École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique de Polytech Tours

Rapport de stage 2019 - 2020

Stage détection de duplicata pour le CBIR TV / VoD

Confidentiel

Entreprise

Laboratoire d'Informatique de
l'Université de Tours (LIFAT)
60 Rue du Plat d'Étain
37000 Tours, France

Tuteur entreprise

Mathieu Delalandre
Maître de conférences
[mathieu.delalandre@univ-tours.fr]

Étudiant

Tom Suchel
4A Département Informatique
[tom.suchel@etu.univ-tours.fr]

Tuteur académique

Christophe Lenté
[christophe.lente@univ-tours.fr]

Version du 23 août 2020

Table des matières

Table des figures

1. Cadre du stage

1 L'entreprise

ToddTV est un projet de start-up co-fondée en 2019 par Jordan Nicot, Ingénieur Informatique, et Mathieu Delalandre, maître de conférence et chercheur au sein du Laboratoire d'Informatique de l'Université de Tours (LIFAT). Ce projet s'inscrit dans un contexte d'innovation autour des domaines des médias, des technologies du web et du traitement d'images. La start-up veut répondre à plusieurs problématiques, comme l'explosion du contenu vidéo aujourd'hui disponible, que ce soit à la télévision ou sur internet (VoD, Replay ou Streaming), ou encore le changement du comportement des utilisateurs, qui naviguent de plus en plus souvent sur plusieurs appareils en même temps à la recherche de contenu.

L'objectif de ToddTV est d'offrir à ses utilisateurs un portail Web donnant accès à de nombreux services numériques autour de la TV/VoD, avec un guide multi-plateformes de qualité, adapté aux goûts et aux habitudes de l'utilisateur. Ce guide aura pour objectif de fournir une expérience de visionnage intelligente, incluant notamment des fonctionnalités de zapping automatique entre les programmes, et un accès direct à des médias ou réseaux sociaux en lien avec le programme en cours.

2 Le projet

À la suite de plusieurs études de marché engagées au début du projet ToddTV, Jordan et Mathieu ont pu définir le périmètre du projet et isoler les fonctionnalités à proposer aux utilisateurs. L'application a pour but de constituer un portail vers différents contenus vidéos, qui resteront exclusifs aux plateformes qui les diffusent.

Face à la quantité de contenu vidéo aujourd'hui disponible, l'objectif est de se démarquer des autres sites de programmes TV/VoD en fournissant un contenu de qualité, avec de très nombreuses informations additionnelles (images, bande-annonces, casting...) qui permettront à l'utilisateur de choisir un programme rapidement et efficacement. De plus, ToddTV souhaite informer le téléspectateur de la diffusion des programmes en temps-réel, et pas seulement sur des plages temporelles larges comme c'est le cas aujourd'hui. Cela passera donc par l'identification automatique des temps publicitaires, des démarrages et des coupures de programmes.

Ainsi, l'application agira comme un "assistant télévisuel", qui donnera plus de contrôle à l'utilisateur lorsqu'il regarde la télévision. L'objectif est de fournir à celui-ci des informations supplémentaires lorsqu'il regarde un programme, ou de lui donner plusieurs façon de gérer la publicité, via des alarmes à la fin de la pause publicitaire, une redirection automatique vers d'autres chaînes ou encore l'affichage de contenu lié à la publicité sur un second écran.

3 Le sujet du stage

Le but de mon stage était de mettre en place un système de détection de duplicatas d'images, robuste aux changements de formats, d'échelle, de contraste ou de luminosité. Ce sujet aborde donc le domaine de l'image matching, et de différents algorithmes utilisés pour comparer deux images entre elles : SAD, SSD, NCC... L'objectif était ensuite d'optimiser ce système en utilisant plusieurs techniques (vectorisation, calcul offline, parallélisation) afin d'atteindre une vitesse de calcul suffisante pour traiter des gros volumes de données. À terme, ce système s'inscrit dans un processus de développement d'un *scrapper web* intelligent, qui permettrait de récolter automatiquement des images de qualité dans des sites des programme TV sans avoir à étudier leur structure au préalable.

Ce stage était assez orienté vers le thème de la recherche, car certaines techniques d'image matching et d'optimisation à aborder sont encore assez récentes et encore en évolution. Il était donc plus question de faire un état de lieux des technologies existantes et de réaliser un prototype fonctionnel, plutôt que de développer une application totalement opérationnelle. De ce fait, il était parfois difficile de prévoir le temps que prendraient les différents points à aborder, ni les résultats offerts par ceux-ci. Les objectifs ont donc évolué au fur et à mesure du stage, et certains nouveaux points ont été abordés tandis que d'autres ont été abandonnés.

2. Organisation du travail

1 Déroulement général

Ce stage s'est déroulé dans le contexte très particulier de la crise sanitaire liée au Covid-19, et de ce fait, l'organisation générale du travail a dû être totalement adaptée. Une des conséquences directe de la crise sanitaire a été que le stage a eu lieu entièrement en télétravail, ce qui était une expérience nouvelle pour moi.

J'ai donc réalisé ce stage depuis chez moi, en travaillant sur mon matériel personnel : un ordinateur fixe dans mon logement étudiant, et un ordinateur portable que j'emportais en déplacement. Le fait de travailler sur mon ordinateur portable, peu puissant, a pu quelque fois ralentir l'exécution de certains de mes programmes, mais ne m'a pas handicapé dans l'avancée du projet.

Concernant le déroulé de mon stage, j'ai passé tout le premier mois à prendre en main la technologie et à comprendre le fonctionnement des différents algorithmes d'image matching, en utilisant le rapport de projet R&D réalisé par Naifen Gan l'année précédente, ainsi que plusieurs ressources fournies par M. Delalandre. Une fois les premiers prototypes réalisés, j'ai abordé une phase d'évaluation des performances, afin d'estimer la précision de chaque algorithme. Les trois semaines qui ont suivi ont été utilisées pour optimiser mon implémentation afin de réaliser des tests sur des grandes bases d'images et ainsi réaliser une évaluation des performances plus fiable. Enfin, la dernière semaine de stage a été consacrée à la mise en oeuvre du système de détection de duplicatas dans la démarche finale du projet, c'est à dire pour extraire des règles de structuration de sites web inconnus.

2 Comptes-rendus et communication

Le contexte de télétravail nous a poussé à repenser la façon de communiquer avec les différents membres du projet. Ainsi, nous avons utilisé l'application Microsoft Teams, que ce soit pour communiquer de manière individuelle avec M. Delalandre, ou avec toute l'équipe de ToddTV.

J'ai pu donc maintenir une communication constante avec mon tuteur professionnel, pour faire des points sur mon avancée et sur les prochains objectifs à aborder (environ une ou deux fois par semaine), ou tout simplement dès que je faisais face à une difficulté lors du développement. Nous avons pu aussi organiser des réunions collectives toutes les deux semaines avec tous les membres de l'équipe de ToddTV. À la manière d'un weekly, chaque stagiaire y résumait en trois minutes environ ses avancées, les difficultés rencontrées et les objectifs à venir pour les deux semaines à venir. Ces réunions me semblaient très importantes car elles me permettaient d'avoir une vision d'ensemble du projet de ToddTV et de mieux appréhender le sens de mon stage au sein de l'entreprise.

Lors de mes trois dernières semaines de stage, j'ai eu l'occasion de travailler en collaboration avec deux autres stagiaires, Jean-Baptiste Huyghe et Jolan Odiot, qui travaillaient respectivement sur l'élaboration d'un crawler d'images performant et sur la récupération automatique de métadonnées. Nos trois stages, bien que très différents, s'inscrivaient dans le même objectif de récolte automatique de données (images et textes) sur les sites de programme TV. Ainsi, j'ai dû communiquer souvent avec Jean-Baptiste pour comprendre le fonctionnement de son crawler, que j'ai utilisé pour récupérer des images provenant de sites web qui n'étaient pas couverts par le crawler de Louis. J'ai aussi été en contact avec Jolan pour discuter et partager des méthodes d'évaluation des performances, nos deux programmes étant très similaires sur ce point.

3 Outils

Comme précisé précédemment, ce stage était plus orienté vers la recherche que vers la production d'un programme abouti, et j'ai donc dû travailler avec un langage de prototypage. Les deux options qui se proposaient à moi étaient le Python et le Matlab, deux langages haut niveaux qui permettent une programmation rapide et aisée grâce à un typage dynamique et à de très nombreuses librairies. J'ai choisi de travailler tout au long de mon stage avec Python, un langage que je ne maîtrisais qu'en surface mais qui m'intéressait fortement de par l'éventail de domaines qu'il couvrait (calcul scientifique, machine learning, programmation web, interfaces utilisateurs...).

Python est un langage de programmation interprété, conçu pour optimiser la productivité de ses utilisateurs en offrant des outils de haut niveau et une syntaxe simple à utiliser. C'est un langage assez ancien qui possède donc une très grande communautée et un nombre important de librairies (pour ne citer que celles que j'ai utilisé : Numpy pour le calcul matriciel, Skimage pour la manipulation d'images, Matplotlib pour le dessin de graphiques...). Cependant, Python a aussi quelques inconvénients, dont notamment sa faible vitesse d'exécution comparée à d'autres langages bas niveau.

Concernant mon environnement de développement, j'ai commencé par travailler sur Jupyter Notebook pour me re-familiariser avec Python et pour appréhender les différents algorithmes d'image matching. Jupyter Notebook est un environnement de développement basé sur IPython, qui prend la forme d'une application web interactive. L'avantage de cet IDE est qu'il permet d'associer code et prise de notes au sein d'un même document (un *notebook*) et d'exécuter des "cellules" de code une par une en conservant le résultat des cellules précédentes. Ces fonctionnalités m'ont permis de prototyper les premières versions du système de reconnaissance de duplicatas, et de lancer une première phase d'évaluation.

Au bout d'un mois de stage, j'ai attaqué la phase d'optimisation de mon programme en travaillant sur Pycharm, un IDE qui propose des fonctionnalités de débogage plus poussées que sur Jupyter Notebook. J'ai eu aussi à utiliser occasionnellement Eclipse pour lancer les crawlers réalisés en Java par Louis Babuchon et Jean-Baptiste Huyghe, ainsi que Atom pour manipuler les fichiers de configuration de ces crawlers et les fichiers de métadonnées, enregistrés au format JSON.

3. Analyse du problème

1 Introduction à la notion de duplicatas

La reconnaissance de duplicatas d'images consiste simplement à détecter si deux images sont identiques ou non. On peut cependant différencier deux types d'images "identiques" :

- Les quasi-doublons, qui représentent visuellement le même sujet, mais qui présentent des différences dans les conditions de prise de la photo (perspective, luminosité de la scène...). Ces deux photos sembleront donc identiques pour un humain, mais auront énormément de différences du point de vue des données informatiques. Les quasi-doublons sont donc bien plus difficiles à détecter.

- Les doublons identiques, qui représentent deux images du même sujet, provenant exactement de la même source, mais qui auraient subi des traitements au cours du temps. Ces traitements peuvent être de plusieurs natures :

  - **Réduction d'échelle** : il est courant qu'un site web affiche toutes ses images avec une taille très précise. Ceux-ci doivent alors redimensionner les images afin qu'elles rentrent dans le cadre imposé, tout en économisant de l'espace de stockage. Les images redimensionnées sont similaires aux originales, mais ont cependant "perdu" de l'information.

  - **Changement de l'espace colorimétrique** : de nos jours, la plus grande majorité des images affichées sur Internet utilisent l'espace colorimétrique RGB (Red/Green/Blue). Cependant, d'autres formats existent (HSV pour Hue/Saturation/Value, ou encore CMYK pour Cyan/Magenta/Yellow/Black) et il arrive que certains sites changent l'espace de couleur de ses images. Les images sembleront encore identiques à vue d'oeil, mais présenteront des différences au point de vue informatique.

  - **Changement d'illumination ou de contraste** : certains sites retouchent automatiquement les images en modifiant légèrement leur contraste pour l'illumination pour faire ressortir le sujet principal.

  - **Compression** : celle-ci est utilisée pour enregistrer les images dans un format optimal, qui permet de diminuer l'espace utilisé par l'image sans pour autant diminuer sa qualité. Il existe des dizaines de formats de compression (JPEG, PNG, BMP, WebP...) qui, à partir d'une même image, donneront tous une image similaire mais différente au niveau des données.

Les doublons identiques sont très courants sur Internet et la détection automatiques de ceux-ci est aujourd'hui extrêmement courante (par exemple pour détecter des images protégées par le droit d'auteur).

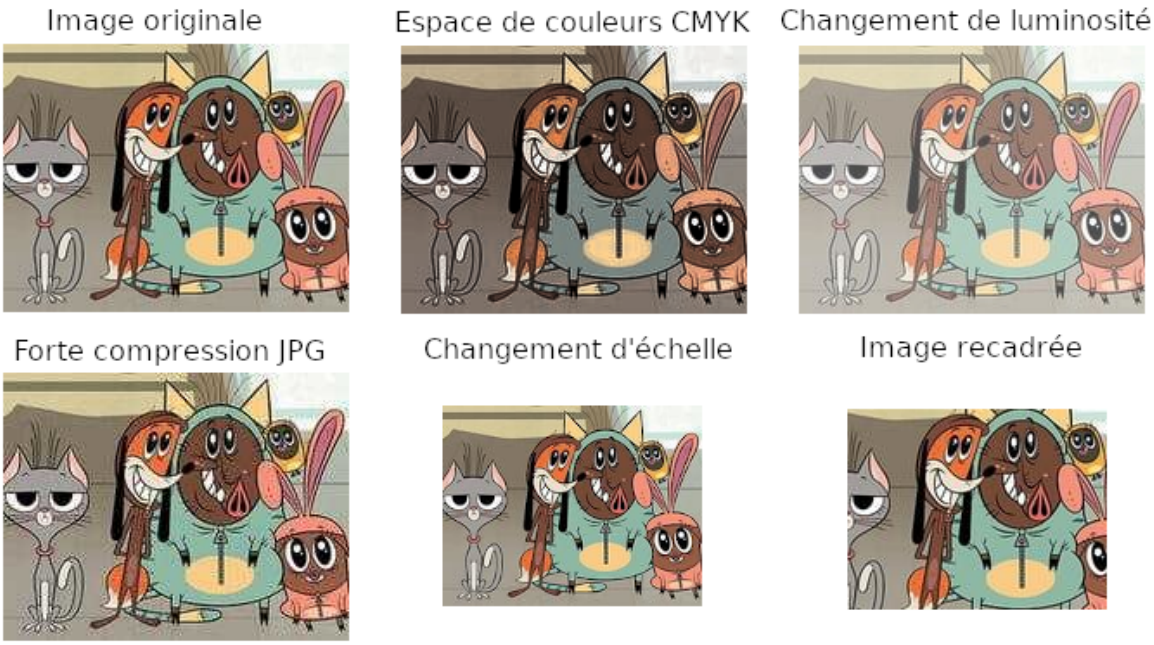

FIGURE 3.1 – Comparaison des différents types de doublons

2 Objectif

Aujourd'hui, on retrouve des images identiques sur des centaines de sites webs. Récupérer du contenu multimédia sur des sites concurrents est devenu un phénomène très courant, et il est donc habituel de retrouver la même image sur plusieurs sites, bien que celle-ci aura été souvent redimensionnée ou retouchée.

Ce phénomène est d'autant plus courant sur les sites de programmes TV, qui doivent se procurer des dizaines de milliers d'images pour couvrir la totalité des émissions programmées à la télévision, et qui préfèrent donc récupérer ces images sur d'autres sites concurrents plutôt que de les créer par eux-mêmes.

Pour récupérer ces images, on utilise des robots d'indexations (*crawlers* en anglais). Ces robots explorent le web à la recherche de contenu (images, vidéos, textes...) pour ensuite le télécharger et le réutiliser plus tard. Un fait à considérer dans l'élaboration d'un crawler est que chaque site web est conçu d'une manière différente. Par conséquent, pour qu'un crawler puisse explorer un nouveau site web, on doit d'abord étudier sa structure, puis en extraire des règles logiques qu'on fournira au crawler. Cette démarche est longue et répétitive, d'autant plus qu'il arrive que certains sites mettent à jour la structure avec laquelle ils gèrent les images.

Un des objectifs de ToddTV est d'exploiter le phénomène de réutilisation d'images, afin d'extraire automatique les règles de structurations des nouveaux sites de programmes TV. On parlera alors de *scrapper*, un type de crawler "intelligent". La démarche pour réaliser cela est la suivante :

1. Réaliser une base d'images labellisées à l'aide d'un crawler codé "en dur".

2. Pour chaque nouveau site web, récupérer le plus d'images possibles sans chercher à les catégoriser. Pour chaque image, récupérer la façon dont elle est structurée dans le site.

3. Comparer les deux bases d'images et détecter les duplicatas.

4. À partir de la liste de duplicatas et des métadonnées, extraire automatiquement des règles qui permettent de reconnaître et labelliser les images utiles (vignettes de programmes TV) et d'ignorer les images inutiles (logos, publicités).

Dans un premier temps, mon rôle sera de réaliser la base d'images labellisées à l'aide du crawler réalisé par Louis Babuchon, puis de développer un système de détection de duplicatas et de l'évaluer sur cette base de données (point 1). Dans un second temps, je devrais optimiser ce système afin de pouvoir comparer des gros volumes de données, et donc être en capacité de réaliser les points 2 et 3. Enfin, si le temps me le permet, je tenterai d'extraire les règles de structurations à partir des listes de duplicatas (point 4).

3 Base d'images labellisées

Pour tester mes premiers algorithmes de détection de duplicatas, j'ai utilisé une base de 1 800 images labellisées à la main par Naifen Gan en 2019. Ces images avaient été récupérées grâce au crawler de Louis Babuchon, puis triées à la main en deux catégories : images identiques et images différentes.

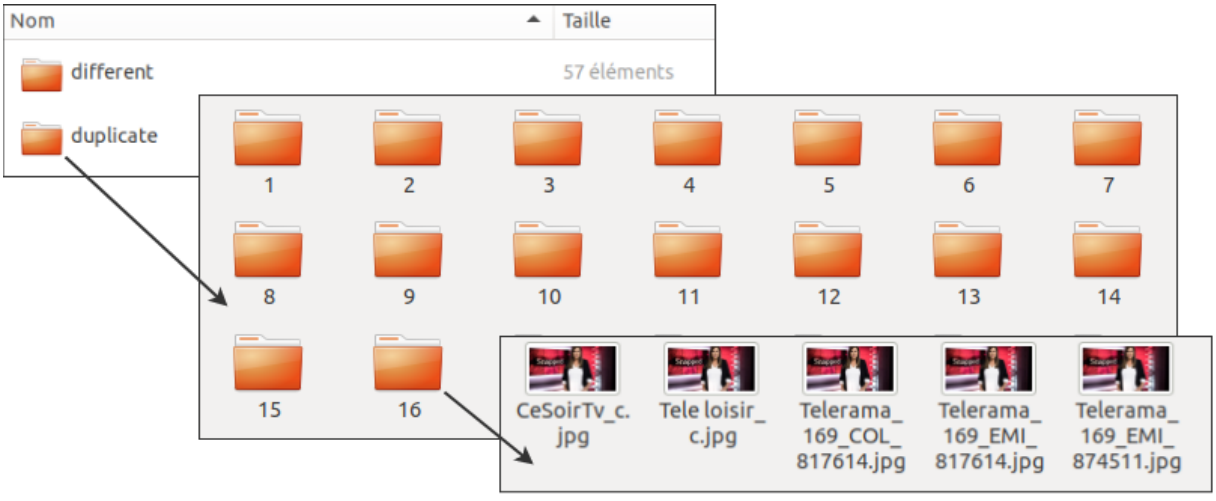

FIGURE 3.2 – Structure de la base d'images

Cette base était suffisante pour réaliser des petits tests, mais pour évaluer correctement mes algorithmes, j'ai du augmenter fortement le nombre d'images. J'ai donc récolté plus de 25 000 images avec le crawler de Louis, puis je les ai triées de manière semi-automatique, en utilisant les algorithmes développés précédemment, couplé à un second tri manuel pour éliminer les erreurs. Ma base finale d'évaluation était donc constituée d'environ 1 200 images identiques réparties en 332 classes, et de 9 000 images différentes.

4. Réalisations

1 Algorithmes d'image-matching

Le premier objectif à accomplir lors de ce stage était de détecter si deux images sont identiques ou non. Pour réaliser cela, j'ai du étudier et implémenter différents algorithmes d'image matching. Ces algorithmes ont tous pour point commun de prendre deux images en paramètre et de renvoyer un "score de similitude", qu'on normalise souvent entre 0 et 1. Ensuite, en fixant un seuil (différent pour chaque algorithme), on pourra facilement déterminer si une image est identique à une autre en fonction de si le score renvoyé par l'algorithme est inférieur ou supérieur à ce seuil.

Les deux images utilisées par mon programme sont d'abord redimensionnées à une même taille fixe, puis passées en noir et blanc, afin de ne réaliser le calcul de différence que sur une seule dimension (une nuance de gris comprise entre 0 et 255). Il serait aussi possible d'utiliser des images en couleur afin d'améliorer les performances, au détriment de la vitesse de calcul, mais j'ai décidé de ne pas aborder cette possibilité lors de mon stage, car les performances étaient déjà très satisfaisantes avec des images en noir et blanc.

Dans les formules mathématiques et les implémentations en Python qui vont suivre, on notera A et B les deux images à comparer, $X(i, j)$ le pixel à la position (i, j) de l'image X, et (n, m) les dimensions des images.

1.1 Sum of Absolutes Differences (SAD)

La "somme des différences absolues" est un algorithme simple permettant de calculer la différence absolue entre les pixels de deux images. En additionnant ces différences, on obtient une mesure analogue à une distance qui caractérise la similitude des deux images comparées.

Interprétation

On peut calculer la distance en utilisant SAD avec la formule ci-dessous :

$$SAD = \frac{1}{n * m} \sum_{i=1}^{n} \sum_{j=1}^{m} |A(i, j) - B(i, j)|$$

L'algorithme de SAD renvoie une distance, normalisée entre 0 et 1. Une distance de 0 correspond donc à deux images identiques, tandis qu'une distance approchant de 1 indique une différence entre les deux images.

Implémentation en Python

On peut très simplement implémenter SAD en python avec la ligne suivante. On considère que A et B sont les deux images à comparer, transformées en matrices de valeurs de gris, et n et m les dimensions de ces matrices.

```
1 sad = (1 / n*m) * np.sum(np.sum(np.absolute(A - B))
```

L'implémentation de l'algorithme SAD a l'avantage de nécessiter peu de calculs, et d'être donc assez rapide comparée à d'autres méthodes d'image matching. Cependant, SAD effectue des comparaisons pixels par pixels, et est donc très sensible au bruit ou aux changements de contraste. C'est donc un algorithme qui aura tendance à réaliser beaucoup de faux négatifs (des images identiques qui seraient reconnues comme différentes).

1.2 Sum of Squared Differences (SSD)

L'algorithme SSD est très similaire à SAD à l'exception qu'on passe la différence des pixels au carré plutôt qu'en valeur absolue. De ce fait, SSD ressemble fortement à un calcul d'erreur quadratique appliqué à deux images.

Interprétation

La formule pour calculer la distance entre deux images en utilisant SSD est très similaire à la précédente :

$$SSD = \frac{1}{n * m} \sum_{i=1}^{n} \sum_{j=1}^{m} [A(i,j) - B(i,j)]^2$$

De la même manière que pour SAD, SSD renvoie une distance comprise entre 0, pour une corrélation parfaite, et 1.

Implémentation en Python

Comme pour l'implémentation en Python de SAD, on peut écrire l'algorithme de SSD en une ligne, en passant la différence entre les images A et B au carré plutôt qu'en valeur absolue.

```
1 ssd = (1 / n*m) * np.sum(np.sum(np.power(A - B, 2)))
```

Les implémentations de SSD sont elles aussi assez rapides du fait du peu de calculs qu'elles nécessitent, mais possèdent le même défaut que SAD. En effet, ce type d'algorithme compare les valeurs des pixels deux à deux, et sont donc très sensibles aux changements d'illumination ou au bruit.

1.3 Zero-Normalized Cross Correlation (ZNCC)

Le calcul de ZNCC diffère des précédents car il adopte une approche plus statistique du problème d'image matching. En effet, il est très courant que deux images aient des différences de contraste, et les algorithmes comme SAD et SSD auront des difficultés à détecter ces différences. ZNCC procède différemment en normalisant les images dans un premier temps, puis en divisant le produit des deux images par le produit de leurs écart-types.

Interprétation

Le coefficient ZNCC entre deux images A et B peut être calculé comme suit :

$$ZNCC = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}[(A(i,j) - \bar{A}) * (B(i,j) - \bar{B})]}{\sigma_A * \sigma_B}$$

Avec $\bar{A}$ représentant la moyenne de tous les pixels de A et :

$$\sigma_A = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(A(i,j) - \bar{A})^2} = ||A - \bar{A}||_F$$

Comme pour les algorithmes SAD et SSD, les images sont d'abord converties en matrices de valeurs de gris, comprises entre 0 et 255. Les coefficients σ_A et σ_B correspondent à l'écart type des matrices images de A et B. ZNCC renvoie un score compris entre -1 (anti-corrélation) et 1 (corrélation parfaite).

Implémentation en Python

Une implémentation classique de l'algorithme ZNCC en Python en utilisant la librairie Numpy ressemblerait à cela :

```
1 mA = A - np.mean(A)
2 mB = B - np.mean(B)
3 zncc = np.sum(np.sum(np.multiply(mA, mB))) /
4         (np.linalg.norm(mA) * np.linalg.norm(mB))
```

L'algorithme ZNCC est donc très avantageux comparé à SAD ou SDD, car il est bien plus robustes aux changements de contraste ou au bruit. Cependant, il demande plus d'étapes de calcul et prend donc légèrement plus de temps à être exécuté. Nous verrons plus tard comment il est possible d'optimiser le calcul de ZNCC en pré-calculant certains termes afin de ne plus avoir à les recalculer à chaque itération.

2 Évaluation des performances

Une fois les trois algorithmes de reconnaissance de duplicatas implémentés, une phase importante de mon stage a été d'évaluer les performances de ces algorithmes, afin de déterminer lequel utiliser dans le système final.

La démarche générale derrière l'évaluation des performances d'un algorithme de classification est de réaliser plusieurs milliers de tests sur une base déjà classifiée, puis de comparer les résultats obtenus aux résultats attendus (appelés la *vérité terrain*). À partir de la base d'images labellisée évoquée dans la partie 3.3, j'ai pu réaliser des comparaisons sur 53 896 couples d'images différentes et 9 622 couples d'images identiques.

Courbes précision-rappel

La plupart des méthodes permettant d'analyser la qualité d'un algorithme de classification commencent par le comptage des vrais/faux positifs et des vrais/faux négatifs. Dans notre cas, ces indicateurs peuvent être définis comme suit :

- Vrai positif (TP) : images correctement détectées comme identiques
- Faux positif (FP) : images détectées comme identiques alors qu'elles sont différentes
- Vrai négatif (TF) : images correctement détectées comme différentes
- Faux négatif (FN) : images détectées comme différentes alors qu'elles sous identiques

À partir de ces mesures, on peut calculer d'autres indicateurs qui permettent de quantifier la qualité de notre prédiction :

- La précision, calculée comme suit : $P = \frac{TP}{TP+FP}$, représente ici le rapport entre le nombre d'images bien classifiées comme identiques, et le nombre d'images détectées comme identiques. Elle peut donc être interprétée comme une mesure de l'exactitude des résultats.

- Le rappel, calculé comme suit : $R = \frac{TP}{TP+FN}$, est la proportion d'images bien classifiées comme identiques parmi l'ensemble des images identiques. Il représente donc une mesure de la quantité de résultats pertinents.

Ces deux mesures sont comprises entre 0 (mauvais score) et 1 (score parfait). Un système parfait détecterait la totalité des images identiques (rappel = 1) sans faire aucune erreurs (précision = 1). Dans notre cas, notre objectif est de maximiser avant tout la précision, car le fait de classifier comme identiques deux images différentes aurait des répercutions sur toutes les autres étapes du projet. Avoir un rappel égal à 1 est moins important, car on peut se permettre de louper quelques images parmi les milliers d'images téléchargées par le crawler de Jean-Baptiste.

La démarche pour calculer ces mesures est de lancer les algorithmes avec différentes valeurs de seuil sur la base de tests, puis de compter à chaque exécution le nombre de vrais positifs, faux positifs et faux négatifs, pour ensuite calculer la précision et le rappel. Évidemment, pour chaque algorithme, la précision et le rappel vont brutalement changer au moment de franchir le seuil optimal de détection. On ferra donc attention à réaliser beaucoup plus de mesures aux alentours de ce seuil.

À l'aide d'un programme écrit en Python, je lance des évaluations sur les quelques 60 000 couples d'images avec les trois algorithmes, en testant à chaque fois une centaine de valeurs de seuils. Après plus de 4h d'exécution, on obtient les courbes ci-dessous :

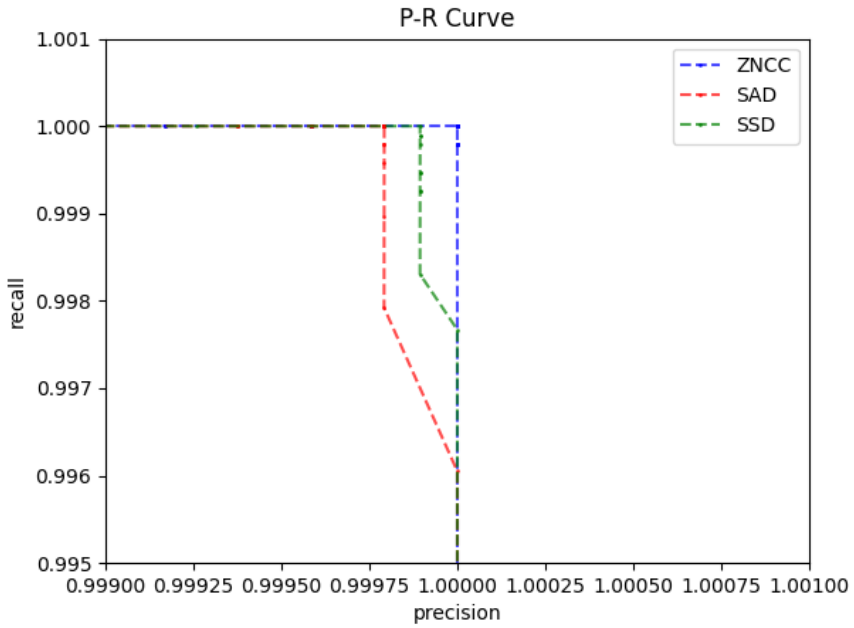

FIGURE 4.1 – Courbe de précision-rappel des trois méthodes

Les trois algorithmes fournissant des résultats très satisfaisants, j'ai du zoomer sur "l'extrémité" des courbes (le point $(1, 1)$, équivalent à un système optimal) afin de distinguer une différence entre celles-ci. Bien que cette figure confirme que l'algorithme ZNCC est bien le plus performant pour détecter des duplicatas, les implémentations de SAD et de SSD présentent aussi des résultats très corrects, et il pourrait donc être intéressant de les utiliser afin d'accélérer le temps de calcul.

Il me semble important de rappeler que la base d'images de test a été conçu semi automatiquement, en classifiant d'abord les images en utilisant ZNCC avec un seuil très haut, pour être plus "strict" dans la sélection d'images (afin de ne classer comme identiques que celles dont l'algorithme est 100% convaincu qu'elles le sont). J'ai ensuite effectué un second tri manuel pour éliminer les erreurs. Il est donc possible que cette phase d'évaluation des performances présente un biais qui favoriserait ZNCC. Cependant, les performances de ZNCC avaient déjà été démontrées lors de tests sur la base d'images classées manuellement par Naifeng et ces résultats restent donc très probables.

Courbes de F-Measure

La F-measure est un indicateur qui combine précision et rappel pour quantifier la qualité globale d'un algorithme de classification. Elle peut se calculer comme suit :

$$F_\beta = \frac{(1+\beta^2) \times (P \times R)}{\beta^2 \times P + R}$$

On utilise généralement la mesure F1, qui considère équitablement la précision et le rappel, et qui s'obtient simplement en calculant la F-measure avec $\beta = 1$:

$$F_1 = 2 \times \frac{(P \times R)}{(P + R)}$$

En calculant la F1-measure pour toutes les valeurs de précisions et de rappels obtenues précédemment, on peut tracer l'évolution de la qualité d'une prédiction en fonction du seuil choisi. À partir des données récoltées dans la partie précédente, on peut donc dessiner la courbe de F1-measure pour chaque algorithme, sans oublier de normaliser les valeurs de seuil entre 0 et 1 pour pouvoir comparer les courbes plus facilement.

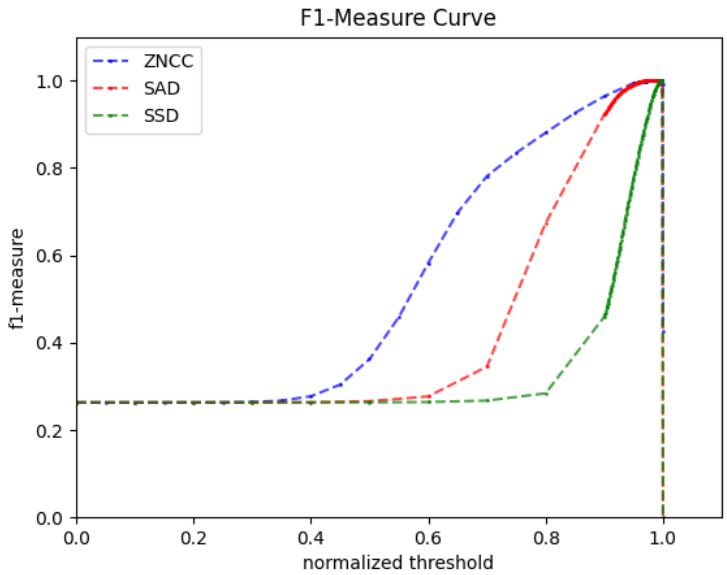

FIGURE 4.2 – Courbe de F-Measure des trois méthodes

On remarque que la F1-measure de ZNCC augmente plus rapidement que celles des autres algorithmes, et que cet algorithme donnera donc de bons résultats sur une grande plage de seuils. Pour trouver le seuil optimal de chaque méthode, il suffit donc de chercher à quel valeur de seuil est obtenu la F1-measure maximum. On trouve donc un seuil de 0.015 pour SAD, 0.003 pour SSD et 0.982 pour ZNCC. On remarque par la même occasion que seule la méthode ZNCC permet d'atteindre le score maximum de 1, ce qui signifie que le problème est séparable avec cet algorithme. Les deux autres méthodes sont elles-aussi presque parfaites, avec des scores maximum de 0,9999 pour SAD et 0,9992 pour SSD.

Matrices de confusions

L'élaboration de matrices de confusions est une autre façon de montrer rapidement la qualité d'un système de classification. Ces matrices sont obtenues en lançant les algorithmes avec leur valeur de seuil optimale, trouvée grâce aux courbes de F-Measure, puis en comptant le nombre de vrais/faux positifs et vrais/faux négatifs.

	Identique	Différent
Identique	TP = 9 622	FP = 2
Différent	FN = 0	TN = 53 894

TABLE 4.1 – Matrice de confusion de SAD (seuil = 0.015)

	Identique	Différent
Identique	TP = 9 622	FP = 16
Différent	FN = 0	TN = 53 880

TABLE 4.2 – Matrice de confusion de SSD (seuil = 0.003)

	Identique	Différent
Identique	TP = 9 622	FP = 0
Différent	FN = 0	TN = 53 896

TABLE 4.3 – Matrice de confusion de ZNCC (seuil = 0.982)

Ces tableaux confirment que le problème de détection de duplicatas est séparable via l'algorithme ZNCC, c'est à dire qu'on peut facilement distinguer la classe des images identiques (TP ou les vrais positifs) de la classe des images différentes (TN ou les vrais négatifs). J'utiliserai donc cet algorithme pour le reste du stage.

3 Optimisation

Les premières versions de mes algorithmes étaient très lentes : elles ne réalisaient pas plus de 10 comparaisons par secondes. Pour comprendre pourquoi il était indispensable d'optimiser mon programme, on peut revenir sur la quantité de données à comparer pour l'objectif final du stage.

Le but de ce projet était de pouvoir comparer deux à deux des images provenant d'un nouveau site de programme TV (environ 3 000 images pour chaque site parcouru par le crawler de Jean-Baptiste) à une base de donnée d'images récentes, récupérées par un crawler fiable (5 000 images en moyenne). Comparer ces deux bases d'images implique plus de 15 millions de comparaisons, soit une quinzaine de jours de calculs (par site) avec l'implémentation actuelle.

Ma mission pour la deuxième partie de ce stage était donc d'étudier plusieurs techniques d'optimisation et de les exploiter afin d'améliorer l'efficacité de mon programme.

3.1 Vectorisation

La vectorisation est un cas particulier de parallélisation, où on utilise des instructions spécialisées pour effectuer plusieurs opérations simultanément, à l'inverse d'un programme classique qui effectue toutes ses opérations en séquentiel. Cette technique d'optimisation implique plusieurs conséquences, notamment le fait d'avoir des données correctement ordonnées en mémoire. La vectorisation accélère particulièrement la vitesse d'exécution des boucles, du fait qu'il soit possible d'effectuer plusieurs itérations en simultané.

Ainsi, une boucle classique de ce type :

```
1 for (i=0; i<1024; i++)
2    C[i] = A[i] * B[i];
```

Pourrait s'interpréter comme ceci en code vectorisé :

```
1  for (i=0; i<1024; i+=4)
2     C[i:i+3] = A[i:i+3] * B[i:i+3];
```

Avec C[i :i+3] qui représente 4 éléments du tableau C, que le processeur calcule donc en même temps.

Dans des langages bas niveau comme le C, la vectorisation passe par l'utilisation d'instructions spéciales (SIMD ou SSE pour la plupart des CPU, AVX pour les CPU plus récents...) qui indiquent au processeur d'appliquer les mêmes opérations sur plusieurs valeurs à la fois. Dans le contexte des langages de haut niveau comme Python, Matlab ou R, le terme de vectorisation décrit plus simplement l'utilisation d'un code optimisé et précompilé, écrit dans un langage de bas niveau (par exemple C) pour effectuer des opérations mathématiques sur une séquence de données. Cela se fait à la place d'une itération explicite écrite dans le code du langage natif (par exemple, une "boucle for" écrite en Python).

Pour vectoriser mon code Python, je n'ai donc pas eu à utiliser d'instructions spécialisées, mais plutôt à utiliser exclusivement *Numpy*, une librairie de calcul scientifique implémentant des fonctions vectorisées.

Pour vérifier les capacités de Numpy à optimiser des calculs, j'ai écris l'algorithme SAD en deux versions différentes : la première utilise des "boucles for" classiques, et la seconde utilise uniquement les fonctions de Numpy.

```
1  def SAD_nonvect(path1, path2):
2      img1 = load(path1)
3      img2 = load(path2)
4
5      (n, m) = img1.shape
6      sad = 0
7
8      for i in range(n):
9          for j in range(m):
10             sad += abs(img1[i][j] - img2[i][j])
11
12     return sad
```

Algorithme 4.1 – Algorithme SAD non-vectorisé

```
1  def SAD_vect(path1, path2):
2      img1 = load(path1)
3      img2 = load(path2)
4
5      sad = np.sum(np.sum(np.absolute(img1 - img2)))
6
7      return sad
```

Algorithme 4.2 – Algorithme SAD vectorisé

En plus d'un gain de place et de clarté évident, l'amélioration des performances est assez remarquable : la version vectorisée prend uniquement 0.03s à s'exécuter contre 0.64s pour la version non-vectorisée (moyenne calculée avec 5 000 itérations de chaque algorithme, avec deux images identiques de 900x450px). En utilisant au maxiumum les fonctions de Numpy, on peut donc diviser par 20 le temps de calcul. Ainsi, afin d'optimiser le programme, on se devra de proscrire les calculs utilisant des boucles classiques en Python, et préférer l'utilisation de fonctions Numpy vectorisées, qui seront ensuite converties en code C optimisé.

3.2 Calcul offline

Les premières versions de mes algorithmes de reconnaissance de duplicatas (SAD, SSD, ZNCC) s'occupaient dans un premier temps de plusieurs tâches qui n'étaient pas directement liées à la détection, mais qui étaient nécessaires pour la bonne réalisation de celle-ci. On pouvait alors résumer les algorithmes en 5 étapes :

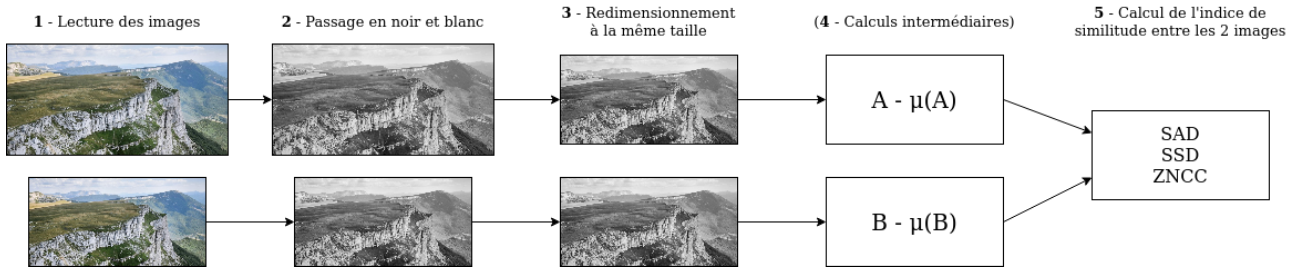

FIGURE 4.3 – Représentation des algorithmes (sans calcul offline)

Après plusieurs sessions de profilage de mon programme, je me suis rendu compte que la majorité du temps d'exécution était due aux manipulations sur les images (passage en noir et blanc et changement d'échelle), ainsi qu'à la lecture de celles-ci. De manière évidente, plus les images seront de grande taille, plus il sera long de les ouvrir et de les manipuler.

```
1  def ZNCC(path1, path2):
2      # 1 - Lecture des images --------------- 22.05ms
3      img1 = io.imread(path1)
4      img2 = io.imread(path2)
5
6      # 2 - Passage en N&B -------------------- 9.79ms
7      img1 = rgb2gray(img1)
8      img2 = rgb2gray(img2)
9
10     # 3 - Redimensionnement ----------------- 44.69ms
11     img1 = resize(img1, (64, 64))
12     img2 = resize(img2, (64, 64))
13
14     # 4 - Pre-Calculs ----------------------- 0.10ms
15     img1 = img1 - np.mean(img1)
16     img2 = img2 - np.mean(img2)
17
18     # 5 - Calcul de ZNCC -------------------- 0.07ms
19     zncc = np.sum(np.sum(np.multiply(img1, img2))) /
20            (np.linalg.norm(img1) * np.linalg.norm(img2))
21
22     return zncc
```

Algorithme 4.3 – Implémentation de ZNCC avec les 5 étapes

Avec une implémentation comme celle-ci, comparer une même image à plus centaines d'autres implique de réaliser des centaines de fois les mêmes opérations sur la même images. Pour éliminer cette perte de temps, on pourrait donc réaliser ces opérations une seule fois, puis sauvegarder l'image "transformée" en plus de l'originale. Via cette simple manipulation, on se passe des étapes 2 et 3, ce qui diminue fortement le temps de calcul. De plus, une fois l'image redimensionnée, sa

lecture est beaucoup plus rapide, ce qui nous fait économiser du temps sur l'étape 1. On remarque aussi qu'il serait possible de réaliser les pré-calculs (étape 4) en même temps que les transformations sur l'image, et enregistrer les résultats de ceux-ci à la place de l'image. C'est là qu'intervient la notion de carte de caractéristique et de calcul offline.

Au lieu de sauvegarder l'image dans un format classique (généralement JPG), on va effectuer certains calculs sur celle-ci (dans notre cas, la valeur de chaque pixel moins la moyenne de tous les pixels de l'image) et enregistrer directement les résultats de ces calculs sous forme des tableaux. On peut choisir d'enregistrer le fichier généré dans un format lisible par un humain (CSV par exemple), ou dans un format binaire. Dans mon cas, j'ai choisi d'enregistrer les résultats en fichiers binaires Numpy, avec l'extension .npy, car ceux-ci offraient de meilleures performances à la lecture et à l'écriture.

J'ai profité de cette phase d'optimisation pour décider de la taille standard à laquelle je redimensionnerais les images avant de les enregistrer. En utilisant les outils d'évaluation des performances élaborés précédemment, j'ai trouvé qu'on pouvait réduire les images à une taille de 64x64 avant que l'algorithme perde en précision. Cette taille peut encore être réduite pour accélérer le programme, au risque d'augmenter le taux d'erreurs, ou être augmentée pour améliorer la précision.

Le nouveau format des "images" est donc un tableau de nombres flottants en 3 dimensions, de taille 64x64x2. La première composante du tableau représente les valeurs de gris de chaque pixel, et la seconde composante représente cette même valeur à laquelle on a soustrait la moyenne de tous les pixels de l'image ($A - \bar{A}$ dans la formule de ZNCC vue précédemment). Cette seconde composante n'est utile que pour le calcul de ZNCC, et c'est aussi la seule valeur nécessaire pour celui-ci, on pourrait donc réduire le fichier à cette simple composante si on choisissait de n'utiliser que cet algorithme.

La nouvelle implémentation utilisant le calcul offline est résumée dans le schéma ci-dessous.

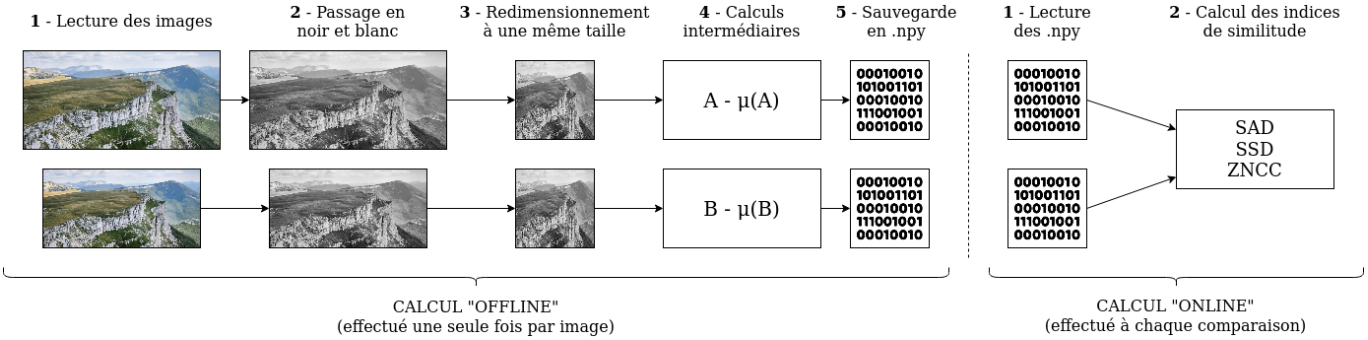

FIGURE 4.4 – Représentation des algorithmes (avec calcul offline)

Cette implémentation propose donc une exécution en deux temps : après avoir récolté un lot d'images à l'aide d'un crawler, il faudra d'abord les traiter avec un premier programme, qui enregistrera pour chaque image sa version sous forme de carte de caractéristique. Ensuite, avec un second programme, on utilisera ces fichiers pour détecter les doublons entre plusieurs images.

```
def ZNCC(path1, path2):
    # Lecture des fichiers binaires ----- 0.57ms
    img1 = load(path1)
    img2 = load(path2)

    # Calcul de ZNCC -------------------- 0.06ms
    zncc = np.sum(np.sum(np.multiply(img1[1], img2[1]))) /
           (np.linalg.norm(img1[1]) * np.linalg.norm(img2[1]))

    return zncc
```

Algorithme 4.4 – Implémentation de ZNCC avec calcul-offline

Avec cette implémentation, on remarque tout d'abord une grande amélioration du temps de calcul total : 0.7ms contre 76ms pour la première version (moyenne effectuée avec 10 000 itérations de chaque algorithme, avec deux images identiques de 900x450px). On remarque aussi que la lecture des fichiers binaires est beaucoup plus rapide. Là où il fallait 22ms pour lire une image de 900x450px, il ne faut plus que 0.57ms pour lire un fichier binaire contenant deux tableaux de 64x64.

3.3 Parallélisation

La dernière technique d'optimisation que j'ai abordé est la parallélisation de mon programme. En effet, celui-ci peut se résumer à une simple boucle effectuant les mêmes opérations sur des millions d'images, et il semble donc logique de paralléliser ce processus afin d'exploiter toute la puissance disponible de la machine.

Pour réaliser cela, j'ai utilisé la librairie de Python *concurrent.futures*, qui fournit une interface haut niveau pour l'exécution de tâches parallèles. Cette librairie dispose de fonctions permettant d'effectuer ces tâches par des threads, ou des processus séparés. L'utilisation de threads n'est pas intéressante dans mon cas, car ceux-ci ne s'exécutent jamais réellement en parallèle.

L'utilisation de threads est très utile pour des tâches qui subissent des latences impossibles à supprimer. Par exemple, lorsqu'on effectue des requêtes à des bases de données ou à des sites webs, le processeur ne fait rien d'autre qu'attendre une réponse. On peut alors utiliser d'autres threads pour réaliser des requêtes pendant que les premiers sont en attente.

À l'inverse, le multiprocessing consiste à réaliser plusieurs tâches en simultané, afin de réduire le temps de calcul total. C'est donc une technique très utilisée pour les tâches très lourdes en consommation CPU, par exemple pour des calculs scientifiques.

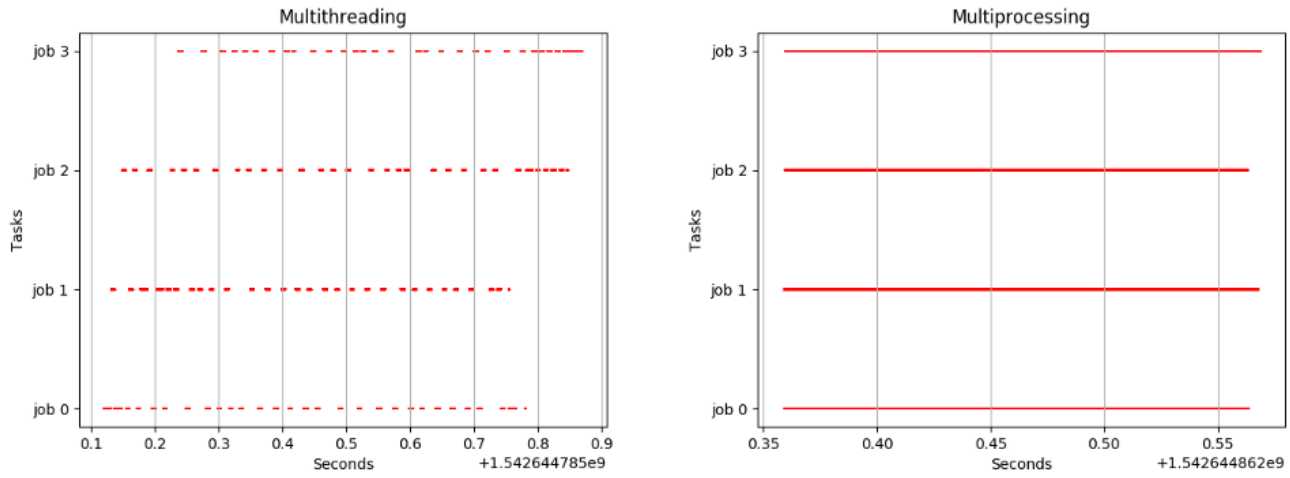

FIGURE 4.5 – Multithreading VS Multiprocessing

Source: Amine Baatout (5 Décembre 2018). *Multithreading VS Multiprocessing in Python*

L'implémentation du parallélisme dans notre programme ne va pas changer nos algorithmes de reconnaissance de duplicatas (SAD, SSD, ZNCC), mais plus la façon de répartir les images à ces algorithmes. Jusqu'à présent, les couples d'images était comparées les uns à la suite des autres, de manière séquentielle, comme dans l'algorithme simplifié ci-dessous.

```
1 results = []
2 for img1 in database1:
3     for img2 in database2:
4         results.append(ZNCC(img1, img2))
```

Algorithme 4.5 – Implémentation séquentielle

Une implémentation parallélisée du code ci-dessus ressemblerait à cela :

```
1 executor = concurrent.futures.ProcessPoolExecutor(max_workers=4)
2 results = executor.map(ZNCC, database1, database2, chunksize=50)
```

Algorithme 4.6 – Implémentation parallélisée

Tout d'abord, on créé un *executor*, un objet possédant des méthodes permettant de gérer des tâches de manière asynchrone. On configure cet executor en lui indiquant un nombre maximum de "workers" (c'est à dire des threads dans le cas d'une ThreadPoolExecutor, ou des process dans le cas d'une ProcessPoolExecutor) à 4. Dans le cadre du multiprocessing, ce nombre ne doit jamais dépasser le nombre de coeurs du CPU, sous peine que les différents processus entrent en concurrence.

Ensuite, la fonction *map* est chargée de "découper" les itérables (database1 et database2) en un certain nombre de blocs qu'elle soumet au pool de processus en tant que tâches distinctes. La taille de ces morceaux peut être spécifiée en définissant le paramètre *chunksize*. La documentation officielle de concurrent.futures indique que dans le cas d'itérables très longs, augmenter la taille des "chunks" améliora nettement la vitesse de calcul. Cependant, il n'est pas précisé comment

déterminer de manière optimale la valeur de ce paramètre. J'ai donc lancé une série de tests avec différentes valeurs de chunksize, et j'ai déterminé que la vitesse maximale était atteinte pour une valeur comprise entre 40 et 60 (tests réalisés avec une base de donnée de 80 000 images).

Les résultats du calcul parallélisé arrivent progressivement dans la liste *results*, qu'on peut commencer à traiter avant la fin de l'exécution. La parallélisation de mon programme aura permis de tripler la vitesse d'exécution, mais celle-ci pourrait montrer de meilleurs résultats sur une machine plus performante.

Récapitulatif

En trois semaines de stage, j'ai pu aborder trois techniques d'optimisation qui agissent chacune sur des points différents mais qui ont toutes montré des résultats très satisfaisants. La version finale de mon programme permet de réaliser jusqu'à **5 000 comparaisons par seconde**, ce qui est une vitesse suffisante pour les objectifs à accomplir. Cependant, d'autres pistes d'optimisation restent encore à étudier :

- L'utilisation de techniques d'*upper bounding* permettraient de trouver rapidement une borne supérieure au calcul de ZNCC, et donc d'ignorer une grande partie du calcul. (Di Stefano L., Mattocia S., Tombari F. (2005) *ZNCC-based template matching using boundedpartial correlation*)

- La réécriture du programme dans d'autres langages pourraient améliorer nettement le temps de calcul. On pourrait par exemple utiliser *Cython*, un ensemble de compilateur et d'instructions spécialisées qui permettent de compiler du code Python en code C optimisé. Une solution encore meilleure serait d'implémenter les algorithmes directement dans un langage bas niveau.

5. Mise en œuvre

Une fois que le système de reconnaissance de duplicatas a été mis au point, testé et optimisé, j'ai pu commencer à mettre en place différents scripts en Python ayant pour but de permettre à un utilisateur de générer facilement les règles d'extraction d'un nouveau site de programme TV.

Il me semble important de rappeler l'objectif général visé par ToddTV à travers la plupart des stages réalisés cet été. Afin de se démarquer des sites de programme TV concurrents, la start-up souhaite proposer à ses utilisateurs un très large contenu additionnel appuyant les émissions ainsi que films et séries en VoD. Ce contenu peut être sous la forme d'images, d'extraits vidéos, ou de texte (par exemple, des bandes annonces, des captures d'écrans du programme, le casting...).

Avec plusieurs stagiaires, nous nous sommes concentrés sur la récupération automatique de contenu image depuis d'autres sites de programme TV. Jean-Baptiste Huyghe, lui aussi étudiant en 4ème année au département Informatique de Polytech Tours, a travaillé sur l'élaboration d'un crawler très performant capable de récupérer plus de 75 000 images sur une vingtaine de sites web en quelques minutes. En associant son travail à mon système de détection de duplicatas, nous espérions pouvoir créer un programme capable de comprendre automatiquement la structuration des images dans les sites web sans avoir à les étudier manuellement.

Dans les parties à venir, j'expliquerai les fonctionnalités de ce programme, les points accomplis et ceux qui restent à réaliser, ainsi que le fonctionnement général du système.

1 Récupération des images

La première étape à réaliser pour comprendre la structure d'un nouveau site web est de constituer deux bases d'images : la première est une base d'images organisée, triée par programme TV. Elle sera élaborée avec le crawler réalisé par Louis Babuchon lors de son projet Recherche & Développement de 5ème année. On utilisera ces images comme base fiable pour chercher des doublons dans la seconde base de donnée. Celle-ci, élaborée grâce au crawler de Jean-Baptiste évoqué plus tôt, est constituée de toutes les images récupérées sur le site web à étudier.

Base d'images labellisées

Le crawler réalisé par Louis Babuchon est très simple à utiliser. Dans son fichier de configuration "config.json", on peut choisir les différents sites à parcourir, parmi 6 sites disponibles : xmltv.ch, programme-television.org, cesoirtv.com, linternaute.fr et tvmag.lefigaro.fr. On peut aussi spécifier le dossier où sauvegarder les images, puis lancer le crawler avec la commande suivante : `java -jar Crawler.jar 'config.json'`. Le programme ira ensuite récupérer uniquement les images des programmes TV et les enregistrera selon la structure résumée dans la capture d'écran ci-dessous. Ce crawler est capable de récupérer environ 4 000 nouvelles images lorsqu'on le lance pour la première fois depuis longtemps.

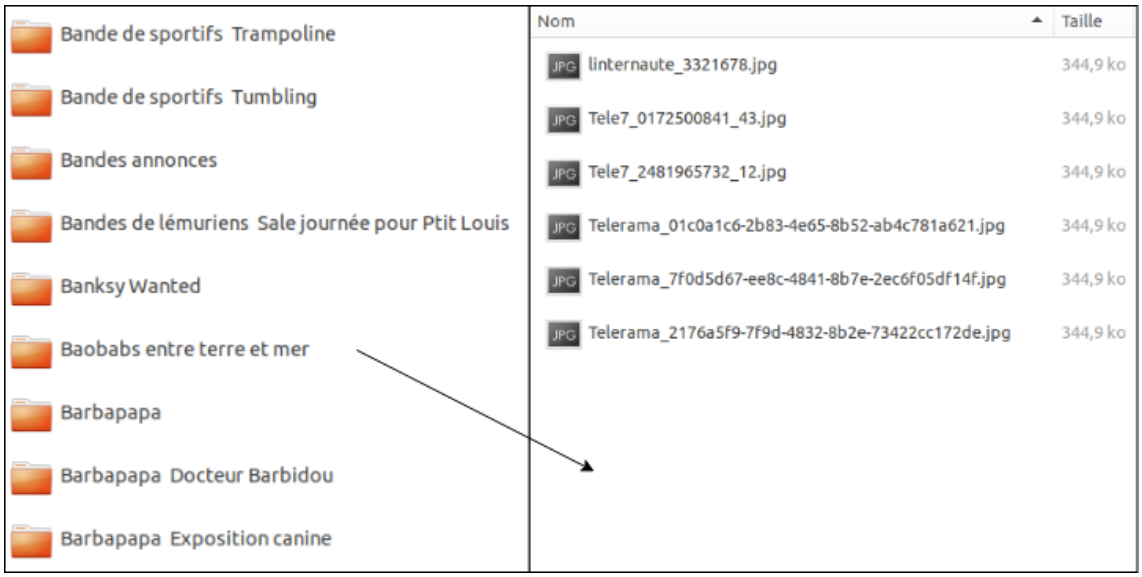

FIGURE 5.1 – Structure de la base d'images labellisées

Nouvelle base d'images

Le crawler de Jean-Baptiste est beaucoup plus performant que celui de Louis mais aussi simple à utiliser. Cependant, son défaut est qu'il récupère toutes les images des sites parcourus et non pas seulement les images intéressantes. Pour l'utiliser (version du 31/07/2020), il faut commencer par éditer le fichier "websites.csv" pour ne conserver que le site à parcourir. Il est possible de modifier le répertoire utilisé pour la sauvegarde des images avec la constante *IMAGES_PATH* définie dans la classe *Downloader.java*. Il suffit ensuite de lancer la classe *Main.java* pour démarrer le crawler, et de l'arrêter à tout moment une fois que assez d'images ont été récoltées. Ce crawler est capable de récupérer en une fois plus de 70 000 images sur 25 sites différentes, ce qui donne une moyenne d'environ 3 000 images récentes par site, mais ce chiffre peut varier énormément d'un site à l'autre. Actuellement, le crawler de Jean-Baptiste stocke les données selon la structure suivante :

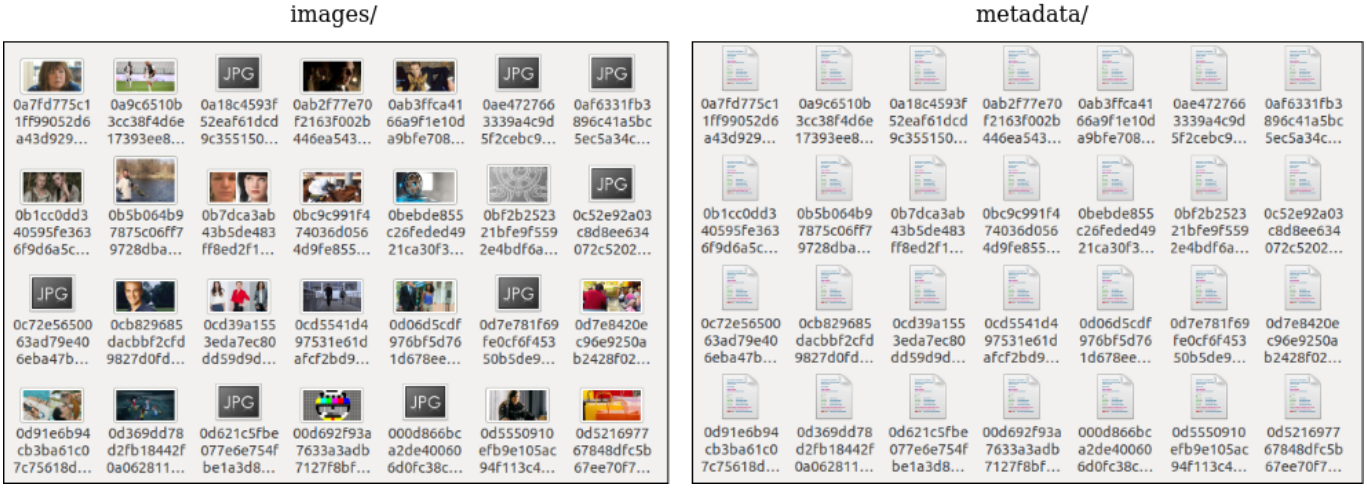

FIGURE 5.2 – Structure de la base de donnée à étudier

2 Pré-traitement des images

Une fois que les deux bases d'images ont été constituées, il faut les traiter avant d'effectuer de la reconnaissance de duplicatas. Les traitements à effectuer (passage en nuances de gris, changement d'échelle et calculs offline) sont prévus avant tout pour gagner en temps de calcul, et ont été évoqués dans la partie 3.2.

Pour effectuer ces traitements, j'ai réalisé un script en Python nommé *preparation.py*, qui prend en paramètres un chemin vers la base d'images "source" et un autre vers la destination. Ce script s'occupe ensuite d'ouvrir toutes les images du dossier source, de les traiter grâce aux fonctions de la librairie *skimage*, puis de les enregistrer sous la forme de fichier binaire dans le dossier de destination, en conservant exactement les mêmes noms et la même structure. Le programme ignorera les images corrompues (assez courantes lorsqu'on arrête le crawler en pleine exécution) et les images dans des formats non-reconnus.

Le pré-traitement des images peut être assez long et varie en fonction de la quantité d'images à manipuler, et surtout de la taille des images. En effet, les opérations de lecture et de redimensionnement dépendent directement de la taille des images, et le temps de calcul lié à ces opérations peut fortement augmenter avec des images en haute résolution. Heureusement, la plupart des sites de programmes TV hébergent directement leurs images dans de faibles résolutions. Le pré-traitement fonctionne aujourd'hui à une vitesse d'environ 50 images par seconde pour des images de 640x360px. Une piste d'amélioration aurait été de paralléliser le programme afin d'utiliser au maximum la puissance du CPU.

3 Comparaison des bases d'images

Maintenant que les deux bases sont sous forme de fichiers binaires, on peut enfin lancer la comparaison entre celles-ci afin de trouver des doublons. Pour cela, on utilise le script *imagematching.py*. Celui ci embarque plusieurs fonctionnalités qui permettent de garder le contrôle sur la détection de duplicatas entre les deux bases de données :

- Le programme a besoin de seulement deux paramètres pour fonctionner : les chemin vers la base d'images classifiée et celui vers la nouvelle base d'images. On parle bien ici des répertoires contenant les fichiers binaires et non les images brutes.

- Il est possible de spécifier l'algorithme et le seuil à utiliser pour la détection de duplicatas avec les paramètres *algo* et *threshold*. Par défaut, on utilise ZNCC avec un seuil de 0.982, ce qui nous donne la meilleure qualité selon les tests effectués précédemment (voir partie 2.)

- Pour avoir plus de contrôle sur la parallélisation de l'algorithme, on peut aussi changer le nombre de processus utilisés et la taille des "blocs de données" à envoyer au pool de processus avec les paramètres *nbProcess* et *chunk*. Pour obtenir des performances maximales, on choisira toujours *nbProcess* égal au nombre de cœurs physiques du processeur et *chunk* un nombre entier

entre 40 et 60 (cf. partie 3.3 sur la parallélisation). Par défaut, *nbProcess* vaut 4 et *chunk* vaut 50.

- Enfin, on peut choisir de limiter le nombre de comparaisons avec la base d'images labellisées avec le paramètre *limit* (qui représente le pourcentage de la base à explorer, compris entre 0 et 1). Il est important de garder à l'esprit que la vitesse maximale de comparaison est d'environ 5 000 images/s, et que même si ce chiffre peut sembler grand, cela reste très peu comparé au volume de données à analyser. Pour étudier un site, j'ai toujours tenté de travailler avec une base d'images labellisée et nouvelle base contenant 3 000 images chacunes. Comparées deux à deux, cela correspond à 9 millions de comparaisons, soit 30 minutes. Si la base d'images labellisées est trop grande, on peut par exemple la diminuer en fixant *limit* à 0.5, ce qui bloquera les comparaisons qu'à la première moitié de la base. Par défaut, *limit* est fixé à 1, soit 100% de la base.

Le fonctionnement du programme est résumé à travers ce code simplifié et commenté :

```
# On récupère la liste d'images à comparer
listUnknown = os.listdir(pathUnknown)
listKnown = os.listdir(pathKnown)

# On mélange la base connue pour ne pas faire des comparaisons
# dans le même ordre à chaque fois
random.shuffle(listKnown)

# On calcule l'exploration maximum de la base d'images labellisée
maxExplo = int(limit * len(listKnown))
unknownSize = len(listUnknown)

duplicates = 0

# Pour chaque image inconnue
for img in listUnknown:

    # On la compare à toutes les images connues
    # (Seule cette ligne est parallélisée)
    results = matching(repeat(img), listKnown[:maxExplo],
                       algo=ZNCC, nbProcess=4, chunk=50)

    for val in results:
        # Si val = True, alors on a repéré un doublon
        if val:
            duplicates += 1
            # On peut ensuite enregistrer sur un fichier l'image détectée
            # en tant que doublon, à quel programme elle correspond, etc.
            # (non-montré ici)

# Enfin, on calcule la couverture du site, comprise entre 0 et 1
coverage = duplicates / unknownSize
```

Algorithme 5.1 – Comparaison des deux bases d'images (simplifié)

La notion de couverture, évoquée dans les dernières lignes de l'algorithme, est importante. La couverture d'un site représente, dans notre cas, la proportion d'images identiques à celles de notre base de donnée labellisée. Elle correspond entre-autre au pourcentage d'images que les sites de programmes TV et VoD se "partagent" entre eux.

Lorsque j'ai commencé ce projet, nous pensions obtenir des taux de couverture très importants – de l'ordre de plusieurs dizaines de pourcents. Cependant, les premiers tests effectués sur une dizaines de sites web ont eu tendance à afficher des taux de couverture très bas, entre 1% et 2% en moyenne. Ce faible taux pourrait s'expliquer de plusieurs façons : tout d'abord, il est probable que la taille de la base d'images labellisées que j'utilise pour les comparaisons soit trop faible (seulement 3 000 images alors qu'on pourrait en utiliser plus de 30 000, mais cela ferait malheureusement grimper le temps de calcul). Il est aussi possible que la reconnaissance de duplicatas soit rendue difficile du fait que ces sites transforment énormément les images avant de les héberger (notamment en les recadrant, ce qui les rends impossible à détecter avec le système actuel). Enfin, il se pourrait tout simplement que ce taux soit correct, et les sites de programmes TV ne se volent pas autant d'images que l'on aurait imaginé. Cette dernière hypothèse reste cependant peu probable, car après une étude rapide de ces différentes sites, on remarque que ceux-ci proposent effectivement les mêmes images dans 90% des cas.

Ce faible taux de couverture a pour conséquence qu'on ne détecte que peu d'images de "bonne qualité" sur les sites à étudier : environ 30 à 100 images en fonction du volume de la base de données. Cependant, cela sera peut être suffisant pour extraire des règles de structurations qui permettraient de reconnaître comment sont intégrées les images importantes au sein du site web.

Extraction des règles de structuration

L'extraction automatique des règles de structuration des sites de programme TV était l'un des objectifs optionnels de mon stage. L'idée était d'utiliser toutes les pages web où l'on trouvait des images "doublons", et de les analyser pour en extraire une logique qui nous permettrait ensuite de reconnaître les images utiles, et de les associer automatiquement à leur programme.

Je n'ai pas eu le temps d'accomplir cet objectif, notamment parce que le crawler de Jean-Baptiste ne récupérait toujours pas le contenu des pages web, mais seulement des liens côté serveur web et côté serveur image. Ces URL auraient pu être suffisantes pour décider quelles images étaient importantes où non (car les icônes ne sont pas stockées au même endroit que les aperçus de programme dans les sites web), mais pas pour extraire le programme TV lié à l'image. Pour cela, il faudrait récupérer pour chaque image les balises HTML autour de celle-ci afin d'extraire des informations utiles (exemple : on trouve très souvent le nom du programme dans le champ *alt* des balises <img>).

```
https://www.programme-tv.net/programme/autre/r305698-m6-music/
https://www.programme-tv.net/programme/culture-infos/17631376-a-la-legere/#chaine=france-3-7
https://www.programme-tv.net/programme/divertissement/r206066-amour-gloire-et-beaute/15806309-amour-gloire-et-beaute/
https://www.programme-tv.net/programme/divertissement/r206066-amour-gloire-et-beaute/15806309-amour-gloire-et-beaute/
https://www.programme-tv.net/programme/culture-infos/17887752-voyage-au-bout-de-la-nuit/
https://www.programme-tv.net/programme/autre/r307489-le-grand-betisier-de-lete/17863044-le-grand-betisier-de-lete/#chaine=tf1-19
https://www.programme-tv.net/programme/series-tv/r768-ma-famille-dabord/803507-mauvais-joueurs/
https://www.programme-tv.net/programme/jeunesse/r235560-a-table-les-enfants/7152414-le-gingembre/
https://www.programme-tv.net/programme/culture-infos/r1549434047-terra-terre/17757721-terra-terre/#chaine=la-chaine-parlementaire-11
https://www.programme-tv.net/programme/autre/r1549474801-kiffons-lete-premiere-partie/17833537-kiffons-lete-premiere-partie/
https://www.programme-tv.net/programme/culture-infos/17484560-trafic-denfants/#chaine=arte-337
https://www.programme-tv.net/programme/sport/17887726-formula-one-le-mag/
```

FIGURE 5.3 – Exemple de liens (serveur web) correspondant à des doublons

Récapitulatif

À travers cette partie, nous avons vu comment utiliser les programmes conçus lors de mon stage et celui de Jean-Baptiste pour récupérer des grandes bases d'images, les comparer, et extraire le taux de couverture de nouveaux sites web. La dernière étape, qui reste à réaliser, consistera à extraire des règles de structuration à partir des métadonnées récupérées par le crawler de Jean-Baptiste et les doublons détectés par mon système. Cette démarche est résumée à travers le schéma ci-dessous.

L'objectif à terme serait d'embarquer toutes ces étapes dans un programme autonome ou semi-autonome, qui se lancerait à chaque ajout d'un nouveau site de programme TV. Il serait aussi intéressant de lancer le programme quelques fois par mois sur les sites utilisés pour mettre à jour le scrapper lorsque les sites changent leur structure.

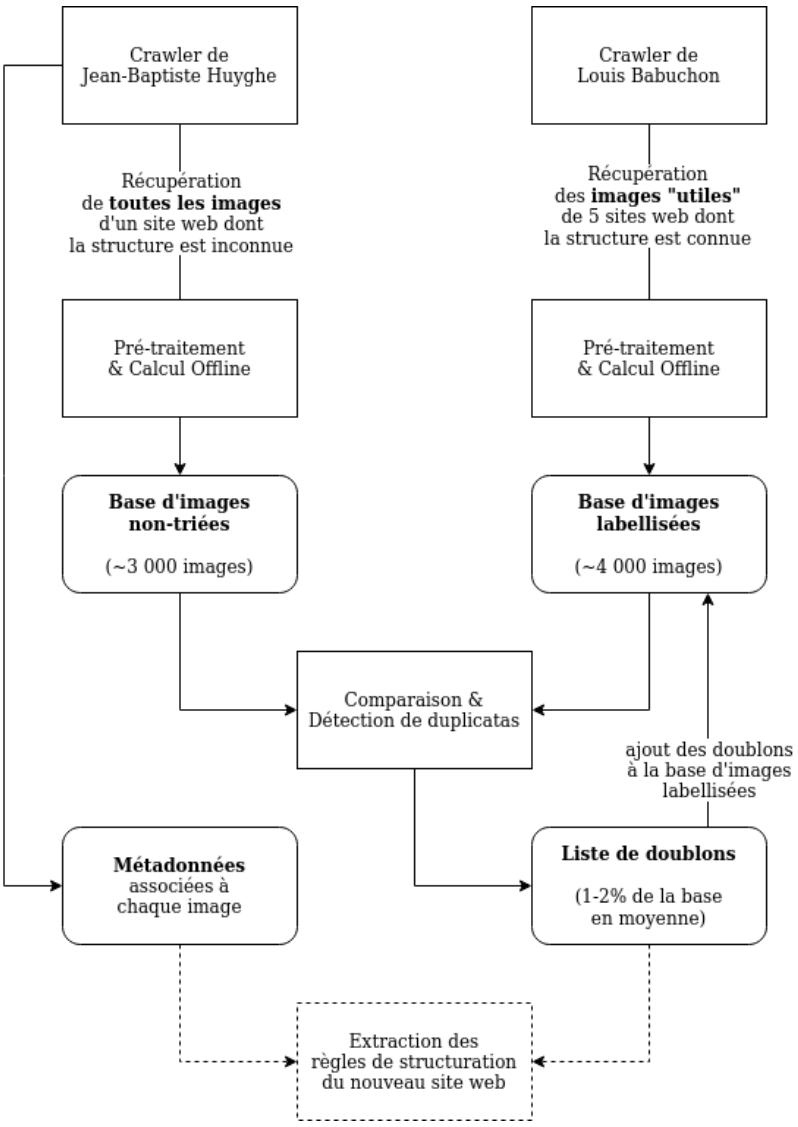

FIGURE 5.4 – Résumé de la méthode

6. Conclusion

Ce stage a été ma première expérience professionnelle en télétravail. L'idée de réaliser un stage de deux mois sans occuper une seule fois les bureaux de l'entreprise m'a d'abord laissé perplexe, mais je ressors finalement satisfait de cette expérience. La communication avec mon tuteur et le reste de l'équipe de ToddTV a été efficace tout au long du stage, grâce à des visioconférences fréquences et des réunions de groupe pour faire un point sur l'avancée du projet. Ce travail, réalisé en autonomie, a renforcé ma capacité à travailler seul et à organiser mon travail.

Ce stage a été d'autant plus formateur pour moi car il s'apparentait plus à un projet de recherche qu'à la production d'un produit fini. Le monde de la recherche étant un milieu qui m'attire, j'aimerai continuer mes études vers un doctorat , et ce stage m'a conforté dans ce projet. Les thématiques abordées – la reconnaissance d'images et l'optimisation – sont deux sujets qui m'intéressent fortement et que j'aimerai approfondir dans de futurs stages.

D'un point de vue technique, ce stage a été aussi très intéressant car j'ai pu me perfectionner en Python, un langage que je ne connaissais qu'en surface mais dont j'ai découvert les avantages au fur et à mesure de mon avancée. Ce langage, bien que peu performant, permet de réaliser des prototypes et des tests très rapidement et efficacement grâce à sa syntaxe simple et son grand nombre de librairies. J'ai aussi pu en apprendre plus sur le domaine de l'image-matching et de l'optimisation, deux sujets que j'avais peu abordé précédemment.

7. Bibliographie

- Nakhmani A, Tannenbaum A. *A New Distance Measure Based on Generalized Image Normalized Cross-Correlation for Robust Video Tracking and Image Recognition*. Pattern Recognit Lett. 2013 ;34(3) :315-321. doi :10.1016/j.patrec.2012.10.025

- Louis Babuchon, *Scraping d'image smart pour portail guide média TV/vidéo*, Projet Recherche & Développement, École Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

- Naifeng Gan, *The Exact Duplicate Image Detection for TV Guide Image Crawling*, Projet Recherche & Développement, École Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

Stage détection de duplicata pour le CBIR TV / VoD

Rapport de stage 2019 - 2020

Résumé : L'objectif du stage est la mise en place et l'optimisation d'un système de détection de duplicata pour le CBIR TV / VoD. Cela couvre différents aspects comme l'étude des méthodes de signatures d'images pour la détection de duplicata (robustes aux déformations en contraste, échelle et crop) et leur optimisation (calcul offline, parallélisation, vectorisation, upper-bounding, etc.). Le stage aura pour principal objectif l'identification de la structuration des images TV / VoD sur les portails web pour le scraping supervisé.

Mots clé : Image matching, Détection de duplicatas, SAD, SSD, ZNCC, Optimisation

Abstract : The objective of the internship is to set up a duplicate detection system for the CBIR TV / VoD. This covers various aspects such as the study of image signature for the detection of duplicates (robust to deformations in contrast, scale and crop) and their optimization (offline calculation, multiprocessing, vectorisation, upper-bounding...). The main objective of the internship will be to identify the structuring of TV / VoD images on web portals for supervised scraping.

Keywords : Image matching, Exact duplicates detection, SAD, SSD, ZNCC, Optimisation

Ce document a été formaté selon le format StagePolytech.cls (N. Monmarché, modifié par A. Friot)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
http://www.polytech.univ-tours.fr