WP3

Classification de l'occupation du sol (LC) par méthode orientée objet

Guide méthodologique et recommandations d'usages (D3.4)

Version révisée

14/02/2020

Subvention financée par le Service Public de Wallonie, DGO3 et le Département de la Géomatique du Secrétariat Général.

Table des matières

1 Introduction

Le projet WALOUS a développé une méthode automatisée pour la cartographie de l'occupation du sol en Wallonie. La démarche prévoit deux processus parallèles de classification automatique sur les orthophotos de la région : une classification par objet et une par pixel. La première est plus performante pour les environnements bâtis tandis que la deuxième plus pour les environnements naturels. Deux autres classifications par pixel, basées sur des images Sentinel à plusieurs dates dans l'année, complètent la série, permettant de mieux caractériser les parties du territoire qui changent de façon systématique au cours des saisons (ex : les champs). L'ensemble de ces classifications est alors fusionné pour obtenir une seule carte d'occupation du sol.

Ce présent document présente le guide méthodologique pour la classification par objets. Il se veut le plus complet possible pour permettre à d'autres de reproduire la démarche. Nous présentons dès lors d'abord une vision d'ensemble de la démarche, pour ensuite rentrer dans les détails pour chaque étape.

L'ensemble du code développé dans le cadre du projet est disponible publiquement sur https://github.com/mlennert/WALOUS.

2 Logiciels utilisés

L'ensemble des traitements de la chaîne OBIA a été implémenté avec des logiciels libres et donc disponibles gratuitement pour tous les systèmes d'exploitation principaux. Tous jouissent d'une communauté d'usagers active et présente donc une grande probabilité de durabilité. Voir l'annexe 7.2 pour le détail des versions de l'ensemble des logiciels et dépendances utilisées lors de l'élaboration de la carte LC-OBIA.

L'outil principal est le logiciel GRASS GIS (https://grass.osgeo.org/) utilisant Python (https://www.python.org/) comme langage de script. Lors du développement de la chaîne la version stable de GRASS GIS était la version 7.6 qui n'était pas encore compatible avec Python 3. Les scripts ont donc été écrits en Python 2. Entre temps, la version 7.8 de GRASS GIS est sortie, compatible cette fois-ci avec Python 3 (mais aussi toujours avec Python 2). La traduction des scripts vers Python 3 ne devrait pas demander un effort démesuré.

En sus de la version de base de GRASS GIS, certaines extensions sont nécessaires et peuvent être installées très simplement avec l'outil g.extension de GRASS GIS

(il est envisageable d'inclure l'installation au début du premier script). Il s'agit des extensions suivantes :

i.superpixels.slic	r.neighborhoodmatrix	v.class.mlR
i.segment.uspo	r.object.geometry	
i.segment.stats	r.texture.tiled	
i.cutlines	r.mapcalc.tiled	

Un des modules de GRASS GIS utilisé utilise le logiciel statistique R (https://www.r-project.org/). Ce logiciel doit donc également être installé.

Voir l'annexe 7.2 pour le détail des logiciels et leurs versions utilisées lors de l'élaboration de la carte LC-OBIA.

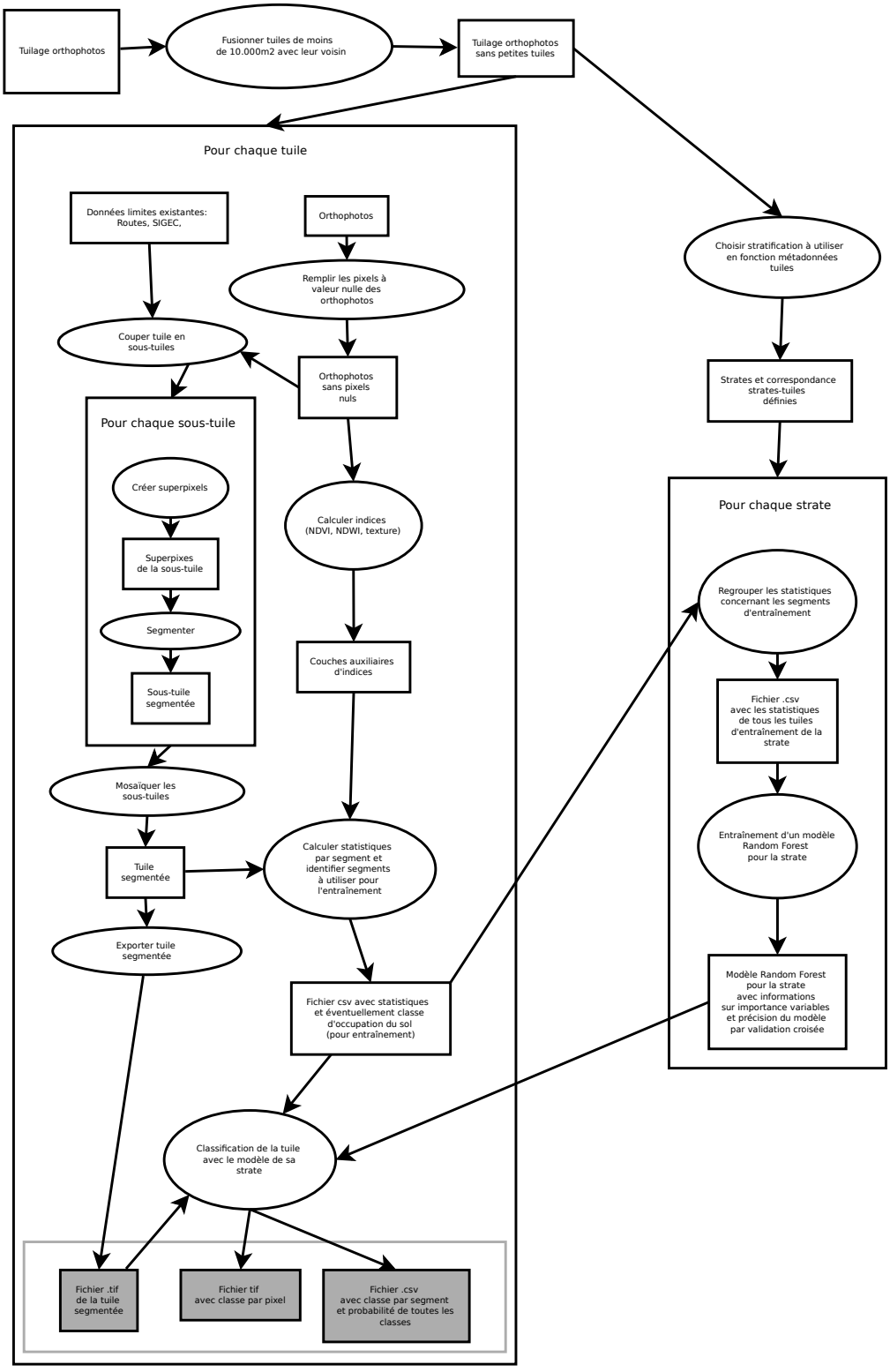
3 Approche méthodologique

La classification par objet dans le cadre de Walous suit la lignée déjà entamée avec le projet SmartPop (ainsi que des projets BELSPO comme MAUPP et REACT), focalisé particulièrement sur la cartographie de l'occupation du sol en milieu bâti. La démarche reflète les principes de base suivants :

- une attention à une parallélisation importante des calculs à travers un traitement par tuiles, tout en évitant le plus possible les effets de bord ;
- une prise en compte des conditions de prise de vue différentes des images en traitant séparément des groupes d'images homogènes.
- une volonté d'automatisation complète

Le schéma suivant (Figure 1) montre l'ensemble des processus mis en œuvre. Dans le chapitre suivant, les différentes étapes sont détaillées en relevant surtout les enjeux de chaque étape et les choix importants à faire. Pour chaque étape, le fichier script Python est référencé et, si pertinent, les modules GRASS GIS utilisés sont précisés.

Figure 1: Vue globale des processus constituant la chaîne OBIA



4 Les étapes en détail

4.1 Remarque préliminaire : la parallélisation

Un aspect important de l'ensemble de la chaîne est la parallélisation de toutes les étapes où une telle parallélisation est pertinente. La parallélisation est implémentée à deux niveaux : une parallélisation à haut niveau, avec l'ensemble de la chaîne conçue pour un traitement tuile par tuile, avec de traiter plusieurs tuiles en parallèle sur un cluster de nœuds de calcul. L'implémentation pratique de cette parallélisation dépend de l'environnement. Le dépôt Github contient les scripts .pbs contenant à titre d'exemple les instructions pour le système de gestion de tâches utilisée dans le système HPC Hydra à l'ULB.

Le deuxième niveau de parallélisation est implémenté dans une bonne partie des modules GRASS GIS utilisés dans la chaîne de traitement. Elle concerne donc les processus appliqués sur chaque tuile/sous-tuile. La possibilité de parallélisation dépend ici des capacités sur chaque nœud de calcul utilisé. A titre d'exemple, lors des traitements à l'ULB nous avons utilisé des nœuds avec 8 processeurs chacun. Ce nombre de processeurs disponible doit être défini dans la variable PROCESSES dans le fichier *walous_obia_config.py*.

4.2 Prétraitements

Suppression de tuiles trop petites

Réalisé en dehors des scripts avec module GRASS GIS v.clean, option=rmarea.

La mosaïque des orthophotos a été élaborée en assemblant des tuiles irrégulières provenant des photos individuelles, en fonction de la qualité des images respectives. Dans certaines situations de toutes petites tuiles ont été utilisées ce qui peut poser problèmes lors des traitements ultérieurs (surtout lors de la création des superpixels et de la segmentation). Nous avons donc décidé de fusionner ces toutes petites tuiles avec leur voisine avec laquelle elles partageaient la plus longue frontière.

Élimination des pixels à valeur nulle

Fichiers : walous_obia_tiles_data_creation.py, walous_obia_functions.py

Les orthophotos peuvent contenir des pixels à valeur nulle et ce à travers toutes les bandes, même si ces pixels sont particulièrement fréquents dans le proche

infrarouge. Nous appliquons donc un filtre qui attribue à ces pixels la moyenne pondérée par la distance des pixels autour, à l'aide du module *r.fill.stats* (dans la fonction auxiliaire fill_band()). La fenêtre utilisée pour calculer cette moyenne est une simple fenêtre 3x3 pour éviter de plus grands sauts de valeurs, et il faut au moins deux valeurs remplies dans cette fenêtre pour calculer la moyenne. Toutes les valeurs existantes sont évidemment préservées. Dans les cas où tout un groupe de pixels adjacents a une valeur nulle, le script répète donc l'appel au module en boucle pour remplir ces trous par les bords vers l'intérieur.

4.3 Sous-tuilage

Fichiers : walous_obia_tiles_data_creation.py, walous_obia_functions.py, walous_obia_config.py)

Une première étape dans la classification par objet est la création des objets par segmentation. Comme l'expérience des autres projets l'a montré, l'hétérogénéité dans les images traitées exige une approche spatialement différenciée de choix des paramètres de cette segmentation. Pour pouvoir assurer une telle différenciation, l'image doit être découpée en plus petites tuiles, de préférence sans que les limites de ces tuiles passent à travers des objets qui nous intéressent. Les orthophotos de la Région sont déjà constituées de tuiles représentant différentes prises de vue. Ces tuiles sont néanmoins trop grandes pour les utiliser comme plus petite unité spatiale de différenciation. La première partie du travail a donc consisté en la mise en place d'une chaîne de traitement assurant un découpage traversant le moins possible les objets que l'on souhaite identifier. Pour ce sous-tuilage, nous utilisons comme base l'image panchromatique, donc la combinaison des trois bandes rouge, vert et bleu par un simple calcul de moyenne (fonction auxiliaire *calculate_panvis*()).

Pour commencer, nous avons défini (hors du script) les polygones existants que nous souhaitons d'office utiliser comme sous-tuiles, à savoir les polygones existants du SIGEC et des forêts du PICC (couche définie dans walous_obia_config.py par la variable EXISTING_TILE_MAP)

Le découpage est réalisé avec le module *i.cutlines*. Pour assurer que les sous-tuiles créées par ce module s'alignent bien sur les sous-tuiles déjà définie par les polygones existants, il permet de définir des lignes préexistantes dont il faut tenir compte (sous forme de lignes ou de polygones). Nous avons combiné (en dehors des scripts Python) les tuiles existantes avec les axes de routes du PICC avec pour définir des limites prioritaires pour les sous-tuiles (couche définie dans walous_obia_config.py avec la variable EXISTING_CUTLINE_MAP). On peut influencer la taille et la forme de tuiles avec les paramètres suivants (entre parenthèses les valeurs effectivement utilisées dans le script) *number_lines* (15),

no_edge_friction (20), *lane_border_multiplier* (500) et *min_tile_size* (15625 m^2= 500x500 pixels). Voir le manuel du module pour plus d'explications.

A la fin, le script combine les sous-tuiles préexistantes avec les sous-tuiles créées par *i.cutlines* pour avoir un fichier définissant les sous-tuiles à utiliser pour l'optimisation de la segmentation.

4.4 Segmentation

Fichiers : walous_obia_tiles_data_creation.py, walous_obia_functions.py, walous_obia_config.py)

La segmentation (découpage de l'image en objets) se fait sous-tuile par sous-tuile. Avant la segmentation proprement dite, nous procédons à un regroupement des pixels de la sous-tuile en superpixels (module i.superpixels.slic). Cela permet de réduire significativement le temps de segmentation. Pour ne pas perdre le détail spatial, ces superpixels ne doivent pas être trop grand, et la priorité est mise sur la nature spectralement homogène et pas sur la compacité des superpixels. Nous créons des superpixels plus grands (et un peu moins homogène) dans les sous-tuiles de forêt ou agricoles (choix défini selon l'origine de la sous-tuile : polygones prédéfinis ou résultats de *i.cutlines*).

La segmentation se fait ensuite par un module d'optimisation automatique de la segmentation (module *i.segment.uspo*). L'algorithme teste différentes valeurs pour le seuil de segmentation (*threshold*). Les gammes des valeurs testées et la taille minimale des segments varient selon le type d'environnement : dans les zones forestières et agricoles les segments recherchés seront plus grands, et une moindre attention est portée à l'homogénéité interne des segments, puisque les variations à l'intérieur d'une forêt ou un champ donné ne sont généralement pas pertinentes pour la classification de l'occupation du sol. Le contraire est vrai pour les autres types d'occupation du sol, notamment le bâti, pour lesquels de plus petits segments sont nécessaire et une attention plus grande à l'homogénéité interne des segments en vue du passage rapide d'une occupation à une autre. Autrement dit, il vaut mieux de plus petits segments en milieu bâti pour éviter de grouper ensemble dans un même segment des pixels d'une maison avec ceux d'un trottoir ou d'une terrasse, par exemple.

L'ensemble des segments des sous-tuile d'une tuile des orthophotos est alors regroupé pour aboutir à une couche raster de la tuile segmentée. Cette couche est exportée vers un fichier .tif pour utilisation future.

Le script contient également une option de calculer un deuxième niveau de segmentation avec des segments significativement plus large (variable

WITH_2_SEGMENTATION_LEVELS dans *walous_obia_config.py*). Nous n'avons pas utilisé cette option à cause du temps de calcul beaucoup plus long nécessaire.

4.5 Caractérisation des segments et sélection des segments d'entraînement

Fichiers : walous_obia_tiles_data_creation.py, walous_obia_functions.py, walous_obia_config.py)

Tous les segments résultant de la segmentation doivent être caractérisés avec des statistiques décrivant leur forme, leurs caractéristiques spectrales, leur hauteur, leur texture, etc. Avant de calculer les statistiques, nous créons d'abord les données auxiliaires : indice de végétation NDVI, indice d'eau NDWI, ainsi que des couches de texture à différentes échelles. La pertinence et l'utilité des différents indices de texture, ainsi que des tailles de fenêtres les plus efficaces, peuvent varier d'une image à l'autre et sont difficiles de déterminer de façon théorique. On peut calculer un grand nombre d'indices utilisant une large série de tailles de fenêtre, pour ensuite faire jouer la sélection de variables pour en extraire les plus utiles, mais une telle approche nécessite beaucoup de temps de calcul et n'est donc pas vraiment efficace dans le cadre de notre chaîne de traitement ici. Sachant que beaucoup d'indices de texture sont fortement corrélés entre eux, nous avons donc décider de travailler de façon empirique avec une sélection manuelle (visuelle) de l'indice et des tailles de fenêtre par approche empirique. Nous avons utilisé l'indice de texture « Inverse Difference Moment », sur des fenêtres de 11x11 pixels (donc un rayon de 1,25m autour du pixel) et 21x21 pixels (donc un rayon de 2,50 autour du pixel) et une distance de comparaison des pixels de 5 pixels. Voir le manuel du module *r.texture* pour plus de renseignements.

Pour accélérer les calculs nous utilisons une version parallélisée du calcul des indices et du calcul des textures : *r.mapcalc.tiled* et *r.texture.tiled.*

Les statistiques sont calculées grâce au module i.segment.stats. Ce module crée une version vectorielle des segments avec dans la table attributaire les différentes statistiques. L'ensemble des rasters supplémentaires dont il faut calculer les statistiques sont définis dans la variable RASTERS, remplie au fur et à mesure de la création de ces bandes dans *walous_obia_tiles_data_creation.py* et combinés à l'ensemble des bandes des orthophotos pour le calcul des statistiques (variable BANDS). Les statistiques à calculer par segments pour ces rasters, ainsi que concernant la forme sont définies respectivement dans RASTER_STATS et AREA_MEASURES dans *walous_obia_config.py*. Voir le manuel de *i.segment.stats* pour plus de renseignements. Nous calculons également pour chaque segment des informations concernant les voisins (option configurable en

incluant la lettre n das ISEGMENTSTATSFLAGS dans le fichier *walous_obia_config.py)*. Si l'option d'un deuxième niveau de segmentation a été activée, on peut aussi définir les options pour le calcul des statistiques de ce deuxième niveau.

Ensuite, pour pouvoir procéder à la classification supervisée, un set de données d'entraînement est indispensable et il faut donc sélectionner parmi l'ensemble des segments créés ceux qui se prêtent le mieux au rôle de segment d'entraînement. Cette sélection se base sur les bases de données vectorielles existantes en Région wallonne (PICC, SIGEC, Lifewatch, Masque forestier de l'Université de Gembloux, etc). Pour certains types d'occupation du sol les données du squelette vectoriel créé dans le WP2 du projet WALOUS ont été utilisées. Seule exception : les routes puisque les calculs sont significativement plus rapidement en travaillant avec les axes de routes qu'avec des routes en polygones. Pour chaque classe une couche vectorielle a été créée en amont des scripts Python. Elles sont définies dans des variables telles que BUILDINGSMAP, ROADSMAP, CONIFEROUSMAP, etc dans le fichier *walous_obia_config.py*. Voir les règles dans l'annexe 7.1 pour le détail sur les couches utilisées dans la sélection.

Pour chaque tuile, les segments sont sélectionnés selon leur superposition avec des données de différentes classes et ensuite soumis à un filtrage supplémentaire (essentiellement sur base de valeurs NDVI et de hauteur – voir annexe pour le détail) pour éliminer de faux positifs (segments identifiés comme appartenant à une classe, mais n'y appartenant visiblement pas - éventuellement puisqu'ils sont recouverts par une autre classe sur l'image) et des valeurs extrêmes. Les algorithmes de sélection doivent aussi tenir compte du décalage spatial entre photos et bases de données vectorielles, par exemple en acceptant comme bâtiment un segment dont la surface recouvre à 95 % la surface d'un bâtiment du PICC.

Pour chaque classe, une version ombre de cette classe est aussi identifiée, sur base de la moyenne et l'écart-type des valeurs de luminance de chaque bande des orthophotos : si un segment identifié comme segment d'entraînement a dans chaque bande une luminance plus basse que la moyenne moins un écart-type de l'ensemble des segments identifiés comme entraînement pour cette classe, alors on le déclare comme un entraînement pour les ombres spécifiques de cette classe.

A la fin, des segments pas encore identifiés comme d'entraînement, mais montrant des caractéristiques d'ombres sont identifiés comme exemples d'ombre générique. Après expérimentation, nous avons décidé de ne pas utiliser ces segments dans la classification, mais l'option existe dans le script et ces segments sont éliminés à l'étape suivante.

Le résultat de cette sélection de segments d'entraînement est inscrit dans la colonne attributaire dont le nom est défini par la variable TRAINING_COLUMN du vecteur des segments de la tuile. A la fin du processus de sélection l'ensemble des attributs est alors exporté vers un fichier .csv contenant l'identifiant des segments, les statistiques calculées et, le cas échéant, la classe identifiée pour les segments sélectionnés pour l'entraînement.

4.6 Stratification

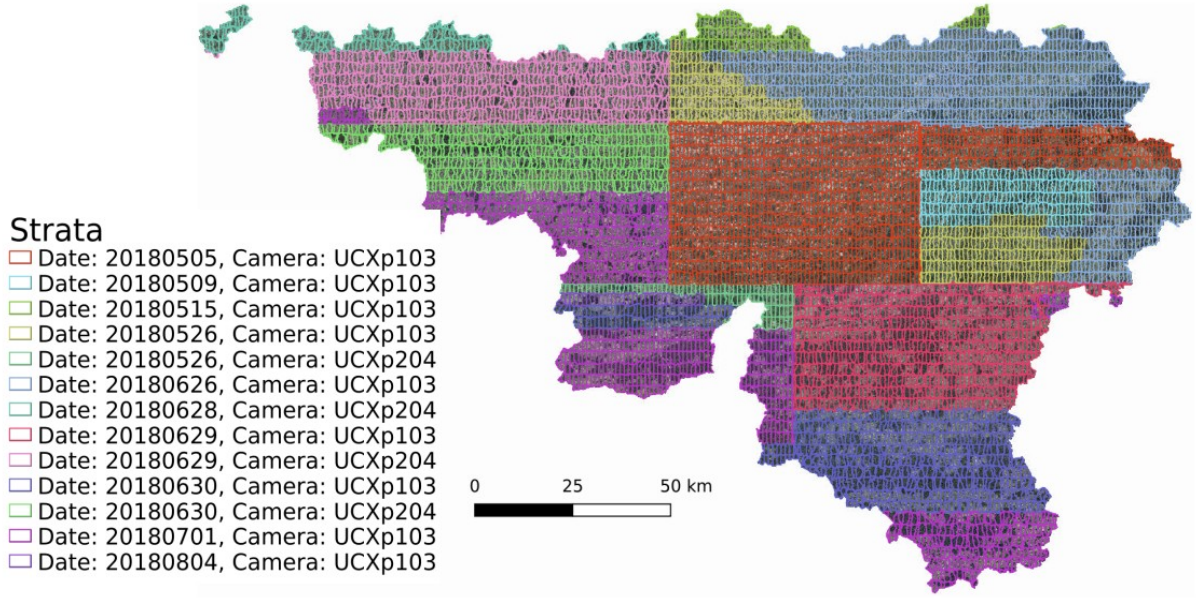
Fichier : walous_obia_trainingfile_creation.py

Les données d'entraînement de plusieurs tuiles avec des conditions de prise de vue similaires (ligne de vol le même jour à presque la même heure et en même direction) sont ensuite regroupées en strates. Pour la chaîne OBIA, cette stratification a été créée sur base de la date de la prise de vue, combinée au type de caméra utilisé (voir figure 2). L'appartenance de strate de chaque tuile a été incluse comme attribut supplémentaire dans la table attributaire de la couche vectorielle du tuilage (après prétraitement mentionné ci-dessus).

Les statistiques calculées pour les tuiles d'une strate donnée sont regroupées en un seul fichier, en retirant l'identifiant des segments (pas nécessaire pour l'entraînement du modèle et non-unique entre tuiles), et en éliminant les segments identifiés comme ombres génériques.

Puisque le nombre de segments d'entraînement peut être très important pour certaines classes, dépassant les capacités de calcul des algorithmes utilisés dans le logiciel R pour la classification, nous procédons à une sélection aléatoire par classe de segments d'entraînement. Dans le script, le nombre maximal de segments par classe est défini par la variable MAX_CLASS_SIZE qui a une valeur par défaut de 100000.

Figure 2: Strates utilisées pour la classification



4.7 Entraînement d'un modèle de classification par strate

Fichier : *walous_obia_modeltuning.pbs*

Pour chaque strate un modèle est entraîné utilisant le module *v.class.mlR* qui fait appel au logiciel R. Nous avons décidé d'entraîner un modèle de Random Forest qui nous paraissait le meilleur compromis en termes de qualité de la classification et temps de calcul, le faible nombre d'hyper-paramètre à définir, etc.[1] Un autre algorithme qui a montré de bons résultats ces dernières années est XGBoost[2], mais dans nos expériences avec les données et la chaîne WALOUS, le nombre d'hyper-paramètres plus élevés à déterminer menait à des temps de calcul significativement plus élevés non justifié par des gains en précision.

1 Belgiu, M. et Drăguţ, L. (2016), « Random forest in remote sensing: A review of applications and future directions », ISPRS Journal of Photogrammetry and Remote Sensing, Volume 114, Pages 24-31, https://doi.org/10.1016/j.isprsjprs.2016.01.011

2 Georganos, S. et al (2018), « Very High Resolution Object-Based Land Use-Land Cover Urban Classification Using Extreme Gradient Boosting », IEEE Geoscience and Remote Sensing Letters, vol. 15, no. 4, pp. 607-611, doi: 10.1109/LGRS.2018.2803259.

L'entraînement se fait strate par strate avec un simple appel au module en question avec les paramètres suivants :

- training_sample_size: le nombre maximal de segments par classe utilisé pour l'entraînement. Normalement ce paramètre peut prendre le nombre utilisé dans la création des fichiers d'entraînement par strate (100000), mais pour certaines strates un nombre plus faible (jusqu'à 50000) était nécessaire pour ne pas dépasser la capacité des algorithmes de R.
- tuning_sample_size : le nombre maximal de segments par classe utilisé pour le tuning du modèle (ici tuning pour le paramètre mtry qui détermine le nombre de variables échantillonnées à chaque étape de construction d'un arbre).
  Puisque cette étape peut prendre beaucoup de temps avec beaucoup de données, nous utilisons un faible échantillon par classe.
- max_features : le nombre maximal de variables à utiliser dans le modèle

Pour cette étape nous n'avons pas créé de script Python puisqu'il s'agit d'un seul appel à un module GRASS GIS que nous avons donc intégrer directement dans le fichier de lancement de tâches pour le cluster HPC. Une intégration dans un script Python est néanmoins possible, évidemment.

Les fichiers en sortie de l'entraînement comprennent : le modèle proprement dit (fichiers .rds), un fichier texte avec une information concernant la qualité du modèle (mesuré par validation croisée lors du tuning du modèle – la médiane de la précision globale variant entre 0,82 et 0,89), un fichier texte avec la liste des variables utilisées dans le modèle avec une indication de leur importance, le fichier script R soumis par le module à R (pour débogage éventuel).

4.8 Classification des tuiles

Fichier : walous_obia_classification.py

Une fois les modèles entraînés pour chaque strate, nous procédons à la classification proprement dite, tuile par tuile. Pour chaque tuile, le script identifie la strate à laquelle elle appartient, lit le modèle pour cette strate l'applique à la tuile sur base du fichier .csv avec les statistiques par segment et le fichier .tif avec les segments.

En sortie, le script crée un fichier .tif avec pour chaque pixel la valeur de la classe, ainsi qu'un fichier .csv avec pour chaque segment l'information de sa classe et les probabilités d'appartenance à chacune des classes.

5 Explication pratique de l'usage des scripts

Les scripts Python peuvent être utilisés dans n'importe quel environnement de calcul avec les outils nécessaires installés (Python, R, GRASS GIS – et leurs dépendances respectives). Ils ont été développés dans un environnement GNU/Linux, mais devrait être indépendants du système d'opération.

Les scripts ont été conçus avec comme input une base de données GRASS GIS contenant les orthophotos et le fichier vectoriel avec leur tuilage, le MNH, ainsi que l'ensemble des données auxiliaires nécessaires, notamment pour la sélection des segments d'entraînement. Pour augmenter la performance, il est conseillé de garder les orthophotos dans leur découpage par maillage et de les regrouper virtuellement, bande par bande, avec le module *r.buildvrt*. La même approche est recommandée pour le MNH (découpé par nous en 100 tuiles pour ne pas trop augmenter le nombre de fichiers, mais il est possible de travailler avec plus). La performance sera d'autant plus augmentée par cette approche si les données sont stockées sur un système de fichier parallélisé.

Il est possible de ne pas importer les données raster dans GRASS GIS, mais de lier (avec le module *r.external*) des fichiers extérieurs existants (ex : .tiff) dans une base de données pour un usage quasi équivalent d'un fichier importé. Nous n'avons pas testé cette option et ne savons pas si cette approche engendre une diminution de performance significative.

La définition des fichiers de lancement des tâches pour un système HPC dépend du système. Des exemples tournant sur le système de l'ULB (Hydra[3]) sont fournis. Selon les étapes les scripts font appels à différentes configurations :

- Segmentation et calculs des statistiques par segment : des nœuds de calculs avec 8 cœurs, 12 GB de RAM et 20 GB d'espace de stockage.
- Extraction des données pour création de jeux d'entraînement par strate : nœuds avec 1 cœur, 1 GB de RAM et 2 GB d'espace disque
- Entraînement des modèles par strate : nœuds avec 10 cœurs, 48 GB de RAM, 20GB d'espace disque

3 Les spécifications actuelles de Hydra sont disponible ici : https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/brussels/tier2_hardware/hydra_hardware.html. Les scripts utilisés faisaient appel à

- Classification de toutes les tuiles : nœuds avec 8 cœurs, 10 GB de RAM et 20 GB d'espace de stockage

Le système de gestion du HPC à l'ULB attribue des nœuds selon les disponibilités. Le nombre total de nœuds actifs en parallèle pouvait varier entre 30 et 100 selon les demandes par d'autres utilisateurs.

La logique générale est la suivante pour les étapes de segmentation et de classification (sur un système de cluster utilisant un système du style GNU/Linux):

- créer sur le nœud une base de données GRASS GIS avec un secteur (« location » en anglais) GRASS GIS dans le système de projection utilisée (ici le Lambert belge 1972, code EPSG 31370).
- intégrer dans ce secteur, par lien symbolique, les jeux de cartes (« mapsets » en anglais) de la base de données centrale (avec les orthophotos, MNH, etc)
- créer un fichier batch avec définition d'un accès lecture aux autres jeux de carte (appel au module *g.mapsets*) et l'appel du script python
- définir la variable d'environnement GRASS_BATCH_JOB avec le chemin vers ce ficher batch
- lancer GRASS GIS en créant un nouveau jeu de carte temporaire, utilisé uniquement pour le traitement
- nettoyer

Pour plus d'informations sur la terminologie de GRASS GIS, voire https://grass.osgeo.org/grass76/manuals/helptext.html (en anglais) https://portailsig.org/content/grass-gis-pas-pas-pour-les-debutants-1-demarrage-de-l-application-secteurs-locations-jeux-de.html (en français).

6 Temps de calcul

En dehors du temps de préparation (importation des données dans GRASS GIS, création du fichier de tuilage retravaillé, etc), voici les temps de calcul moyens nécessaires :

- segmentation, calcul des statistiques et identification des segments d'entraînement avec 8 processeurs disponibles par nœud : 1h15 / tuile (forte variation selon les tuiles : 45 minutes à 3h),
  → augmentation du temps sans parallélisation interne : ± 30-40 %

- regroupement des données d'entraînement : maximum 15 minutes / strate
- entraînement du modèle avec 10 processeurs par nœud : 2-4 heures / strate
  → augmentation du temps sans parallélisation interne : quasi x 10
- classification : 30 secondes / tuile

On voit donc que la parallélisation implémentée dans les modules à déjà un impact significatif sur les temps de calculs, mais l'ensemble des étapes peut-être de nouveau parallélisée à un niveau supérieur en lançant en parallèle les traitements tuile par tuile ou strate par strate sur différents nœuds de calcul. La diminution du temps total de calcul est alors directement proportionnelle au nombre de nœuds. Si on lance la partie segmentation + sélection de segments d'entraînement sur un seul nœud à 8 processeurs, on arrive donc à un total de 6372 tuiles x 75 minutes ~ 330 jours de calcul. En faisant tourner sur 30 nœuds de calcul en parallèle en permanence, on arrive à moins de 8 jours.

7 Annexes

7.1 Règles pour la sélection des segments d'entraînement

- Bâti (référence = squelette vectoriel du WP 2):
  - au moins 95 % du segment doit se trouver dans un polygone bâtiment du squelette vectoriel (pour tenir compte du décalage spatial entre les deux couches)
  - le NDVI moyen du segment doit être à moins d'un écart-type de la moyenne des NDVI moyens de tous les segments se superposant avec des polygones de bâti du PICC
  - l'écart-type du NDVI dans le segment doit être inférieur à l'écart-type moyen de tous les segments se superposant avec des polygones de bâti du PICC
  - la hauteur moyenne du segment doit être de plus de 3 m
- Sols imperméabilisés (routes – référence = axes des routes du PICC):
  - le NDVI du segment doit être inférieur à 0,1

- l'écart-type du NDVI dans le segment doit être inférieur à l'écart-type moyen de tous les segments croisés par un axe de route du PICC
- la moyenne de hauteur dans le segment doit être inférieur à 1,5 m

- Sols imperméabilisés (rail – référence = axes des rails du PICC) :
  - le NDVI du segment doit être inférieur à 0,1
  - l'écart-type du NDVI dans le segment doit être inférieur à l'écart-type moyen de tous les segments croisés par un axe de route du PICC
  - la moyenne de hauteur dans le segment doit être inférieur à 1,5 m
  - la taille du segment doit être plus petit que 250m^2 (pour éviter les segments qui combinent éléments du rail avec routes)
- Végétation basse (référence = SIGEC sans type 'autres') :
  - au moins 95 % du segment doit se trouver dans un polygone SIGEC (pour tenir compte du décalage spatial entre les deux couches)
  - le NDVI moyen du segment doit être supérieure à la moyenne des NDVI moyens des segments se superposant avec des polygones du SIGEC moins 2,5x la moyenne des écart-types du NDVI des mêmes segments
  - la hauteur moyenne du segment doit être inférieur à 1m
- Sol nu (référence = SIGEC type 'cultures') :
  - au moins 95 % du segment doit se trouver dans un polygone SIGEC (pour tenir compte du décalage spatial entre les deux couches)
  - le NDVI moyen du segment doit être inférieure à la moyenne des NDVI moyens des segments se superposant avec des polygones du SIGEC moins 2,5x la moyenne des écart-types du NDVI des mêmes segments
  - la hauteur moyenne du segment doit être inférieur à 1m
- Sol nu (référence = lignes carrières dans IGN VectorTopo10) :
  - le NDVI moyen du segment doit être inférieure à la moyenne des NDVI moyens des segments se superposant avec des lignes des carrières plus 0,5x la moyenne des écart-types du NDVI des mêmes segments

- le NDWI moyen du segment doit être inférieure à la moyenne des NDWI moyens des segments se superposant avec des lignes des carrières plus 0,5x la moyenne des écart-types du NDWI des mêmes segments
- la hauteur moyenne du segment doit être inférieur à 1m

- Conifères (référence = masque forestier conifères Gembloux)
  - la hauteur moyenne du segment doit être supérieur 3 m
  - le NDVI moyen du segment doit être supérieure à la moyenne des NDVI moyens des segments à l'intérieur avec des polygones de forêt conifères moins 2x la moyenne des écart-types du NDVI des mêmes segments
- Feuillus (référence = masque forestier feuillus Gembloux)
  - la hauteur moyenne du segment doit être supérieur 3 m
  - le NDVI moyen du segment doit être supérieure à la moyenne des NDVI moyens des segments à l'intérieur avec des polygones de forêt conifères moins 2x la moyenne des écart-types du NDVI des mêmes segments
- Eau (référence = classe eau de Lifewatch)
  - au moins 95 % du segment doit se trouver dans un polygone eau de Lifewatch
  - le NDWI moyen du segment est plus grand que la moyenne des NDWI moyens de tous les segments se superposant avec la couche de référence moins 1x la moyenne des écart-types du NDWI des mêmes segments
  - le NDVI moyen du segment est plus petit que la moyenne des NDVI moyens de tous les segments se superposant avec la couche de référence moins 1x la moyenne des écart-types du NDVI des mêmes segments
  - la valeur moyenne du segment dans le proche infrarouge est plus petite que la moyenne des valeurs moyennes de tous les segments se superposant avec la couche de référence moins 1x la moyenne des écart-types de la valeur dans le proche infrarouge des mêmes segments
  - la hauteur moyenne du segment doit être inférieure à 1 m

- Ombres génériques
    - la valeur moyenne du segment dans chacune des 4 bandes est inférieure à la moyenne des valeurs moyennes dans la même bande respective de l'ensemble des segments moins 1x la moyenne des écarts-type de la même bande respective dans les mêmes segments

7.2 Versions des logiciels utilisés pour les analyses

La liste ci-dessous donne le détail de version des différents logiciels et librairies, ainsi que la version de l'outil de compilation, utilisés sur le système de calcul de l'ULB au moment de la création de la carte LC par OBIA. De nouvelles versions sortent régulièrement de la plupart de ces outils et il faut donc s'assurer de la bonne combinaison au moment de l'usage de la chaîne de traitement. Il suffit de suivre les instructions sur les sites respectifs des trois outils principaux (GRASS GIS, Python, R) pour obtenir les bonnes combinaisons.

1) GCCcore/7.3.0

2) binutils/2.30-GCCcore-7.3.0

3) GCC/7.3.0-2.30

4) zlib/1.2.11-GCCcore-7.3.0

5) numactl/2.0.11-GCCcore-7.3.0

6) hwloc/1.11.10-GCCcore-7.3.0

7) OpenMPI/3.1.1-GCC-7.3.0-2.30

8) OpenBLAS/0.3.1-GCC-7.3.0-2.30

9) gompi/2018b

10) FFTW/3.3.8-gompi-2018b

11) ScaLAPACK/2.0.2-gompi-2018b-OpenBLAS-0.3.1

12) foss/2018b

13) bzip2/1.0.6-GCCcore-7.3.0

14) XZ/5.2.4-GCCcore-7.3.0

15) libxml2/2.9.8-GCCcore-7.3.0

16) ncurses/6.1-GCCcore-7.3.0

17) gettext/0.19.8.1-GCCcore-7.3.0

18) libreadline/7.0-GCCcore-7.3.0

19) Tcl/8.6.8-GCCcore-7.3.0
20) SQLite/3.24.0-GCCcore-7.3.0
21) GMP/6.1.2-GCCcore-7.3.0
22) libffi/3.2.1-GCCcore-7.3.0
23) Python/2.7.15-foss-2018b
24) libpng/1.6.34-GCCcore-7.3.0
25) Szip/2.1.1-GCCcore-7.3.0
26) HDF5/1.10.2-foss-2018b
27) cURL/7.60.0-GCCcore-7.3.0
28) netCDF/4.6.1-foss-2018b
29) expat/2.2.5-GCCcore-7.3.0
30) GEOS/3.6.2-foss-2018b-Python-2.7.15
31) NASM/2.13.03-GCCcore-7.3.0
32) libjpeg-turbo/2.0.0-GCCcore-7.3.0
33) JasPer/2.0.14-GCCcore-7.3.0
34) LibTIFF/4.0.9-GCCcore-7.3.0
35) PCRE/8.41-GCCcore-7.3.0
36) PROJ/5.0.0-foss-2018b
37) libgeotiff/1.4.2-foss-2018b
38) GDAL/2.2.3-foss-2018b-Python-2.7.15
39) FreeXL/1.0.5-GCCcore-7.3.0
40) libspatialite/4.3.0a-foss-2018b
41) freetype/2.9.1-GCCcore-7.3.0
42) x264/20181203-GCCcore-7.3.0
43) LAME/3.100-GCCcore-7.3.0
44) x265/2.9-GCCcore-7.3.0
45) util-linux/2.32-GCCcore-7.3.0
46) fontconfig/2.13.0-GCCcore-7.3.0

47) X11/20180604-GCCcore-7.3.0

48) FriBidi/1.0.5-GCCcore-7.3.0

49) FFmpeg/4.1-foss-2018b

50) pixman/0.34.0-GCCcore-7.3.0

51) GLib/2.54.3-GCCcore-7.3.0

52) cairo/1.14.12-GCCcore-7.3.0

53) nettle/3.4-foss-2018b

54) libdrm/2.4.92-GCCcore-7.3.0

55) LLVM/6.0.0-GCCcore-7.3.0

56) Mesa/18.1.1-foss-2018b

57) libGLU/9.0.0-foss-2018b

58) ATK/2.28.1-foss-2018b

59) Gdk-Pixbuf/2.36.12-foss-2018b

60) HarfBuzz/2.2.0-foss-2018b

61) Pango/1.42.4-foss-2018b

62) GTK+/2.24.32-foss-2018b

63) DBus/1.13.6-GCCcore-7.3.0

64) dbus-glib/0.110-GCCcore-7.3.0

65) GObject-Introspection/1.54.1-foss-2018b-Python-2.7.15

66) Perl/5.28.0-GCCcore-7.3.0

67) XML-Parser/2.44_01-GCCcore-7.3.0-Perl-5.28.0

68) intltool/0.51.0-GCCcore-7.3.0-Perl-5.28.0

69) GConf/3.2.6-foss-2018b

70) GStreamer/0.10.36-foss-2018b

71) GST-plugins-base/0.10.36-foss-2018b

72) wxPython/3.0.2.0-foss-2018b-Python-2.7.15

73) gzip/1.9-GCCcore-7.3.0

74) lz4/1.9.0-GCCcore-7.3.0

75) zstd/1.4.0-foss-2018b

76) GRASS/7.6.0-foss-2018b-Python-2.7.15

77) Java/1.8.0_181

78) Tk/8.6.8-GCCcore-7.3.0

79) NLopt/2.4.2-GCCcore-7.3.0

80) libsndfile/1.0.28-GCCcore-7.3.0

81) ICU/61.1-GCCcore-7.3.0

82) UDUNITS/2.2.26-foss-2018b

83) GSL/2.5-GCC-7.3.0-2.30

84) R/3.5.1-foss-2018b-Python-2.7.15