Mines ParisTech

Novembre 2018

MIG Systèmes Embarqués
Urbanisme et IA
–
Rapport écrit

CMA Centre de Mathématiques Appliquées

MINES ParisTech

n a m.R

Gaël Ancel, Maya Boumerdassi, Martin Briand, Colin Cointe,
Vincent Coz, Théo Dumont, Anthony Gandon, Sarah Grislin,
Victor Guillot, Félix Kahane, Alexiane Laude, Tanguy Magne,
Félix Marty, Tom Szwagier, Raphaël Toledano

Remerciements

Nous tenons tout d'abord à remercier nos encadrants, Valérie Roy et Matthieu Denoux, pour la confiance qu'ils nous ont accordée, leur disponibilité durant ces trois semaines et leurs conseils, qui nous ont permis de mener notre projet à terme.

Nous remercions aussi Nicolas Berthelot, Guillaume Larcher et Sébastien Ohleyer de nam.R pour nous avoir proposé ce projet, présenté leur travail, et avoir su répondre à nos questions durant ces trois semaines.

Pour nous avoir fait découvrir pendant ce projet une multitude de facettes des Systèmes Embarqués, nous tenons à remercier : Xavier Rival et Laurent Théry, qui nous ont permis d'entrevoir les principes mathématiques sur lesquels reposent ces systèmes complexes ; Guillaume Foeillet, Jean-Michel Lécuna et Lionel Daniel, pour nous avoir respectivement fait visiter la Rame *Iris* de la Sncf, Dassault Aviation (*Istres*) et de Thales Alenia Space (*Cannes*). Ces trois visites ont été très enrichissantes et nous ont fait découvrir des applications industrielles de ces systèmes embarqués.

Nous remercions également Pauline Frey et Vincent Rey de l'école 42, ainsi que tous les élèves de l'école 42 ayant participé au projet commun, d'avoir organisé cette journée durant laquelle nous avons pu développer de vraies compétences de travail de groupe.

Et enfin, nous adressons nos remerciements à Nathalie Borgeaud, à Daniel Florentin et à Mélusine Hucault, qui ont su nous faire entrevoir les problématiques auxquelles doivent faire face les urbanistes en terme de développement urbain durable et. Un grand merci à Lyes Khacef de l'UCA, qui nous a présenté avec brio les bases du *machine learning* et sans qui les premiers pas de notre projet auraient été erratiques.

Table des matières

Table des figures

Liste des tableaux

Partie I
Motivation et cahier des charges

1 Motivation

1.1 Le développement durable, un aspect incontournable de l'urbanisme

Le développement durable est devenu une des problématiques majeures dans le développement des villes et la mise en place de plans d'urbanisme [17]. Aujourd'hui, il faut considérer de nombreux facteurs liés au climat, à l'énergie ou à la mobilité des individus. Ces nouvelles contraintes se traduisent entre autres par de nouveaux écrits encadrant le développement des villes : par exemple, le plan d'aménagement et de développement durable (PADD) accompagne maintenant le plan local d'urbanisme (PLU).

Une urbanisation durable aboutit en effet à des résultats concrets : une bonne organisation des axes de mobilité de la ville permet de réduire les temps de trajet, et donc les émissions de gaz à effet de serre ; la mise en place de corridors verts (appelés également trames vertes) permet une meilleure circulation de la faune (très riche dans les villes) ; un accès facile aux espaces verts augmente la qualité de vie des habitants.

Les idées en la matière sont nombreuses et peuvent avoir un impact substantiel sur la transition vers des villes plus durables.

1.2 Le besoin d'information dans un monde qui se numérise

1.2.1 Un besoin d'information

Pour se développer de manière plus durable, une municipalité peut souhaiter disposer d'informations sur l'état de sa végétation. Il lui est par exemple utile de connaître la répartition des arbres ou espaces verts dont elle dispose au sein de son agglomération.

Cela lui permettrait notamment de visualiser directement sur une carte les potentielles nouvelles trames vertes, en s'appuyant sur des alignements végétaux déjà existants. Elle pourrait aussi calculer le taux de CO_2 absorbé par la végétation de son territoire et adapter ainsi ses décisions concernant la gestion des espaces verts et de la qualité de l'air. Il s'agirait donc d'un outil très puissant d'aide à l'éco-gestion du territoire urbain.

1.2.2 Un contexte technologique porteur

Les municipalités s'efforcent de numériser leurs données papiers pour créer des bases communes et plus facilement accessibles. Ce processus s'inscrit dans le Projet de loi pour une République numérique [1], qui demande aux communes de plus de 3500 habitants de mettre à disposition du public leurs données. Ces données sont donc ouvertes à tous et permettent aux citoyens, ainsi qu'aux entreprises, de les utiliser.

Ce n'est cependant pas toujours suffisant car les archives contiennent parfois des données partielles : par exemple, si les arbres publics sont souvent connus par les services de la mairie, ils sont rarement référencés numériquement et la connaissance des arbres privés est souvent manquante.

D'où l'intérêt de compléter ces bases avec des données extrapolées, par des méthodes automatiques et rapides comme les réseaux de neurones utilisés dans le cadre d'analyse d'image, ou des analyses de colorimétrie. En effet, les progrès technologiques permettent d'avoir des images aériennes de certaines zones de plus en plus précises et des outils d'analyses toujours plus puissants.

Ces méthodes, bien qu'encore trop imprécises pour produire une analyse détaillée d'un objet en particulier, ont atteint un niveau de développement suffisant pour donner des résultats statistiques exploitables sur de plus grandes échelles.

1.3 Un projet collaboratif

1.3.1 Notre projet

Notre projet vise à aider cette éco-gestion urbaine. Nous nous sommes fixé pour objectif de produire une application à l'interface intuitive qui permette de synthétiser et comprendre rapidement les caractéristiques d'une zone géographique. Ces caractéristiques sont obtenues :

- à partir des différentes techniques présentées (*machine learning*, analyse d'images) ;
- complétées par des bases de données pertinentes, proposées directement par la mairie de Bordeaux.

1.3.2 Un projet en collaboration avec nam.R

L'entreprise Le projet que nous avons réalisé nous a été proposé par l'entreprise nam.R. Cette société française a pour but de rassembler les nombreuses bases de données existantes à partir desquelles elle construit un jumeau numérique du territoire français grâce aux outils d'intelligence artificielle. Ces données sont utiles aux projets d'urbanisme, et ont vocation à être utilisées par des entreprises, des particuliers ou des collectivités territoriales. À titre d'exemple, nam.R possède des données sur plus de 44 millions de bâtiments sur le territoire national, et est en mesure de déduire de ces données de nouveaux attributs, comme par exemple la nécessité de renouveler la toiture, de réaliser la rénovation de façade, ou encore la possibilité de poser des panneaux photovoltaïques.

Une collaboration Nous avons rencontré une partie des membres de l'entreprise au début du projet. Ils nous ont exposé leurs attentes, nous ont présenté leurs méthodes et nous ont donné quelques pistes pour démarrer. L'entreprise nous a ensuite accompagné durant le projet en répondant à nos interrogations techniques.

2 Cahier des charges

2.1 Objectif

L'objectif de ce projet était de créer une interface graphique permettant un accès rapide et synthétique aux statistiques environnementales d'une zone géographique sélectionnée. À cette fin, la première étape consistait à créer des bases de données adaptées à nos objectifs de développement durable et à proposer des statistiques utiles au développement urbain.

Notre projet nécessitait un grand nombre de données labellisées, notamment pour entraîner nos algorithmes de *machine learning* ou tester nos algorithmes de traitement d'images. La ville de Bordeaux est très en avance en terme de publication des données et dispose d'images aériennes de très bonne qualité sur tout le territoire. C'est donc vers cette ville que nam.R nous a orientés.

2.2 Résultats souhaités

En utilisant ces différentes bases de données brutes, nous souhaitons en produire de nouvelles identifiant des objets importants dans l'optique d'une analyse environnementale.

2.2.1 Sélection d'objets pertinents

Les objets suivants ont été retenus comme utiles pour une analyse environnementale de l'espace urbain de Bordeaux, car particulièrement révélateurs de l'utilisation des sols de la ville.

- arbres ;
- voitures ;
- espaces verts ;
- espaces bitumés ;
- parcelles ;
- bâtiments.

On tracera pour cela une boîte autour de chaque arbre ou structure détectée, comme sur la figure suivante :

Une voiture encadrée

Les données concernant les parcelles et les bâtiments (délimitations et type du terrain) étaient déjà enregistrées sous forme de cadastres informatisés dans les données de Bordeaux et n'ont nécessité aucun traitement de notre part. Quant aux autres objets (arbres, espaces verts...), il nous a fallu créer des données plus exploitables que des images brutes (localisation, taille...).

2.2.2 Indicateurs de développement durable

Une fois les données créées, encore faut-il pouvoir les comprendre et les analyser. Nous avons donc sélectionné un certain nombre de statistiques s'appuyant sur des données géographiques :

1. la densité de végétation (surface végétale par unité d'aire du territoire) ;
2. le taux d'occupation des parcelles (partie construite sur surface de la parcelle) ;
3. la densité d'espaces verts publics pour évaluer l'inégalité d'accès à des parcs ;
4. des propositions de plans d'urbanisme (création de trames vertes ou de parcs plus grands).

Les trois premiers points ont bien été implémentés dans l'application. La création intelligente de trames vertes a bien été programmée mais n'a pas pu être ajoutée à l'interface graphique, faute de temps. Nous avons cependant implémenté une fonctionnalité intermédiaire qui crée de nouveaux espaces verts en cohérence avec ceux déjà existants.

2.3 Logiciels utilisés

Lors de notre travail, nous avons utilisé de nombreux logiciels. On y retrouve de multiples modules de Python, le langage de programmation que nous avons choisi, comme `Rasterio` pour la gestion des images localisées, `GeoPandas` et `Pandas` pour la gestion des données, `OpenCV` pour le traitement des images, `Keras` pour le *machine learning* et `Tkinter` pour le développement de l'application graphique.

D'autres logiciels comme `QGis` et `GeoJSON.io` ont été utilisés pour la visualisation et la vérification de notre travail.

Partie II
Réalisation du projet

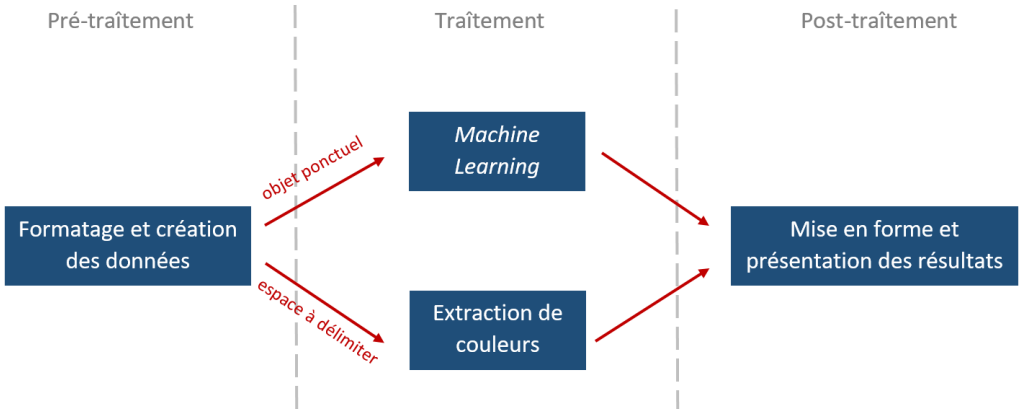


FIGURE 0.1 – Schéma global de notre travail

1 Reconnaissance d'objets par *machine learning*

Pour la reconnaissance d'arbres et de voitures, nous avons choisi d'utiliser le *machine learning*, technique fondée sur les réseaux de neurones. En effet, nous disposions déjà d'une base de données de voitures (DOTA [1]) pouvant servir de base d'entraînement, ce qui facilitait l'implémentation et le test des algorithmes.

1.1 Qu'est-ce que le *machine learning* ?

Nous avons implémenté le *machine learning* sous sa forme nommée « apprentissage supervisé ». Elle se déroule en trois temps : une phase d'apprentissage, une phase de test et une phase d'utilisation.

1.1.1 Apprentissage

Pour la phase d'apprentissage, les données d'entrée sont des images labellisées qui constituent la base d'entraînement.

Un modèle de réseaux de neurones en apprentissage supervisé fonctionne comme suit : pour chaque image de la base d'apprentissage, les différentes couches de neurones [2] s'activent successivement. Quand une couche de neurones s'active, chaque neurone de la couche reçoit des valeurs produites par les neurones de la couche précédente, applique sa fonction d'activation sur la somme pondérées des entrées et émet une valeur de sortie pour les couches suivantes.

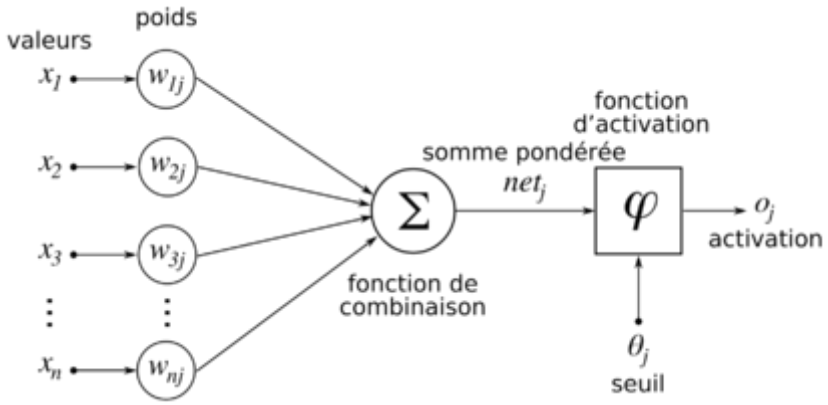


Figure 1.2 – Schéma d'un neurone (crédit : Chrislb, CC-BY-SA 3.0)

Les neurones de la dernière couche effectuent la prédiction finale : le label supposé par le modèle ainsi qu'une probabilité pour quantifier la certitude de cette supposition (présence ou non du véhicule/position du quadrilatère). Cette prédiction est comparée au label associé à chaque image, via la fonction d'erreur choisie, et le module `Keras` (une API pour les réseaux de neurones écrite en Python) implémente un procédé de mises à

1. pour *Dataset for Object Detection in Aerial Images.*
2. Un neurone possède des entrées, des poids pour pondérer les entrées, une fonction d'activation (linéaire ou non), et des sorties.

jour des poids de toutes les entrées de chaque neurone pour se rapprocher du label réel (*optimizer* `Adam` [15]).

1.1.2 Test

L'étape suivante est une phase de test qui consiste à appliquer le réseau de neurones entraîné à des images dont on cache le label. L'algorithme essaye d'extrapoler, de prédire ce dernier mais ne met plus à jour les poids des neurones. Il utilise uniquement les labels pour évaluer ses propres performances, en pourcentage de labels correctement identifiés. Avec les les réseaux de neurones sur lesquelles nous les avons entraînés, nos modèles atteignent les 87% de labels corrects sur cette phase de test.

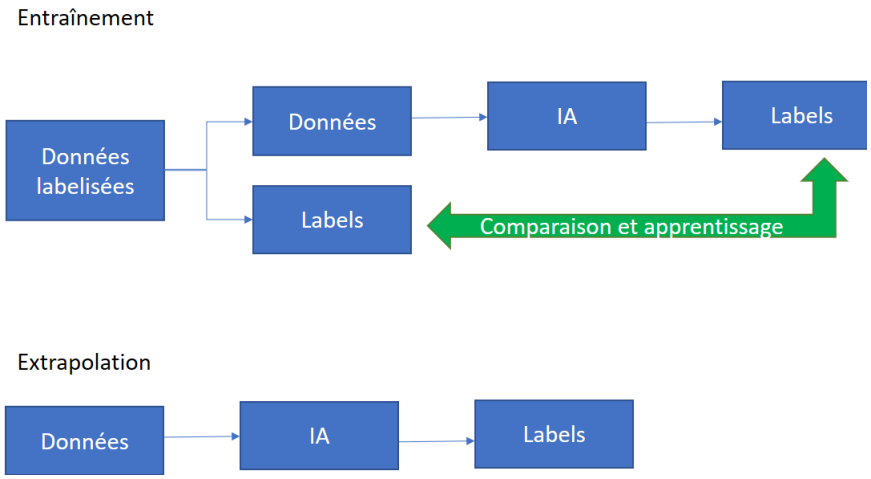


Figure 1.3 – Processus d'entraînement et de prédiction d'un algorithme de *machine learning*

1.1.3 Prédiction

La dernière phase consiste à utiliser l'algorithme testé et évalué sur les images de la ville de Bordeaux (base d'images assez différente de celle de l'entraînement et du test) pour produire des labels, qui seront ensuite utilisés pour caractériser l'urbanisation de la ville.

1.1.4 Le réseau, concrètement

Pour construire le réseau de neurones chargé « d'apprendre », nous devons créer la structure des couches d'apprentissage, qui sont à choisir parmi les types exposés ci-après (liste non-exhaustive) :

Objet du module Keras	**Description**
`Dense()`	couche de neurones avec ou sans biais
`Flatten()`	aplatit une matrice de pixels en un vecteur
`Dropout()`	attribue la valeur 0 à un pourcentage p de neurones de la couche précédente, pour empêcher le sur-entraînement
`Convolution2D()`	fait glisser sur l'image un filtre appliquant une transformation élémentaire sur chaque sous-image
`MaxPooling2D()`	fait glisser une fenêtre sur une matrice de pixels et ne conserve pour chaque fenêtre que celui de norme maximale

Table 1 – Description des modules `Keras`

Les choix à réaliser lorsque l'on crée un modèle de réseau de neurones sont dictés par l'expérience plutôt que par la théorie. En partant d'une première version de *machine learning* avec une structure basique de couches, on a testé l'influence de chaque type de couche et de leur position dans le réseau. On a systématiquement fait varier chaque paramètre et chaque position des couches pour trouver empiriquement quelle combinaison s'adaptait le mieux à notre problème.

Nous avons finalement utilisé les trois versions ci-dessous, qui donnaient les meilleurs résultats, dont un réseau de neurones à convolution.

Structure	***Accuracy***	***Loss***
`Flatten`, `Dense(128)`, `Dense(128)`, `Dense(2)`	0.8747	0.3474
`Flatten`, `Dense(128)`, `Dropout(0.2)`, `Dense(2)`	0.8705	0.3694
`Convolution2D`, `MaxPooling2D`, `Dense(128)`, `Dropout`, `Dense(2)`	0.8940	0.3278

Table 2 – Meilleurs résultats de l'étude comparative et empirique du rôle de chaque couche

1.2 Idée générale

1.2.1 But

Nous voulions créer un algorithme de reconnaissance qui, en plus, encadre l'élément reconnu. Plus précisément, on applique l'algorithme sur une image et ce dernier doit renvoyer une probabilité caractérisant la présence (ou non) d'un arbre (ou une voiture), ainsi qu'un 8-uplet contenant les coordonnées (x, y) des quatre points formant le polygone encadrant.

1.2.2 Détails des étapes

Une question se pose : que renvoyer comme polygone si l'algorithme ne détecte pas d'arbre ? Entraîner l'algorithme à renvoyer une figure aléatoire ou de taille nulle aurait un impact hasardeux sur les cas où il détecte bien des arbres.

Implémentation Nous avons donc séparé le problème en deux algorithmes différents : le premier servira uniquement à la reconnaissance d'arbres (ou de voitures). Le deuxième algorithme aura la tâche de placer le polygone encadrant sur l'image. Il ne prendra donc en entrée que des images sur lesquelles la probabilité d'avoir un arbre (probabilité obtenue par le premier algorithme) sera suffisante, *i.e.* plus grande qu'un certain seuil prédéfini au préalable.

1.3 Récupération de la base de données

L'apprentissage, comme son nom l'indique, nécessite des données préalablement labellisées sur lesquelles notre modèle peut s'entraîner, pour ensuite être capable de faire des prédictions raisonnables sur des cas inconnus.

Il est donc nécessaire de constituer une base de données de photos de l'objet à reconnaître (arbre ou voiture) prises depuis une vue aérienne, et une autre base de photos aériennes d'objets différents.

1.3.1 Images de voitures

Pour cela, nous avons choisi d'utiliser la base de données DOTA contenant en particulier des voitures en vue aérienne, déjà labellisées.

On extrait ainsi 10 000 images de voitures de taille 48×48 pixels. Puisque notre modèle de classification devra ensuite déterminer si une image est une voiture ou non, il est aussi nécessaire de constituer une base d'images aériennes de même taille (48×48 pixels), qui ne sont pas des voitures, ce que nous avons également fait grâce à la base DOTA en extrayant 10 000 images de non-voitures. Des images de non-voitures sont également extraites aléatoirement des images aériennes en format `.jpg` de la ville de Bordeaux pour enrichir la base de données (environ 25 000 images choisies dans des zones avec peu de véhicule).

Il s'agit ensuite d'importer les images sur Python et de leur associer un label, idéalement : [1,0] pour les voitures et [0,1] pour les autres images [3].

3. Ce format équivaut à un booléen mais il permet en plus d'utiliser les fonctions d'erreurs préconçues

1.3.2 Images d'arbres

L'algorithme de reconnaissance d'arbre va s'entraîner sur les données fournies par Open Data Bordeaux, via le fichier recensant les arbres publics. Plus particulièrement, les données prises en entrée sont des fichiers contenant :

- une matrice de taille $(100, 100, 3)$, représentant l'image de taille 100×100 pixels en RGB [4] ;
- un tableau de 8 éléments, contenant les coordonnées du polygone encadrant ;
- un couple de probabilités $(p_a, 1 - p_a)$: p_a vaut 1 si l'objet est un arbre, 0 sinon [5] ;
- un tableau de deux éléments contenant la géolocalisation de l'image.

L'algorithme se servira de l'image, de la probabilité (qui nous sert de label), et des coordonnées du polygone. Les données sont enregistrées sous un format binaire [6].

1.4 Réseaux de neurones

Nous utilisons des réseaux de neurones à convolution [7] comme décrits en section (1), page 8.

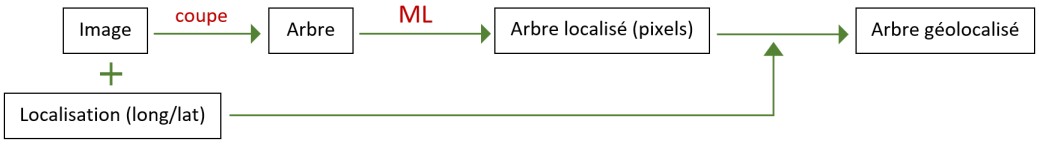


Figure 1.4 – Schéma du traitement par *machine learning*

1.4.1 Modèle de classification

Cet algorithme s'occupe uniquement de la séparation des *inputs* en deux catégories : image d'un arbre ou non, en donnant en sortie un pourcentage indicatif. La fonction d'erreur est alors classiquement la fonction de `loss binary_crossentropy` [4].

Résultats Voici un exemple de résultats obtenus sur les images aériennes découpées de la ville de Bordeaux :

pour la classification d'images.

4. *Red Green Blue*, abrégé en RGB, est un système de codage des couleurs se basant sur la synthèse additive des couleurs.

5. Même remarque que pour les labels des voitures dans le paragraphe précédent.

6. Le processus d'obtention des données pour les arbres est plus amplement expliqué dans la section (2), page 14.

7. Un réseau de neurones contenant au moins une couche de convolution.

(a) Un arbre (b) Aussi un arbre (c) Pas un arbre

Figure 1.5 – Ce que nous considérons comme étant un arbre

L'algorithme de détection d'arbres fournit des résultats concluants. En effet, après apprentissage, nous obtenons une *accuracy* (évaluation de l'exactitude des prédictions sur la base de test) de 96.63% et une *loss* (valeur moyenne des erreurs calculée avec la fonction d'erreur utilisée) de 0.5378, ce qui est empiriquement satisfaisant. Pour les représentations graphiques ci-dessous, on affiche l'évolution de la précision (à maximiser) et de l'erreur (à minimiser) au cours des phases d'apprentissage (*epochs*). Cela nous permet d'éviter l'écueil du « sur-apprentissage », c'est à dire lorsque le modèle apprend par cœur les caractéristiques des données d'entraînement, et ne réalise plus de prédictions valides pour de nouvelles images à analyser faute de capacité de généralisation à des données pas encore vues.

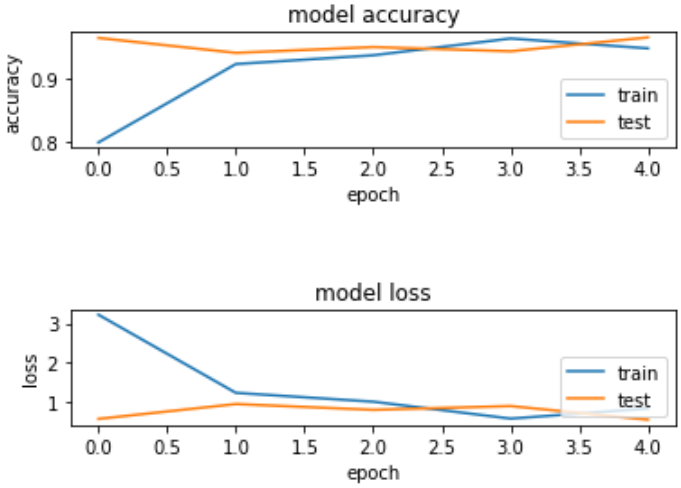


Figure 1.6 – Fonctions *loss* et *accuracy* pour l'algorithme de reconnaissance d'arbres pour le set d'images d'entraînement (*train*) et de test

En pratique, l'algorithme détecte bien les arbres, surtout ceux isolés. Néanmoins, il rencontre quelques difficultés pour différencier les pelouses et buissons des arbres.

1.4.2 Modèle d'encadrement

Grâce au traitement des données publiques concernant les arbres de la ville de Bordeaux, nous avions déjà des données labellisées avec le polygone d'intérêt. Nous pouvions donc directement utiliser ces images pour les bases d'apprentissage et de test.

Objectif Cette fois-ci, l'algorithme doit renvoyer un 8-uplet correspondant aux coordonnées (x, y) en pixel des coins du rectangle encadrant l'arbre ou la voiture. Ces données doivent être normalisées [8] avant d'être transmises à l'algorithme pour l'apprentissage.

Le principe de fonctionnement est assez différent de l'algorithme de reconnaissance, et certaines modifications ont dû être faites dans l'architecture du réseaux de neurones. En effet, le problème considéré n'est plus de prédire exactement le label objet ou non-objet pour une image donnée, mais de prédire suffisamment précisément les 8 coordonnées du polygone encadrant, en sachant qu'il n'y a pas unicité de celui-ci, et que la définition même de ce polygone est très empirique.

Nouvelle mesure La nouvelle fonction d'erreur doit prendre cela en compte en donnant une mesure de l'écart entre les bonnes coordonnées et celles prédites. Les deux fonctions les plus courantes sont l'intersection sur union et la norme euclidienne. La première serait la plus pertinente mais n'est pas différentiable, ce qui nous a conduit à implémenter la seconde (*mean_ square_ error* [4]).

Accuracy L'*accuracy* n'a plus de sens à cause de la multiplicité des solutions possibles ; seule la valeur de la fonction d'erreur nous donne empiriquement une indication sur la précision de l'algorithme. Pour la délimitation des arbres, nous obtenons une *loss* de 0.34 après apprentissage, ce que nous avons jugé satisfaisant après des calculs probabilistes d'espérance de notre métrique pour une image donnée.

Résultats En pratique, lorsqu'un arbre isolé est détecté, l'algorithme a tendance à tracer un polygone pertinent, quoique légèrement décalé vers le centre ; les résultats sont moins bons sur les zones forestières denses (où encadrer un arbre en particulier devient plus compliqué quand leurs branches s'entrelacent vu du ciel) et sur les champs, où certains faux positifs demeurent. Les résultats restent cependant exploitables.

8. Les données sont entre 0 et 100, on les ramène par proportionnalité entre 0 et 1.

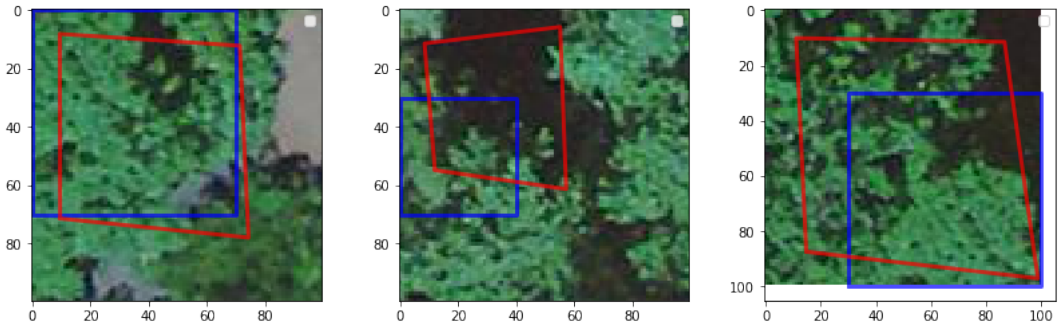


Figure 1.7 – Exemples d'encadrements d'arbres (en bleu la délimitation attendue, en rouge la délimitation produite par l'algorithme)

1.5 Algorithme d'encadrement des voitures

Pour plus de précision quant à notre démarche à travers le *machine learning*, voir l'Annexe (A), page 39.

1.6 Format de sortie

L'algorithme d'encadrement ne prend une image en entrée que si la probabilité qu'un arbre soit présent sur celle-ci est supérieure à un certain seuil fixé, et ne la considère pas sinon. Le seuil a été établi empiriquement en fonction de la précision de la reconnaissance d'arbre.

Enfin, grâce à la localisation de l'image et l'échelle des images satellites, on récupère les coordonnées géographiques du cadre fourni en sortie de l'algorithme, et l'on peut créer une constellation de fichiers géographiques (un par image de Bordeaux) exploitables par l'application graphique finale [9].

1.7 Synthèse sur le *machine learning*

Nous avons donc réussi à implémenter des algorithmes de reconnaissance d'objets et de détection de leur position sur une image. Ces algorithmes sont génériques et pourraient s'appliquent à d'autres types d'objets plus ou moins isolés – comme des piscines ou des toits de bâtiments –, mais nous ne les avons appliqués qu'à des véhicules et des arbres, objets qui nous intéressaient dans notre étude.

Pour les autres types d'objets (des zones par exemple), nous avons préféré chercher d'autres méthodes plus efficaces. Ces méthodes seront abordées en Section (3), page 22.

9. *cf.* Annexe (C.1), page 63.

2 Pré-traitement des données pour le *machine learning*

Comme cela a été dit dans la partie précédente, l'intelligence artificielle permet d'analyser de grandes bases de données à condition que celles-ci soient au bon format. Elle a aussi besoin d'un jeu d'entraînement conséquent et au même format. Cette partie vise à expliquer comment ont été obtenus le jeu d'entraînement et la base de prédiction pour les arbres.

2.1 Les données disponibles

La majorité des données qui nous ont servi pour les arbres proviennent du site [9] Open Data Bordeaux. Nous avons utilisé :

- des photos aériennes et géolocalisées de Bordeaux et son agglomération (résolution de 10 cm par pixel) ;
- un fichier `.csv`[10] recensant certains arbres publics et contenant beaucoup d'informations sur chaque arbre[11].

Les donnés d'entraînement pour les voitures proviennent quant à elles du site [10] Dota-DataSet.

2.2 Définition du format des données

Pour pouvoir travailler en groupe sur le même projet, il nous a fallu nous mettre d'accord sur le format des données que nous souhaitions produire et de celles que nous voulions fournir aux algorithmes de *machine learning*.

Pour entraîner un algorithme de *machine learning*, il faut un jeu de données labellisées et un jeu de données de test que l'intelligence artificielle tentera de labelliser.

On distingue les données d'entraînement et de prédiction.

2.2.1 Les données d'entraînement

On rappelle[12] que les données prises en entrée sont des fichiers contenant :

- une matrice de taille $(100, 100, 3)$;
- un tableau de 8 éléments, contenant les coordonnées du polygone encadrant ;
- un couple de probabilités $(p_a, 1 - p_a)$;
- un tableau de deux éléments contenant la géolocalisation de l'image.

Nous avons également élaboré une base de données d'images ne contenant pas d'arbre. Le format pris en compte par l'algorithme est le même que pour les images d'arbres. Pour s'assurer qu'il n'y avait pas d'arbre sur ces images, nous les avons extraites de photos de zones construites comme des routes ou des bâtiments.

10. *comma separated values*, format de stockage de données brutes sous forme de tableau.
11. localisation, famille, genre, âge, diamètre, hauteur, localisation exacte.
12. *cf.* section (1.3.2), page 10

Pour ce faire, nous avons superposé les zones bâties déjà géolocalisées avec les images satellites de Bordeaux, puis nous avons découpé les images et les avons converties au format souhaité.

2.2.2 Les données de prédiction

Pour la prédiction, nous avons découpé les images de Bordeaux en plusieurs sous-images géolocalisées encore une fois exportées dans un fichier binaire. Les algorithmes de *machine learning* ont ensuite pu prédire des quadrilatères d'encadrement (coordonnées en pixels) pour les images où un arbre a été détecté.

Enfin, en remontant à la localisation de l'image, on peut créer des bases de données localisées recensant les arbres pour chaque image de Bordeaux [13].

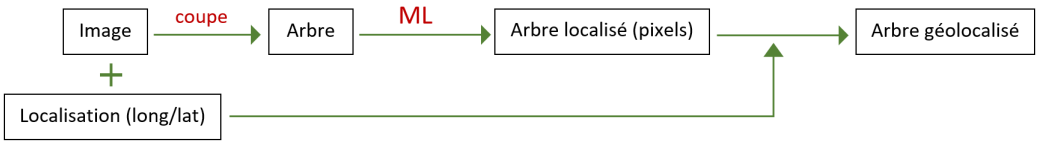


Figure 1.3 – Schéma du traitement par *machine learning*

2.3 Processus de création des bases de données

Pour parler de photo géolocalisée, il est nécessaire de choisir un système de référence spatiale adapté à la localisation étudiée. Comme les photos étaient référencées en `epsg3945` et que ce système est extrêmement précis pour les latitudes qui nous intéressent (entre 44°et 46°), nous l'avons conservé.

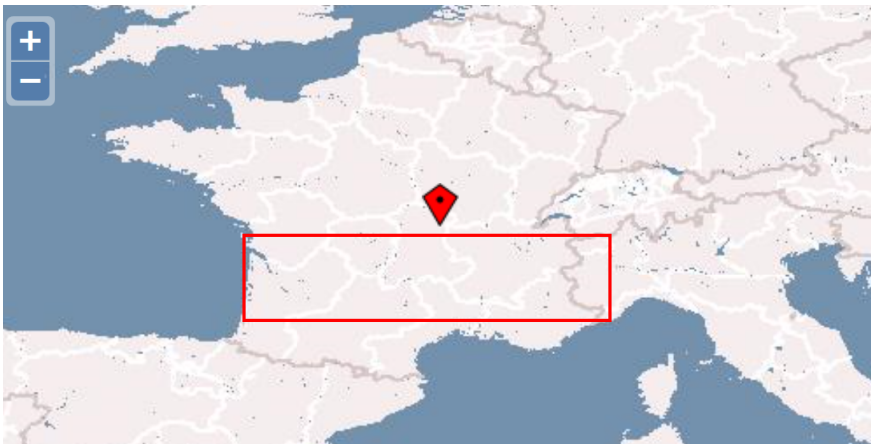


Figure 2.1 – Zone de prédilection de `epsg3945` (source : `spatialreference.org`)

13. *cf.* Annexe (C.1), page 63

2.3.1 Les données d'entraînement

Ces données sont utilisées pour entraîner l'algorithme et doivent donc être labellisées. Détaillons ce processus.

Étape 1 – Récupération des données On récupère les données libres du site Open Data Bordeaux :

- Il s'agit d'un fichier `.csv` contenant la géolocalisation des arbres (en `epsg 4326`, latitude longitude) et diverses informations (*cf.* section (2.1), page 14).
- On utilise ensuite le module `Pandas` de Python pour créer une structure de base données (`DataFrame` [14]) que l'on pourra plus facilement traiter ensuite.

Étape 2 – Création de la boîte L'objectif est ensuite de créer une boîte autour des arbres.

1. Certaines informations étant inutiles, on ne conserve que les données concernant la hauteur de l'arbre, le diamètre de sa couronne [15] ainsi que ses coordonnées. On obtient une table de ce type :

	HAUTEUR	DIAMETRE_COURONNE	X_LONG	Y_LAT
0	10.0	8.0	-0.580935	44.840838
1	5.0	4.0	-0.580936	44.840897
10	13.0	8.0	-0.580945	44.841451
100	10.0	6.0	-0.577515	44.832995
10000	15.0	6.0	-0.572678	44.844497
10001	19.0	8.0	-0.574264	44.843875
10002	18.0	4.0	-0.574113	44.843895
10003	22.0	10.0	-0.574045	44.843900
10004	22.0	8.0	-0.573899	44.843913
10005	22.0	9.0	-0.573756	44.843929
10006	22.0	9.0	-0.573610	44.843941
10007	22.0	7.0	-0.573462	44.843956
10009	9.0	5.0	-0.558371	44.812501

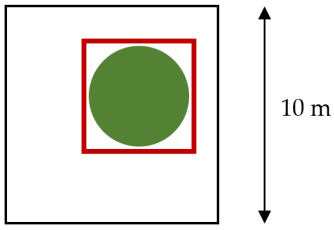
Figure 2.2 – Table de données obtenue à partir d'un fichier `.csv`

2. Pour construire la boîte, on crée 4 colonnes qui contiennent les coordonnées des coins de la boîte de l'arbre.
3. On doit ensuite convertir la table en un fichier `.geojson` en prenant garde à bien choisir la projection utilisée (`epsg3945`).

Étape 3 – Découpe

1. Il nous faut créer les coordonnées des points de découpe de l'image : on choisit de produire des images de 10 mètres par 10 mètres [16].

14. objet propre à `Pandas`, qui correspond en fait à une table de données comme on pourrait en trouver en SQL.
15. diamètre de l'arbre vu de dessus avec les feuilles.
16. Ce choix de dimensions est expliqué dans la sous-section (2.3.2), page 19.



On crée donc un carré englobant l'arbre de côté 10 mètres. On obtient finalement un fichier `.geojson` avec pour `properties` [17] :

- `HAUTEUR`
- `DIAMETRE_COURONNE`
- `X_LONG` la longitude du centre de l'arbre ;
- `Y_LAT` la latitude du centre de l'arbre ;
- `Xi_BOX` et `Yj_BOX` les coordonnées de la boîte ;
- `geometry` qui contient un polygone avec les coordonnées des bords de l'image à découper.

2. Il faut enfin découper les images. On les lit à l'aide du module `Rasterio` qui permet d'obtenir les coordonnées de l'image. On les stocke dans une liste. Puis, pour découper, on utilise deux boucles imbriquées :
   (a) on parcourt toutes les images en les ouvrant avec `opencv` pour obtenir la matrice de pixels en RGB ;
   (b) on parcourt tous les arbres.

   On vérifie ainsi si l'arbre est bien dans l'image et si c'est le cas, on découpe notre matrice au bon format. Dans une liste de longueur quatre, on met ensuite : la matrice de l'image, les coordonnées en pixels de l'arbre dans l'image de cent pixels, $p = 1$ [18] et les coordonnées du coin supérieur gauche de l'image :

```
[img, coord_box, 1, géolocalisation]
```

17. *cf.* Annexe (C.1), page 63
18. la probabilité d'avoir un arbre sur l'image

Après découpe, une image ressemble à ceci :

Figure 2.3 – Image après découpe

On fait le choix de découper l'arbre de plusieurs façons pour assurer une plus grande diversité des images en entrée :

- l'arbre en entier ;
- l'arbre coupé au milieu ;
- l'arbre décentré.

Ceci est nécessaire si l'on veut pouvoir repérer les arbres n'étant pas présents entièrement sur une image ou étant décentrés. On apprend ainsi à l'algorithme de *machine learning* à détecter des portions d'arbres (méthode conseillée par nam.R).

Figure 2.4 – Diversité des découpes d'arbres

On peut finalement résumer le processus de cette manière :

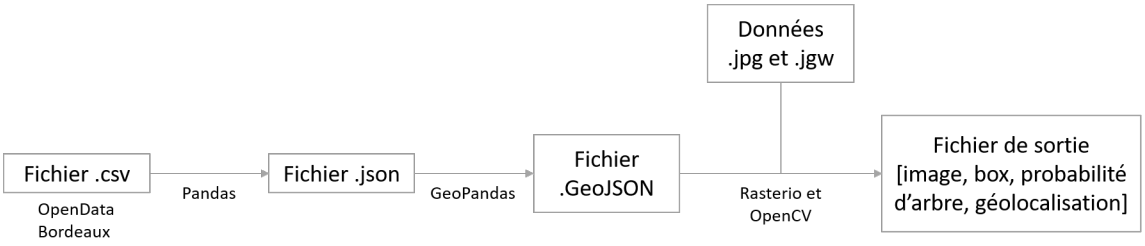


Figure 2.5 – Processus de formatage des données

2.3.2 Les données de prédiction

Une fois entraîné, l'algorithme aura pour tâche de prédire sur ces données si un arbre est présent, et dans ce cas, l'encadrer.

Il faut donc, à partir des photos de Bordeaux, découper des centaines de milliers de sous-images et les localiser.

Figure 2.6 – Découpage d'une des 527 images (quadrillage grossi)

Deux contraintes s'imposent cependant :

- avoir des photos de même taille tout en conservant l'intégralité de l'image. En Figure 2.7, les zones rouges sont perdues si le découpage n'est pas adapté.
- choisir une taille d'image adaptée : suffisamment grande pour qu'un arbre puisse y tenir et suffisamment petite pour qu'il n'y ait que rarement plus d'un arbre par image.

Les tailles de découpe possibles Nos photos découpées doivent être carrées puisqu'un arbre n'a *a priori* pas d'orientation privilégiée. Pour éviter de perdre des morceaux

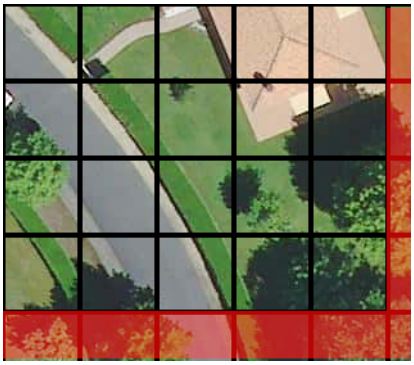


Figure 2.7 – Une contrainte de découpe : conserver la totalité de l'image traitée

d'image, il faut donc choisir une longueur de découpe qui soit un diviseur de ces deux nombres. Or

$$\text{pgcd}(14000, 10000) = 2000 = 2^4 \times 5^3.$$

Avec notre résolution au décimètre, on se limite donc à des images de 0.2, 0.4, 0.5, 0.8, 1.0, 1.6, 2.0, 2.5, 4.0, 5.0, 8.0, 10.0, 12.5, 20.0, 25.0, 40.0, 50.0, 100.0 ou 200.0 mètres.

Choisir une taille adaptée à nos arbres Le mieux est d'étudier les arbres qui existent déjà. La base de données produite à partir du fichier `.csv` permet de produire l'histogramme suivant sur le diamètre couronne des arbres :

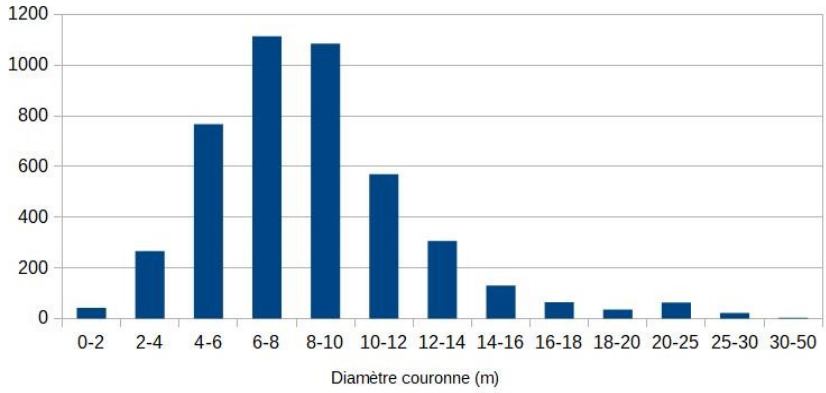


Figure 2.8 – Nombre d'arbres référencés par diamètre de couronne

Quitte à apprendre à notre algorithme à reconnaître des demi-arbres (méthode conseillée par nam.R), le meilleur choix pour s'assurer que l'on ait peu de photos avec plus d'un arbre est de prendre des carrés de dix mètres de côté (soit 100 pixels).

Nécessité de restreindre notre zone d'étude Lancer une telle opération d'extraction sur toute la commune de Bordeaux (527 images) nécessite 19 heures de calcul et 200 Go de stockage. Il fallait donc réduire le nombre d'images à explorer. En ne conservant que les images sur lesquelles il y a au moins un arbre déjà référencé dans les données de Bordeaux, on se retrouve à découper 29 photos et on obtient 406 000 fichiers `.bin` pour un total de 11.4 Go.

Conclusion Finalement, des données d'entraînement et de prédiction ont été créées pour les arbres. Celles-ci ont servi au *machine learning* décrit dans la partie précédente. Des fichiers d'arbres localisés ont donc été ensuite créés et servent directement dans l'interface graphique décrite dans la section (4), page 31.

3 Une autre approche : détection d'espaces verts par extraction de couleurs

3.1 Introduction

3.1.1 Motivation

Pour détecter les zones vertes, nous avons tenté une approche différente du *machine learning*. Ces zones sont en effet de tailles et de géométries très inégales, ce qui rend difficile leur détection par un réseau de neurones. En revanche, le module `opencv` de Python est bien adapté à la détection de couleurs, et nous a paru être une bonne solution alternative. Cette méthode de détection s'applique à toute zone caractérisée par une couleur prédominante, et est donc parfaitement adaptée au cas des espaces verts.

3.1.2 Objectif

Nous souhaitons délimiter nettement les zones vertes de Bordeaux (forêts, parcs, jardins publics ou particuliers). Cela permet notamment d'étudier leur répartition, de mesurer leur surface en vue de proposer de nouveaux modèles urbains.

La délimitation passe par la construction de contours autour de ces zones vertes sur la carte, et par l'atténuation des autres zones.

On ne souhaite pas détecter la présence d'arbres individuels, cette tâche étant déjà réalisée par *machine learning*.

3.2 Méthode utilisée

Le module `opencv` fournit un grand nombre de fonctions pour le traitement d'images : dessin, transformations géométriques, gradients, segmentation, etc.

Pour délimiter les zones vertes, nous avons principalement utilisé la fonction `findContours`, qui nécessite d'avoir extrait l'espace vert de l'image au préalable.

Pour cela, nous nous sommes basés sur une méthode de détection d'objets en 3 étapes afin de délimiter au mieux les espaces verts en enlevant les bruits :

1. extraction de teintes de couleurs ;
2. *thresholding* : élimination du bruit par seuillage d'image ;
3. *smoothing* : lissage d'image.

Dans la suite, nous détaillons le processus de traitement des images. Les codes par lesquels nous avons implémenté ce processus peuvent être trouvés en Annexe (B), page 55.

3.2.1 Extraction des zones vertes par Object Tracking

Le vert de l'image est isolé. La méthode la plus habile consiste à convertir l'image du code BGR vers le code HSV [19]. Après avoir choisi la teinte verte de référence que nous voulons sélectionner, il n'y a qu'à faire varier le premier paramètre `teinte` (H) pour extraire une gamme de couleurs.

Premier problème Les teintes de vert caractérisant les espaces verts peuvent varier d'une image à l'autre, comme on peut le voir sur les images suivantes :

Figure 3.1 – Vues aériennes d'espaces verts

En prenant $(R, G, B) = (25, 50, 0)$ comme valeur de vert (vert « pelouse »), le programme reconnaît mieux les grands terrains verts que les parcs avec des arbres :

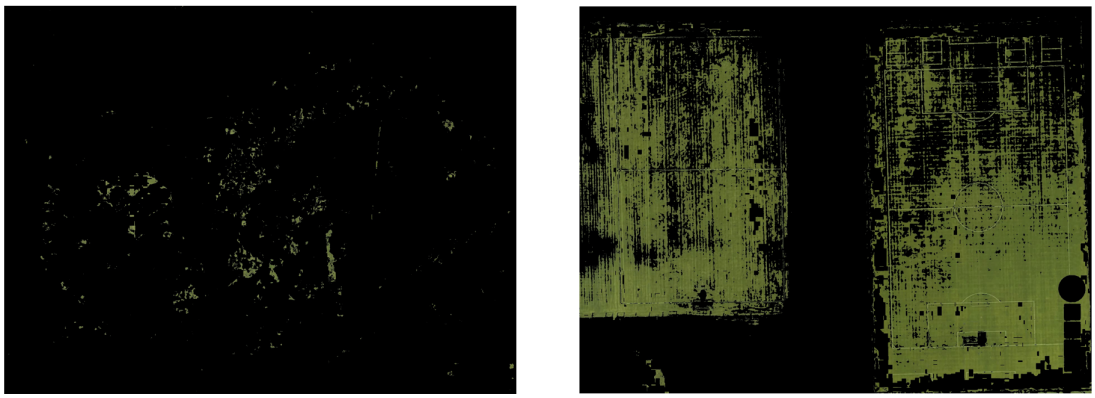


Figure 3.2 – Variation des teintes par terrain

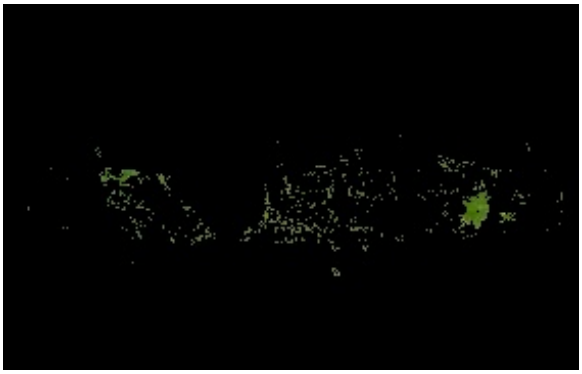
19. Hue Saturation Value, ou Teinte Saturation Lumière.

Second problème Certains espaces verts sont caractérisés par plusieurs teintes de vert à la fois, notamment en raison des ombres qui donnent des verts très sombres. Le programme précédent ne reconnaît alors que des zones partielles :

Figure 3.3 – Zone verte initiale

(a) $(0, 255, 0)$ avec $\Delta H = 20$

(b) $(102, 102, 0)$ avec $\Delta H = 20$



(c) $(102, 204, 0)$ avec $\Delta H = 20$

(d) $(250, 200, 100)$ avec $\Delta H = 20$

Figure 3.4 – Reconnaissance en fonction de (R, G, B) et ΔH

Le fait d'agrandir l'intervalle de teintes à reconnaître (ΔH) – et donc de diminuer la précision – n'est pas une solution, puisqu'alors certaines zones beiges et turquoises commençaient à apparaître :

Figure 3.5 – Reconnaissance de $[0, 150, 150]$ avec $\Delta H = 200$

Première idée : méthode par superposition de verts Nous avons donc dans un premier temps analysé nos images pour trouver les valeurs de vert les plus représentatives des zones que nous souhaitions encadrer [20]. Notons que, pour cela, nous avons travaillé uniquement avec les valeurs RGB de ces teintes, et que nous les avons ensuite converties en HSV afin de traiter les images.

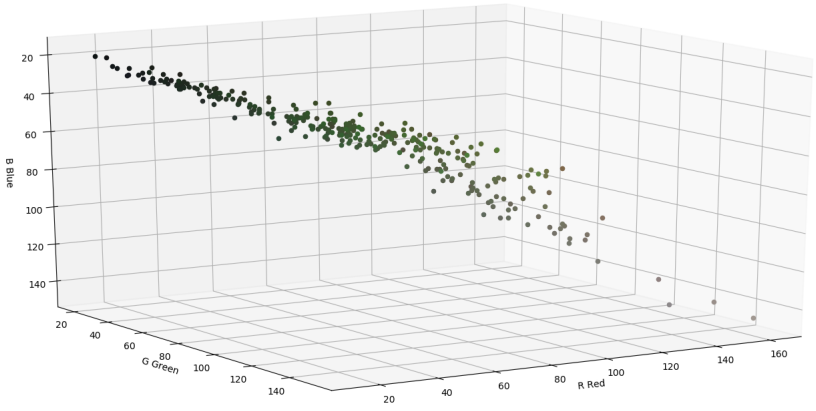


Figure 3.6 – Localisation des teintes vertes d'une photo d'espace vert

Nous avons ensuite superposé les résultats de reconnaissance pour chaque teinte de vert, avec des intervalles de teinte moindres (ΔH faible), pour améliorer la précision. Après lissage, le résultat était nettement meilleur :

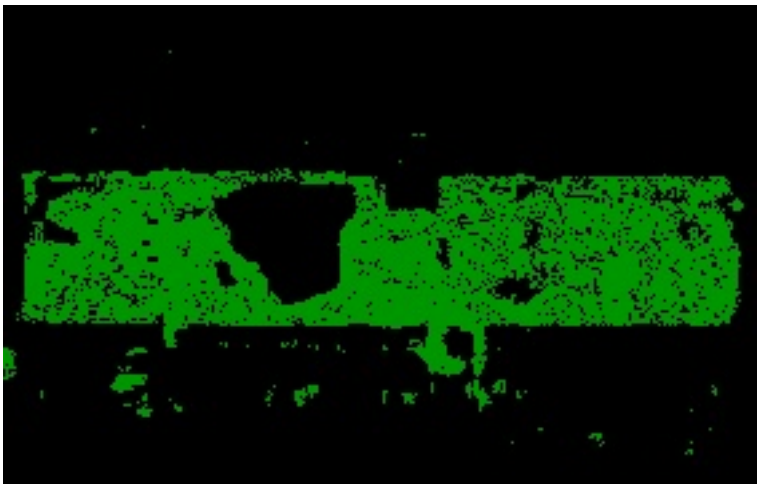


Figure 3.7 – Reconnaissance par addition d'images

20. Pour le code, voir Annexe (B.1), page 55.

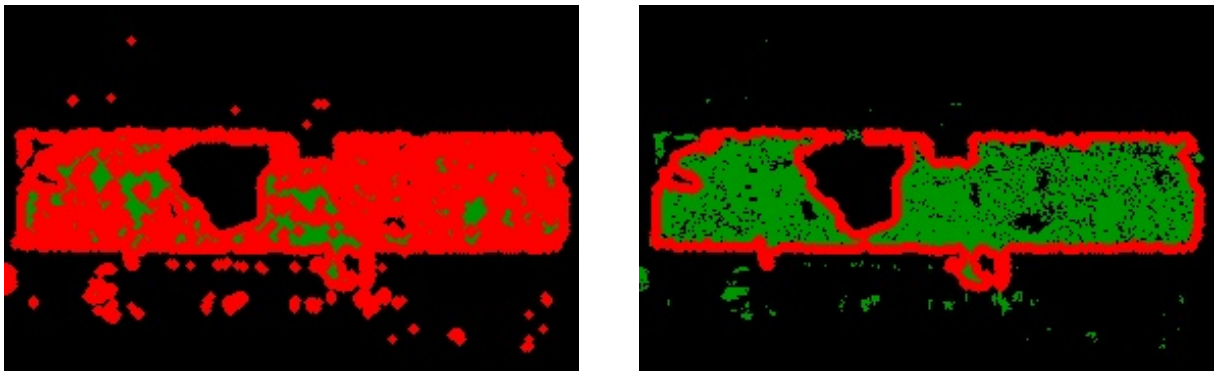
Seconde idée : méthode HSV Bien que très précise, l'approche précédente présentait un problème évident de temps d'exécution, puisqu'il fallait appeler un programme long plusieurs dizaines de fois. Notre dernière approche, moins fastidieuse et fonctionnant pour la plupart des espaces verts, a été de travailler directement avec la valeur du vert en HSV, sans passer par une conversion de BGR vers HSV. Nous avons en effet constaté que dans ce système de représentation des couleurs, la plupart des verts ont une teinte $H \in [20, 160]$. En sélectionnant cette plage de valeurs, on obtient des résultats légèrement meilleurs que précédemment, mais surtout, l'algorithme est bien plus efficace : nous n'appliquons qu'un seul filtre.

3.2.2 Tracé des contours

Idée La zone verte étant maintenant presque isolée, on peut procéder à sa délimitation. La fonction `findContours` trace automatiquement des contours autour de zones privilégiées. L'idée est alors de tracer l'enveloppe convexe du contour pour obtenir un polygone simple à manier et à représenter.

Problème Les petites taches vertes alentour sont des bruits qui gênent la délimitation du parc. Par ailleurs, le tracé de l'enveloppe convexe serait hautement perturbé par ces bruits, qui agrandiraient et déformeraient considérablement la zone. Il faudrait donc les éliminer.

Amélioration En ne sélectionnant que les contours de périmètre assez grand, on réussit à ne garder que le contour global de la zone souhaitée :



(a) Tracé naïf des contours (b) Tracé rationnel des contours

Figure 3.8 – Amélioration du tracé

Tracé Après un *thresholding* et un *smoothing*, qui, par une augmentation du contraste et un filtre, permettent d'éliminer les petites imperfections à l'intérieur de la grande zone délimitée, on peut maintenant utiliser la fonction `convexHull` d'`opencv` pour tracer l'enveloppe convexe de la zone.

Figure 3.9 – Tracé de l'enveloppe convexe

Un autre exemple de traitement d'image peut être vu en Annexe (B.2.1), page 56.

3.3 Discussions sur le modèle

3.3.1 Difficultés rencontrées

Limiter le nombre de contours renvoyés par l'algorithme Dans sa version initiale, il en détectait en effet plusieurs dizaines de milliers par image (le vert des images n'étant pas uniforme). Ce problème a principalement été réglé en modifiant la valeur du vert sélectionné, et en élargissant la fourchette de sélection. [21]

Enveloppe convexe Bien qu'utile pour détecter des espaces verts constitués de plusieurs sous-espaces verts, la méthode de tracé d'enveloppe convexe a été cependant abandonnée. Elle présentait en effet de nombreux inconvénients, notamment la déformation des espaces en zones géométriques simples, la création d'un unique espace vert à partir d'une photo satellite présentant plus de deux grands parcs (par principe des barycentres de l'enveloppe convexe).

À la place, nous avons gardé les contours, plus précis mais moins esthétiques. Cependant, pour alléger les fichiers `.geojson` renvoyés par l'algorithme et éviter une précision superflue, nous n'avons gardé qu'un point sur 20 des contours initiaux.

3.3.2 Limites et bilan

Défauts Ces défauts sont dus à la méthode choisie qui repose uniquement sur la détection d'une couleur.

- seulement certains champs sont détectés par l'algorithme ;
- imperfections : bâtiments détectés, parcelles vertes ignorées ou sectionnées en plusieurs zones…
- les surfaces d'eau « naturelles » (rivières, étangs) sont souvent détectées par la méthode HSV car elles sont également de couleur verte ;
- il n'est pas envisageable de généraliser l'algorithme à la détection de bâtiments ou de surfaces d'eau car leur couleur est moins homogène que le vert des arbres.

Bilan La précision des contours renvoyés est meilleure que prévue, mais la fiabilité de l'algorithme est à certains endroits décevante.

L'algorithme permet toutefois d'estimer la surface de verdure sur une parcelle, et donne ainsi de bons indicateurs pour une étude urbanistique de la zone.

21. *cf.* section (3.2.1), page 23.

Figure 3.10 – Résultat de la détection d'espaces verts

4 Mise en forme et présentation des résultats

Finalement, grâce aux résultats du *machine learning* et de l'analyse d'image par reconnaissance des couleurs, de nombreux fichiers `.geojson` ont été produits. Pour poursuivre la démarche d'affichage commencée avec le site `GeoJSON.io`, et pour permettre plus de personnalisation, nous avons préféré développer une interface graphique intuitive qui permette à la fois d'avoir un aperçu des données par dessus les images, et de leur donner un sens en produisant des statistiques intéressantes ou des propositions d'aménagement, le tout dans un esprit de développement urbain durable.

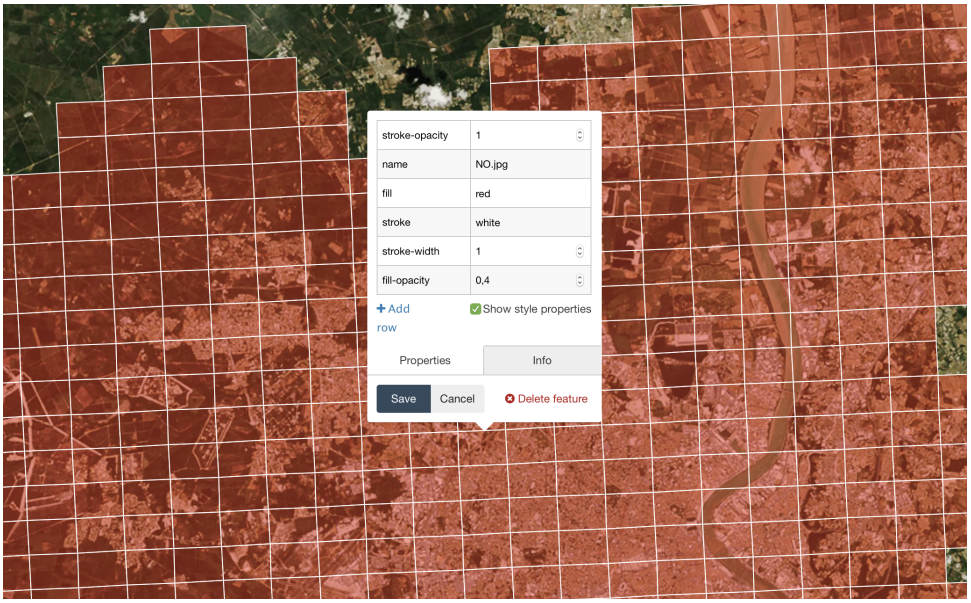


Figure 4.1 – Affichage interactif de la localisation des images via `GeoJSON.io`

4.1 L'interface

L'interface repose sur un principe simple : l'utilisateur repère la photo de Bordeaux qu'il souhaite étudier sur une carte. Il choisit ensuite cette dernière via l'interface graphique de l'application et celle-ci va ouvrir automatiquement les fichiers `.geojson` de l'image. Les différentes `properties` du fichier `.geojson` seront directement accessibles sur l'interface.

Figure 4.2 – Application : pas de filtre appliqué

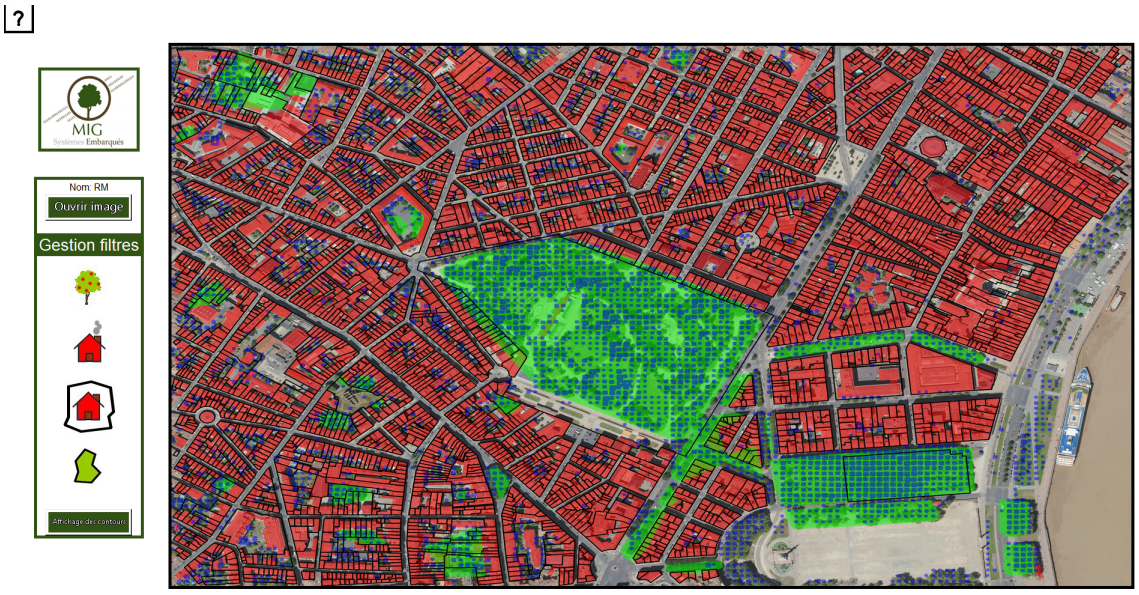


Figure 4.3 – Application avec filtres : arbres (bleu), bâtiments (rouge), cadastre (traits noirs), espaces verts (vert)

4.2 Comprendre les données et présenter des indicateurs de développement durable

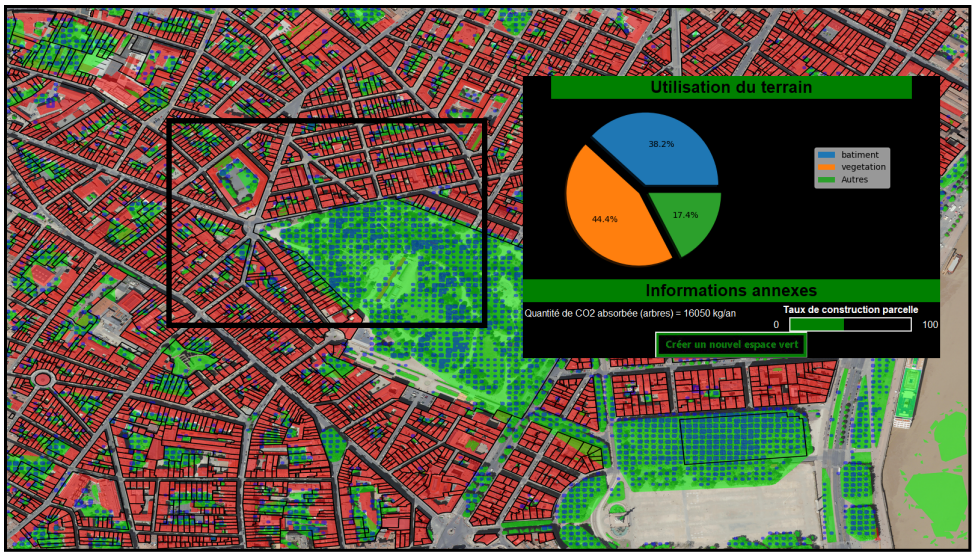


Figure 4.4 – Affichage des statistiques sur l'application

Les informations collectées sont très précieuses car elles permettent de calculer des indicateurs associés au développement durable. Par exemple, l'OMS [22] a défini des recommandations pour les villes en matière de disponibilité et d'accessibilité des espaces verts.

4.2.1 Statistiques codées puis implémentées

Les statistiques proposées sont les suivantes :

1. le nombre d'arbres, dont on déduit la quantité de CO_2 absorbé par an en moyenne. Il serait plus judicieux de déterminer la quantité de CO_2 à partir de la surface totale des arbres mais les délimitations fournies par le *machine learning* ne représentent pas réellement la surface de chaque arbre ;
2. un diagramme circulaire sur l'utilisation du terrain ;
3. le taux de construction des parcelles.

4.2.2 Statistiques codées

D'autres statistiques ont été codées sans être intégrées à l'application :

4. la distance moyenne à un espace vert dans l'image (l'OMS préconise que chaque citoyen soit à moins de 5 minutes à pied d'un espace de plus de 2 hectares) ;
5. la possibilité de trouver le point de l'image le plus isolé en terme d'espaces verts.
6. des propositions de voies vertes reliant les espaces verts entre eux [23].

22. Organisation Mondiale de la Santé, *cf.* [16].
23. *cf.* Section (4.3.2), page 35.

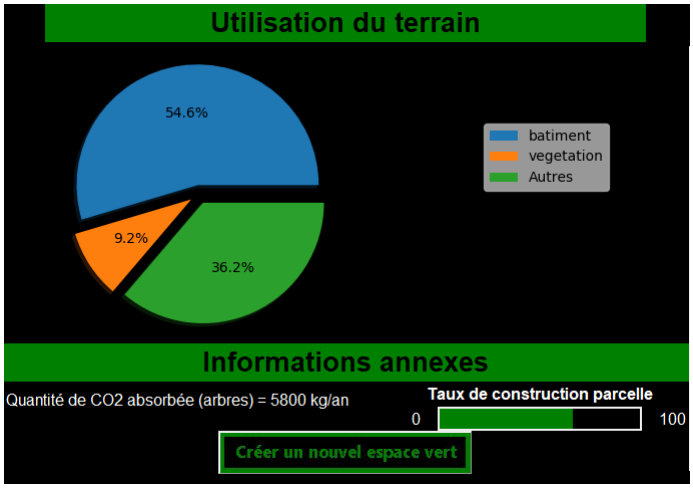


Figure 4.5 – Fenêtre des statistiques

4.3 Proposer de nouveaux espaces verts

4.3.1 Méthode barycentrique

Enfin, l'application propose de créer de nouveaux espaces verts en reliant des espaces verts existants par une méthode barycentrique. L'application indique le nombre de bâtiments présents sur le nouvel espace qu'il faudrait potentiellement détruire.

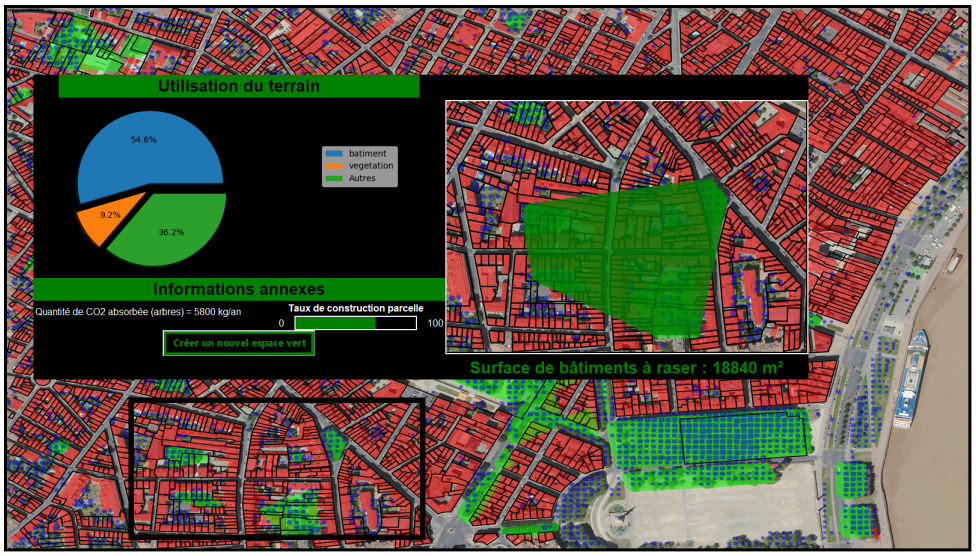


Figure 4.6 – Création d'un nouvel espace vert

Si cette méthode peut paraître éthiquement absurde et esthétiquement douteuse pour des lieux historiques comme le centre de Bordeaux (l'exemple de Central Park reste marginal [24]), elle a plus d'intérêt pour des grands projets d'urbanisme en banlieue.

24. il a été créé en détruisant un certain nombre d'habitations.

4.3.2 Trames vertes

Une autre piste plus raisonnée est de construire des trames vertes reliant les espaces verts entre eux. Ces chemins ont pu être tracés à partir des coordonnées des contours des espaces verts par un principe de plus proche voisin, et peuvent encore être optimisés en prenant en compte la géométrie des voies préexistantes. Bien qu'implémentées, ces méthodes n'ont pas encore pu être ajoutées à l'application, faute de temps.

Figure 4.7 – Création de corridors verts

4.4 Améliorer la base de données des espaces verts publics

La base de données proposée par la commune de Bordeaux pour cartographier ses espaces verts publics avait l'inconvénient de ne recouvrir que partiellement la zones réelle occupée par un espace vert [25]. Ils nous a donc fallu retravailler cette base de données pour avoir un encadrement des parcs et jardins publics plus proche de la réalité. Bien que tracer l'enveloppe convexe soit une méthode qui présente le défaut théorique de supposer le parc convexe, les résultats de son application sur l'ensemble des parcs de Bordeaux sont très satisfaisants comme le montrent les photos ci-dessous. Tous les parcs ont ainsi été remplacés par leur enveloppe convexe.

(a) Avant tracé (b) Après tracé

Figure 4.8 – Espaces verts publics avant et après le tracé des enveloppes convexes

Figure 4.9 – Résultats de l'analyse d'image : espaces verts publics et privés

25. Sûrement la commune voulait-elle mettre en valeur certaines spécificités de ses parcs.

4.5 Traitement des espaces verts trouvés par analyse d'image

4.5.1 Un premier obstacle

Les polygones fournis par l'analyse d'image représentant les espaces verts présentent parfois le défaut de se recouper eux-même (comme un sablier) et il est alors impossible de faire des statistiques dessus. Pour pallier ce problème, nous avons fait le choix de prendre l'enveloppe convexe de tous les parcs, car un polygone convexe est toujours « valide ». D'autant plus que la plupart des parcs sont convexes, ce qui permet de ne pas trop modifier la forme des polygones.

4.5.2 Un deuxième obstacle

Prendre l'enveloppe convexe des polygones amène à un autre problème dans notre cas : des polygones peuvent se superposer suite à ce traitement. En effet, la surface de l'enveloppe convexe d'un polygone étant plus grande que celle du polygone, on « gonfle » alors les espaces verts, ce qui amène des recoupements et falsifie les calculs d'aire. Par exemple, si deux polygones se superposent sur une surface S, la surface calculée par notre algorithme sur ce recoupement sera alors $2S$. La surface calculée sera toujours plus grande que la surface réelle, pouvant amener à des proportions dépassant 1.

Figure 4.10 – Problème du calcul de surfaces à cause des recoupements

Pour régler ce problème, nous avons simplement procédé à une suppression des recouvrements. Dès que deux polygones s'intersectent, on retire leur intersection à un des deux polygones. On conserve ainsi la surface totale sans avoir de superposition pour autant.

Conclusion

Malgré un cahier des charges de départ assez vaste et ouvert, avec de nombreux objets possiblement détectables, nous avons réussi à nous fixer des objectifs concrets s'inscrivant dans une démarche de développement urbain durable. Les résultats pourront toujours être améliorés, mais nous sommes très satisfaits vis-à-vis de ce que nous avons su produire en trois semaines.

Objectifs atteints Nous avons réussi à atteindre la majorité des objectifs que nous nous étions fixés :

- détecter les arbres et les délimiter avec des quadrilatères ;
- détecter les voitures [26], avec une proportion de faux-positifs raisonnable ;
- délimiter les zones de verdure ;
- réaliser une interface graphique pouvant :
  - donner un accès facile aux données déjà en notre possession et à celles que nous avons produites ;
  - nous permettre de les comprendre par l'affichage dynamique de statistiques.

Développement durable En matière d'application au développement durable et à l'urbanisme, nous pouvons désormais :

- déterminer la densité d'espace végétal sur une zone donnée ;
- connaître le taux moyen d'occupation des parcelles ;
- faire des propositions de plans pour un urbanisme plus vert en se basant sur l'ensemble des espaces verts pré-existants.

Améliorations de l'interface Notre interface peut encore être améliorée de plusieurs façons :

- en augmentant la fiabilité de détection des arbres, nous pourrions fournir des statistiques relatives à l'absorption de CO_2 beaucoup plus précises ;
- nous pourrions également ajouter une fonctionnalité permettant de proposer de nouvelles voies vertes et améliorer le mode de calcul de celle permettant de proposer de nouveaux espaces verts.

26. *cf.* Annexe (A), page 39.

Annexes

A Processus d'encadrement des voitures

Le schéma de l'algorithme plus spécifique aux voitures s'effectue comme suit :

1. création d'une base de données avec des voitures en prise de vue aérienne et conception d'un algorithme de détection de véhicules ;
2. élaboration d'un algorithme de fenêtre glissante permettant de traiter des images en format `.jpg` de grande taille, où se situent plusieurs voitures ;
3. application sur les images aériennes de Bordeaux ;
4. suppression des doublons ;
5. prédiction du polygone encadrant dans chaque sous-fenêtre.

A.1 La fenêtre glissante

A.1.1 De petites voitures sur de grandes images

Comme décrit précédemment, la classification de voitures ou non-voitures s'effectue uniquement sur des images de taille 48×48 pixels, où se situe potentiellement un unique véhicule. Or, nos données à traiter sont des images aériennes `.jpg` de la ville de Bordeaux, de taille $14\,000 \times 10\,000$ pixels et contenant pour la plupart de très nombreuses voitures.

La question s'est alors rapidement posée : peut-on utiliser un modèle de classification d'image pour faire de la détection d'objets ? Plus simplement, peut-on détecter des voitures sur une « grande » image avec un algorithme permettant de les détecter seulement sur une « petite » image ?

(a) Ce que notre modèle prend en entrée (b) Un exemple d'image à traiter

Figure A.1 – Illustration du problème rencontré

A.1.2 La fenêtre glissante

Une solution à notre problème est l'utilisation d'une fenêtre glissante de taille 48×48 pixels qui parcourt notre grande image. Ceci revient en fait à découper les grandes images en plus petites de taille 48×48, qui, elles, peuvent être analysées par notre modèle.

C'est en pratique ce qui est fait dans une boucle qui permet de déplacer la fenêtre sur l'image, comme l'illustre la figure ci-dessous. Nous avons donc choisi de faire du recouvrement : c'est-à-dire que pour avoir plus de chance de repérer un objet, qui ne sera pas toujours centré dans la sous-image, on fait glisser la fenêtre avec un pas inférieur à la taille de celle-ci. En contrepartie, cette implémentation à le défaut de repérer certains objets plusieurs fois, défaut que nous avons essayé de corriger dans la suite.

Figure A.2 – Illustration du principe de la fenêtre glissante

A.2 Application à des images aériennes de Bordeaux

Un obstacle au passage d'une fenêtre glissante sur les images aériennes `.jpg` de la ville de Bordeaux est leur taille : $14\,000 \times 10\,000$ pixels. En sachant que la taille choisie pour les voitures est de 48×48 et que le pas choisi pour la fenêtre glissante est 5, un calcul élémentaire montre que nous avons 5 552 100 sous-images à analyser.

Dans la suite, pour faciliter la présentation, nous choisissons donc la sous-image ci-dessous, de plus petite taille :

Figure A.3 – L'image de référence pour la suite de la présentation

A.2.1 Premiers résultats

Le résultat obtenu est le suivant :

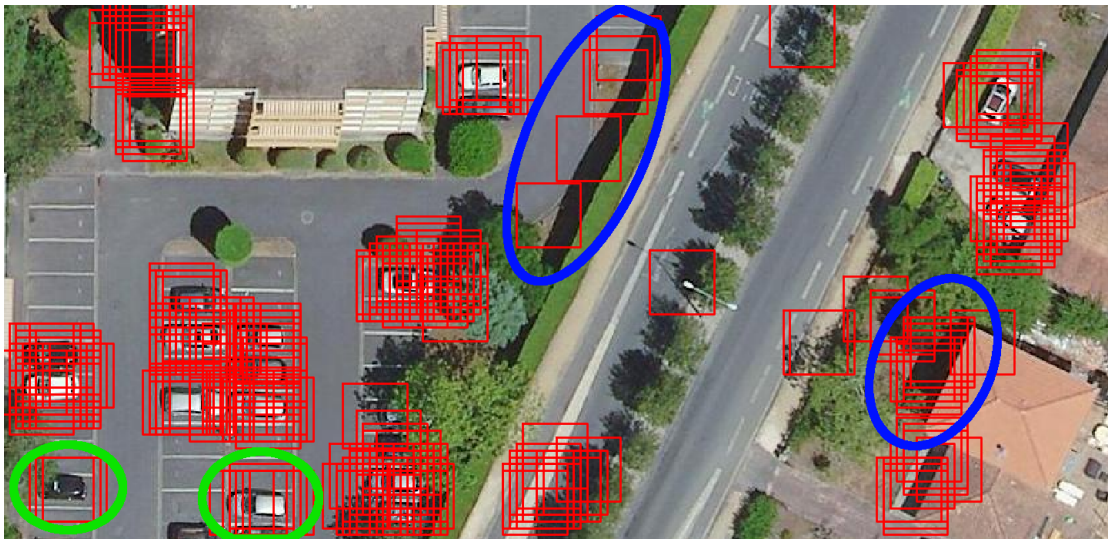


Figure A.4 – Les voitures détectées

Deux problèmes se posent :

- des faux-positifs (en bleu) ;
- plusieurs cadres détectés pour une seule voiture (en vert).

Ce cas d'application naïf de l'algorithme à une image de Bordeaux permet de mettre en évidence deux difficultés introduites par le *machine learning* :

La détection de faux-positifs Des éléments du décor (ombres, climatisation, etc.) sont faussement identifiés comme des véhicules. Cela est dû aux différences subtiles entre la base de données d'entraînement et celle sur laquelle nous utilisons le modèle pour faire des prédictions (par exemple les couleurs de l'image, le contraste, la luminosité), ainsi qu'au fait que la précision de l'identification n'est que de l'ordre de 90%.

La détection de plusieurs cadres pour un objet donné Ce défaut apparaît avec le choix de faire du recouvrement.

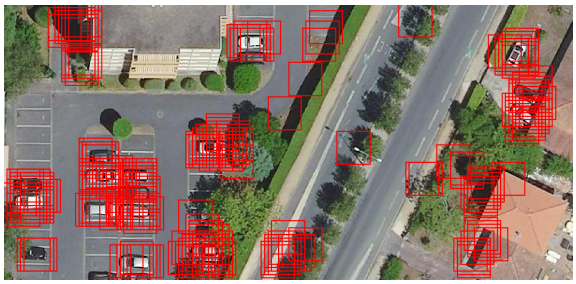
A.2.2 Perfectionnement

Pour parvenir à mieux distinguer les faux-positifs des objets à identifier, nous créons une base de données annexes contenant des faux-positifs labellisés à la main comme des non-objets, et ce à partir d'images différentes de celle de la ville de Bordeaux sur lesquelles les prédictions sont réalisées. On joint alors cette nouvelle base d'apprentissage avec celle de départ, pour aboutir à une base de données d'apprentissage finale atteignant alors environ 34 000 images de non-voitures – contenant donc les faux-positifs précédemment détectés – et un peu plus de 10 000 images de voitures.

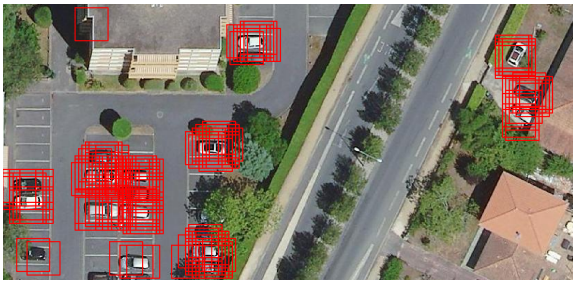
Figure A.5 – Ressemblance entre des voitures et des ombres de bâtiments

On entraîne à nouveau notre réseau neuronal avec ces nouvelles données, dans le but de réduire le nombre de faux-positifs.

Nous obtenons alors, confirmant la pertinence de cette amélioration :



(a) Avec les images de DOTA



(b) Avec la base de données enrichie

Figure A.6 – Voitures détectées avant et après l'enrichissement de la base données

On constate que les faux positifs sont bien moins nombreux : les cadres rouges ne sont présents presque qu'autour des voitures, et beaucoup moins autour des ombres ou des arbres.

A.3 Suppression des doublons

Il reste cependant un problème majeur déjà évoqué : une seule voiture donne lieu à plusieurs détections, ce qui conduit à plusieurs encadrements. Nous voudrions au contraire qu'à une voiture corresponde un unique cadre. Pour réaliser cela, nous avons implémenté [27] une méthode de post-traitement des cadres rouges nommée « *Multi-Detection Suppression* » [8].

A.3.1 Première implémentation

Une première implémentation de cette méthode consiste à utiliser une donnée de sortie du modèle de *machine learning* : la probabilité que telle photo corresponde à tel label. On peut alors calculer pour une image donnée une matrice de probabilités, telle que A_{ij} soit la probabilité que la sous-fenêtre F_{ij} contienne une voiture. Dès lors, en ne conservant que les maxima locaux de notre matrice, on parvient à identifier les zones d'intérêt.

A.3.2 Seconde implémentation

Une seconde implémentation de cette méthode, d'apparence plus naïve, consiste à parcourir tous les cadres, et à évaluer toutes les intersections sur union [28]. Si ce rapport est proche de 1, c'est que les deux cadres se superposent presque et qu'ils identifient donc le même véhicule. On en supprime alors un des deux et on continue.

Les implémentations vectorisées de cette dernière méthode la rendent quasiment instantanée, même pour un grand nombre de cadres initiaux, raison pour laquelle nous l'avons choisie. De plus, ses performances sont encourageantes, comme le représente la photo : les résultats obtenus sont maintenant assez pertinents pour procéder à des études statistiques.

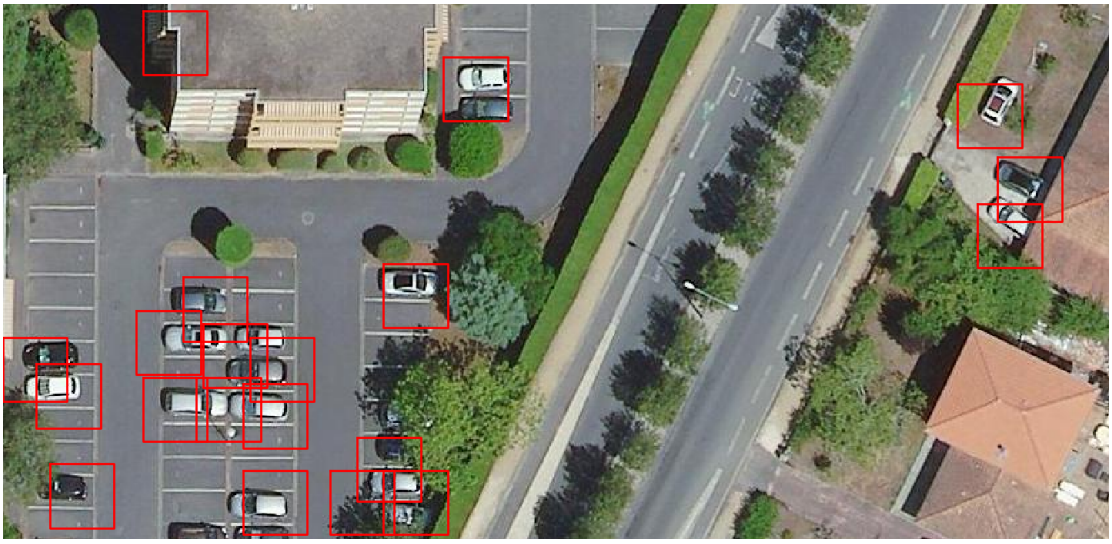


Figure A.7 – Encadrement des voitures après traitement

27. *cf.* Annexe (A.7), page 52.
28. aire de l'intersection de deux cadres divisée par l'aire de leur union.

A.4 Segmentation des voitures

Nous avons importé de la base DOTA les coordonnées des plus petits quadrilatères encadrant les véhicules : les polygones encadrants. L'association de ceux-ci et des images de véhicule associées nous a permis d'entraîner un second réseau de neurones capable, à partir d'une image inconnue de véhicule, d'extrapoler le quadrilatère associé.

Figure A.8 – Exemple de quadrilatère associé à un véhicule.

Les résultats de ce modèle sur la base de donnée de test étaient très satisfaisants. Cependant, les bases de données de test et d'entraînement (DOTA) sont assez différentes de la base d'utilisation (la ville de Bordeaux). Les prédictions sur les quartiers de Bordeaux sont donc peu précises. En effet, si les véhicules bien visibles sont précisément délimités, d'autres plus éloignés des profils à partir desquels le modèle a appris sont associés à un quadrilatère visiblement aléatoire.

Les pistes de correction suivantes ont été testées, mais fautes de temps, elles n'ont pas été utilisées.

A.4.1 Augmentation de la base de données d'entraînement

Pour chaque image, on ajoute des copies altérées en contraste, en couleurs, et en netteté.

Comme attendu, la base de test n'ayant pas été modifiée, et ne contenant pas ces altérations, la précision sur celle-ci a diminué.

Cependant, lorsqu'on étudie les résultats sur les images de la ville de Bordeaux qui peuvent contenir ces altérations, on observe qualitativement des améliorations dans l'identification de faux-positifs comme non-objets.

(a) Sans altération (b) Altération 1 (c) Altération 2 (d) Altération 3

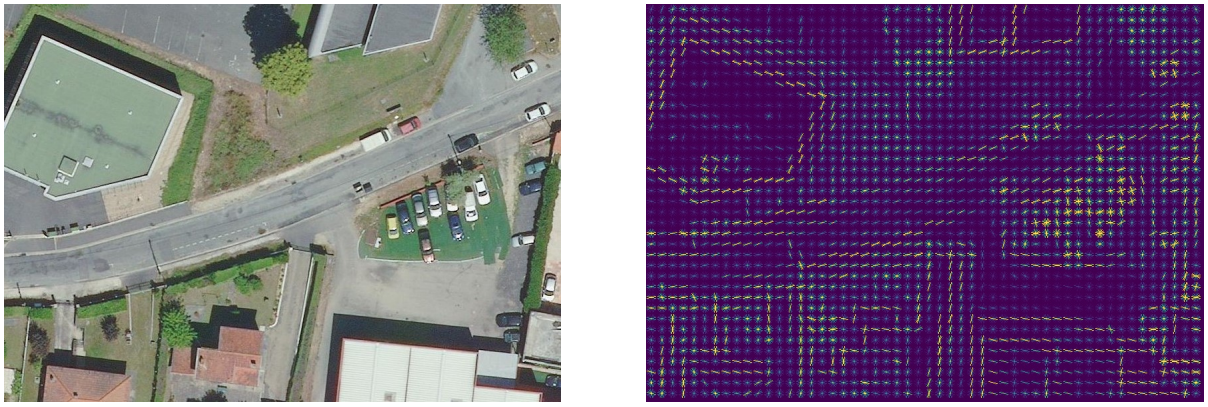
Figure A.9 – Altération des images

A.4.2 Pré-traitement des images

Principe On effectue une transformation des images de la base d'entraînement et de test de RGB vers des teintes de gris. Puis on applique une transformation pertinente dans notre cas, nommée HOG [29].

L'idée est de faire ressortir des contours d'objet sur une image en calculant une matrice des gradients de pixels : les transitions entre deux teintes au niveau d'un contours donnent des valeurs importantes et sont mis en évidence. Ces altérations sont censées permettre à l'algorithme de mieux saisir au niveau des petits groupes de pixels (7×7) l'essence d'un véhicule. On utilise ensuite des classificateurs d'images de type `Support Vector Machine` [18] pour caractériser les ensembles de sous-propriétés qui définissent bien un véhicule.

Implémentation Nous avons tenté d'adapter cette méthode [30] à notre cas d'application sans parvenir à obtenir des résultats, mais cela aurait constitué une signifiante amélioration en terme de précision pour le placement du cadre.



(a) Sans altération (b) Après application du filtre

Figure A.10 – Application du filtre HOG

29. *Histogramm of Oriented Gradient*, [8].
30. *cf.* Annexe (A.8), page 53.

A.5 Algorithme – Apprentissage

```
## Import de la base de donnée de voitures
TrainVraiNoms = os.listdir('/home/user/TrainVrai/')
TestVraiNoms = os.listdir('/home/user/TestVrai/')
TrainFauxNoms = os.listdir('/home/user/TrainFaux/')
TestFauxNoms = os.listdir('/home/user/TestFaux/')

train_data_size = len(TrainVraiNoms)+len(TrainFauxNoms)
test_data_size = len(TestVraiNoms)+len(TestFauxNoms)

train_images = [None] * train_data_size
train_labels = [[0,0]] * train_data_size
test_images = [None] * test_data_size
test_labels =[[0,0]] * test_data_size

for i,nom in enumerate(TrainVraiNoms):
    train_images[i] = cv2.imread("/home/user/TrainVrai/" + nom)
    train_labels[i] = [1,0] # 1 (true) pour voiture ; 0 (false) pour non-voiture.

for i,nom in enumerate(TestVraiNoms):
    test_images[i] = cv2.imread("/home/user/TestVrai/" + nom)
    test_labels[i] = [1,0]

for i,nom in enumerate(TrainFauxNoms):
    train_images[i + len(TrainVraiNoms)] = cv2.imread("/home/user/TrainFaux/" + nom)
    train_labels[i+len(TrainVraiNoms)] = [0,1]
    # 0 (false) pour voiture ; 1 (true) pour non-voiture.

for i,nom in enumerate(TestFauxNoms):
    test_images[i + len(TestVraiNoms)] = cv2.imread("/home/user/TestFaux/" + nom)
    test_labels[i + len(TestVraiNoms)] = [0,1]

def shuffle_in_unison(a, b):
    assert len(a) == len(b)
    shuffled_a = np.empty((len(a),48,48,3), dtype=float)
    shuffled_b = np.empty((len(b),2), dtype=float)
    permutation = np.random.permutation(len(a))
    for k in range(len(permutation)):
        shuffled_a[permutation[k]]= a[k]
        shuffled_b[permutation[k]] = b[k]
    return shuffled_a, shuffled_b

# Pour permuter nos bases de test et d'apprentissage, de façon à ne pas avoir
# au début que des images de voitures, puis à la fin que des images de non-voitures.
train_images, train_labels = shuffle_in_unison(train_images, train_labels)
test_images, test_labels = shuffle_in_unison(test_images, test_labels)
```

```
train_data_size = len(train_images)
test_data_size = len(test_images)
```

```
data_depth, data_dim_1, data_dim_2 = 3,48,48
# data_depth = profondeur. RGB ou niveau de gris ? En couleur, on a 3.
num_classes = 2 # Nombre de classes d'objets à identifier
num_epochs = 10 # Nombre de fois où on va apprendre de notre base d'apprentissage. e.
batch_size = 128
valid_data = 0.05 # Proportion qu'on va utiliser pour la validation

# Redimensionnement des images pour keras
train_images = train_images.reshape(train_data_size,data_depth,data_dim_1,data_dim_2)
train_images = train_images.astype('float32')
train_images = train_images / 255
test_images = test_images.reshape(test_data_size,data_depth,data_dim_1, data_dim_2)
test_images = test_images.astype('float32')
test_images = test_images / 255 # Entre 0 et 1

# Réseau neuronal
model = Sequential()

# Partie convolution du réseau
model.add(Convolution2D(16, (3,1), activation = 'relu', input_shape = (data_depth,data
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Convolution2D(16, (1,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

# Partie neuronale du réseau
model.add(Flatten())
model.add(Dense(128,activation='relu')) # Couche de neurones profonds
model.add(Dense(128,activation='relu')) # Couche de neurones profonds
model.add(Dense(num_classes,activation='softmax')) # Couche de sortie

model.compile(optimizer='adam',loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, batch_size = batch_size, epochs = num_epochs, ve

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(test_loss,test_acc)

## Exemple de prédiction
img = cv2.imread('/home/user/pasVoiture.jpg')
# img de bonne taille (ici 48*48)
img = img.astype('float32')
img = img / 255
img = img.reshape(3,data_dim_1,data_dim_2)

res = model.predict(np.array([img]))
print(res)
```

A.6 Algorithme – Fenêtre glissante

```
# Application de l'algorithme de fenêtre glissante à une image.
def fenetre_glissante(image, pas, taille_fenetre):
    # Taille de l'image
    taille_x = image.shape[1]
    taille_y = image.shape[0]

    # Taille des fenêtres
    fen_x = taille_fenetre[0]
    fen_y = taille_fenetre[1]
    nb_subImg = int((taille_x-49)/pas) * int((taille_y-49)/pas)
    count = 0

    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB)) # Fond d'affichage
    image = image.astype('float32')
    image = image / 255

    # On parcourt l'image à l'aide de la fenêtre glissante
    # (une boucle en y et une en x).
        for y in range(0, taille_y - fen_y - 1, pas):
            print(count/nb_subImg)
                for x in range(0, taille_x - fen_x - 1, pas):
                    count = count + 1
            subImg = img[y:y+fen_y,x:x+fen_x] # Sous-image
            subImg = subImg.reshape(3,48,48)
            resSubImg = model.predict(np.array([subImg]))

            if resSubImg[0][0] >= 0.999: # Voitures détectées à partir d'un
                                         # certain seuil
                plt.plot([x,x,x+fen_x,x+fen_x,x],[y, y+fen_y,y+fen_y,y,y],color='r',li

    plt.axis('off')
    plt.show()

# Permet de réaliser une nouvelle base de données à partir des détections.
# On trie manuellement, en appuyant sur deux touches, les voitures effectivement
# détectées et les faux-positifs.
def fenetre_glissante_correction(image, pas, taille_fenetre):
    taille_x = image.shape[1]
    taille_y = image.shape[0]
    fen_x = taille_fenetre[0]
    fen_y = taille_fenetre[1]
    extent = (0, max(fen_x,fen_y), 0, max(fen_x,fen_y))
    nb_subImg = int((taille_x-49)/pas) * int((taille_y-49)/pas)
    count = 0
    for y in range(0, taille_y - fen_y - 1, pas):
```

```
                for x in range(0, taille_x - fen_x - 1, pas):
                    count = count + 1
            subImg = img[y:y+fen_y,x:x+fen_x]
            subImg = subImg.reshape(3,48,48)
            resSubImg = model.predict(np.array([subImg]))
            if resSubImg[0][0] >= 0.9999:
                subImg = subImg.reshape(48,48,3)
                print(count/nb_subImg)

                # Tri des fenêtres détectées comme voitures
                winname = 'image'
                cv2.namedWindow(winname,cv2.WINDOW_NORMAL)
                cv2.resizeWindow(winname, 500,500)
                cv2.moveWindow(winname,100,300)
                cv2.imshow(winname,subImg)
                k = chr(cv2.waitKey())

                if k == 'v':
                    cv2.imwrite('/home/user/VoituresBonnes/' + str(y) + '-' + str(x)

                elif k == 'b':
                    cv2.imwrite('/home/user/FauxPositifs/' + str(y) + '-' + str(x) +

# Application de l'algorithme de fenêtre glissante à une image, avec partitionnement
# final des boîtes.
def fenetre_glissante_amelioree(image, pas, taille_fenetre):
    taille_x = image.shape[1]
    taille_y = image.shape[0]
    fen_x = taille_fenetre[0]
    fen_y = taille_fenetre[1]
    nb_subImg = int((taille_x-49)/pas) * int((taille_y-49)/pas)
    count = 0
    plt.figure()
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

    image = image.astype('float32')
    image = image / 255
    boxes = []
        for y in range(0, taille_y - fen_y - 1, pas):
            print(count/nb_subImg)
                for x in range(0, taille_x - fen_x - 1, pas):
                    count = count + 1
            subImg = img[y:y+fen_y,x:x+fen_x]
            subImg = subImg.reshape(3,48,48)
            resSubImg = model.predict(np.array([subImg]))
            if resSubImg[0][0] >= 0.999:
```

```
                # On sauvegarde les coordonnées des boîtes détectées.
                boxes.append([x,y,x+fen_x,y+fen_y])

                plt.plot([x,x,x+fen_x,x+fen_x,x],[y, y+fen_y,y+fen_y,y,y],color='r',li

    boxesbis = np.array(boxes)
    # On effectue un partitionnement (clustering) des boîtes, via un algorithme exter
    # de non maxima suppression.
    pick = non_max_suppression_fast(boxesbis, 0.3)

    # Affichage
    plt.figure()
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    for (startX, startY, endX, endY) in pick:
        plt.plot([startX,startX,startX+fen_x,startX+fen_x,startX],[startY, startY+fen_

    plt.axis('off')
    plt.show()
```

A.7 Algorithme – Suppression des doublons

Ce programme permet de supprimer les doublons efficacement. On entend par doublon un couple de cadres qui identifient le même objet sur une image. On introduit un pourcentage de recouvrement autorisé entre deux cadres pour éviter d'avoir plusieurs cadres pour le même objet détecté. Un équilibre est à trouver, dans le cas où on aurait par exemple deux objets différents très proches — le pourcentage de recouvrement autorisé ne peut alors pas être nul, car les deux cadres pour les deux objets seraient fusionnés en un seul.

```
def multiDetectionSuppresion(boxes,recouvrement=0.3):
    '''On applique cette fonction à des cadres carrés. EN pratique, on
    ↪  applique l'algorithme de détection qui identifie les
    ↪  sous-fenêtres carrées qui contiennent un objet, puis on applique
    ↪  multiDetectionSuppression.'''
    if len(boxes)==0:
        return []
    index_retenus = []
    x1 = boxes[:,0]
    y1 = boxes[:,1]
    x2 = boxes[:,4]
    y2 = boxes[:,5]
    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    #print(x1[0],x2[0])
    idxs = np.argsort(y2)
    while len(idxs) > 0:
        last = len(idxs) - 1
        i = idxs[last]
        index_retenus.append(i)

        xx1 = np.maximum(x1[i], x1[idxs[:last]])
        yy1 = np.maximum(y1[i], y1[idxs[:last]])
        xx2 = np.minimum(x2[i], x2[idxs[:last]])
        yy2 = np.minimum(y2[i], y2[idxs[:last]])
        # largeur et hauteur de la Bounding Boxes
        w = np.maximum(0, xx2 - xx1 + 1)
        h = np.maximum(0, yy2 - yy1 + 1)

        # calcule l'intersection sur union
        overlap = (w * h) / area[idxs[:last]]
        # supprime les indexes tel que overlap>seuil
        idxs = np.delete(idxs, np.concatenate(([last],
            np.where(overlap > recouvrement)[0])))
    return boxes[index_retenus].astype("int")
```

A.8 Algorithme – Histogramme des gradients orientés et classification

Ce programme permet :

- d'importer la base de donnée en ayant déja appliqué la transformation HOG de `skimage.feature`.
- d'entraîner un modèle des K-neighbors (autre type d'apprentissage supervisé implémenté par le module `sklearn`). Celui-ci va classifier les *features* qui ont été créées pour décrire efficacement une image, en deux catégories, les images contenant l'objet et celles ne le contenant pas ;
- de faire des prédiction sur des images inconnues et donc de tester la précision du modèle sur la base de test.

```
testV = os.listdir('voiture bien decoupe/testV')
trainV = os.listdir('voiture bien decoupe/trainV')
test = os.listdir('voiture bien decoupe/test')
train = os.listdir('voiture bien decoupe/train')

# MODELE DES K-NEIGHBORS

trainHOG = []
hog_features = []
labels = []

# On applique la transformation HOG à chaque image de la base de donnees
↪  d'entrainement puis de test

for nom in train:
    image = mpimg.imread("voiture bien decoupe/Train/" + nom)
    imageGrey = color.rgb2gray(image)
    #print(imageGrey)
    fd, hog_image = hog(imageGrey, orientations=8, pixels_per_cell=(16,
    ↪  16),cells_per_block=(3,3), visualize=True, block_norm= 'L1')
    labels.append([0.,1.])
    hog_features.append(fd)
    trainHOG.append(hog_image)

print("1")

for nom in trainV:
    image = mpimg.imread("voiture bien decoupe/TrainV/" + nom)
    fd, hog_image = hog(color.rgb2gray(image), orientations=8,
    ↪  pixels_per_cell=(16, 16),cells_per_block=(3,3), visualize=True,
    ↪  block_norm= 'L1')
    labels.append([1.,0.])
    hog_features.append(fd)
```

```
    trainHOG.append(hog_image)
print("2")

# On met en forme les données
labels = np.array(labels)
hog_features = np.array(hog_features)
data_frame = np.hstack((hog_features,labels))
np.random.shuffle(data_frame)

# On implémente un modèle de SUpport vector Machine/ KNeighbors
model = KNeighborsClassifier(n_neighbors=1)
model.fit(data_frame, labels)
print("[INFO] evaluating...")

# AFFICHAGE DES IMAGES APRES TRANSFORMATION

for (i, imagePath) in enumerate(paths.list_images(args["test"])):
    # imoport en teintes de gris
    image = cv2.imread(imagePath)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # extraction du HOG
    (H, hogImage) = feature.hog(gray, orientations=9,
    ↪  pixels_per_cell=(10, 10),
        cells_per_block=(2, 2), transform_sqrt=True, block_norm="L1",
        ↪  visualise=True)
    pred = model.predict(H.reshape(1, -1))[0]

    # affichage
    hogImage = exposure.rescale_intensity(hogImage, out_range=(0, 255))
    hogImage = hogImage.astype("uint8")
    cv2.imshow("HOG Image #{}".format(i + 1), hogImage)
    cv2.putText(image, pred.title(), (10, 35), cv2.FONT_HERSHEY_SIMPLEX,
    ↪  1.0,
    (0, 255, 0), 3)
    cv2.imshow("Test Image #{}".format(i + 1), image)
    cv2.waitKey(0)
```

B Analyse d'image

B.1 Détermination des teintes vertes

Ce programme affiche sur un graphe 3D une partie (choisie aléatoirement) des teintes trouvée sur une image (ici une des images de Bordeaux).

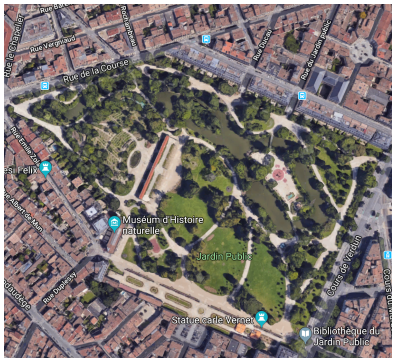
Le graphique a sur chacun de ses axes une des composantes RBG. Cela permet alors de donner une idée des teintes rencontrées sur l'image en question et de mieux cerner la plage de couleur qui nous intéresse.

```
import sys
sys.path.append("/home/felix/Documents/INFORMATIQUE/Python_Mines/Modules_perso")

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import cv2
from utile_plt import invRB
from random import randrange

def convCouleur(a,b,c):
    return (a/255, b/255, c/255)

# Image d'origine
img = cv2.imread("foret1.png",1)
invRB(img)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# On selectionne et on affiche un certain nombre de pixels pris au hasard
↪  sur l'image
for i in range(img.shape[0]//2):
    j=randrange(0,img.shape[1])
    coul = convCouleur(img[2*i,j,0], img[2*i,j,1], img[2*i,j,2])
    ax.scatter(img[2*i,j,0], img[2*i,j,1], img[2*i,j,2], c=coul,
    ↪  marker='o')

ax.set_xlabel('R Red')
ax.set_ylabel('G Green')
ax.set_zlabel('B Blue')

plt.show()
```

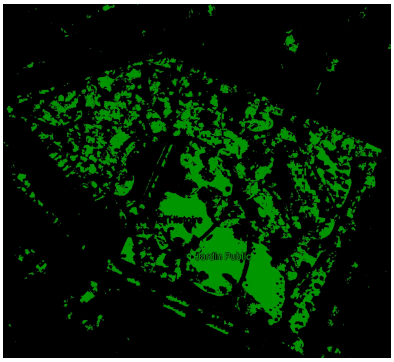
B.2 Délimitation des zones vertes

B.2.1 Un autre exemple

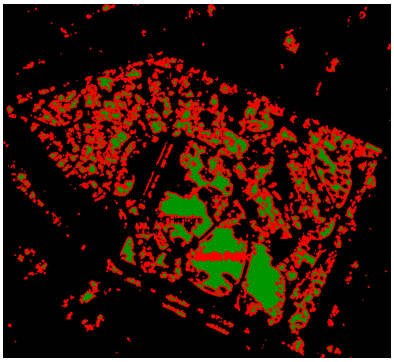
Voici un autre exemple de délimitation des zones vertes qu'en Section (3.2), page 22. Celui-ci concerne un parc de Bordeaux. La différence est que plusieurs grandes zones ont été délimitées par le programme, et qu'il a donc fallu les unifier pour n'en faire qu'une seule zone de plus grande taille.



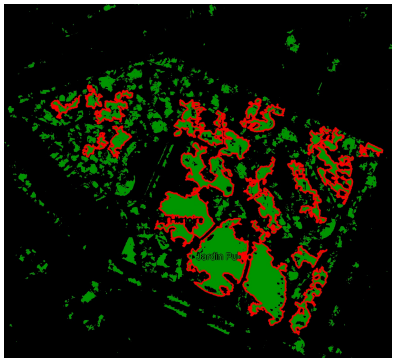
(a) Vue satellitaire d'un parc



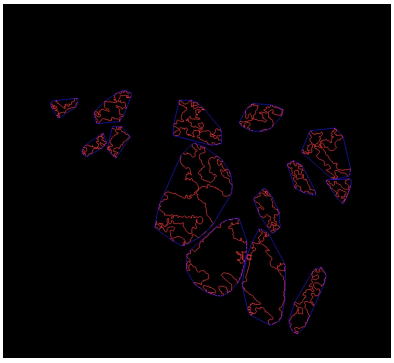
(b) Extraction du vert



(c) Tracé naïf des contours



(d) Tracé rationnel des contours



(e) Enveloppes convexes multiples

(f) Enveloppe convexe simple

Figure B.1 – Processus de traitement d'image – Bordeaux

B.2.2 Algorithme de délimitation

Algorithme qui permet d'isoler les zones vertes et d'en tracer les contours (sans utiliser d'enveloppe convexe ici). Ces contours sont ensuite transformés en polygones geojson afin d'être exploités par l'interface graphique.

```
import cv2
import numpy as np
import matplotlib
import os
matplotlib.rcParams['backend'] = "TkAgg"
from matplotlib import pyplot as plt
import pandas as pd
import geopandas as gpd
from shapely.geometry import Polygon
import rasterio as rst

#On noircit tous les points de l'image qui ne sont pas verts
def colorspace(im):
    lower = np.array([20,50,50])
    upper = np.array([160,255,255])
    hsv = cv2.cvtColor(im, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, lower, upper)
    res = cv2.bitwise_and(im,im,mask = mask)
    return res

 #On transforme les points verts en points blancs + lissage de l'image
  ↪  (meilleure precision pour les contours)
def bon_format_contours(im):
    res = colorspace(im)
    res_median = cv2.medianBlur(res,7)
    res_gray = cv2.cvtColor(res_median,cv2.COLOR_BGR2GRAY)
    ret,thresh = cv2.threshold(res_gray,50,255,cv2.THRESH_BINARY)
    dil   = cv2.dilate(thresh, None, 10)
    thresh = cv2.erode(dil, None, 10)
    return thresh

#On néglige les zones de verdures en dessous de cette aire minimale(10
 ↪  m^2)
min_area = 1000

#Contournement des zones de verdure
def contours(im):
    tr = bon_format_contours(im)
    image, cnts, hierarchy =
     ↪  cv2.findContours(tr,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
    cnts_filtres = []
```

```
    for cnt in cnts:
        if cv2.contourArea(cnt) > min_area:
        cnt_simple = []
            for i in range(0,len(cnt),min(20,len(cnt)//3)):#On ne garde
            ↪ qu'un point sur 20, mais il faut au moins 3 points pour
            ↪ faire un polygone
                cnt_simple.append(cnt[i])
            cnts_filtres.append(np.array(cnt))
    return cnts_filtres

#Transformation en geojson:
def transfo(cnts,bounds):
    l = []
    xo,_,_,yo = bounds
    for c in cnts:
        coord = c.reshape((len(c),2))
        for i in range(len(coord)):
            x,y = coord[i]
            coord[i] = xo + x*0.1, yo-y*0.1
        l.append(Polygon(coord))
    cnts = np.array(l)
    df = {'geom':pd.Series(cnts)}
    gdf = gpd.GeoDataFrame(df, geometry = 'geom')
    return (gdf)

#Fonction principale (à lancer sur l'ensemble des images)
path = '/Users/Alexiane/Documents/MIG_SE/Photos_Bordeaux/'
path_res = '/Users/Alexiane/Documents/MIG_SE/Resultats2/'
dossier = os.listdir(path)
dossier_jpg = [f for f in dossier if f[-4:] == ".jpg"]


def main():
    for nom in dossier_jpg:
        print(nom)
        im = cv2.imread(path+nom)
        r = rst.open(path+nom)
        bounds = r.bounds
        r.close()
        cnts = contours(im)
        gdf = transfo(cnts,bounds)
        surface = pd.DataFrame(gdf.area,columns = ['area'])
        gdf['area'] = surface
        gdf.to_file(path_res+'vegetation_prive'+nom[:2]+'.geojson',
        ↪ driver = 'GeoJSON')
```

B.3 Tracé des contours et de l'enveloppe convexe

L'algorithme de tracé des contours et de l'enveloppe convexe par méthode de superposition prend la forme suivante :

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def invRB(img):
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            img[i,j,0], img[i,j,2] = img[i,j,2], img[i,j,0]


def zotero_gray(n,p):
    """crée un ndarray (format de prédilection des images en OpenCV) de
    ↪  taille n,p"""
    return np.uint8([[0 for j in range(p)]for i in range(n)])

def tracking(frame,R,G,B):
    """Détecte les pixels de l'image qui ont une teinte de vert
       "très proche" de la teinte RGB entrée, et renvoit une image
       en noir et blanc où le blanc correspond aux zones détectées """
    color=np.uint8([[[B,G,R]]])
    frame=cv2.imread(frame,1)
    hsv_color=cv2.cvtColor(color,cv2.COLOR_BGR2HSV)
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower = np.array([hsv_color[0,0,0]-10,50,50])
    upper = np.array([hsv_color[0,0,0]+10,255,255])
    mask = cv2.inRange(hsv, lower, upper)
    return mask

def enveloppe_cvx(frame,ech):
    """
       Prend en entrée une image, détecte les espaces verts par la
↪   méthode de superposition des verts,
       trace l'enveloppe convexe de l'ensemble de ces espaces verts, qui
↪   constitue alors une nouvelle zone verte,
       et calcule l'ancienne et la nouvelle surface verte sur l'image.
    """

    #détection des espaces verts par superposition de teintes

    #teintes les plus répandues en RGB

    ↪  teintes_vert=[[51,51,0],[25,51,0],[0,51,0],[0,51,25],[100,150,100],[102,102,0
```

```
im_frame=cv2.imread(frame,1)
mask=zotero_gray(im_frame.shape[0],im_frame.shape[1])
for i in range(len(teintes_vert)):

    ↪  mask+=tracking(frame,teintes_vert[i][0],teintes_vert[i][1],teintes_vert[i]

#prétraitement
blur=cv2.blur(mask, (3, 3))
ret, thresh = cv2.threshold(blur, 50, 255, cv2.THRESH_BINARY)

#détermination des contours de taille assez importante
im2, contours, hierarchy =
↪  cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
L=[]
Aire_det=0
for i in range(len(contours)):
    if len(contours[i])>200:
            L.append(contours[i])
            Aire_det+=cv2.contourArea(contours[i])
contours=L

#conversion de la liste de contours en liste de points pour pouvoir
↪  tracer l'enveloppe convexe
points=[]
for k in range(len(contours)):
    for i in range(len(contours[k])):
        points.append(contours[k][i])
new_contours=np.int32(points)
hull=[]
hull.append(cv2.convexHull(new_contours, True))
Aire_cvx=cv2.contourArea(hull[0])

#tracé de l'enveloppe convexe
color_contours = (0, 0, 255)
color = (255, 0, 0)
for i in range(len(contours)):
    img1=cv2.drawContours(im_frame, contours, i, color_contours, 2,
    ↪  10)
img2=cv2.drawContours(im_frame, hull, 0, color, 3, 15)
cv2.imwrite('NYcvx.jpg',img2)
return Aire_det*ech,Aire_cvx*ech
```

B.4 Tracé des corridors verts

L'algorithme de tracé des corridors verts prend quant à lui la forme suivante :

```
def dist(a,b):
    """donne la distance euclidienne entre deux points
    """
    return np.sqrt(((a[0]-b[0])**2)+((a[1]-b[1])**2))

def relier_parc_à(liste_contours,indice):
    """relie un parc (ie un élément de la liste des contours détectés par
    ↪ l'algorithme précédent) au parc le plus proche
    """
    d_min=9223372036854775807
    point_depart=[0,0]
    point_arrivee=[0,0]
    for i in range(len(liste_contours)):
        if i!=indice:
            for q in range(len(liste_contours[indice])):
                for p in range(len(liste_contours[i])):
                    if
                    ↪ dist(liste_contours[indice][q][0],liste_contours[i][p][0])<d_m

                        ↪ d_min=dist(liste_contours[indice][q][0],liste_contours[i]
                        point_depart=liste_contours[indice][q][0]
                        point_arrivee=liste_contours[i][p][0]
    return point_depart,point_arrivee


def corridors_verts(frame):
    """ BILAN: algorithme qui prend une image en entrée, détecte tous les
    ↪ espaces verts et les relie entre eux
            selon le principe de plus proche voisin précédent.
    """
    #détection des espaces verts

    ↪ teintes_vert=[[51,51,0],[25,51,0],[0,51,0],[0,51,25],[100,150,100],[102,102,0
    im_frame=cv2.imread(frame,1)
    mask=zotero_gray(im_frame.shape[0],im_frame.shape[1])
    for i in range(len(teintes_vert)):

        ↪ mask+=tracking(frame,teintes_vert[i][0],teintes_vert[i][1],teintes_vert[i

    #prétraitement de l'image
    blur=cv2.blur(mask, (3, 3))
    ret, thresh = cv2.threshold(blur, 50, 255, cv2.THRESH_BINARY)
```

```
#détermination des contours de taille assez importante
im2, contours, hierarchy =
↪  cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
L=[]
for i in range(len(contours)):
    if len(contours[i])>200:
            L.append(contours[i])
contours=L

#tracé des contours
color_contours = (0, 0, 255)
for i in range(len(contours)):
    img1=cv2.drawContours(im_frame, contours, i, color_contours, 2,
    ↪  8)

#tracé des corridors verts
for i in range(len(contours)):
    point_depart,point_arrivee=relier_parc_à(contours,i)
    point_depart=tuple(point_depart)
    point_arrivee=tuple(point_arrivee)
    img1=cv2.line(img1,point_depart,point_arrivee,(255,0,0),20)


cv2.imwrite("corridors_"+frame,img1)
```

C Les différents types de fichiers

C.1 Fichiers `.geojson`

C.1.1 Écriture

Un fichier `.geojson` se présente sous cette forme :

```
"type": "FeatureCollection",
   "features": [{
      "type": "Feature",
      "properties": {"fill":"green","stroke-width":"1",
"fill-opacity":0.6},
      "geometry": {
         "type": "Polygon",
            "coordinates": [
            [[-76.9702148,40.1788733143],
             [-74.0258789,39.8422860207],
             [-73.4326174,41.7139300733],
             [-76.7944335,41.9431487473],
             [-76.9702148,40.1788733143]]]}}]
```

Les propriétés du fichier `.geojson` regroupent les paramètres d'affichage des différentes formes : opacité, couleur ou taille de contour.

Il est également possible d'y ajouter des informations non spatiales : pour les arbres, on peut par exemple référencer leur espèce ou leur date de plantation.

C.1.2 Affichage

Le site `GeoJSON.io` [2] permet cette superposition. Il est possible de visualiser un fond de carte du monde en images satellites et d'y ouvrir un fichier `.geojson`.

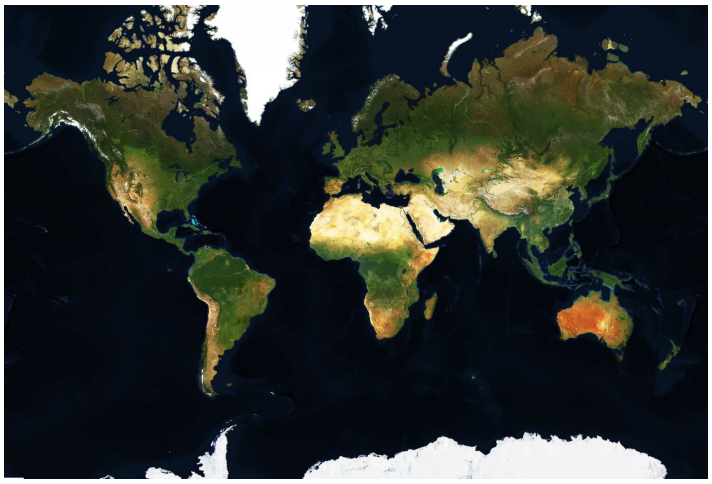


Figure C.1 – Fond de carte de `GeoJSON.io`

Les différentes formes géolocalisées seront donc directement associées à une image.

L'exemple qui suit montre la superposition d'une carte et des carrés repérés au-dessus des arbres, issus d'un fichier `.geojson`. Le site `GeoJSON.io` est donc un intermédiaire précieux pour visualiser efficacement les données manipulées.

Figure C.2 – Superposition d'un fond de carte satellite avec une base `.geojson`d'abres

Il est également possible d'accéder aux différentes propriétés non spatiales sur l'interface de `GeoJSON.io`.

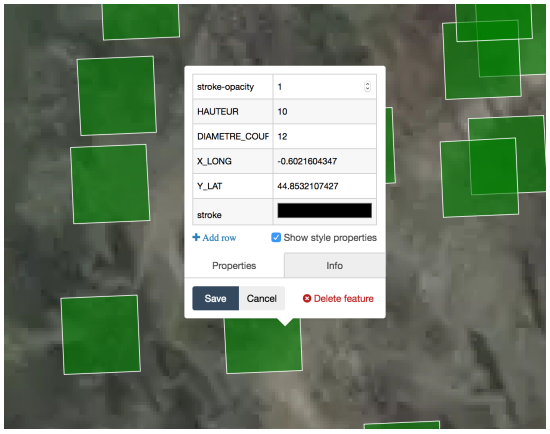


Figure C.3 – Affichages des propriétés des arbres

C.1.3 Perspectives

Outre son utilité dans l'interface graphique comme expliqué plus haut [31], cette interface permet de vérifier les modifications et donc de tester le fichier `.geojson` et sa localisation ou bien la taille des formes contenues. Cela permet par exemple de remarquer si un polygone n'est pas trop grand par rapport à l'entité qu'il souhaite repérer ou déboguer nos programmes.

31. *cf.* Section (4), page 31.

C.2 Images localisées

Lors de notre travail, nous avons utilisé deux formats d'images localisées :

1. une association d'une image (`.jpg`) et de sa localisation (`.jgw`[32]) pour les images aériennes localisées de Bordeaux ;
2. un tableau Python avec l'image et sa localisation stockés dans un fichier binaire (`.bin`).

C.2.1 Premier format

Le format Le premier format est assez classique et est automatiquement lu par le module `Rasterio` de Python. Le fichier `.jgw`, qui a le même nom que l'image, contient six lignes :

1. l'échelle d'un pixel en abscisse (0.1 dans notre cas)
2. la rotation de l'axe des ordonnées (0 dans notre cas)
3. la rotation de l'axe des abscisses (0 dans notre cas)
4. l'échelle d'un pixel en ordonnée (-0.1 dans notre cas)
5. la position en abscisse du pixel en haut à gauche de l'image
6. la position en ordonnée du pixel en haut à gauche de l'image

Limites Ce système présente le défaut de ne pas préciser la projection utilisée. Dans notre cas, il s'agissait du format `epsg 3945` et les distances étaient par conséquent en mètres.

C.2.2 Second format

Le format Nous avons déjà évoqué ce second format à plusieurs reprises à la section (2). Il s'agit d'un couple ayant comme premier élément l'image dans une matrice RGB, et comme second un tableau à deux éléments avec la position du pixel en haut à gauche en `epsg3945`.

Limites Ce format est encore moins précis que le précédant puisque les rotations d'axes, la projection, les unités, et les dimensions d'un pixel sont omises. Cependant, comme nous voulions créer des centaines de milliers d'images avec à chaque fois les mêmes informations, cela aurait été un gaspillage d'espace mémoire et nous avons préféré optimiser le stockage. Ce format est donc moins universel que le précédant mais plus avantageux pour notre cadre d'étude précis.

32. `JPG` World file.

D Pré-traîtement des données

D.1 Mise en forme .csv et conversion .json

```
import geopandas as gpd
import pandas as pd

path = "C:\\Users\\Tanguy\\arbres.csv"
path2 = "C:\\Users\\Tanguy\\arbres2.csv"

fichier1 = open(path, "r")
fichier2 = open(path2, "w")

while True:
    c = fichier1.read(1)
    if not c:
        break
    elif c==';':
        fichier2.write(',')
    elif c==',':
        fichier2.write('.')
    else:
        fichier2.write(c)

fichier1.close()
fichier2.close()

df =
↪  pd.read_csv("C:\\Users\\Tanguy\\Documents\\MIG\\arbres2.csv",low_memory=False)

df.to_json("C:\\Users\\Tanguy\\Documents\\MIG\\arbres.json")
```

D.2 Conversion de .json vers .geojson

```
from shapely.geometry import Point, Polygon
import pandas as pd
import geopandas as gpd

#On ouvre le fichier json
s = pd.read_json("C:\\Users\\Tanguy\\Documents\\MIG\\arbres2.json")


#On selectionne seulement les colonnes que l'on veut conserver
df = s.loc[lambda x: x.STATUT_ARBRE == "Vivant", ['HAUTEUR',
↪  'DIAMETRE_COURONNE', 'X_LONG', 'Y_LAT']]
df=df[df['DIAMETRE_COURONNE'].notnull()]

#On créer les colonnes contenant les coordonnées des bords de l'arbre et
↪  le polygone de découpe de l'image
df['box_arbres'] = Polygon([(0,0),(0,1),(1,1),(1,0)])
df['boxArbreGauche'] = 0
df['boxArbreBas'] = 0
df['boxArbreDroite'] = 0
df['boxArbreHaut'] = 0
df['Coordinates'] = list(zip(df.X_LONG, df.Y_LAT))
df['Coordinates'] = df['Coordinates'].apply(Point)

#On transforme la DataFrame en GeoDataFrame et on précise le système de
↪  projection
gdf = gpd.GeoDataFrame(df, geometry='Coordinates')

gdf.crs = {'init' :'epsg:4326'}
gdf = gdf.to_crs({'init': 'epsg:3945'})

#On modifie les coordonnées des bords de l'arbre pour qu'elles collent
↪  avec celui ci
for i in range(df.shape[0]):
    absi = gdf.iloc[i,9].x
    ordo = gdf.iloc[i,9].y
    ray = gdf.iloc[i,1] / 2
    demi_cote_boite = min(ray, 5)
    gdf.iloc[i, 4] = Polygon([(absi - demi_cote_boite,ordo -
    ↪  demi_cote_boite), (absi - demi_cote_boite, ordo +
    ↪  demi_cote_boite), (absi + demi_cote_boite, ordo +
    ↪  demi_cote_boite), (absi + demi_cote_boite, ordo -
    ↪  demi_cote_boite)])
    gdf.iloc[i, 5] = gdf.iloc[i, 4].bounds[0]
    gdf.iloc[i, 6] = gdf.iloc[i, 4].bounds[1]
    gdf.iloc[i, 7] = gdf.iloc[i, 4].bounds[2]
```

```
    gdf.iloc[i, 8] = gdf.iloc[i, 4].bounds[3]

del gdf['box_arbres']

#On modifie le polygone de découpe pour que toutes les découpes soient un
↪  carré de côté 10 mètres
for i in range(df.shape[0]):
    absi = gdf.iloc[i,8].x
    ordo = gdf.iloc[i,8].y
    gdf.iloc[i,8] = Polygon([(absi - 5,ordo - 5), (absi - 5, ordo + 5),
    ↪  (absi + 5, ordo + 5), (absi + 5, ordo - 5)])

#On enregistre le fichier au format GeoJSON
gdf.to_file("C:\\Users\\Tanguy\\Documents\\MIG\\arbresgeo.geojson",
↪  driver = "GeoJSON")
```

D.3 Découpe d'arbres

```
import rasterio as rst
import geopandas as gpd
import pickle
import os
import numpy as np
import cv2

pathPhotoFolder =
↪  'C:\\Users\\Tanguy\\Documents\\MIG\\BaseDonneeArbre\\ortho-bordeaux'
pathgeopandas =
↪  'C:\\Users\\Tanguy\\Documents\\MIG\\BaseDonneeArbre\\arbresgeo.geojson'

#On ouvre toutes les photos et on garde uniquement les fichiers .jpg
pathphotos_all = os.listdir(pathPhotoFolder)
pathphotos_jpg = []
for f in pathphotos_all:
    if '.jpg'in f:
        pathphotos_jpg.append(f)

pathphotos_jpg = np.array(pathphotos_jpg)

#On crée une liste dans laquelle chaque élément est un quadruplé avec les
↪  coordonnées dans le système epsg:3945 des bords de l'image
photosBoundsList = []

for i in range(len(pathphotos_jpg)):
    img = rst.open(pathPhotoFolder + '\\' + pathphotos_jpg[i])
    left, bottom, right, top = img.bounds[0], img.bounds[1],
    ↪  img.bounds[2], img.bounds[3]
    photosBoundsList.append([pathphotos_jpg[i], left, bottom, right,
    ↪  top])
    img.close

#on ouvre le fichier GeoJSON contenant les arbres déjà référencé
↪  contenant la position de leur contour
#et les coordonnées de l'image de côté 10m à découper
listeArbre = gpd.read_file(pathgeopandas, driver = 'GeoJSON')

compteur = 0
temps = 0

#On parcourt tous les arbres
for i in range(listeArbre.shape[0]):
    #On prend les coordonées du carré à découper
    cleft, cbottom, cright, ctop = listeArbre.iloc[i, 8].bounds
```

```
    temps += 1
    for photo in photosBoundsList:  #On parcourt toutes les images
        if photo[1] < cleft and photo[2] < cbottom and photo[3] > cright
         ↪  and photo[4] > ctop:   #On verifie que l'arbre est bien dans
         ↪  l'image
            img = cv2.imread(pathPhotoFolder + '\\' + photo[0])
            cote_gauche = photo[1]
            cote_bas = photo[2]
            decal = 5 - (listeArbre.iloc[i, 1] / 2)
            #Cette liste permet de décaler pour l'avoir a différents
             ↪  endroits de l'image et pas seulement au centre
            liste_modif_decoup = [[ 0 ,0 ,0 ,0], [ 0, 5, 0, 5], [ 5, 0,
             ↪  5, 0],
                      [ 0,-5, 0,-5], [-5, 0,-5, 0],
                      [ 0, decal, 0, decal], [ decal, 0, decal, 0],
                      [ 0,-decal, 0,-decal], [-decal, 0,-decal, 0],
                      [ decal, decal, decal, decal],
                       ↪  [-decal,-decal,-decal,-decal],
                      [ decal,-decal, decal,-decal], [-decal,
                       ↪  decal,-decal, decal],]
            compteur2 = 0
            for x in liste_modif_decoup:
                #On parcourt cette liste pour avoir les variations de
                 ↪  l'arbre
                #On convertit les coordonées du système epsg:3945 en
                 ↪  coordonées pixel de découpe en prenant en compte le
                 ↪  décalage pour que l'arbre ne soit pas au centre
                pix_left =  int((cleft - cote_gauche + x[0])/ 0.1)
                pix_top = int(10000 - (cbottom - cote_bas + x[1])/0.1)
                pix_right = int((cright - cote_gauche + x[2])/0.1)
                pix_bottom = int(10000 - (ctop - cote_bas + x[3])/0.1)
                decoupe = img[pix_bottom: pix_top ,  pix_left:
                 ↪  pix_right]
                if decoupe.shape[0] == 100  and decoupe.shape[1] == 100:
                    #On verifie que l'image fasse 10m de côté (si l'arbre
                     ↪  est au bord d'une des grandes images, il est
                     ↪  possible qu'en la déplacant une partie ne soit
                     ↪  plus dedans)
                    On convertit les contour de la boite contenantl
                     ↪  l'arbre en pixel
                    boxPixLeft =  int(max(pix_left, (listeArbre.iloc[i,4]
                     ↪  - cote_gauche)/ 0.1)) - pix_left
                    boxPixBottom = int( max(pix_bottom, (10000 -
                     ↪  (listeArbre.iloc[i,7] - cote_bas)/0.1))) -
                     ↪  pix_bottom
```

```
                    boxPixRight = int( min(pix_right,
                    ↪  (listeArbre.iloc[i,6] - cote_gauche)/0.1)) -
                    ↪  pix_left
                    boxPixTop = int(min(pix_top, (10000 -
                    ↪  (listeArbre.iloc[i,5] - cote_bas)/0.1)))-
                    ↪  pix_bottom
                    liste = [decoupe, [boxPixLeft, boxPixBottom,
                    ↪  boxPixRight, boxPixTop], 1, [cleft, ctop]]
                    res_bin =
                    ↪  open('C:\\Users\\Tanguy\\Documents\\MIG\\BaseDonneeArbre\\Arbr
                    ↪  + str(compteur) + '_' + str(compteur2) + '.bin'
                    ↪  ,'wb')
                    pickle.dump(liste, res_bin)
                    #On enregistre la liste obtenue dans un fichier
                    ↪  binaire
                    res_bin.close
                    #La ligne suivante permet si on le souhaite
                    ↪  d'enregistrer l'image découper au format png
                    ↪  (sous forme d'image)

                    ↪  #cv2.imwrite('C:\\Users\\Tanguy\\Documents\\MIG\\BaseDonneeArb
                    ↪  + 'arbre' + str(compteur) + str(compteur2) +
                    ↪  '.png', decoupe)
                compteur2 += 1
            compteur+=1
    print(temps)
print ("Finalement on a gardé", compteur)
```

D.4 Création des bases de données de prédiction

```
# Création des bases de données de prédiction pour le \ml

# Script (on fait sur les images avec au moins un arbre, sinon, on meurt
↪  (200Go et 19h pour les arbres et plus encore pour les voitures))
# Chaque arbre appartient à une et une seule image (c'est vérifié)
compteurimages = 0
cote = 6 #en m (se règle à 10 pour les arbres)
#une image est de dim (ligne, colonne, couleurs)
for k in range(len(pathphotos)):
    if nbArbresParImage[k] > 0:
        #initialisation avant exportaiton
        pathphoto = pathphotos[k]
        geodata = open(pathPhotoFolder + pathjgw[k], 'r')
        geodatalist = geodata.readlines()
        geodata.close
        stepx = float(geodatalist[0][:-2])
        stepy = float(geodatalist[3][:-2])
        xim = float(geodatalist[4][:-2])
        yim = float(geodatalist[5][:-2])
        compteurimages +=1
        print ('Image :', compteurimages, '/', nbImages)
        img = cv2.imread(pathPhotoFolder + pathphoto) la grande image en
        ↪  np.array
        H = int(len(img)/cote * (-stepy))
        L = int(len(img[0])/cote * stepx)
        tot = (H+1)*(L+1)
        for i in range (H):
            for j in range (L):
                toExport = [] #ce qui sera stocké en bin
                subimg =
                ↪  img[int(i*cote/(-stepy)):int((i+1)*cote/(-stepy)),
                ↪  int(j*cote/stepx):int((j+1)*cote/stepx)] #la sous
                ↪  image en np.array
                toExport.append(subimg)
                toExport.append(np.array([xim + j*cote, yim - i*cote]))
                export = open('C:\\Users\\Martin
                ↪  BRIAND\\Documents\\MIGSE\\testcarsphotos\\' +
                ↪  imagesName[k] + '_' + str(i) + '_' + str(j) + '.bin',
                ↪  mode = 'wb')
                pickle.dump(toExport, export)
                export.close


# Création d'un GeoJSON pour visualiser sur GeoJSONio l'ensemble des
↪  photos
```

```
# Création d'une data frame avec le bon format
polyslist = []
for k in range (len(pathphotos)):
    a, b, c, d = boundsImages[k]
    polyslist.append(Polygon(((a, d), (c, d), (c, b), (a, b))))

polyslist = np.array(polyslist)
df = pd.DataFrame({'name': pathphotos, 'geom': polyslist})

# Transformation en gdf avec le bon system de coord
gdf = gpd.GeoDataFrame(df, geometry = 'geom')
gdf.crs = {'init': 'epsg:3945'}
gdf = gdf.to_crs({'init': 'epsg:4326'})

# Exportation
gdf.to_file('C:\\Users\\Martin
↪  BRIAND\\Documents\\MIGSE\\Application\\PhotosBordeaux.geojson',
↪  driver = 'GeoJSON')
```

D.5 Découpe d'images de non-arbres pour l'entraînement

```
import rasterio
import rasterio.mask
from rasterio.plot import show
import geopandas as gdp
import numpy as np
import pickle
import numpy.ma as ma
import os

## Decoupe d'image ne contenant pas d'arbres pour la phase d'entraînement

chemin_image = "/home/vg/Bureau/MIG/données/image"
chemin_enregistrement = "/home/vg/Bureau/MIG/données/binaire_vegetation/"


#liste (String) les images jpg contenues dans fichier donne
liste_fichier1 = os.listdir(chemin_image)
liste_fichier2 = os.listdir("/home/vg/Bureau/MIG/données/vegetation")
liste_fichier = []
for x in liste_fichier1:
    if x[-3:] == 'jpg' and x[:2]+ '.geojson' in liste_fichier2:
        liste_fichier.append(x)



for x in liste_fichier1:
    if x[-3:] == 'jpg':
        liste_fichier.append(x)
for x in liste_fichier:
    geodataframe =
    ↪  gdp.read_file("/home/vg/Bureau/MIG/données/vegetation/" + x[:2] +
    ↪  ".geojson")
    raster = rasterio.open(chemin_image + x,"r")
    for i in range(len(geodataframe)-1):
        #ouvre image et effectue le calque
        out_image, out_transform = rasterio.mask.mask(raster,
        ↪  geodataframe[i:i+1].geometry, filled = False, crop=True)
        l=out_image.shape
        #test pour couper l'image
        if l[1]>100 and l[2]>100:
            out_image =
            ↪  out_image[:,l[1]//2-50:l[1]//2+50,l[2]//2-50:l[2]//2+50]
            #transpose l'image pour modifier dimension en (100,100,3)
            out_image = np.transpose(out_image)
            #enleve le maskedarray
```

```
out_image = out_image.data
liste_image = [out_image,[],0,[]]
#ouvre fichier binaire
fichier_binaire = open(chemin_enregistrement + "vegetation"
↪  +x[:2] + str(i) + ".bin","wb")
pickle.dump(liste_image,fichier_binaire)
#ferme le fichier binaire
fichier_binaire.close
```

E Élargissement des bases de données

E.1 Augmenter la base de données de voitures avec des voitures non détectées

```
def onclick(event):

    print(event.xdata, event.ydata)
    if event.dblclick: # Il suffit de cliquer au centre de la voiture non détectée
                       # pour sauvegarder ses coordonnées dans la liste L
        print(event.xdata, event.ydata)
        L.append([event.xdata, event.ydata])

img = cv2.imread('/home/user/resultat.jpg')

fig, ax = plt.subplots()
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB),aspect='auto')
L = []
plt.axis('equal')

# Permet d'interagir avec l'utilisateur via la souris
cid = fig.canvas.mpl_connect('button_press_event', onclick)
plt.show()

for i,mid in enumerate(L):
    xmid = int(mid[0])
    ymid = int(mid[1])
    subImg = img[ymid-24:ymid+24,xmid-24:xmid+24]
    cv2.imwrite('/home/user/VoituresOubliees/' + str(i),subImg)
```

E.2 Opérations sur les images

Ce programme permet de créer des variantes d'images à partir d'images d'une base de donnée, de façon à augmenter le nombre d'images de la base (nous l'avons utilisé pour les images de voitures).
Il s'utilise en ligne de commande dans le terminal, prend en argument une image et crée à partir de celle-ci trois variantes en modifiant le contraste et la luminosité.

Le mode verbeux (option -v) est très couteux car il faut convertir les images plusieurs fois de RGB à BGR et *vice versa*. D'habitude, cela n'est pas un problème car il n'a pas vocation à être utilisé pour traiter un grand nombre d'image mais bien pour n'en traiter qu'une seule.

Nous avons optimisé cette version du code au maximum pour permettre le traitement d'un grand nombre d'images.

```
import cv2
import argparse
import os
import matplotlib.pyplot as plt
from utile_plt import invRB
import numpy as np

# Récupération du nom de fichier et mise en place ou non du mode verbeux
# Mise en place des arguments pour l'appel de la fonction dans la console
parser=argparse.ArgumentParser()
parser.add_argument("image", help="Nom de l'image")
parser.add_argument("-v", "--verbose", help="affiche les nouvelles
↪  images",
                    action="store_true")
parser.add_argument("-a", "--affichage_seulement",
                    help="permet de désactiver le téléchargement des
                     ↪  images",
                    action="store_true")
parser.add_argument("-d", "--directory", help="répertoire où enregistrer
↪  les variantes")
args=parser.parse_args()
path=args.image
v=args.verbose      # v est un bool
a=args.affichage_seulement

# Nom des variantes successives construit avec la fonction nom
path_dir=os.path.dirname(path) # Nom du répertoire de l'image
im_name, _ = os.path.splitext(os.path.basename(path))

if args.directory:
    path_dir=args.directory

def nom(n):
```

```
    "Permet de générer le nom des variantes des images crées"
    nombre_de_zeros = 2 - bool(n//10) - bool(n//100)  # 3 -> 003 et 16 ->
    ↪ 016
    return path_dir+"/"+im_name+"_"+"0"*nombre_de_zeros+str(n)+".png"

# Fonction principale de création des variantes
def transfo(img, a, b):
    # Ne modifie pas l'image d'origine
    "variation de contraste et de luminosité"
    dimx, dimy, _ = img.shape
    f = lambda x: int(max(min(255, a*x+b), 0))
    f = np.vectorize(f)
    return f(img)

def transfoCouleur(img, a, b, k):
    # Modifie l'image d'origine par effet de bord
    "variation de contraste et luminasité seulement pour la couleur
    ↪ 0=<k<=2"
    dimx, dimy, _ = img.shape
    f = lambda x: int(max(min(255, a*x+b), 0))
    f = np.vectorize(f)
    for i in range(dimx):
        for j in range(dimy):
            img[i,j,k] = f(img[i,j,k])

def var(img):
    n=3 # nombre de variantes
    images=[]
    # Variante 1
    images.append(transfo(img, 1, 70))
    # Variante 2
    images.append(transfo(img, 0.6, 50))
    # Variante 3
    images.append(transfo(img, 1.4, -30))

    # Enregistrement si il n'a pas été déactivé
    if not a:
        for k in range(n):
            cv2.imwrite(nom(k),images[k])
    # Affichage si le mode verbeux est activé
    if v:
        for k in range(n):
            invRB(images[k])
            afficher(images[k], k+2, "Variante "+str(k+1))

def afficher(img, n, nom):
```

```
    plt.subplot(2,2,n)
    plt.imshow(img, interpolation = 'bicubic')
    plt.title(nom)
    plt.xticks([]), plt.yticks([])  # to hide tick values on X and Y axis

# Ouverture et conversion de l'image
img=cv2.imread(path,1)
if v:
    # On affiche l'image seulement si le mode verbeux est activé
    invRB(img)
    afficher(img, 1, "Original")
    invRB(img)

try:
    var(img)
except:
    print("Ceci n'est pas une image")

plt.show()
```

F Algorithme de *machine learning*

```
#importation des modules utilisés
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Convolution2D,
↪  MaxPooling2D
import pickle
import os
```

F.1 Algorithme de reconnaissance

```
#ALGORITHME DE RECONNAISSANCE

#création des bases de données d'entraînement et de test
train_data, train_label = np.empty((30000,3,100,100)),
↪  np.empty((30000,2))
test_data, test_label = np.empty((10000,3,100,100)), np.empty((10000,2))

#recherche des photos et ajout à la base de données
dossier = os.listdir('E:/Mines/MIG/git/ArbreDecoupeBinaire')
np.random.shuffle(dossier) #pour mélanger

for k in range (30000):
    element = dossier[k]
    #lecture des fichiers binaires
    fichier = open("E:/Mines/MIG/git/ArbreDecoupeBinaire/" + element,
    ↪  "rb")
    objet = pickle.load(fichier)
    if objet[0].shape != (100,100,3):
        print(element)
    else:
        train_data[k]=objet[0]
        if objet[2]==1: #c'est un arbre
            train_label[k]=[0,1]
        elif objet[2]==0: #ce n'est pas un arbre
            train_label[k]=[1,0]
        else:
            print("problème de données")
    fichier.close()
for k in range (10000):
    element = dossier[k+30000]
    fichier = open("E:/Mines/MIG/git/ArbreDecoupeBinaire/" + element,
    ↪  "rb")
    objet = pickle.load(fichier)
```

```
    if objet[0].shape != (100,100,3):
        print(element)
    else:
        test_data[k]=objet[0]
        if objet[2]==1: #c'est un arbre
            test_label[k]=[0,1]
        elif objet[2]==0: #ce n'est pas un arbre
            test_label[k]=[1,0]
        else:
            print("problème de données")
    fichier.close()


#création du réseau de neurones
data_depth, data_dim_1, data_dim_2 = 3, 100, 100
num_classes = 2
model = Sequential ()

model.add(Convolution2D(32, (3,3), activation = 'relu',
                        input_shape=(data_depth, data_dim_1, data_dim_2),
                        data_format='channels_last'))
model.add(Convolution2D(32, (3,3), activation='relu' ))
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Convolution2D(60, (3,3), activation='relu' ))
model.add(Convolution2D(60, (3,3), activation='relu' ))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes,activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',
↪  metrics=['accuracy'])

#entraînement du modèle
training = model.fit(train_data, train_label, epochs=5, verbose = 1)
#validation du modèle
test_loss, test_acc = model.evaluate(test_data, test_label)


#affichage des caractéristiques de l'entraînement
fig = plt.figure()
```

```
plt.subplot(2,1,1)
plt.plot(training.history['acc'])
plt.plot(training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')

plt.subplot(2,1,2)
plt.plot(training.history['loss'])
plt.plot(training.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')

#sauvegarde du modèle
model.save('model_reconnaissance.h5')
del model
```

F.2 Algorithme d'encadrement

```
#ALGORITHME D'ENCADREMENT

#création de la base de données d'entraînement et de test
train_data, train_box = np.empty((30000,100,100,3)), np.empty((30000,4))
test_data, test_box = np.empty((10000,100,100,3)), np.empty((10000,4))



#récupération des images
dossier = os.listdir('E:/Mines/MIG/git/ArbreDecoupeBonneBox')
np.random.shuffle(dossier)

#on a besoin de normaliser les labels
def normalisation(x):
    return(x/100)


for k in range (30000):
    element = dossier[k]
    #lecture du fichier binaire
    fichier = open("E:/Mines/MIG/git/ArbreDecoupeBonneBox/" + element,
    ↪  "rb")
    objet = pickle.load(fichier)
    if objet[0].shape != (100,100,3):
        print(element)
    else:
        if objet[2] == 1:
            #on ne traite que les arbres dans cet algorithme
            #mais normalement, il n'y a que des arbres dans ce dossier
            train_data[k]=objet[0]
            train_box[k]=list(map(normalisation,objet[1]))
    fichier.close()
for k in range (10000):
    element = dossier[k+30000]
    fichier = open("E:/Mines/MIG/git/ArbreDecoupeBonneBox/" + element,
    ↪  "rb")
    objet = pickle.load(fichier)
    if objet[0].shape != (100,100,3):
        print(element)
    else:
        if objet[2] == 1:
            test_data[k]=objet[0]
            test_box[k]=list(map(normalisation,objet[1]))
    fichier.close()
```

```
#création du réseau de neurones
data_depth, data_dim_1, data_dim_2 = 3, 100, 100
num_classes = 4
model = Sequential ()

model.add(Dropout(0.2))
model.add(Convolution2D(32, (3,3), activation = 'relu',
                        input_shape=(data_depth, data_dim_1, data_dim_2),
                        data_format='channels_last'))
model.add(Convolution2D(32, (3,3), activation='relu' ))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.2))
model.add(Convolution2D(32, (3,3), activation='relu' ))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))

model.add(Dense(num_classes, activation='linear'))

model.compile(loss='mean_squared_error', optimizer='adam',
↪  metrics=['accuracy'])



#entraînement du modèle
model.fit(train_data, train_box, epochs=5,verbose = 1)
#évaluation du modèle
score = model.evaluate(test_data, test_box)
print(score)

#enregistrement du modèle
model.save('model_box.h5')
del model
```

F.3 Algorithme global

```
#ALGORITHME GLOBAL
from keras.models import load_model

#récupération des deux modèles entraînés
modelrec = load_model('model_reconnaissance.h5')
modelbox = load_model('model_box.h5')

#chargement des données à traiter
dossier = os.listdir('E:/Mines/MIG/git/donneestest')
for element in dossier:
    if element.endswith('.bin') :
        #lecture d'un fichier binaire
        fichier = open("E:/Mines/MIG/git/donneestest/" + element, "rb")
        objet = pickle.load(fichier)
        image = objet[0]
        x, y = objet[1][0], objet[1][1]
        #coordonnées gps du pixel (0,0) de l'image
        testarbre = modelrec.predict(np.array([image]))
        if testarbre[0][0]<0.000001 and testarbre[0][1]>0.999999:
            #si on pense que c'est un arbre
            coordPred = modelbox.predict(np.array([image]))*100
            listXPred = []
            listYPred = []
            #on récupère les données
            for i in range(0,8,2) :    # on sépare les x et y
                listXPred.append(coordPred[0][i])
            for i in range(1,8,2) :
                listYPred.append(coordPred[0][i])
            #on convertit la pixel_box en box_gps
            box = []
            box.append(x + 0.1*listXPred[0])
            box.append(y - 0.1*listYPred[0])
            box.append(x + 0.1*listXPred[1])
            box.append(y - 0.1*listYPred[2])
            #écriture dans un fichier binaire
            res_bin = open('E:/Mines/MIG/git/resultats/box'+element
             ↪  ,'wb')
            pickle.dump(box, res_bin)
            res_bin.close()
        fichier.close()
```

G Post-traîtement des données

G.1 Amélioration des espaces verts existant déjà

```
# Script pour améliorer les enregistrements des espaces verts publics en
↪ traçant les enveloppes convexes


# Initialisation
pathgjson = 'C:\\Users\\Martin
↪ BRIAND\\Documents\\MIGSE\\Application\\vegetation\\'
pathnewgjson = 'C:\\Users\\Martin
↪ BRIAND\\Documents\\MIGSE\\Application\\vegetation2\\'
listOfGeoJSON = os.listdir(pathgjson)

# Création des enveloppes convexes
print (0, "/", len(listOfGeoJSON))
compteur = 0
for filename in listOfGeoJSON:
    gdf = gpd.read_file(pathgjson+filename, driver = 'GeoJSON')
    for k in gdf.index:
        gdf.loc[k, 'geometry'] = gdf.loc[k, 'geometry'].convex_hull
    z = gdf.columns.tolist().index('geometry')
    for i in range(len(gdf)-1):
        poly1 = gdf.iloc[i, z]
        for j in range (i+1, len(gdf)):
            poly2 = gdf.iloc[j, z]
            if poly1.within(poly2):
                gdf.iloc[i, z] = Point((0., 0.))
            elif poly2.within(poly1):
                gdf.iloc[j, z] = Point((0., 0.))
    gdf = gdf[~(gdf['geometry'].geom_type == 'Point')] #j'ai découvert
    ↪ après qu'il aurait été mieux de faire des références nulles
    ↪ plutôts que des points mais il s'agit ici du code qui a tourné
    gdf.to_file(pathnewgjson+filename, driver = 'GeoJSON')
    compteur+=1
    print (compteur, "/", len(listOfGeoJSON))


# Un script pour alléger les geojson d'Alexiane en prenant l'enveloppe
↪ convexe des parcs puis en retirant les recouvrement pour éviter les
↪ absurdités lors de calculs d'aires


#Executé sur 1/5ème des geojson en environ 13h

compteur = 0
```

```
diantrelist = [] #liste des geojson d'Alexiane qui pour une raison
↪  obscure n'ont pu être exportés (1% environ)
totorigin = len(listOfOriginFiles) #il s'agit de la liste des chemins
↪  vers le geojson d'alexiane
print (0, "/", len(listOfOriginFiles)) #suivi du script
for filename in listOfOriginFiles:
    gdf = gpd.read_file(pathOrigin+filename, driver = 'GeoJSON')
    for k in gdf.index: #tracé des enveloppes convexes
        gdf.loc[k, 'geometry'] = gdf.loc[k, 'geometry'].convex_hull
        if gdf.loc[k, 'geometry'].geom_type != 'Polygon':
            gdf.drop(k, axis = 0, inplace = True)
    z = gdf.columns.tolist().index('geometry')
    for i in range(len(gdf)-1): #Suppression des recouvrement en
    ↪  conservant l'aire totale.
        for j in range (i+1, len(gdf)):
            try:
                if not (gdf.iloc[i, z] - gdf.iloc[j, z]).is_empty:
                    gdf.iloc[i, z] = gdf.iloc[i, z] - gdf.iloc[j, z]
            except:
                print("pas dans le sens 1")
            try:
                if not (gdf.iloc[i, z] - gdf.iloc[j, z]).is_empty:
                    gdf.iloc[i, z] = gdf.iloc[i, z] - gdf.iloc[j, z]
            except:
                print("pas dans le sens 2") #des subtilités dues au
                ↪  polygones forcent l'existence des try
                print(i, j)
    try:
        gdf.to_file(pathDestinationConvex+filename, driver = 'GeoJSON')
    except:
        print ("Diantre", filename, "nous a quittés")
        diantrelist.append(filename)
    compteur += 1
    print(compteur, "/", len(listOfOriginFiles))
```

G.2 Création d'un fichier .geojson pour chaque image

```
import rasterio as ro
import geopandas as gdp
import pandas as pd
import json as js
import os


## Creation d'un fichier .geojson pour chaque image

path_image = "/home/vg/Bureau/MIG/données/image"
path_geojson = "/home/vg/Bureau/MIG/données/cadastre_parcelles1.geojson"
path_enregistrement = "/home/vg/Bureau/MIG/données/cadastre_parcelles/"

#liste (String) les images jpg contenu dans fichier donnees
liste_fichier1 = os.listdir(path_image)
liste_fichier = []
for x in liste_fichier1:
    if x[-3:] == 'jpg' :
        liste_fichier.append(x)

#importe le fichier geojson general
dataset = gdp.read_file(path_geojson)


#construction liste des coordonnees extremaux des polygones
boite_polygone=[]
for i in range (len(dataset)-1):
    boite_polygone.append(dataset.geometry[i].bounds)


#fonction retournant les coordonnees maximales du cadre
def cadre(photo):
    image = ro.open(path_image + "/" +photo)
    return image.bounds

l=[]
for photo in liste_fichier:
    #stocke les frontieres de l'image
    frontiere = cadre(photo)
    l=[]
    #parcourt tous les geojson
    for i in range(len(boite_polygone)-1):
        xMin,yMin,xMax,yMax=frontiere
        #test si le polygone est dans l'image consideree
```

```
        if boite_polygone[i][0]>xMin and boite_polygone[i][1]>yMin and
        ↪  boite_polygone[i]        [2]<xMax and
        ↪  boite_polygone[i][3]<yMax:
            l.append(dataset[i:i+1])
    if l != []:
        liste_polygone = pd.concat(l)
        #creee fichier geojson contenant tous les batiments dans l'image
        liste_polygone.to_file(path_enregistrement + "cadastre_parcelles"
        ↪  + photo[0:2]+'.geojson', driver = "GeoJSON")
```

H L'interface graphique

H.1 Corps du code

```
from tkinter import filedialog
import numpy as np
import cv2
from tkinter import *
from PIL import Image, ImageTk
from functools import partial
import ctypes
import time
import rasterio as rst
import geopandas as gpd
import pandas as pd
from shapely.geometry import Polygon, MultiPolygon
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
Image.MAX_IMAGE_PIXELS = None
plt.figure.max_open_warning = 2000

nom_fichier = []
first_time = [True]

## Le code n'est pas forcément dans l'ordre de lecture le plus explicite,
↪  néanmoins changer l'ordre de définition des fonctions et des classes
↪  peut rendre impossible le lancement de l'application


def essentiel(c):
    """ à partir d'un répertoire, ne donne que ce qu'il y a après le
    ↪  dernier /"""
    n = len(c)
    k = n-1
    essentiel = []
    while c[k] != "/":
        essentiel.append(c[k])
        k -=1
    l = essentiel.reverse()
    result = ""
    for j in range(len(essentiel)):
        result += essentiel[j]
    return result

def __from3945toPixels__ (a, x, y):
    """conversion 3945 -> pixel"""
    return np.array([int((a[0] - x)/0.1), int((y-a[1])/0.1)])
```

```
def __fromPixelstoGeographicPolygon3945__(l, x, y):#top left corner,
↪  abscisse and ordonnée
    """conversion pixel -> 3945"""
    left = min(l[0][1], l[1][1])
    right = max(l[0][1], l[1][1])
    top = min(l[0][0], l[1][0])
    bottom = max(l[0][0], l[1][0])

    left = x + 0.1*left
    right = x + 0.1*right
    top = y - 0.1*top
    bottom = y - 0.1*bottom
    return Polygon([(left, top), (left, bottom), (right, bottom), (right,
    ↪  top)])

class LoadImage:
    """classe pour l'image principale"""
    def __init__(self,root,bool,liste_activés):
        """constructeur"""
        if first_time[0]:
            frame_param.destroy()
            self.frame_param = Frame(root, background = "white",
            ↪  highlightbackground="#315414",highlightcolor="#315414",highlightthickn
            ↪  0)
            self.frame_param.grid(sticky = "n",row = 2, column = 1)
            root.bouton = Image.open("bouton.jpg")
            root.bouton = ImageTk.PhotoImage(root.bouton)
            Button(self.frame_param, command = partial(f,True),
            ↪  font="Helvetica 12 bold", background = 'white',
            ↪  foreground =
            ↪  "black",highlightcolor="black",highlightthickness=0.1, bd
            ↪  = 2, image = root.bouton).grid(column = 0, row = 3,
            ↪  columnspan = 2)
        else:
            root.l_app[0].frame_param.destroy()
            self.frame_param = Frame(root, background = "white",
            ↪  highlightbackground="#315414",highlightcolor="#315414",highlightthickn
            ↪  0)
            self.frame_param.grid(sticky = "n",row = 2, column = 1)
            # roo.bouton = Image.open("bouton.jpg")
            # self.bouton = ImageTk.PhotoImage(self.bouton)
```

```
        Button(self.frame_param, command = partial(f,True),
        ↪  font="Helvetica 12 bold", background = 'white',
        ↪  foreground =
        ↪  "black",highlightcolor="black",highlightthickness=0.1, bd
        ↪  = 2, image = root.bouton).grid(column = 0, row = 3,
        ↪  columnspan = 2)


    first_time.pop()
    first_time.append(False)
    if bool:
        root.filename =  filedialog.askopenfilename(initialdir =
        ↪  "/",title = "Select file",filetypes = (("jpg
        ↪  files","*.jpg"),("png files","*.png"), ("all
        ↪  files","*.*")))
        if len(nom_fichier)>0:
            nom_fichier.pop()

        nom_fichier.append(root.filename[:-4])
        c = essentiel(nom_fichier[0])

    else:
        self.liste_Martin = liste_activés
        root.filename = nom_fichier[0] +".jpg"
        c = essentiel(nom_fichier[0])

    self.liste_Martin = liste_activés
    self.bounds = None
    self.tab = None
    self.img_weighted = None
    self.img = None
    self.canvas = None
    self.filename = root.filename
    self.img2 = None
    self.comprX = None
    self.comprY = None
    self.imgf = None
    self.apresmaintenant = None
    self.zoomcycle = 0
    self.zimg_id = None
    self.coord = []
    self.rect = None
    self.infos = None

    #Gestion des filtres
    if bool:
```

```
            c = essentiel(nom_fichier[0])
            #ICONES
            #voitures = Icone("voiture2.jpg", "voiturerouge.jpg",c,
            ↪  "voiture", 6)
            self.arbres = Icone("arbrenoir.jpg", "arbrecouleur.jpg",c,
            ↪  "arbre", "arbre/", 8, 0, (255, 0, 0), 0, 0.5, 45, self)
            self.batiment = Icone("maisonnoir.jpg",
            ↪  "maisoncouleur.jpg",c, "batiment","batiment/", 10,0, (0,
            ↪  0, 255), 1, 0.5, 0, self)
            self.cadastre_parcelles = Icone("pasparcelle.jpg",
            ↪  "parcelle.jpg",c,
            ↪  "cadastre_parcelles","cadastre_parcelles/", 12, 0, (0, 0,
            ↪  0), 0, 1, 15, self)
            self.vegetation =
            ↪  Icone("pascadastre.jpg","cadastre.jpg",c,"vegetation","vegetation/",14
            ↪  0, (0, 255, 0), 1, 0.5, 80, self)

            self.listIcones = [self.batiment,self.vegetation,
            ↪  self.arbres, self.cadastre_parcelles]
            for icone in self.listIcones:
                if icone.exists:
                    self.liste_Martin.append(icone.var)

        else:
            c = essentiel(nom_fichier[0])
            #voitures = Icone("voiture2.jpg", "voiturerouge.jpg",c,
            ↪  "voiture", 6)
            self.arbres = Icone("arbrenoir.jpg", "arbrecouleur.jpg",c,
            ↪  "arbre", "arbre/", 8, self.liste_Martin[2], (255, 0, 0),
            ↪  0, 0.5, 45, self)
            self.batiment = Icone("maisonnoir.jpg",
            ↪  "maisoncouleur.jpg",c, "batiment", "batiment/", 10,
            ↪  self.liste_Martin[0], (0, 0, 255), 1, 0.5, 0, self)
            self.cadastre_parcelles = Icone("pasparcelle.jpg",
            ↪  "parcelle.jpg",c,
            ↪  "cadastre_parcelles","cadastre_parcelles/", 12,
            ↪  self.liste_Martin[3], (0, 0, 0), 0, 1, 15, self)
            self.vegetation =
            ↪  Icone("pascadastre.jpg","cadastre.jpg",c,"vegetation","vegetation/",14
            ↪  self.liste_Martin[1], (0, 255, 0), 1, 0.5, 80, self)

            self.listIcones = [self.batiment, self.vegetation,
            ↪  self.arbres, self.cadastre_parcelles]
```

```
a = rst.open(root.filename)
self.bounds = a.bounds
a.close()

Label(self.frame_param, text = "Nom: " +
 ↪  essentiel(nom_fichier[0]), background = "white", foreground =
 ↪  "black", font="SegoeUI").grid(sticky = "n", row = 2, column =
 ↪  0, columnspan = 2)
img = cv2.imread(root.filename, 1)
self.tab = img
Label(self.frame_param, text = "",bg = "white").grid(sticky =
 ↪  "n", row = 4, column = 0, columnspan = 2)
img_weighted = self.returnCarte()
b,g,r = cv2.split(img_weighted)
img_weighted = cv2.merge((r,g,b))
self.img_weighted = Image.fromarray(img_weighted)


t = np.shape(img)

if (t[1]> w_img) or (t[0]>h_img):
    self.canvas =
     ↪  Canvas(frame_img,width=w_img,height=h_img,highlightthickness=0)
    self.canvas.grid(row = 0, column = 0)

    self.filename = root.filename

    self.img2 = self.img_weighted.resize((w_img, h_img),
     ↪  Image.ANTIALIAS)
    self.comprY = t[1]/w_img  #Facteur de compression de l'image
    self.comprX = t[0]/h_img
else:
    self.canvas =
     ↪  Canvas(frame_img,width=t[1],height=t[0],highlightthickness=0)
    self.canvas.grid(row = 0, column = 0)
    self.img2 = self.img_weighted
    self.comprY = 1
    self.comprX = 1
if t[1] > t[0]:
    self.imgf = ImageTk.PhotoImage(self.img2)
else:
    self.imgf = ImageTk.PhotoImage(self.img2.rotate(-np.pi/2))
self.canvas.create_image(0,0,image=self.imgf, anchor="nw")
self.label = Label(self.frame_param, text = "Gestion filtres",
 ↪  background = "#315414", foreground = "#FFFFFF", font="SegoeUI
 ↪  20").grid(row = 5, column = 0, columnspan = 2)
```

```
        self.bouton2 = Image.open("bouton2.jpg")
        self.bouton2 = ImageTk.PhotoImage(self.bouton2)



        self.bouton = Button(self.frame_param, command = partial(f,
        ↪  False), background = 'white', foreground =
        ↪  "black",highlightcolor="black",highlightthickness=0.1, bd =
        ↪  2, image = self.bouton2).grid(column = 0, row = 17,
        ↪  columnspan = 2)


        self.label2 = Label(self.frame_param, text = "",bg =
        ↪  "white",height = 1).grid(sticky = "n", row = 16, column = 0)



        root.bind("<MouseWheel>",self.zoomer)
        self.canvas.bind("<Motion>",self.crop)
        self.canvas.bind("<Button-1>", self.pos)
        self.canvas.bind("<B1-Motion>", self.mouvement)
        self.canvas.bind("<ButtonRelease-1>", self.fin)

    def zoomer(self,event):
        """actualise la puissance de zoom par la molette"""
        if (event.delta > 0):
            if self.zoomcycle != 4: self.zoomcycle += 1
        elif (event.delta < 0):
            if self.zoomcycle != 0: self.zoomcycle -= 1
        self.crop(event)

    def crop(self,event):
        """effectue l'affichage du zoom"""
        if self.zimg_id: self.canvas.delete(self.zimg_id)
        aX = self.comprX
        aY = self.comprY
        if (self.zoomcycle) != 0:
            x,y = event.x, event.y
            if self.zoomcycle == 1:
                tmp =
                ↪  self.img_weighted.crop((aY*(x-45),aX*(y-30),aY*(x+45),aX*(y+30)))
            elif self.zoomcycle == 2:
```

```
                tmp =
                ↪ self.img_weighted.crop((aY*(x-30),aX*(y-20),aY*(x+30),aX*(y+20)))
            elif self.zoomcycle == 3:
                tmp =
                ↪ self.img_weighted.crop((aY*(x-15),aX*(y-10),aY*(x+15),aX*(y+10)))
            elif self.zoomcycle == 4:
                tmp =
                ↪ self.img_weighted.crop((aY*(x-6),aX*(y-4),aY*(x+6),aX*(y+4)))
            size = w_crop, h_crop
            self.zimg = ImageTk.PhotoImage(tmp.resize(size))
            self.zimg_id =
            ↪ self.canvas.create_image(event.x,event.y,image=self.zimg)

    def pos(self,event):
        """detection du clic pour la formation de la sélection du
        ↪ rectangle"""
        if len(self.coord) == 2:
            del self.coord[:]
            self.canvas.delete(self.rect)
            if self.infos != None:
                self.infos.window.destroy()
            self.infos = None
            return

        if len(self.coord) >0:
            del self.coord[:]
            self.canvas.delete(self.rect)
        self.coord.append((event.x, event.y))
        self.rect = self.canvas.create_rectangle(event.x, event.y,
        ↪ event.x, event.y, width = 9, outline = "black")

    def mouvement(self,event):
        """actualise le rectangle si le clic est maintenu et que la
        ↪ souris bouge"""
        curx, cury = event.x, event.y
        if curx<w_img and cury<h_img and curx >=0 and cury>=0 and
        ↪ len(self.coord) >0:
            self.canvas.coords(self.rect, self.coord[0][0],
            ↪ self.coord[0][1], curx, cury)

    def fin(self,event):
        """lorsque le clic est relaché, trace le rectangle final et lance
        ↪ les stats"""
        if len(self.coord)>0:
            self.coord.append((event.x, event.y))
            self.canvas.delete(self.rect)
```

```
            self.rect = self.canvas.create_rectangle(self.coord[0][0],
            ↪  self.coord[0][1], self.coord[1][0], self.coord[1][1],
            ↪  width = 9, outline = "black")
            x1, y1 = self.coord[0][0], self.coord[0][1]
            x2, y2 = event.x, event.y
            aX = self.comprY
            aY = self.comprX
            l = [(int(aX*x1),int(aY*y1)), (int(aX*x2), int(aY*y2))]
            res = self.statistics(l)
            self.affichage_info(res,l)


    def newGreen(self, poly, transparency):
        """création du nouvel espace vert"""
        icone = self.vegetation
        geos = icone.gdf.geometry.intersection(poly)
        b = MultiPolygon(list(geos))
        newpoly = b.convex_hull
        background = self.returnCarte()
        b,g,r = cv2.split(background)
        background = cv2.merge((r,g,b))
        foreground = background.copy()
        pts = np.array(newpoly.exterior.coords)[:-1]
        x, _, _, y = self.bounds
        for i in range(len(pts)):
            pts[i] = __from3945toPixels__(pts[i], x, y)
        pts = pts.reshape((-1,1,2))
        pts = pts.astype(np.int32)
        cv2.fillPoly(foreground, [pts],(0,130,0),)
        alpha = transparency
        cv2.addWeighted(foreground, alpha, background, 1- alpha, 0,
        ↪  background)

        left, bottom, right, top = newpoly.bounds
        left = int((left-x)/0.1)
        right = int((right - x)/0.1)
        bottom = int((y-bottom)/0.1)
        top = int((y- top)/0.1)
        left = max(0, left-1000)
        right = min(14000, right+1000)
        bottom = min(10000, bottom+1000)
        top = max(0, top-1000)
        background = background[top:bottom, left:right,:]
        return (self.statistics2(newpoly), background)

    def statistics(self, l):
```

```
        """return a dictionnary of tupple (see __qttetAireDans__) for
        ↪  each icon in listOfIcones"""
        x, _, _, y = self.bounds
        new_l = [(l[0][1],l[0][0]),(l[1][1],l[1][0])]
        poly = __fromPixelstoGeographicPolygon3945__(new_l, x, y)
        return self.statistics2(poly)

    def statistics2(self, poly):
        """renvoie les aires et le nombre de chaque type d'objet"""
        res = {'me': (poly.area, 1)}
        for icone in self.listIcones:
            if icone.exists:
                if icone.var:
                    geos = icone.gdf.geometry.intersection(poly)
                    geos = geos[~(geos.is_empty)]
                    res[icone.typename] = (int(geos.area.sum()),
                    ↪  int(geos.count()))
        return res


    def returnCarte(self):
        """return a map of the loaded app with all the acivated icones"""
        x, _, _, y = self.bounds
        background =self.tab.copy()
        for icone in self.listIcones:
            if icone.exists:
                if icone.var:
                    foreground = background.copy()
                    for poly in icone.gdf['geometry']:
                        if poly.geom_type == 'Polygon':
                            pts = np.array(poly.exterior.coords)[:-1]
                            for i in range(len(pts)):
                                pts[i] = __from3945toPixels__(pts[i], x,
                                ↪  y)
                            pts = pts.reshape((-1,1,2))
                            pts = pts.astype(np.int32)
                            if icone.fill:
                                cv2.fillPoly(foreground,
                                ↪  [pts],icone.color)
                            else:
                                cv2.polylines(foreground, [pts],
                                ↪  True,icone.color,icone.width)

                        elif poly.geom_type == 'MultiPolygon':
                            polys = list(poly)
```

```
                            for poly2 in polys:
                                pts =
                                ↪ np.array(poly2.exterior.coords)[:-1]
                                for i in range(len(pts)):
                                    pts[i] = __from3945toPixels__(pts[i],
                                    ↪ x, y)
                                pts = pts.reshape((-1,1,2))
                                pts = pts.astype(np.int32)
                                if icone.fill:
                                    cv2.fillPoly(foreground,
                                    ↪ [pts],icone.color)
                                else:
                                    cv2.polylines(foreground, [pts],
                                    ↪ True,icone.color,icone.width)

                    alpha = icone.transparency
                    cv2.addWeighted(foreground, alpha, background, 1-
                    ↪ alpha, 0, background)
        return background

    def affichage_info(self,l, rect):
        """affiche les stats dans une nouvelle fenêtre que l'on peut
        ↪ faire glisser avec la souris"""
        surf_tot = l['me'][0]
        if surf_tot > 0 :
            self.infos = FloatingWindow()
            #self.infos.window.geometry("600x600")
            self.infos.window.configure(background = "black")
            #Frame de l'info
            frame = Accrochable(self.infos)
            frame.widget = Frame(self.infos.window, bg = "black")
            frame.accroche(0,0, 1, 1, "")
            #Labels de l'info
            label = Accrochable(self.infos)
            label.widget = Label(frame.widget, text = "Informations
            ↪ annexes", foreground = "black", font = "SegoeUI 20 bold",
            ↪ bg = "green")
            label.accroche(1, 0, 1, 3,"nesw")
            label_r = Accrochable(self.infos)
            label_r.widget = Label(frame.widget, text = "", foreground =
            ↪ "black", font = "SegoeUI 20 bold", bg = "black", width =
            ↪ 3)
            label_r.accroche(2,1, 5, 1,"")
            if self.arbres.exists:
                if self.arbres.var:
                    nb_arbres = l["arbre"][1]*25
```

```
            label2 = Accrochable(self.infos)
            label2.widget = Label(frame.widget, text = "Quantité
            ↪ de CO2 absorbée (arbres) = {}
            ↪ kg/an".format(nb_arbres), foreground = "white",
            ↪ font = "SegoeUI 12", bg = "black")
            label2.accroche(2, 0, 1, 1, "w")
    # if self.voitures.exists
        # if self.voitures.var:
        #     nb_voitures = l["voitures"][1]*25
        #     label3 = Accrochable(self.infos)
        #     label3.widget = Label(frame.widget, text =
        ↪ "Quantité de CO2 émise (voitures) =
        ↪ {}".format(nb_voitures), foreground = "white", font =
        ↪ "SegoeUI 12", bg = "black")
        #     label.accroche(3, 0, 1, 1, "w")


    #Camembert

    l_surf = []
    l_nom = []
    surf_restante = 1
    for icone in self.listIcones:
        if icone.exists:
            if icone.var and icone.typename != "arbre" and
            ↪ icone.typename != "cadastre_parcelles" :
                a = l[icone.typename][0]/surf_tot
                l_surf.append(a)
                l_nom.append(icone.typename)
                surf_restante -= a
    l_nom.append("Autres")
    l_surf.append(surf_restante)


    afficher_camembert(l_nom,l_surf, "Utilisation du
    ↪ terrain",self,frame, 0, 0, 1, 3)
    #Barre
    # if batiment.var and parcelles.var:
    #     afficher_barre(l["batiment"][0]/l["parcelles"][0],
    ↪ "Occupation des parcelles", self,frame, 2, 1, 1, 1, "e")
    if self.vegetation.exists:
        if self.vegetation.var and l["vegetation"][1] > 1:
            boutonfutur = Image.open("futur.jpg")
            root.boutonfutur = ImageTk.PhotoImage(boutonfutur)
```

```
                    Button(frame.widget, command = partial(self.turfu,
                    ↪  rect), font="Helvetica 12 bold", background =
                    ↪  'white', foreground =
                    ↪  "black",highlightcolor="black",highlightthickness=0.1,
                    ↪  image = root.boutonfutur).grid(column = 0, row =
                    ↪  10, columnspan = 3,sticky = "s")

            if self.batiment.exists and self.cadastre_parcelles.exists:
                if self.batiment.var and self.cadastre_parcelles.var:
                    p = l["batiment"][0]/l["cadastre_parcelles"][0]
                    afficher_barre(p,"Taux de construction parcelle",
                    ↪  self, frame, 2, 2, 2, 1, "n")



    def turfu(self, l):
        """affichage de la création du nouvel espace vert"""
        x, _, _, y = self.bounds
        new_l = [(l[0][1],l[0][0]),(l[1][1],l[1][0])]
        poly = __fromPixelstoGeographicPolygon3945__(new_l, x, y)
        stats, img = self.newGreen(poly, 0.8)
        t = np.shape(img)
        img = Image.fromarray(img)
        if t[1] > t[0]:
            h = int(600 * t[0]/t[1])
            img = img.resize((600, h), Image.ANTIALIAS)
            img = ImageTk.PhotoImage(img)
            t = (600, h)
        else:
            w = int(600 * t[1]/t[0])
            img = img.resize((w,600), Image.ANTIALIAS)
            img = img.rotate(-90)
            img = ImageTk.PhotoImage(img)
            t = (600, w)
        root.img = img
        self.apresmaintenant = Futur(stats,root.img,t)


class Aide:
    """affichage de la fenêtre de renseignements"""
    def __init__(self):
        """constructeur"""
        self.active = 0
        self.img = Image.open("aide.jpg")
        self.img = self.img.resize((500,500), Image.ANTIALIAS)
```

```
        self.img = ImageTk.PhotoImage(self.img)
        self.fenetre = None
        self.canvas = None

    def active_ou_desactive(self):
        """affiche ou quitte la fenêtre"""
        if self.active:
            self.fenetre = FloatingWindow()
            self.fenetre.window.configure(background = "black")
            self.fenetre.window.geometry("{}x{}+{}+{}".format(500, 500,
             ↪  screensize[0]-700, int(screensize[1]/4)))
            self.canvas = Accrochable(self.fenetre)
            self.canvas.widget = Canvas(self.fenetre.window, width = 500,
             ↪  height = 500,highlightthickness=0, background = "black")
            self.canvas.accroche(0,0,1,1,"")
            self.canvas.widget.create_image(0,0,image = self.img, anchor
             ↪  = "nw")
        else:
            self.fenetre.destroy()
            self.fenetre = None
            self.canvas = None




    def reaction_bouton(self):
        """detection du clic"""
        self.active = 1 - self.active
        self.active_ou_desactive()



class Futur:
    """classe pour la création du nouvel espace vert"""
    def __init__(self, stats, img, t):
        self.frame_image = Accrochable(root.l_app[0].infos)
        self.frame_image.widget = Frame(root.l_app[0].infos.window, bg =
         ↪  "black")
        self.frame_image.accroche(0, 2, 11, 1, "")
        self.canvas = Accrochable(root.l_app[0].infos)
        self.canvas.widget = Canvas(self.frame_image.widget,width =t[0]
         ↪  ,height = t[1], bg = "black")
        self.canvas.accroche(0, 0,1, 1, "nesw")
        self.canvas.widget.create_image(0, 0, image = img,anchor = "nw")
```

```
        if root.l_app[0].batiment.exists:
            if root.l_app[0].batiment.var:
                label = Accrochable(root.l_app[0].infos)
                label.widget = Label(root.l_app[0].infos.window, text =
                ↪  "Surface de bâtiments à raser : {}
                ↪  m²".format(stats["batiment"][0]), foreground =
                ↪  "green", font = "SegoeUI 20 bold", bg = "black")
                label.accroche(1, 2, 1, 1,"n")
```

```
def afficher_camembert(labels, values, stat, app, accrochable, i , j, k ,
↪  l):
    """fonction pour créer un widget camembert"""
    camembert = Accrochable(app.infos)
    camembert.widget = Camembert(labels,values, stat, accrochable.widget)
    camembert.widget1 = camembert.widget.frame
    camembert.widget2 = camembert.widget.label
    camembert.widget3 = camembert.widget.canvas
    camembert.widget4 = camembert.widget.canvas2
    camembert.widget1.grid(row = i, column = j, rowspan = k, columnspan =
    ↪  l)
    camembert.accroche_bis(camembert.widget1)
    camembert.accroche_bis(camembert.widget2)
    camembert.accroche_figurecanvastkagg(camembert.widget3)
    camembert.accroche_figurecanvastkagg(camembert.widget4)

def afficher_barre(p,stat, app, accrochable, i, j, k, l, c):
    """fonction pour créer un widget barre de proportion"""
    pourcentage = Accrochable(app.infos)
    pourcentage.widget = Pourcentage(p, stat, accrochable.widget)
    pourcentage.widget1 = pourcentage.widget.frame
    pourcentage.widget2 = pourcentage.widget.label
    pourcentage.widget3 = pourcentage.widget.label0
    pourcentage.widget4 = pourcentage.widget.label100
    pourcentage.widget5 = pourcentage.widget.canvas
    if c == "":
        pourcentage.widget1.grid(row = i, column = j, rowspan = k,
        ↪  columnspan = l)
    else:
        pourcentage.widget1.grid(row = i, column = j, rowspan = k,
        ↪  columnspan = l, sticky = c)
    pourcentage.accroche_bis(pourcentage.widget1)
```

```
    pourcentage.accroche_bis(pourcentage.widget2)
    pourcentage.accroche_bis(pourcentage.widget3)
    pourcentage.accroche_bis(pourcentage.widget4)
    pourcentage.accroche_bis(pourcentage.widget5)



def f(bool):
    """fonction pour actualiser la grande image"""
    liste_Martin = []

    if len(root.l_app)>0:
        liste_Martin = root.l_app[0].liste_Martin

    App = LoadImage(root,bool, liste_Martin)

    if len(root.l_app)>0:
        old_App = root.l_app[0]
        old_App.canvas.destroy()
        del root.l_app[0]
    root.l_app.append(App)
```

H.2 Lancement de l'application et réglages

```
"""lancement de l'application et réglages"""
root = Tk()
aide = Aide()
root.title("DUD")
root.state('zoomed')


#Adaptation à la taille de l'écran

user32 = ctypes.windll.user32
screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)

root.geometry("{}x{}+{}+{}".format(screensize[0], screensize[1], 0, 0))
w_img = int(0.833*screensize[0])
h_img = int(0.833*screensize[1])
w_crop = int(0.15625*screensize[0])
h_crop = int(0.1851*screensize[1])
w_logo = int(0.0833*screensize[0])
h_logo = int(0.1204*screensize[1])
w_icon = int(0.0521*screensize[0])
h_icon = int(0.0926*screensize[1])


root.configure(background='white')



#Recentrage et outils annexes


frame_r = Frame(root, background = "white")
frame_r.grid(sticky = "nw", row = 0, column = 1)
Label(frame_r, text = "",bg = "white",height = 4).grid(sticky = "n", row
↪  = 0, column = 0)

boutonaide = Image.open("boutonaide.jpg")
boutonaide = ImageTk.PhotoImage(boutonaide)

frame_aide = Frame(root, background = "white")
frame_aide.grid(sticky = "nw", row = 0, column = 0)
```

```
Button(frame_aide, command = partial(aide.reaction_bouton), background =
↪  'white', foreground =
↪  "black",highlightcolor="black",highlightthickness=0, bd = 0, image =
↪  boutonaide).grid(column = 0, row = 0)


frame_r5 = Frame(root, background = "white",
↪  highlightbackground="#315414",highlightcolor="#315414",highlightthickness=5,bd=
↪  0)
frame_r5.grid(row = 1, column = 1)
Label(frame_r5, text = "",bg = "white",height = 1).grid(sticky = "n", row
↪  = 0, column = 0, rowspan = 3)

frame_r2 = Frame(root, bg ="white")
frame_r2.grid(row = 1, column = 2, rowspan = 2)
Label(frame_r2, text = "",bg = "white",width = 5).grid(sticky = "n", row
↪  = 0, column = 0, rowspan = 3)

frame_r3 = Frame(root, bg ="white")
frame_r3.grid(row = 0, column = 3)
Label(frame_r3, text = "",bg = "white",height = 1).grid(sticky = "n", row
↪  = 0, column = 0)


frame_r4 = Frame(root, bg ="white")
frame_r4.grid(row = 2, column = 0, rowspan = 2)
Label(frame_r4, text = "",bg = "white",width = 6).grid(sticky = "n", row
↪  = 0, column = 0)

# LOGO
frame_param = Frame(root, background = "white",
↪  highlightbackground="#315414",highlightcolor="#315414",highlightthickness=5,bd=
↪  0)
frame_param.grid(sticky = "n",row = 2, column = 1)
canvas1 = Canvas(frame_r5,width = w_logo,height =
↪  h_logo,highlightthickness=0, background = "white")
canvas1.grid(sticky = "nw",row = 0, column = 0, columnspan = 2)
#Si tu lis ce code en entier bravo
logo = Image.open("logomig.jpg")
logo = logo.resize((w_logo, h_logo), Image.ANTIALIAS)
logo = ImageTk.PhotoImage(logo)
canvas1.create_image(0,0,image=logo, anchor = "nw")
#Ouverture d'image
frame_img = Frame(root,
↪  bg="white",highlightbackground="black",highlightcolor="black",highlightthickness=
↪  0)
```

```
frame_img.grid(row = 1, column = 3, rowspan = 2)



root.l_app = []


canvas2 = Canvas(frame_img,width = w_img,height =
↪  h_img,highlightthickness=0)
canvas2.grid(sticky = "nw",row = 0, column = 0)
insertimg = Image.open("insertimg.jpg")
insertimg = insertimg.resize((w_img, h_img), Image.ANTIALIAS)
insertimg = ImageTk.PhotoImage(insertimg)
canvas2.create_image(0,0,image=insertimg, anchor="nw")

bouton = Image.open("bouton.jpg")
bouton = ImageTk.PhotoImage(bouton)


Button(frame_param, command = partial(f,True), font="Helvetica 12 bold",
↪  background = 'white', foreground =
↪  "black",highlightcolor="black",highlightthickness=0.1, bd = 2, image
↪  = bouton).grid(column = 0, row = 3, columnspan = 2)
```

H.2.1 Icônes

```
#Icones

class Icone:
    """classe représentant un type d'objet (exemples: voitures, arbres,
    ↪ etc)"""
    def __init__(self,filename,filename2,nom_img, typename, geopath, i,
    ↪ b,c,fill,transparency, width, app):
        if nom_img in img_compatible[typename]:
            self.exists = 1
            self.width = width
            cote = h_icon
            self.frame = app.frame_param
            self.canvas = Canvas(self.frame, width = cote, height =
            ↪ cote,highlightthickness=0,background = "white",bd = 0)
            self.fill = fill
            self.nom_img = nom_img
            self.typename = typename
            self.path = geopath + typename + nom_img + ".geojson"
            self.gdf = gpd.read_file(self.path, driver = 'GeoJSON')
            self.color = c
            self.transparency = transparency
            self.canvas.grid(row = i, column = 0, columnspan = 2)
            self.canvas.bind("<Button-1>", self.detection_click)
            self.img = Image.open(filename)
            self.img = self.img.resize((cote,cote), Image.ANTIALIAS)
            self.img = ImageTk.PhotoImage(self.img)
            self.img2 = Image.open(filename2)
            self.img2 = self.img2.resize((cote,cote), Image.ANTIALIAS)
            self.img2 = ImageTk.PhotoImage(self.img2)

            self.var = b
            if b:
                self.img_on_canv = self.canvas.create_image(0,0,image =
                ↪ self.img2, anchor = "nw")
            else:
                self.img_on_canv = self.canvas.create_image(0,0,image =
                ↪ self.img, anchor = "nw")
        else:
            self.exists = 0


    def detection_click(self,event):
        a = self.var
        self.var = 1-a
        if self.typename == "all":
```

```
            selectall()
        else:
            update(self)



def update_liste():
    """actualise l'état activé ou non des icones"""
    n = len(root.l_app[0].liste_Martin)
    for k in range(n):
        root.l_app[0].liste_Martin.pop()
    #liste_Martin.append(voitures.var)

    if root.l_app[0].batiment.exists:
        root.l_app[0].liste_Martin.append(root.l_app[0].batiment.var)
    else:
        root.l_app[0].liste_Martin.append(0)
    if root.l_app[0].vegetation.exists:
        root.l_app[0].liste_Martin.append(root.l_app[0].vegetation.var)
    else:
        root.l_app[0].liste_Martin.append(0)
    if root.l_app[0].arbres.exists:
        root.l_app[0].liste_Martin.append(root.l_app[0].arbres.var)
    else:
        root.l_app[0].liste_Martin.append(0)
    if root.l_app[0].cadastre_parcelles.exists:

        ↪  root.l_app[0].liste_Martin.append(root.l_app[0].cadastre_parcelles.var)
    else:
        root.l_app[0].liste_Martin.append(0)

def update(icone):
    """actualise l'image d'un icone selon son état"""
    b = icone.var
    if b == 0:
        icone.canvas.itemconfig(icone.img_on_canv, image = icone.img)
    if b == 1:
        icone.canvas.itemconfig(icone.img_on_canv, image = icone.img2)
    update_liste()
```

H.2.2 Classes utiles

```
#Classes utiles

class FloatingWindow:
    """classe pour créer une fenêtre que l'on peut glisser avec le clic
    ↪ de la souris"""
    def __init__(self):
        self.window = Toplevel(root)
        self.window.overrideredirect(True)


    def StartMove(self, event):
        self.x = event.x
        self.y = event.y

    def StopMove(self, event):
        self.x = None
        self.y = None

    def OnMotion(self, event):
        deltax = event.x - self.x
        deltay = event.y - self.y
        x = self.window.winfo_x() + deltax
        y = self.window.winfo_y() + deltay
        self.window.geometry("+%s+%s" % (x, y))
    def destroy(self):
        self.window.destroy()

class Accrochable:
    """classe pour rendre un widget accrochable par la souris"""
    def __init__(self, floatingwindow):
        self.floatingwindow = floatingwindow
        self.widget = None
        self.widget1 = None
        self.widget2 = None
        self.widget3 = None
        self.widget4 = None
        self.widget5 = None



    def accroche(self, i ,j, k, l, c):
        if c == "":
            self.widget.grid(row = i, column = j, rowspan = k, columnspan
            ↪ = l)
        else:
```

```
            self.widget.grid(row = i, column = j, rowspan = k, columnspan
            ↪  = l, sticky = c)
        self.widget.bind("<ButtonPress-1>",
        ↪  self.floatingwindow.StartMove)
        self.widget.bind("<ButtonRelease-1>",
        ↪  self.floatingwindow.StopMove)
        self.widget.bind("<B1-Motion>", self.floatingwindow.OnMotion)

    def accroche_bis(self, widget):
        widget.bind("<ButtonPress-1>", self.floatingwindow.StartMove)
        widget.bind("<ButtonRelease-1>", self.floatingwindow.StopMove)
        widget.bind("<B1-Motion>", self.floatingwindow.OnMotion)

    def accroche_figurecanvastkagg(self,widget):
        widget.get_tk_widget().bind("<ButtonPress-1>",
        ↪  self.floatingwindow.StartMove)
        widget.get_tk_widget().bind("<ButtonRelease-1>",
        ↪  self.floatingwindow.StopMove)
        widget.get_tk_widget().bind("<B1-Motion>",
        ↪  self.floatingwindow.OnMotion)


class Pourcentage:
    """classe pour créer un widget barre de pourcentage"""
    def __init__(self,p,stat, master):
        self.frame = Frame(master, bg = "black")
        self.label = Label(self.frame, text = stat, font = "SegoeUI 12
        ↪  bold",bg = "black", fg = "white")
        self.label.grid(row = 0, column = 1, sticky = "n")
        self.label0 = Label(self.frame, text = "0", font = "SegoeUI 12",
        ↪  bg = "black", fg = "white")
        self.label0.grid(row = 1, column = 0, sticky = "n")
        self.label100 = Label(self.frame, text = "100", font = "SegoeUI
        ↪  12", bg = "black", fg = "white")
        self.label100.grid(row = 1, column = 3, sticky = "n")
        self.canvas = Canvas(self.frame, width = 200, height = 20, bg =
        ↪  "black")
        self.canvas.grid(row = 1, column = 1)
        barre = Image.open("barre.jpg")
        barre = ImageTk.PhotoImage(barre)
        self.canvas.create_image(0,0,image = barre)
        self.canvas.create_rectangle(0,0, int(p*200), 22, fill = "green")


class Camembert:
    """classe pour créer un widget camembert"""
```

```
    def __init__(self,labels,values,stat, master):
        self.frame = Frame(master, bg = "black")
        self.fig = plt.figure(figsize = (3,3), facecolor = "black")
        self.explode = list()
        for k in labels:
            self.explode.append(0.1)
        self.camembert= plt.pie(values, explode = self.explode, shadow =
        ↪  True, autopct='%1.1f%%', radius = 1.3)
        self.fig2 = plt.figure(figsize = (3,2), facecolor = "black")
        ax2 = self.fig2.add_subplot(1, 1, 1)
        plt.legend(self.camembert[0], labels, loc="upper right",
        ↪  framealpha = 0.6)
        ax = plt.gca()
        ax.set_facecolor('black')
        self.label = Label(self.frame, text = stat, bg = "green", fg =
        ↪  "black", font = "SegoeUI 20 bold")
        self.label.grid(row = 0, column = 0, columnspan = 2, sticky =
        ↪  "nesw")
        self.canvas2 = FigureCanvasTkAgg(self.fig2, self.frame)
        self.canvas = FigureCanvasTkAgg(self.fig, self.frame)
        self.canvas.get_tk_widget().grid(row = 1, column = 0)
        self.canvas.get_tk_widget().configure(bg ="black")
        self.canvas2.get_tk_widget().configure(bg = "black")
        self.canvas2.get_tk_widget().grid(row = 1, column = 1)

        self.canvas.show()
        self.canvas2.show()


"""Pour chaque type d'objet, ce dictionnaire donne la liste des images
↪  contenant au moins une instance de cet objet"""
img_compatible = {
        "vegetation" : ['AP', 'BN', 'BO', 'BP', 'BQ', 'BR', 'BS', 'CM',
        ↪  'DN', 'DO', 'DP', 'DQ', 'DR', 'DS', 'DU', 'DV', 'EL', 'EM',
        ↪  'EN', 'EO', 'EP', 'ES', 'ET', 'EU', 'EV', 'FL', 'FM', 'FN',
        ↪  'FO', 'FQ', 'FR', 'FS', 'FT', 'FU', 'FV', 'FW', etc],
        "batiment": ['1.', 'AP', 'AQ', 'BR', 'CR', 'CS', 'CU', 'DL',
        ↪  'DQ', 'DR', 'DS', 'DU', 'DV','DW', 'EM', 'EO', 'EQ', 'ER',
        ↪  'ET', 'EU', etc],
        "arbre" : ["RM","RL","OL","MP"],
        "cadastre_parcelles" : ['AN', 'AP', 'AQ', 'AR', 'BM', 'BN', 'BS',
        ↪  'CK', 'CL', 'CM', 'CR','CS', 'CT', 'CU', 'DK', 'DL', 'DM',
        ↪  'DR', etc]}

root.mainloop()
```

Références

[1] PL 802, Projet de loi pour une République numérique, *Session extraordinaire de 2015-2016, 14e lég.*, sanctionné le 20 juillet 2016 ;

[2] GeoJson.io, `https://geojson.io/`, permet l'affichage du `.geojson` par dessus des images satellites ;

[3] Stack Overflow. `https://stackoverflow.com/` ;

[4] Keras Documentation, `https://keras.io/`, documentation sur les fonctions de base de Keras ;

[5] OpenCV Documentation, `https://opencv-python-tutroals.readthedocs.io`, tutoriels sur l'utilisation d'OpenCV ;

[6] Rasterio Documentation, `https://rasterio.readthedocs.io/en/latest/`, documentation sur les fonctions de bases de Rasterio ;

[7] Geopandas Documentation, `http://geopandas.org/`, documentation sur les fonctions de bases de Geopandas ;

[8] *S. Tuermera, J. Leitloffa, P. Reinartza, U. Stillab*, Evaluation of selected features for car detection in aerial images, ISPRS Hannover 2011 Workshop (Germany), `https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XXXVIII-4-W19/341/2011/isprsarchives-XXXVIII-4-W19-341-2011.pdf` ;

[9] Open Data Bordeaux, `http://opendata.bordeaux.fr/` ;

[10] Dota Data Set, `https://captain-whu.github.io/DOTA/dataset.html` ;

[11] *P. Ferlet*, Capturing your dinner, a deep learning story, `https://medium.com/smileinnovation/capturing-your-dinner-a-deep-learning-story-bf8f8b65f26f` ;

[12] Pngtree, `https://fr.pngtree.com/`, pour l'interface graphique ;

[13] Freepik, pareil `https://fr.freepik.com/` ;

[14] *K. Liu* & G. Mattyus, Fast multiclass vehicle detection on aerial images, contient une étude des features (caractéristiques) les plus efficaces pour la détection aérienne d'objets, `https://elib.dlr.de/96765/1/liu_mattyus_jrnl.pdf` ;

[15] *J. Brownlee*, `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/` ;

[16] *Office régional de l'OMS pour l'Europe de Copenhague*, Urban green spaces and health, `http://www.euro.who.int/__data/assets/pdf_file/0005/321971/Urban-green-spaces-and-health-review-evidence.pdf`, documentation sur les indicateurs sur la santé en ville ;

[17] *A. Bridges*, The rise of cities in the battle against climate change, `https://phys.org/news/2018-03-cities-climate.html`, Earth Institute, Columbia University, 2018 ;

[18] *Université de Mathématiques de Toulouse* Machines à Vecteurs Supports `https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-m-app-svm.pdf`

[19] *F. Precioso*, From Shallow to Deep representation for multimedia data classification : Artificial Neural Networks, Convolutional Neural Networks and Adversarial Examples, Support de cours de Frédéric Precioso remanié par Lyes Khacef pour son exposé du 19 novembre 2019 ;