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

Rapport de projet
Détection de visages

Auteur :
Emma Parois

Semestre :
Printemps 2020

1er mai 2020

Table des matières

1 Introduction

Ce rapport retrace la réflexion et les analyses qui ont accompagnées la construction d'un détecteur de visage. Ce projet a été réalisé dans le cadre de l'UV SY32 (Analyse et synthèse d'images) à l'Université de Technologie de Compiègne. 1000 images d'apprentissages annotées nous ont été fournies à partir desquelles un classifieur a été appris. Ce classifieur doit être capable de déterminer si l'extrait d'image qui lui est soumis est un visage ou non. Il s'agit d'une classification binaire : une classe correspond à celle des visages (1) et une à celle des non visage (-1). Le classifieur appris est ensuite utilisé sur 500 images de test, fournies aussi. Les détections ont été inscrites dans le fichier `detection.txt`. Ce détecteur a été construit à l'aide des librairies `scikit-image` et `scikit-learn`.

2 Construction des données d'apprentissage

La première étape du projet a été de mettre en forme les données d'apprentissage. En effet, les annotations étaient rectangulaires et de taille variable, pour simplifier l'apprentissage, il fallait les transformer en carrés. Cette première étape consistait donc en la création des données dites positives, c'est à dire contenant des visages. Ensuite, il fallait créer un grand nombre d'exemples négatifs.

2.1 Création des données positives

2.1.1 Transformation des annotations

Le fichier `label_train.txt` contient les annotations des visages des images d'entraînement. Pour transformer ces annotations rectangulaires en annotations carrées, j'ai d'abord choisi d'extraire des rectangles, un carré dont la longueur des côtés correspondait au minimum entre la hauteur et la largeur des rectangles. J'avais choisi le minimum car cela évitait de rencontrer des situations où le carré déborderait de l'image dans le cas d'un visage sur le bord. En visualisant les carrés extraits, tels que le carré vert de la figure 1, cette solution m'a paru être mauvaise car les visages étaient bien souvent incomplets.

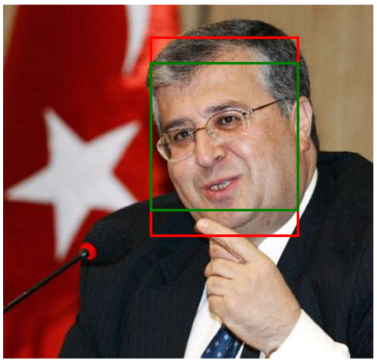

FIGURE 1 – Exemple de transformation d'une annotation rectangulaire (en rouge) en annotation carrée (en vert) en prenant le minimum entre hauteur et largeur du carré.

C'est pourquoi j'ai finalement opté pour le maximum entre la hauteur et la largeur des rectangles, en vérifiant que les carrés extraits ne sortaient pas de l'image. Le résultat est visible sur la figure 2.

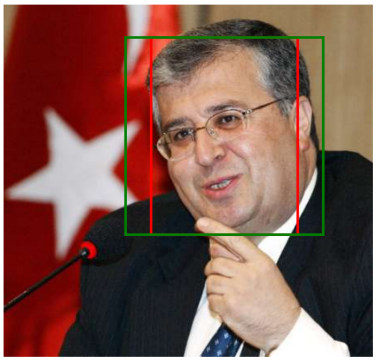

FIGURE 2 – Exemple de transformation d'annotation rectangulaire (en rouge) en annotation carrée (en vert) en prenant le maximum entre hauteur et largeur du carré.

Les boîtes carrées obtenues à l'aide du maximum coupaient bien souvent le menton. C'est pourquoi j'ai choisi d'élargir ces carrés de 6 pixels avant que les carrés soient mis à la taille "standard" évoquée dans le paragraphe suivant.
La partie 7 reviendra sur ce choix de méthode de transformation des annotations.

2.1.2 Choix d'une taille "standard" de boîte carrée

Afin que les carrés extraits correspondent à des tableaux de pixels de même dimension, j'ai choisi de redimensionner les carrés extraits en carrés de 48x48 pixels. La partie 4 reviendra sur ce choix de taille de fenêtre.

2.1.3 Augmentation du jeu de données positives

En observant la taille du jeu de données fourni dans le TD 3 : 3000 exemples positifs pour 12000 négatifs, j'ai constaté que je disposais de seulement 1284 exemples positifs pour les dizaines de milliers d'exemples négatifs que l'algorithme est en mesure de créer. C'est pourquoi, j'ai choisi d'augmenter le nombre d'exemples positifs en ajoutant les symétriques des visages et en appliquant une rotation d'un angle aléatoire compris entre 25 et -25 degré suivant l'axe vertical de l'image. La figure 3 illustre ces transformations. Cela permet donc d'obtenir 3852 exemples positifs.

FIGURE 3 – Exemple de transformations de boîtes. De gauche à droite : boîte originale, rotation aléatoire et symétrique.

Selon moi, un des risques liés à l'augmentation du jeu de données positives de cette façon est que le classifieur appris soit trop lié aux visages du jeu de données d'entraînement présent en triple dans l'ensemble d'entraînement. c'est pourquoi, je n'ai pas souhaité augmenter plus le jeu de données.

2.1.4 Résultats

À l'issu de cette étape de création du jeu de données positives on obtient donc :
— Un vecteur `X_train` contenant les visages en nuances de gris sous forme de carrés de taille 48x48.
— Un vecteur `y_train` contenant les classes des données de `X_train`, pour l'instant seulement des 1.
— Un tableau de labels carrés `squared_labels`, qui va permettre par la suite de manipuler les labels carrés.

2.2 Création des données négatives

Pour créer les exemples négatifs, j'ai choisi d'utiliser une fenêtre glissante. En effet, les possibilités offertes par `scikit-learn` ne me satisfaisaient pas car elles ne permettent pas de connaître les annotations des boîtes négatives extraites d'une image.

Le principe est simple : une fenêtre de taille fixe choisie dès la création des exemples positifs, comme l'explique le paragraphe 2.1.2, parcourt les images d'entraînement pour différentes échelles de l'image. Afin d'obtenir des exemples négatifs variés tout limitant le temps de calcul, les échelles choisies pour créer les exemples par fenêtre glissantes sont : 2 (double l'image), 1.5, 1, et 0.5 (diminue par deux l'image).

Les extraits d'images obtenus par fenêtre glissante sont ensuite soumis à un algorithme *Intersection Over Union* (IoU), pour vérifier que les boîtes extraites ne superposent pas trop les boîtes contenant des visages. En effet, on ne souhaite pas avoir des visages entiers dans les données négatives. C'est pourquoi, le seuil de recouvrement choisi à ne pas dépasser est de 0.3. J'ai dans un premier temps hésiter à modifier légèrement l'algorithme IoU pour qu'aucune boîte contenant un visage de loin n'apparaissent dans les données négatives. En effet, plusieurs cas de visages entiers se trouvaient dans les exemples négatifs malgré un seuil de 0.3, car la boîte négative englobait de loin la boîte positive contenant le visage. J'ai finalement laissé l'algorithme IoU original car j'avais peur d'avoir trop de faux positif sur des visages lointain dans les détections, si aucun exemple de ce type ne se trouvait dans le jeu de données d'apprentissage.

Le principal défaut de cette méthode de construction de l'ensemble des négatifs est que le nombre d'exemples obtenus est très important. Afin de limiter ce nombre, j'ai choisi de tirer 15 extraits aléatoirement parmi les extraits fournis par l'algorithme de fenêtre glissante pour chaque image et chaque échelle. Le choix de ce nombre sera évoqué dans la partie 4.

Pour ce qui est du pas du pas de la fenêtre glissante, j'ai choisi un très grand pas de 100 pixels. J'ai fait ce choix car une fenêtre glissante avec un pas petit met beaucoup de temps à s'exécuter, or seulement 15 extraits par image et par échelle étaient tirés au sort. Beaucoup de temps était donc perdu pour créer un grand nombre d'exemples négatifs pour au final n'en garder que 15 par échelle utilisée.

2.2.1 Résultats

À l'issu de cette étape, le jeu de données d'entraînement `X_train` est maintenant composé d'environ 51922 exemples positifs et négatifs confondus.

3 Apprentissages

Le jeu de données d'apprentissage sous forme de carrés en nuances de gris de taille fixe est maintenant créé. Il faut ensuite calculer les vecteurs descripteurs associés qui seront utilisés pour apprendre un premier classifieur. Une première détection est par la suite réalisée sur les images d'entraînement par fenêtre glissante, puis un deuxième classifieur qui fera office de classifieur final est appris.

3.1 Vecteurs descripteurs et classifieurs

Le descripteur choisi est le descripteur HOG. Les caractéristiques pseudo-Haar auraient aussi pu être choisies mais leur implémentation s'avère bien plus complexe. Le classifieur utilisé a été `sklearn.svm.SVC()` avec les paramètres suivants :

- `C = 1`
- `kernel = 'poly'`
- `degree = 4`
- `gamma = 0.1`

Le choix du classifieur et de ses paramètres sera détaillé dans la partie 4, une critique en sera faite dans la partie 7.

3.2 Première détection

Cette première détection se fait sur l'ensemble des images d'apprentissage et a pour but d'enrichir les exemples négatifs utilisés pour l'apprentissage du premier classifieur avec les faux positifs détectés.

J'utilise donc un algorithme de fenêtre glissante, chaque image est parcourue à différentes échelles. J'ai choisi 4 échelle différentes (1.25, 1, 0.75 et 0.5) car au de là le temps de calcul était inacceptable. Ces 4 échelles permettent néanmoins d'obtenir de faux positifs de bonne qualité tels que des morceaux de visage. À chacun des extraits obtenus par la fenêtre glissante, une classe est attribuée à l'aide de l'algorithme IoU. Le seuil de recouvrement est 0.4 afin d'éviter que des visages trop entier se retrouvent dans les données négatives mais que des morceaux de visage y apparaissent tout de même.

Le pas choisi pour la fenêtre glissante est cette fois très petit : 7 pixels. J'ai fait ce choix à force d'observations sur les images d'entraînement : un pas plus grand ne permettait pas de bonnes détections et un pas plus petit faisait augmenter considérablement le temps de calcul. Ce pas sera réutilisé dans l'algorithme de détection sur les images de test décrit dans la partie 6.

3.3 Résultats

À l'issu du premier apprentissage et de la première détection, le jeu de données `X_train` enrichi par les faux positifs contient maintenant 58837 extraits positifs et négatifs des 1000 images d'apprentissage. Ce jeu de données est utilisé pour apprendre un deuxième classifieur qui sera utilisé par `test.py` pour les détections sur les images de test.

3.4 Second apprentissage

Le second apprentissage est donc réalisé à l'aide du jeu de données enrichi par les faux positifs. La libraire `pickle5` est utilisée pour sauvegarder le classifieur dans un fichier afin que le script `test.py` puisse l'utiliser pour réaliser les détections sur les images de test.

Le classifieur utilisé pour ce second apprentissage est le même que celui utilisé pour le premier apprentissage (voir le paragraphe 3.1 pour les paramètres), la partie 4 reviendra sur la raison pour laquelle aucune évolution n'a été réalisée sur le classifieur entre les deux apprentissages.

4 Optimisation

L'optimisation s'est faite suivants plusieurs aspects :

1. Le choix de la taille de la boîte carrée "standard" et d'un classifieur
2. Le choix des paramètres du classifieur
3. Le choix de la taille du jeu de données d'apprentissage

Cette optimisation s'est faite par validation, non croisée, car le temps de calcul était trop important sinon. En effet, un essai de validation croisée à pris pas loin de 10h à s'exécuter entièrement.

La validation a été réalisée lors du premier apprentissage, c'est pour cette raison qu'aucune évolution n'a eu lieu entre le classifieur de premier apprentissage et second apprentissage. Cependant, le point 2 (choix des paramètres du classifieur), aurait plutôt dû avoir lieu lors du second apprentissage afin de correspondre au mieux au jeu de données complet.

Dans un premier temps, j'ai essayé d'affiner les paramètres du classifieur choisi lors du premier apprentissage puis de chercher plus précisément les paramètres lors du second apprentissage. J'ai malheureusement utilisé la méthode `sklearn.model_selection.GridSearchCV()` qui a rendu l'exécution du programme interminable : après 30 heures de calcul, dont 24 heures par cette méthode, j'ai dû stopper le programme. J'ai donc perdu beaucoup de temps et n'ai donc pas pu faire de validation lors du deuxième apprentissage. C'est pourquoi, la validation a été entièrement faite lors du premier apprentissage.

Le critère utilisée pour la validation a été soit l'erreur empirique, soit l'aire sous la courbe précision/rappel calculée par la méthode `sklearn.metrics.roc_auc_score()`.

4.1 Taille de la fenêtre et classifieur

Une première taille de fenêtre a été choisie de manière arbitraire pour démarrer le projet. Afin de déterminer quelle taille semblait être la plus optimale, j'ai réalisé une validation, non croisée pour des raisons de temps de calcul, sur des tailles de fenêtres 24, 32, 40, 48, 56, 64, 72, 80 et 100. Cette optimisation a été réalisée lors du premier apprentissage car j'ai considéré que les performances n'avaient pas besoin d'être maximisées pour déterminer ce paramètre. En effet, j'ai supposé que la taille qui allait se démarquée lors de cette optimisation aurait été la même si l'optimisation avait été réalisée lors de l'apprentissage du deuxième classifieur. De plus, cela permettait de réduire le temps de calcul.

En supposant que les performances des différents classifieurs, KNeighbors, Adaboost, RandomForest, DecisionTree et SVM pouvaient être influencées par les tailles de fenêtres, j'ai choisi de faire une validation sur les couples (taille, classifieur) et de choisir le plus performant. La mesure de la performance durant cette optimisation a été faite selon le taux d'erreur empirique. Pour des raisons de temps, je n'ai pas recommencer cette validation avec l'aire sur la courbe.

Un aperçu sous forme de diagrammes en barres, des résultats obtenus, est visible dans la figure 10 en annexe. La validation a été faite en utilisant 70% des données d'apprentissage pour apprendre les classifieurs et 30% des données pour réaliser les prédictions. Une seconde validation a ensuite été effectuée avec les tailles et classifieur que la première validation avait retenu, à savoir : 40, 48, 56, 64, 72 (à la place de 70, 2 pixels de différence étant peu signifiant) et 80 avec le classifieur SVM. Cette validation a été effectuée 5 fois afin d'obtenir une meilleur estimation. Les résultats obtenus sont retranscrits dans la figure 4. On observe que le taux d'erreur le plus faible se situe pour la taille de fenêtre de 48 pixels, c'est donc le couple (48, SVM) qui a été retenu. Le fait que ce classifieur soit retenu n'avait rien d'étonnant, ses performances avaient déjà été observées lors du TP 3.

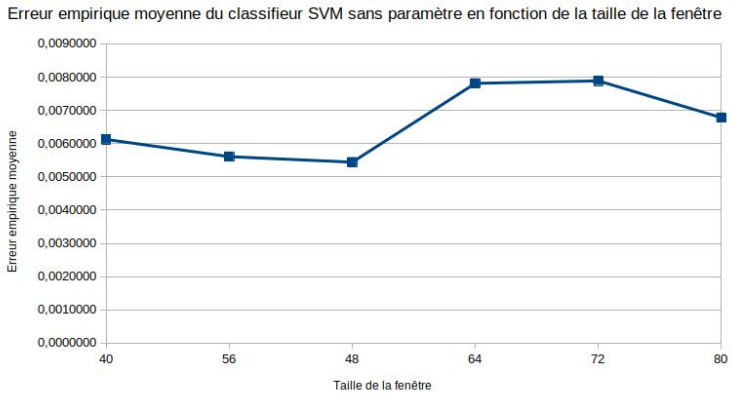

FIGURE 4 – Graphique des estimations du taux d'erreur empirique obtenues à l'issu de 5 validations avec les tailles de fenêtres 40, 48, 56, 64, 72 et 80 et le classifieur SVM.

4.2 Paramètres du classifieur SVM

Le classifieur `sklearn.svm.SVC()` possèdent 4 principaux paramètres sur lesquels je me suis focalisée : `C`, `kernel`, `degree` (si le noyau est polynomial), et `gamma`. Afin de ne pas avoir à chercher `C`, `gamma` et `degree` dans l'espace des paramètres entiers, j'ai choisi de les restreindre aux valeurs suivantes :

- `C` $\in \{0.1, 1, 5, 10, 50\}$
- `gamma` $\in \{0.001, 0.01, 0.1, 1\}$ Afin d'éviter le surapprentissage j'ai choisi de tester des gamma assez bas.
- `degree` $\in \{2, 3, 4, 5\}$ Afin de ne pas trop complexifier le classifieur et ne pas faire de surapprentissage, j'ai choisi de ne pas tester de degrés inférieurs à 5.
- `kernel` $\in \{$`linear`, `poly`, `rbf`, `sigmoid`$\}$

Comme mentionné en introduction de cette partie, j'ai d'abord choisi de réaliser cette validation lors du premier apprentissage afin que les calculs ne prennent pas trop de temps.

Cette première validation, basée sur l'aire sous la courbe précision/rappel retournée par `sklearn.metrics.roc_auc_score()`, a fait ressortir les paramètres suivants :

$$\texttt{C} = 0.1, \texttt{degree} = 5, \texttt{gamma} = 1, \texttt{kernel} = \texttt{poly}$$
$$\texttt{C} = 10, \texttt{gamma} = 0.1, \texttt{kernel} = \texttt{rbf}$$

La figure 5 représente l'aire sous la courbe pour les différents noyaux et donne leur paramètres optimaux.

J'ai par la suite réalisé une validation avec la méthode `sklearn.model_selection.GridSearchCV()` qui permet de tester les combinaisons possibles à l'aide de la validation croisée. Cette méthode à l'avantage de retourner un classifieur et ses paramètres optimaux. Cette méthode nécessite cependant un temps de calcul très long. J'ai d'abord utilisée cette méthode lors du premier apprentissage, en complément de la validation déjà réalisée. Elle m'a retourné les paramètres suivants :

$$\texttt{C} = 1, \texttt{degree} = 4, \texttt{gamma} = 0.1, \texttt{kernel} = \texttt{poly}$$

Ce sont les paramètres qui ont été fournis au classifieur utilisée pour les premières détections mentionnées dans la partie 3.2 et pour les détections sur les images de test décrites dans la partie 6.

Comme expliqué dans l'introduction de cette partie, j'ai ensuite tenté d'utiliser cette méthode `GridSearchCV()` en guise de validation lors du deuxième apprentissage, qui s'est soldée par un échec à cause du temps de calcul (24 heures).

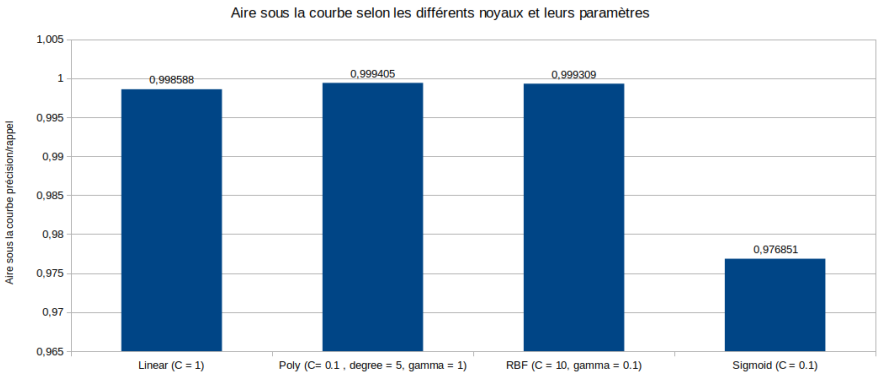

FIGURE 5 – Diagramme en barres de l'aire sous la courbe/précision rappel, précisant les kernels et les paramètres maximisant l'aire.

4.3 Taille du jeu de données d'entraînement et ratio d'exemples positifs et négatifs

J'ai choisi de faire une validation sur la taille du jeu de données afin d'avoir un ordre d'idée de la taille optimale de l'ensemble à fournir au classifieur choisi dans le paragraphe précédent. Les validations réalisées précédemment ont été faites avec un jeu de données de faible taille, cette étape m'a donc permis de voir s'il était possible d'optimiser un peu plus les résultats. La figure rend compte des résultats de cette validation. On observe que la taille optimale, pour le classifieur choisi dans le paragraphe précédent, devrait se trouver aux alentours de 53000 données, ce qui est très supérieur au premier choix que j'avais fait

(30000 données).

Les résultats de cette validation sont très approximatifs car la proportion d'exemples positifs et négatifs n'était pas égale dans tous les jeux de données testés car la méthode `sklearn.model_selection.train_test_split()` a été utilisée pour séparer l'ensemble.

La façon dont j'ai réalisé la validation est la suivante : je disposais de 3852 exemples positifs. Je ne voulais pas faire plus que tripler le jeu de données positives de peur que le classifieur s'adapte de façon trop précise aux visages utilisés pour l'entraîner, 3852 était donc la taille maximum du jeu de données positives. J'ai donc choisi de faire varier la taille du jeu de données négatives, la proportion de données positives variait donc en même temps que la taille de l'ensemble d'entraînement grossissait. Le taux d'erreurs minimum observable sur la figure 6 a été réalisé pour un ensemble de taille 53178 et une proportion de positif s'élevant à 0.06 environ. C'est pourquoi, j'ai opté pour 54985 exemples négatifs pour 3852 exemples positifs, soit une proportion de 0.065 positifs. Obtenir 54985 exemples négatifs a l'issu du premier apprentissage (première détection inclue) a été possible en tirant aléatoirement 15 données par images et par échelles, cela explique donc ce choix mentionné dans la partie 2.2.

Cette validation était peu rigoureuse mais a tout de même permis de donner un ordre de grandeur important quant à la taille du jeu de données d'apprentissage. Elle s'est avérée d'autant plus importante que le classifieur choisi dans le paragraphe précédent est un classifieur complexe et sensible aux données d'entraînement donc nécessitant un jeu de données d'entraînement de taille importante.

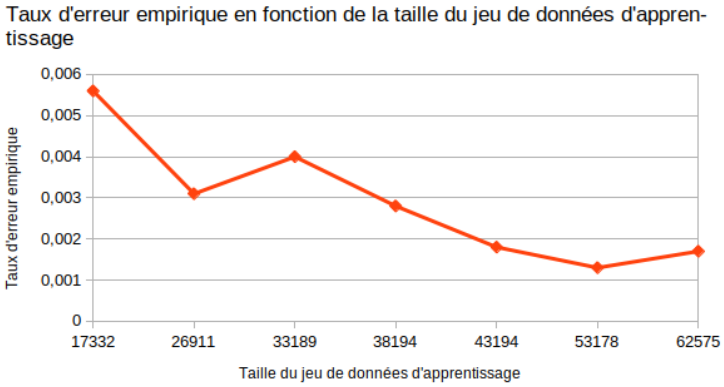

FIGURE 6 – Graphique des estimations du taux d'erreur empirique obtenues à l'issu d'une validation pour des jeux de données de tailles : 17332, 26911, 33189, 38194, 43194, 53178 et 62575.

5 Estimation de la performance du classifieur

Avant de tester le classifieur sur les données de test, il vaut mieux faire une validation de celui-ci sur les images d'apprentissage pour savoir s'il est utile de revenir à une des étapes précédentes. J'ai donc appris un classifieur sur les 700 premières images d'apprentissage que j'ai ensuite testé sur les 300 dernières images d'apprentissage. Le code utilisé pour réaliser cette estimation est disponible en annexe. Les résultats sont les suivants :

— **Score F1 : 84.93**
— **Aire sous la courbe précision / rappel : 97.16**

La figure 7 représente la courbe précision/rappel. Elle est plutôt encourageante quant aux résultats que l'on pourrait obtenir sur les images de test. Si j'avais eu plus de temps à ma disposition j'aurais probablement tenté de l'améliorer encore.

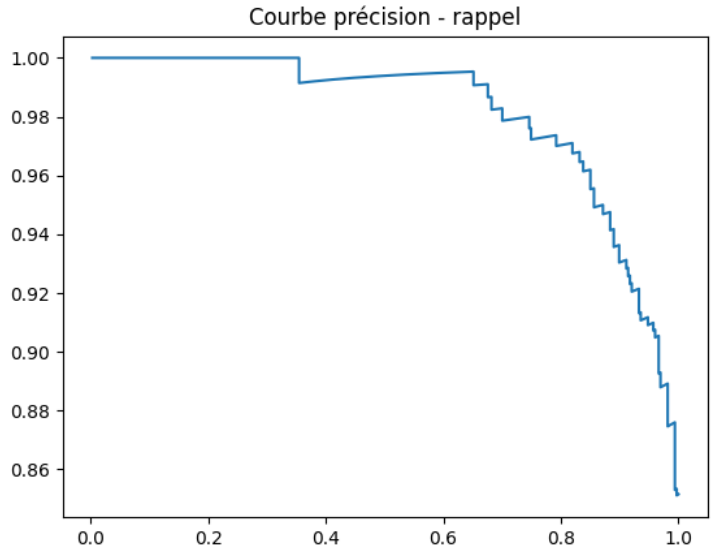

FIGURE 7 – Courbe précision/rappel obtenues à l'aide des détections sur 300 données d'apprentissage.

En observant les résultats des détections sur ces 300 images d'apprentissage, j'ai remarqué que les gros visages, allongés et cadrés de près, comme celui de la figure 8, n'étaient pas détectées. La partie 7 reviendra sur ce point important et ce qui aurait pu être fait pour l'éviter.

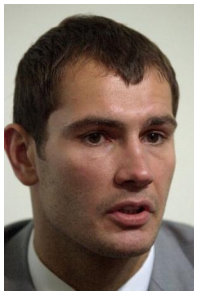

FIGURE 8 – Exemple de visage non détecté.

6 Détection sur les données de test

J'ai choisi d'utiliser une fenêtre glissante pour parcourir les images de test à différentes échelles et en extraire des boîtes à soumettre à la prédiction du classifieur. La discrétisation

de l'échelle pour parcourir les images est ici plus fine que celle mentionnée dans les parties 2.2 et 3.2, pour construire le jeu de données négatives. En effet, pour espérer détecter des visages de toutes tailles, il vaut mieux plus discrétiser l'échelle. De plus, en testant mon classifieur sur les données d'apprentissage, j'ai pu observer que les visage très gros n'étaient pas détectés. C'est pourquoi, j'ai choisi d'élargir l'intervalle contenant les échelles utilisées pour la fenêtre glissante. J'avais initialement choisi les échelles : 2, 1.5, 1.25, 1, 0.75 et 0.5 mais j'ai finalement opté pour : 1.25, 1, 0.75, 0.5, 0.4, 0.35, 0.30, 0.35, 0.2, 0.15, 0.1, 0.05. Cette discrétisation fait un compromis entre temps de calcul (très important pour les échelles supérieures à 1), et détection des visages de tailles variées. Il fallait cependant faire attention a ce que l'image mise a échelle ne devienne pas plus petite que la fenêtre glissante de 48 pixels.

Le pas choisi pour la fenêtre glissante est de 7 pixels, j'ai choisi de garder ce pas suite à des observations faites sur les données d'entraînement comme l'explique le paragraphe 3.2.

Pour calculer les scores des détections, j'utilise la méthode `decision_function` du classifieur SVM de `scikit-learn`. Le score retourné peut être négatif selon le coté de l'hyperplan où se trouve la donnée, c'est pourquoi j'ai choisi de remettre les scores en valeur absolue pour les manipuler plus facilement.

Pour supprimer les non-maxima, j'ai choisi un seuil de recouvrement de 0.5 que l'algorithme IoU doit dépasser pour que la boîte de plus faible score soit supprimer des détections. Il s'agit d'un seuil assez élevé car aucune détection ne doit figurer en double dans les résultats.

7 Problèmes et améliorations possibles

7.1 Transformation des annotations et qualité du jeu de données d'apprentissage

Comme mentionné dans la partie 5, les visages longs et cadrés de près ne sont pas détectés. J'ai débord pensé que cela venait des échelles choisies qui ne permettaient pas de "zoomer" assez sur les images pour détecter les gros visages. Le problème ne venait pas des échelles mais de la qualité des exemples d'apprentissage positifs. Deux options auraient probablement pu permettre de résoudre ce problème :

— Garder une forme de boîte et fenêtre glissante carrées mais faire en sorte que les exemples positifs soient plus près des visages, à la manière de l'illustration 9. Une boîte carrée associée à des exemples positifs plus "zoomés" auraient permis de réduire le nombre de longs visages, mais probablement pas d'éliminer le phénomènes.

— Une autre solution aurait été d'opter pour de boîtes rectangles. En effet, les visages humains ne sont pas carrés mais plutôt rectangles. De telles boîtes auraient sûrement permis de non seulement réduire, mais d'éliminer ces faux négatifs.

FIGURE 9 – Exemple de transformation de visage en boîtes carrés plus "zoomées".

7.2 Taille du jeu de données d'apprentissage et classifieur

Le classifieur que j'ai choisi est un classifieur complexe (SVM polynômial), dépendant de plusieurs paramètres et qui a tendance à trop correspondre au jeu de données d'apprentissage, notamment si le degré du polynôme augmente. Heureusement, j'avais choisi de ne pas aller au de là du degré 5 lors de la validation. Cependant, pour que ce classifieur soit plus performant, je pense que j'aurais dû grossir le jeu de données d'apprentissage en y ajoutant plus d'exemples positifs et négatifs. Ainsi, ce classifieur complexe ce serait moins adapté au jeu de données et aurait permis de limiter le surapprentissage.

7.3 Discrétisation de l'échelle

En observant les détections sur les données d'apprentissage, j'ai remarqué que certains visages étaient détectés à deux reprises sans que l'algorithme IoU ne permette de supprimer la détection de score non maximale. Cela est probablement dû à deux choses : la qualité du jeu de données, déjà commentée dans le paragraphe 7.1, et la discrétisation des échelles. Une meilleur discrétisation aurait probablement pu éviter ce phénomène, il aurait cependant fallut faire un compromis entre limiter les doubles détections de visages et détecter les visages de toutes tailles. La meilleur solution aurait probablement été de revoir la qualité du jeu de données positives.

7.4 Optimisation

L'optimisation réalisée a, selon moi, été de mauvaise qualité pour plusieurs raisons :

1. Les validations n'étaient pas croisées et dans certains cas n'ont pas été répétées plus d'une fois. Les résultats obtenus sont donc peu représentatifs de la réalité.
2. Les validations répétées n fois ont été faites sur un jeu de données séparées n fois à l'aide de la méthode `sklearn.model_selection.train_test_split()`. Les n séparations ont donc aboutis à des jeux de données peu éloignés les uns des autres. Les résultats ont donc dû être peu représentatifs.
3. Les validations ont été réalisées lors du premier apprentissages et ne tiennent pas compte des nouvelles données négatives ajoutées lors de la première détection avec le premier classifieur.

De meilleurs solutions pour réaliser l'optimisation auraient été :
— Plutôt que de séparer un jeu de données fixe, il aurait fallut séparer l'ensemble des images d'apprentissages. L'apprentissage aurait été fait sur une partie et la détections sur l'autre partie à l'aide d'une fenêtre glissante, un peu à la manière dont l'estimation des performances du classifeur est réalisée, comme expliqué dans la partie 5. Cependant, une telle solution aurait démultiplié le temps de calcul et d'apprentissage.
— Réaliser des validations croisées sur le jeu de donnés.

8 Instructions d'exécution

Vous devez trouver dans l'archive :
— Le fichier `train.py` : apprend un classifieur. Ce fichier contient aussi quelques méthodes qui ont été utilisées pour l'optimisation mais qui ne seront pas appelées si le script est exécuté tel qu'il est.

— Le fichier `test.py`
— Le fichier `svm_model.sav` qui contient le classifieur décrit dans ce rapport et qui a permis d'obtenir le fichier `detection.txt` soumis sur le site.
— Le fichier `detection.txt`
— Ce rapport.

Pour lancer le script `train.py`, il faut vous assurer que le fichier figure dans le même répertoire que le dossier `project_train`. S'il n'est pas possible de les mettre dans le même répertoire, il faut modifier la variable `path` située en bas du fichier, après les méthodes. Cette variable doit contenir le chemin de dossier où se trouve les images d'apprentissage de la forme `0001.jpg`.

Pour lancer le script `test.py`, il faut vous assurer que le fichier figure dans le même répertoire que `svm_model.sav` et que le dossier `project_test`. S'il n'est pas possible de les mettre dans le même répertoire, il faut modifier la variable `path` située en bas du fichier, après les méthodes. Cette variable doit contenir le chemin de dossier où se trouve les images de test de la forme `0001.jpg`.

Pour lancer les différents scripts, il faut disposer des librairies suivantes :
— `numpy`
— `scikit-image`
— `scikit-learn`
— `matplotlib.pyplot`, qui est utilisé pour tracer quelques figures lors de la phase d'optimisation.
— `pickle5`, qui est utilisé par `train.py` pour sauvegarder le classifieur dans un fichier et par `test.py` pour charger le fichier du classifieur.

9 Conclusion

Les résultats du classifieur sur les images de tests sont assez proches de ceux estimés dans la partie 5. Ses performances ne sont donc pas trop mauvaises mais largement améliorables, notamment si les gros visages étaient mieux détectés.

La démarche globale est elle aussi améliorable, mais je suis tout de même satisfaite des résultats obtenus.

Annexes

1 Résultats obtenus suite à la validation pour déterminer le couple (taille de fenêtre, classifieur)

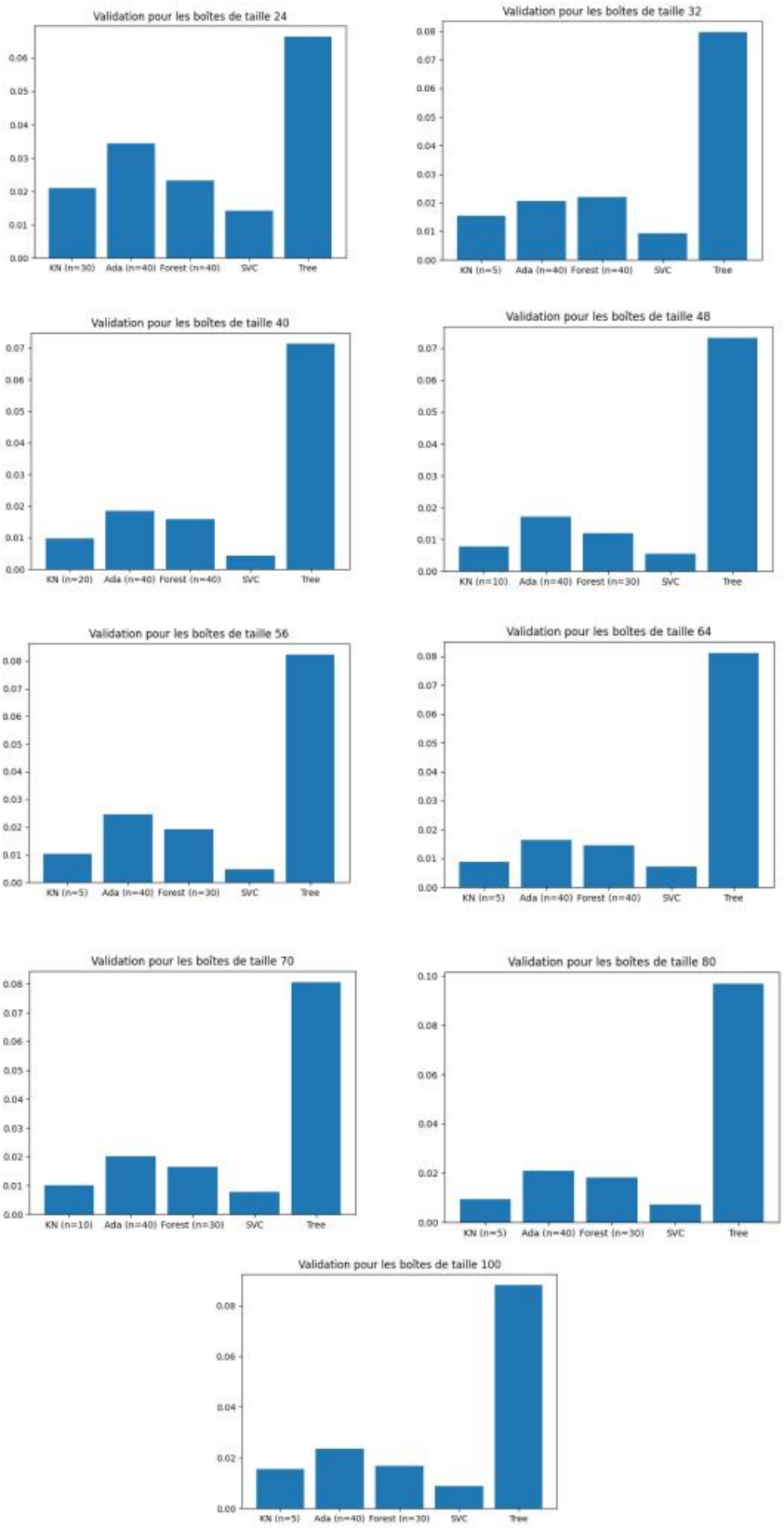

FIGURE 10 – Histogrammes des estimations obtenues à l'issu de la validation avec les tailles de fenêtres 24, 32, 40, 48, 56, 64, 70, 80 et 100 et les différents classifieurs

2 Code utilisé pour réaliser une estimation des performances du classifieur

```
# -*- coding: utf-8 -*-

import numpy as np
from skimage import io, util
import matplotlib.pyplot as plt

results = np.loadtxt('detection_val.txt')
labels = np.loadtxt('./project_train/label_train.txt')
labels = labels[labels[:,0] >= 700]
squared_labels = np.zeros(labels.shape)

k = 0
for img_idx in range(700, 1000):
    image = io.imread('./project_train/train/%04d.jpg'%(img_idx+1))
    # Keep only labels related to img_idx
    labels_filter = labels[labels[:,0] == (img_idx + 1)]
    for label in labels_filter:
        Ni, Nj = label[3:5]
        i, j = label[1:3]
        taille = int(max(Ni+6, Nj+6))
        i_prime = min(max(i-2 + Ni/2 - taille/2, 0), image.shape[0])
        j_prime = min(max(j-2 + Nj/2 - taille/2, 0), image.shape[1])
        squared_labels[k] = [(img_idx+1), i_prime, j_prime, taille, taille]
        k += 1


def IoU(box1, box2):
    i1, j1, h1, l1 = box1[1:5]
    i2, j2, h2, l2 = box2[1:5]

    l_intersection = min(j1 + l1, j2 + l2) - max(j1, j2)
    h_intersection = min(i1 + h1, i2 + h2) - max(i1, i2)

    if l_intersection <= 0 or h_intersection <= 0:
        return 0

    I = l_intersection * h_intersection

    U = l1 * h1 + l2 * h2 - I

    return I / U

print('results shape', results.shape)
print('squared_labels shape', squared_labels.shape)
# Order scores
results = np.array(sorted(results, key = lambda box: box[5], reverse = True
    ))

results_VP = []
results_FP = []
# Add a column to check if the box has been detected and to decide whether
    it's a VP or a FP
checking_column = np.zeros(squared_labels.shape[0], dtype = bool)
squared_labels = np.c_[squared_labels, checking_column]
# Add a column to classify detections : -1 or +1
checking_column = np.zeros(results.shape[0], dtype = bool)
```

```
results = np.c_[results, checking_column]

for box in results:
    for box_face in squared_labels:
        if box[0] == box_face[0]:
            if IoU(box, box_face) > 0.5:
                if box_face[5]:
                    box[6] = -1
                else:
                    results_VP.append(box)
                    box_face[5] = True
                    box[6] = 1


results_VP = np.array(results_VP)
VP = results_VP.shape[0]
FP = results.shape[0] - VP
print('FP', FP)
print('VP', VP)
results_FN = squared_labels[squared_labels[:,5] == False]
FN = results_FN.shape[0]
print('FN', FN)

precision = VP / (VP + FP)
recall = VP / (VP + FN)
F1 = 2 * (precision*recall)/(precision+recall)
print('F1', F1)

plot_x = []
plot_y = []
FN = VP
VP = 0
FP = 0
for box in results:
    if box[6] > 0:
        VP += 1
        FN -= 1
    else:
        FP += 1
    precision = VP / (VP + FP)
    recall = VP / (VP + FN)
    plot_x.append(recall)
    plot_y.append(precision)

plt.plot(plot_x, plot_y)
plt.title("Precision - recall curve")
plt.show()

area = 100*np.sum(plot_y)/len(plot_x)
print('Area', area)
```

Listing 1 – Estimation des performances