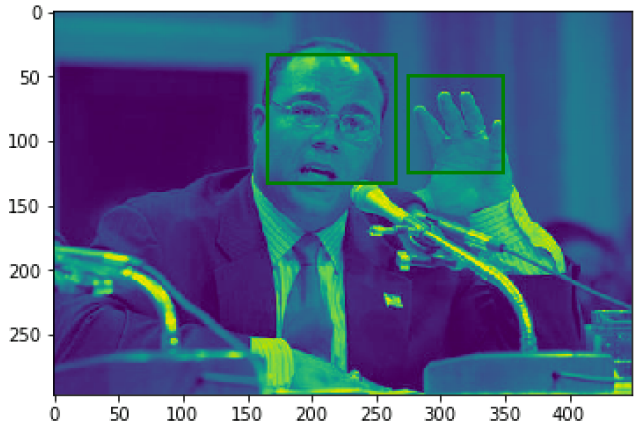

FIGURE 12: Exemple d'un vrai positif (à gauche) et d'un faux positif (à droite)

3.4 Apprentissage des faux positifs

Notre algorithme de détection touche à sa version finale. Nous pouvons néanmoins l'utiliser afin d'améliorer notre classifieur et lui faisant apprendre de ses erreurs :

`calculResultat()` **:** Nous lançons notre algorithme de détection de visage sur notre base de données.

`calculResultatsTrain()` Nous regardons les visages détectés (avec un score de validation $>$ `score`) par notre algorithme.

`fauxPositifs()`, `np.concatenate()` Nous ajoutons les images qui avaient été faussement détectées (recouvrement $<$ `facteur`) à la base de données d'apprentissages, ce qui permet d'améliorer notre classifieur.

En mesurant la performance du classifieur avant et après l'apprentissage des faux positifs, nous obtenons les chiffres suivants :

	Avant apprentissage des faux positifs	Après apprentissage score de validation > 1
Rappel	74,0 %	42,4 %
Précision	63,2 %	89,8 %

L'impact de l'apprentissage des faux positifs est clair : la précision augmente drastiquement, même si cela se fait au détriment du taux de rappel.

3.5 Courbe Précision / Rappel

La fonction `courbePrecisionRappel()` permet d'obtenir la courbe Précision / Rappel d'un calcul sur nos données d'entraînements. L'algorithme utilisé est celui décrit en cours, c'est à dire :

— Tri des données selon leur score
— Calculer précision et rappel pour la donnée avec le plus haut score
— Faire de même en ajoutant les données unes par unes, en procédant dans l'ordre décroissant de score.
— Afficher la courbe de la précision en fonction du rappel.

Avec une telle fonction, nous obtenons les courbes visibles à la figure 13. La tendance globale de la courbe confirme bien que plus le score est élevé plus on est assuré de la précision de notre mesure (cela se traduit par la courbe constante à la précision la plus élevée).

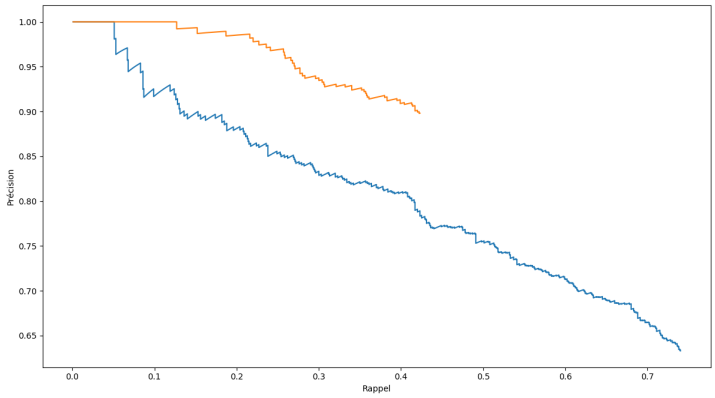

FIGURE 13: Le rappel en fonction de la précision. En bleu avant apprentissage des faux positifs, en orange après apprentissage des faux positifs.

4 Résultats

Adaboost Le fichier `res_adaboost.txt` contient les résultats que nous avons obtenu, avec une précision de 28.96%, un rappel de 23.71%, un score F1 de 26.08% ainsi qu'une précision moyenne de 10.17%.

SVM Avec $C = 7.1$, nous avons obtenu une précision de 59.20%, un rappel de 49.66%, un score F1 de 54.01% ainsi qu'une précision moyenne de 38.64%, ce qui est bien mieux qu'avec Adaboost. De plus, avec $C = 0.08$, nous avons obtenu une précision de 56.04%, un rappel de 48.77%, un score F1 de 52.15% ainsi qu'une précision moyenne de 37.46%, ce qui n'est pas mieux que précédemment.

5 Pistes d'amélioration du classifieur

5.1 Un visage par image ?

Afin d'améliorer les performances (notamment de rappel), il serait possible de forcer la détection d'un visage par image. Ainsi, pour chaque image, une seule "case" ne sera retenue, celle avec le plus grand score, indépendemment du fait qu'elle ait été reconnue ou non comme étant un visage ou non.

5.2 Restriction de la zone de détection

Comme nous l'avons vu en cours, en considérant qu'un visage est rond, il est possible de caractériser l'échelle d'un visage par le noyau qui maximise la réponse au Laplacien (détecteur de "blob"). De plus, on peut imaginer que l'on utilise successivement cette technique sur des parties restreintes de l'image afin de déterminer quelles parties de l'image contiennent des visages, on n'utiliserait alors la fenêtre glissante que dans ces zones-là, ce qui pourrait diminuer considérablement le temps de calcul.

5.3 Autres idées

Ajouter des exemples positifs et négatifs, augmenter la précision de la fenêtre glissante (diminuer les pas (x, y) et la variation du ratio), augmenter la taille du descripteur HoG, tester d'autres descripteurs (caractéristiques de Haar...), calculer la précision moyenne selon différents paramètres du classifieur... en revanche, tout cela demande bien évidemment un temps colossal de calcul...