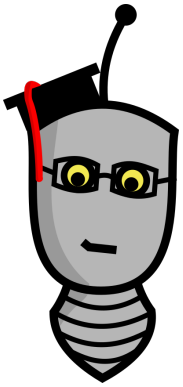
Projet d'Analyse statistique : *Forest Type Cover Prediction* (2014 Kaggle Challenge)

Mina Pêcheux (Monôme) - Polytech Sorbonne **Hiver 2018-2019**
Encadrée par : **Patrick Gallinari, Olivier Schwander, Arthur Pajot et Eloi Zablocki**



Résumé

Contexte
Depuis plusieurs années, le *machine learning* est en plein essor. De nouveaux algorithmes sont développés régulièrement et les chercheurs s'attaquent à toutes sortes de problèmes à l'aide de ce nouvel outil. D'abord théorisée dans les laboratoires, la science de l'apprentissage statistique s'est depuis propagée à l'industrie et, aujourd'hui, Google ou IBM sont de grands acteurs du domaine.

A l'heure actuelle, il semble que les applications possibles du *machine learning* soient presque sans fin : médecine, transport, analyse financière, étude de l'ADN, prédictions juridiques et judiciaires, jeux vidéos... Bien sûr, certaines utopies sont peut-être des buts difficiles à atteindre. Mais il est vrai que le *machine learning* est prometteur et qu'il a eu des succès dans de nombreuses tâches, comme par exemple la reconnaissance d'images, la génération de données artificielles mais réalistes ou bien la classification de données complexes.

A notre époque où la quantité de données ne cesse d'augmenter et que celles-ci déferlent de sources variées parfois difficiles à identifier, il est vital de comprendre comment ces algorithmes fonctionnent et "apprennent" à l'aide de cette matière de base. Si on a parfois l'impression que n'importe qui peut aller piocher dans ces données, les analyser et en tirer des conclusions, il ne faut pas oublier de prendre du recul et d'avoir un regard critique. Quand nous sommes confrontés à un jeu de données, il faut savoir demander aux experts d'apporter leur connaissance *a priori* du sujet, et il faut décortiquer les informations sans se laisser entraîner par les effets de mode et les intuitions hâtives.

Objectifs
Dans ce projet, nous allons appliquer des méthodes classiques d'analyse statistique et d'apprentissage supervisé pour classifier des types de zones de forêt. On souhaite ainsi, à l'aide d'un jeu de données relativement important, trouver le meilleur algorithme pour déterminer la catégorie de parcelles de forêts caractérisées par différentes variables cartographiques (son élévation, sa pente, sa distance au plus proche point d'eau...).

Le but est d'appréhender les concepts de base de l'apprentissage statistique, de découvrir comment charger et traiter correctement un jeu de données puis de mettre en pratique divers algorithmes de *machine learning* à l'aide d'outils usuels dans le domaine (**`NumPy`**, **`scikit-learn`**, etc.). Nous allons chercher à comparer des modèles habituels comme les KNN, les réseaux de neurones ou encore les arbres de décision.

Ce projet est inspiré du Kaggle challenge de 2014.

Table des matières

1 Description générale et préparation du jeu de données

1.1 Jeu de données et notations

Dans ce projet, on s'intéresse à la classification de différents types de forêts à partir d'observations faites sur le terrain. Il s'agit d'un problème de classification multi-classe avec 7 classes. Le jeu de données compte 581 012 échantillons (ou individus statistiques) pour lesquels on a les informations suivantes - nos 12 features[1] :

Variables quantitatives	Variables qualitatives
– **Elevation** : altitude	– **Wilderness** : catégorie de "wilderness"
– **Aspect** : orientation (en degrés)	– **Soil_Type** : type de sol
– **Slope** : pente (en degrés)	
– **HDist_Hydro** : distance horizontale au point d'eau le plus proche	
– **VDist_Hydro** : distance verticale au point d'eau le plus proche	
– **HDist_Roads** : distance horizontale à la route la plus proche	
– **HDist_Fire** : distance horizontale au départ de feu le plus proche	
– **Hillshade_9am** : ombrage à 9h au solstice d'été	
– **Hillshade_12am** : ombrage à 12h au solstice d'été	
– **Hillshade_3pm** : ombrage à 15h au solstice d'été	

Pour procéder à l'étude, on découpe ce jeu de données en données d'apprentissage et données de test avec un ratio de 80%/20%. On a donc finalement 464 810 échantillons d'apprentissage et 116 202 échantillons de test pris au hasard parmi nos données.

1.2 Objectifs, méthodes et outils principaux

Le projet a été codé en **Python**. Mon objectif a été d'appréhender le jeu de données, d'appliquer différentes techniques vues ou non en cours et de proposer des interfaces faciles d'utilisation pour :

- le chargement, le traitement et l'analyse des données
- l'utilisation des données nettoyées et traitées avec différents modèles usuels
- l'affichage de résultats sous une forme adaptée

Afin de gagner en efficacité et en praticité, les données sont manipulées avec des tableaux **NumPy** et des *dataframes* **Pandas**. Les modèles de *machine learning*, quant à eux, sont ceux implémentés dans la bibliothèque **scikit-learn**.

Par exemple, le code suivant permet de charger le jeu de données en ne conservant que les features numériques et en équilibrant le jeu de données, puis d'en extraire le jeu d'apprentissage et d'afficher diverses informations à son sujet en plus d'afficher ses histogrammes pour chaque feature, regroupés par classe :

```
from scripts.dataset import Dataset
from scripts.displayer import Displayer
from scripts.extractor import only_numerical as extr_only_numerical

# load dataset (and take train samples)
dataset = Dataset('covtype.data', autobalance='both', debug=True,
                  extractor={'func': extr_only_numerical})
df = dataset.train()
# display information and histogram
displayer = Displayer(df)
print(displayer.desc)
displayer.plot(['hist'])
```

Je propose donc dans ce petit projet deux classes utiles : **Dataset** et **Displayer** qui permettent d'aisément charger nos données et les analyser.

[1] On se trouve ici dans un cas académique où il n'y a pas de données manquantes.

1.3 Problématiques et prétraitements

1.3.1 Déséquilibre entre les variables quantitatives et qualitatives

Une remarque évidente sur ce jeu de données est qu'il présente une certaine inégalité entre le nombre de variables quantitatives et qualitatives (10 contre 2). Selon le modèle étudié, on peut donc être confronté à différents cas :

- nous pouvons exploiter les deux types de variables (réseaux de neurones, arbres de décision...)
- nous devons éliminer les variables quantitatives (régression logistique...)
- nous devons éliminer les variables qualitatives (k-Nearest Neighbors...)

Pour transformer nos variables quantitatives en variables qualitatives, on peut utiliser des seuils pour regrouper les échantillons en paquets (ou *bins*). En fonction du nombre d'intervalles choisis, on peut en apprendre plus sur notre jeu de données en dégageant des liens entre une classe et une plage de valeurs. Cependant, on perd de l'information puisque les détails de chaque individu disparaissent au profit de caractéristiques de groupes.

Etant donné le faible nombre de variables qualitatives et les résultats généralement peu probants quand on essaie de convertir une variable catégorielle en variable quantitative, j'ai préféré éliminer les deux colonnes `Wilderness` et `Soil_Type` lorsque le modèle n'acceptait pas de variable qualitative.

1.3.2 Chargement des données et récupération des modalités qualitatives

Dans les fichiers fournis, les valeurs des features qualitatives sont encodées sous le format *one-hot*, c'est-à-dire que, pour chaque variable qualitative, on dispose d'autant de colonnes que de modalités, et qu'un échantillon possédant la modalité i pour la variable aura un 1 dans la i-ème colonne et des 0 dans les autres.

Pour mieux communiquer avec les API des bibliothèques utilisées, on peut commencer par convertir ce format *one-hot* en indices entiers. Pour une variable avec n modalités, on a donc finalement une seule colonne dont la valeur est i pour notre échantillon (avec $1 \leq i \leq n$ entier)[2].

1.3.3 Déséquilibre important dans la répartition des individus

Un rapide examen des données nous permet d'identifier une répartition très inégale de nos individus entre les 7 classes. En effet, un regroupement par classes de notre jeu de données (données d'apprentissage et de test confondues) nous donne les pourcentages suivants :

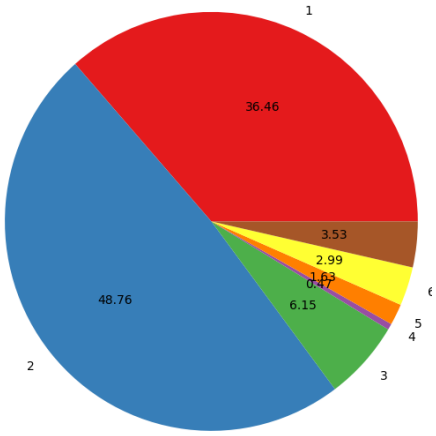


Figure 1 – *Répartition des échantillons par classe*

Classe	Pourcentage d'échantillons
1	36.5%
2	48.8%
3	6.2%
4	0.5%
5	1.6%
6	3.0%
7	4.0%

On observe donc une grande prépondérance des classes 1 et 2 dans notre jeu de données. Afin d'obtenir un apprentissage plus pertinent, pour certains modèles, il est utile d'équilibrer notre jeu, et on peut penser à trois idées pour effectuer ce prétraitement.

[2] A l'aide de bibliothèque comme **Pandas**, on peut ensuite aisément spécifier que ces colonnes sont de type qualitatif.

Idée n°1 : Sous-échantillonnage par rapport à la classe minoritaire
On peut envisager un premier prétraitement pour notre jeu de données où l'on ne garde qu'une partie des échantillons pour réduire les différences de fréquence entre les classes. Précisément, on va considérer une variable N_{lim} et traiter le jeu de sorte que :

- $N_{\text{lim}} = 2 \times N_{\text{min}}$ (avec N_{min} le nombre d'individus de la classe la plus petite)[3]
- pour chacune des classes : si la classe compte moins de N_{lim} individus, on garde l'ensemble des échantillons ; sinon, on en choisit N_{lim} aléatoirement

Mais le problème est que l'on ne conserve alors que 7% de notre jeu de données, et on en oublie 93% !

Idée n°2 : Sur-échantillonnage par *SMOTE (Synthetic Minority Over-sampling Technique)*
L'idée de la technique de *SMOTE* est de créer des points synthétiques, ou artificiels, à partir de nos données : à partir d'un point dans notre jeu de données initial, on peut considérer l'un de ses plus proches voisins[4] et créer un nouvel item entre les deux qui, a priori, doit être dans la même classe.
Le problème, c'est que cette fois on a généré 70% de notre jeu de données à partir de "rien"... encore une fois, on n'utilise finalement que peu de nos données initiales.

Idée n°3 : Un mélange des deux
Pour optimiser notre rééquilibrage, on peut essayer de combiner les deux techniques. On va procéder en deux étapes : un sous-échantillonnage moins violent que dans l'idée n°1, puis un sur-échantillonnage par *SMOTE* sur le nouveau jeu de données intermédiaire.
On choisit cette fois $N_{\text{lim}} = 0.35 \times N_{\text{max}}$ (avec N_{max} le nombre d'individus de la classe la plus grande) : la première étape de sous-échantillonnage élimine alors 50% de nos données (au lieu de 93%).
Ensuite, l'étape de sur-échantillonnage recrée 56% de notre jeu de données final à partir de ces données intermédiaires.
Au final, on obtient un jeu de données balancé avec 2% de données en plus.

1.3.4 Echelles de valeurs différentes

Enfin, nos données sont mesurées dans des échelles de valeurs très différentes (par exemple, l'élévation s'exprime en milliers de mètres alors que la pente est en degré, leurs valeurs sont donc situées dans des intervalles très différents).
Même si cela n'a pas nécessairement d'influence pour nos modèles, il ne faudra pas l'oublier si on souhaite faire du *features engineering*, et notamment si on envisage de créer une nouvelle *feature* à partir de deux variables dans des plages de valeurs éloignées.

2 Premières analyses et réflexion sur les modèles adaptés

2.1 Critères de validation

Pour évaluer l'efficacité des modèles, on peut envisager plusieurs critères : l'*accuracy*, la précision, le rappel... Ici, on a affaire à une classification à 7 classes.

J'ai décidé de mesurer, selon les scripts, l'*accuracy*[5] ou le score de *cross-validation* du modèle entraîné sur le jeu d'apprentissage pour ses prédictions sur le jeu de test.

2.2 Analyse statistique et choix des modèles de travail

A l'aide d'un **`boxplot`**, on peut étudier les moyennes et les variances de nos features pour chaque classe (voir la *Figure* 2).
On peut également construire la matrice de corrélation de notre jeu de données comme sur la *Figure* 3.

Cette première analyse (ainsi qu'un parcours du paragraphe "Relevant Information Paragraph" du fichier d'information fourni `covtype-info.txt`) nous permet d'identifier les *features* pertinentes pour notre classification et en particulier de repérer quelles variables pourraient être regroupées ou transformées lors de la phase de *features engineering*.

[3] On prend ce facteur 2 pour perdre un peu moins de données...
[4] Du point de vue des features et distances considérées.
[5] Je considère ici la mesure fournie par la méthode **`balanced_accuracy_score()`** de **`scikit-learn`** qui prend en compte les problèmes de déséquilibrage et gère les classifications multi-classes.

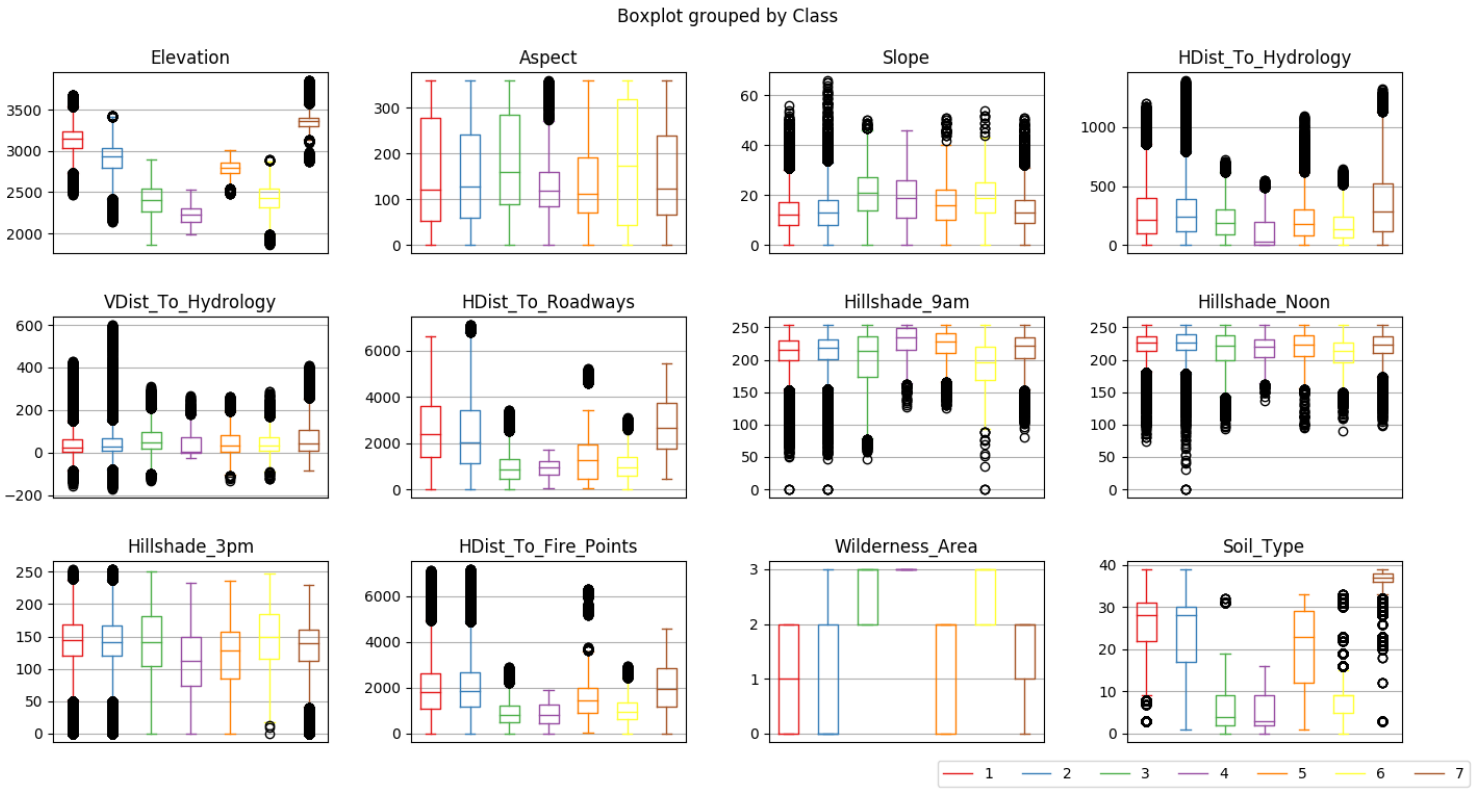


Figure 2 – *Analyse statistique de nos features par classe*

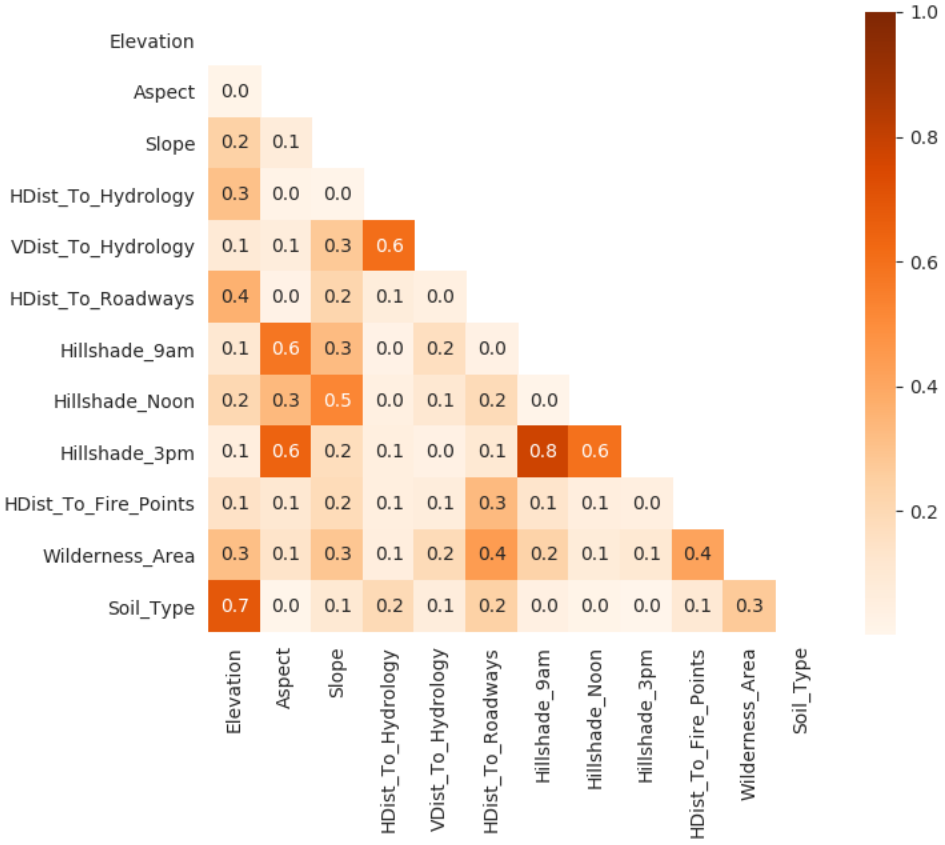


Figure 3 – *(Demi-)Matrice de corrélation de nos features*

On remarque notamment que :

- **`Elevation`**, **`HDist_Roads`** et **`Soil_Type`** semblent mieux discriminer nos échantillons que les autres
- **`VDist_Hydro`** possède également des valeurs négatives (jusqu'à -200) : même si cela a un sens physique, nos modèles risquent d'apprendre des relations d'écarts invalides
- **`HDist_Hydro`** et **`VDist_Hydro`** sont liées : elles pourraient être regroupées en une seule *feature* qui calcule la distance totale (i.e. : la norme euclidienne) au point d'eau le plus proche
- **`Hillshade_9am`** et **`Hillshade_3pm`** sont également corrélées : on pourrait les regrouper en créant une *feature* qui calcule leur différence (puisqu'elles évoluent de la même façon)
- **`Hillshade_9am`**, **`Hillshade_3pm`** et **`Aspect`** semblent liées
- **`Hillshade_9am`**, **`Hillshade_12am`** et **`Hillshade_3pm`** présentent ici beaucoup d'*outliers* : on peut envisager de les transformer en *features* binaires avec un seuil intelligent

J'ai également réalisé une Analyse en Composantes Principales (ACP) présentée en annexe afin d'en apprendre un peu plus sur nos données qui confirme certaines de ces intuitions.

2.3 K-Nearest Neighbors

On peut vérifier avec quelques tests rapides qu'une fois rééquilibré, notre jeu de données semble bien analysé par un modèle k-Nearest Neighbors (pour les paramètres de distance et nombre de voisins par défaut de **`scikit-learn`**) : on obtient une *accuracy* autour de 97%. Même si cet algorithme a le mérite d'être très simple, il est difficile à interpréter dans notre cas car, comme on dispose de beaucoup de *features*, on doit raisonner dans un espace à 12D pour comprendre la "proximité" entre deux individus. De plus, on ne tire absolument pas avantage de la connaissance *a priori* que l'on peut avoir sur nos données.

Cette méthode, même si elle donne des résultats intéressants, est donc laissée de côté car trop opaque pour notre étude.

2.4 Classifieur Bayésien

Un modèle simple et classique pour la classification multi-classe est le classifieur Bayésien. Contrairement à l'approche fréquentiste, un classifieur Bayésien part d'une estimation initiale de la pertinence de chaque variable et affine, à partir des données, l'importance de chaque variable dans la prédiction de la classe.

Ce type de classifieur a l'avantage de nécessiter peu d'informations sur les données (généralement seulement la moyenne et la variance des *features*) et de ne pas être sujet au sur-apprentissage.

Cependant, la classification Bayésienne suppose d'une part une connaissance *a priori* de l'influence des variables sur la classe et d'autre part une indépendance des variables. On remarque que, dans notre cas, on ne dispose pas de connaissance préalable et que beaucoup des *features* sont corrélées (voir la *Figure* 3). On peut donc supposer que ce modèle n'est pas adapté à notre jeu de données.

2.5 Réseau de neurones

Malgré l'engouement actuel pour les réseaux de neurones, des essais très rapides sur notre cas nous montrent que cet outil est peu adapté. En effet, bien que ce modèle puisse utiliser à la fois nos variables qualitatives et quantitatives, il se trouve ici face à trop de variables. Le déséquilibre du jeu de données empêche le réseau d'apprendre correctement les spécificités des classes rares et les poids finaux seront très difficiles à interpréter.

L'utilisation du **`MLPClassifier`** (*Multi-Layer Perceptron*) de la bibliothèque **`scikit-learn`** nous donne un résultat assez inutile puisqu'il est autour de 50% d'*accuracy*, soit la même prédiction qu'un modèle aléatoire ! De plus, le temps d'apprentissage très important d'un réseau de neurones complique l'étude et ralentit les tests.

2.6 Arbres de décision

Au vu du grand déséquilibre de nos jeux de données, et puisqu'il faut mélanger des variables qualitatives et quantitatives, nous pouvons nous attendre à de bons résultats avec des arbres de décision (*Decision Tree*) et des forêts aléatoires (*Random*

Forest). Nous nous concentrerons donc sur ces 2 modèles dans le dossier en comparant d'abord l'efficacité des modèles de **`scikit-learn`** avec les paramètres par défaut puis en optimisant les hyperparamètres de notre meilleur modèle.
Un autre avantage des *Decision Tree* et *Random Forest* est leur interprétabilité : en examinant l'arbre, on voit quels critères ont été sélectionnés pour réaliser la classification et lorsque l'on dispose de l'arbre de décision, pour classer un échantillon, il suffit de "descendre" le long des branches et de vérifier les seuils. On comprend donc immédiatement comment le modèle prédit une classe. En revanche, il n'est pas toujours aisé d'identifier comment ces seuils et critères ont été choisis...

Ces modèles sont également un bon compromis entre expressivité et temps d'apprentissage.

Remarque : On peut aussi noter que, dans l'écosystème des compétitions de *machine learning* (en particulier les compétitions Kaggle), les *Decision Tree* et les *Random Forest* sont connus pour avoir de très bons résultats et pour souvent gagner.

3 *Features engineering*

Remarque : Le script **`features_engineering.py`** permet de tester ces différentes idées de *features engineering*.

3.1 Combiner deux variables en une seule

On a vu dans notre analyse que les variables **`HDist_Hydro`** et **`VDist_Hydro`** sont corrélées, et que la variable **`VDist_Hydro`** présentent des valeurs négatives qui ne sont pas forcément pertinentes pour l'apprentissage. On peut imaginer de les regrouper en une seule variable (qui remplacerait les deux colonnes) : **`TotDist_Hydro`**. Celle-ci est définie comme la norme euclidienne de la forêt considérée à la surface d'eau la plus proche, autrement dit :

$$\texttt{TotDist\_Hydro} = \sqrt{\texttt{HDist\_Hydro}^2 + \texttt{VDist\_Hydro}^2}$$

3.2 Remplacer des variables

Les variables **`Hillshade_9am`** et **`Hillshade_3pm`** étant très corrélées (avec un coefficient d'environ 0.8), on peut envisager de les remplacer par une seule variable, leur différence : **`Hillshade_Diff`**. Celle est calculée simplement comme : $\texttt{Hillshade\_Diff} = |\texttt{Hillshade\_9am} - \texttt{Hillshade\_3pm}|$.

3.3 Rechercher des informations supplémentaires sur le *dataset*

En examinant le fichier d'information sur le jeu de données (`covtype-info.txt`), on apprend que les 40 types de sol (la *feature* **`Soil_Type`**) correspondent à des codes ELU[6] qui représentent la zone climatique et la zone géologique. On pourrait donc créer deux nouvelles variables **`Climate`** et **`Geology`** qui récupèrent le type de zone à partir du code ELU stocké dans **`Soil_Type`**.

3.4 "Binariser" des variables

Si on observe le boxplot des variables **`Hillshade_9am`**, **`Hillshade_12am`** et **`Hillshade_3pm`**, on remarque qu'elles peuvent être "binarisées" (autrement dit, transformées en variables qualitatives à 2 modalités) autour de seuils spécifiques. Cela simplifie à la fois l'apprentissage pour notre modèle et l'interprétation pour l'utilisateur.

3.5 Résultat des expérimentations

En regardant quelles modifications ont une influence, j'ai remarqué que la plupart des idées évoquées ci-dessus donnent des résultats aussi bons, voire moins bons, que les variables initiales. En particulier, l'utilisation des informations supplémentaires fournies par le code ELU n'améliore pas l'*accuracy* mais ralentit sensiblement le chargement des données.

J'ai donc décidé de simplement binariser **`Hillshade_9am`**, **`Hillshade_12am`** et **`Hillshade_3pm`** car la transformation de ces variables semble donner une petite hausse de l'*accuracy*.

[6] Le *United States Forest Service* (USFS) est une agence du département de l'Agriculture des États-Unis qui gère les forêts nationales du pays. Il donne entre autres des codes *Ecological Landtype Units* (ELU) à différentes zones pour regrouper à la fois le type de climat et le type de géologie.

4 Mise en place des modèles et optimisation des hyperparamètres

4.1 Premiers essais : comparaison *Decision Tree* / *Random Forest*

J'ai utilisé l'implémentation des *Decision Tree* et *Random Forest* disponible dans **`scikit-learn`**. Pour la *Random Forest*, pour commencer, j'ai demandé à **`scikit-learn`** d'utiliser **`n_estimators`** $= 20$ arbres. Les 2 modèles traitent intrinsèquement le problème de déséquilibrage de nos données, il suffit donc de les importer à l'aide de la classe **`Dataset`**, d'appliquer notre *features engineering*, puis d'utiliser les classifieurs de la bibliothèque :

```
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from scripts.dataset import Dataset

# load data
dataset = Dataset('covtype.data', debug=True)
# split train datasets into features and matching labels
train_data, train_labels = dataset.train(split=True)
# MODIFY feature: make "Hillshade_9am", "Hillshade_Noon" and "Hillshade_3pm" categorical
train_data['Hill_9bin'] = train_data['Hillshade_9am'] > 175
train_data['Hill_12bin'] = train_data['Hillshade_Noon'] > 200
train_data['Hill_3bin'] = train_data['Hillshade_3pm'] > 150
train_data.drop(columns=['Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm'], inplace=True)
train_data['Hill_9bin'] = train_data['Hill_9bin'].astype('category')
train_data['Hill_12bin'] = train_data['Hill_12bin'].astype('category')
train_data['Hill_3bin'] = train_data['Hill_3bin'].astype('category')
# compare the 2 models (with 5-fold cross-validation)
clf_dt = DecisionTreeClassifier()
clf_rf = RandomForestClassifier(n_estimators=20)
scores_dt = cross_val_score(clf_dt, train_data, train_labels, cv=5)
scores_rf = cross_val_score(clf_rf, train_data, train_labels, cv=5)
print('Mean Score of Decision Tree: {}'.format(np.mean(scores_dt)))
print('Mean Score of Random Forest: {}'.format(np.mean(scores_rf)))
```

Avec ces quelques lignes de code, on obtient déjà d'excellents résultats : on a un score de *cross-validation* de 93.5% pour le *Decision Tree* et de 95.9% pour la *Random Forest* !

Il est logique que la *Random Forest* nous donne un meilleur résultat puisqu'elle contient une phase de sélection des meilleurs arbres et optimise donc plus l'arbre de décision qu'un simple *Decision Tree*. Dans la suite, on va chercher à améliorer notre *Random Forest* en optimisant ses hyperparamètres.

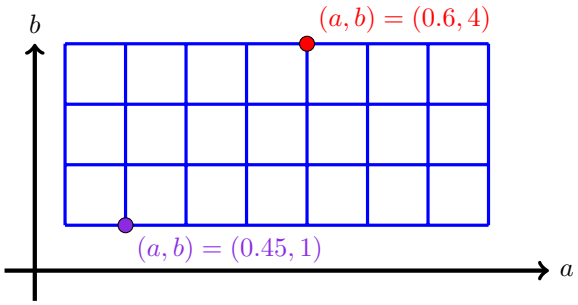
4.2 Ajustement des hyperparamètres

Pour une *Random Forest*, on peut faire varier un certain nombre d'hyperparamètres : le critère de séparation, le nombre minimal/maximal d'échantillons par noeud avant séparation, le nombre d'arbres à générer...

Pour la plupart de ces paramètres, les options par défaut de **`scikit-learn`** nous donnent de très bons résultats. Il est cependant intéressant de faire varier un peu les valeurs pour essayer de gagner encore en précisions sur nos prédictions.

Ce processus peut être automatisé par la technique de "*gridsearch*", ou "recherche par grille". L'idée est de considérer un ensemble de valeurs possibles pour k de nos hyperparamètres ($k \geq 1$) et de construire une grille de points dans un espace de dimension k qui représentent toutes les combinaisons possibles d'hyperparamètres à partir de ces valeurs.

Par exemple, sur un cas fictif de modèle acceptant deux hyperparamètres a et b, on peut construire la grille de recherche (donc pour $k = 2$) présentée *Figure* 4 qui propose 32 configurations différentes.



Evaluation par grille, pour :

- $a \in \{0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75\}$
- $b \in \{1, 2, 3, 4\}$

On travaille bien dans un espace 2D et on voit que la méthode nous propose toutes les combinaisons possibles de paramètres pour a et b.

Figure 4 – *Grille de recherche pour l'ajustement des hyperparamètres a et b d'un modèle fictif*

J'ai fait ici un *gridsearch* exhaustif avec *cross-validation*, c'est-à-dire que j'ai effectué un apprentissage avec validation croisée pour chacune des configurations d'hyperparamètres.

J'ai travaillé sur 3 hyperparamètres de ma *Random Forest* : **`n_estimators`** (le nombre d'arbres générés), **`criterion`** (le critère de sélection à la création d'un noeud) et **`max_features`** (le nombre de variables à considérer lorsque l'on sépare l'arbre). Les résultats (dont le rapport complet est présenté en annexe) nous montrent que la meilleure configuration pour maximiser le score de *cross-validation* est :

$$\texttt{n\_estimators} = 80, \qquad \texttt{criterion} = \text{entropy}, \qquad \texttt{max\_feature} = \text{None}$$

Autrement dit, on génère 80 arbres parmi lesquels on sélectionne le meilleur, la séparation au niveau d'un noeud se fait par le critère de l'entropie (qui minimise l'impureté des noeuds en essayant d'avoir une distribution uniforme de toutes les classes) et on choisit parmi toutes nos *features* pour nos séparations.

Avec cette optimisation des hyperparamètres, on obtient un score de 96.7% ($\pm$0.1%) sur une validation croisée à 5 *folds*. La *Figure* 6 en annexe montre un exemple de noeuds supérieurs d'un arbre obtenu au cours d'un des lancers.

Remarque :

- Même si cette méthode est simple et exhaustive, elle présente une complexité exponentielle et peut donc être relativement longue si l'on considère beaucoup de configurations. Il peut être parfois plus utile de procéder à une recherche aléatoire (**`RandomizedSearchCV`**) ou d'éliminer auparavant certaines configurations par sa connaissance des données.
- On peut aussi décider d'optimiser la recherche pour maximiser un critère de validation particulier ; ici, on a choisi le score de *cross-validation* mais pour d'autres problèmes on pourrait préférer minimiser les faux positifs, maximiser le *recall*...

5 Conclusions et perspectives

Pour ce projet, j'ai utilisé des méthodes de *machine learning* traditionnelles plutôt que des réseaux de neurones et du *deep learning* car ils étaient moins adaptés à nos données : en plus de nécessiter un nombre très important de paramètres pour un réseau, notre jeu de données présente un immense déséquilibre d'apparition des classes ce qui empêche un réseau d'apprendre correctement la classification des cas rares et induit des comportements erratiques.

Pour améliorer nos résultats, on peut envisager :

- de pousser les méthodes de ré-équilibrage pour prétraiter nos données et créer des individus dans les classes minoritaires (on a déjà vu que notre première implémentation basique fournit des résultats intéressants pour des modèles sensibles à ce problème comme le k-Nearest Neighbors)
- de créer de nouvelles *features* en obtenant une connaissance plus avancée des données
- de mettre en place des modèles plus expressifs et plus complexes (par exemple des réseaux de neurones) mais qui demandent une phase de paramétrisation plus importante

Ce projet m'a permis de mieux appréhender les techniques vues en cours et les outils usuels dans le domaine du *machine learning* (notamment **`scikit-learn`**). De plus, j'ai pu voir l'importance d'avoir de bonnes données de travail et de les traiter correctement avant d'y appliquer un modèle.

Annexes

Analyse en Composantes Principales (ACP)

Informations essentielles : le script **`data_analysis.py`** permet de facilement procéder à une ACP sur nos données qui donne les informations suivantes :

```
[PCA] Running a PCA analysis...
------------------------------
With parameters:

PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)

PCA extracted 10 components.

1. Explained variance ratio depending on the number of components:

# components  %age of variance explained
-----------------------------------------
           1                 25% (0.2586)
           2                 47% (0.4736)
           3                 64% (0.6475)
           4                 75% (0.7553)
           5                 83% (0.8323)
           6                 88% (0.8870)
           7                 93% (0.9334)
           8                 96% (0.9690)
           9                 99% (0.9997)
          10                100% (1.0000)

2. Broken-stick test
   Eigenvalues  Thresholds
0     2.586119    2.928968
1     2.149755    1.928968
2     1.739228    1.428968
3     1.077536    1.095635
4     0.770292    0.845635
5     0.546648    0.645635
6     0.464254    0.478968
7     0.356314    0.336111
8     0.306452    0.211111
9     0.003402    0.100000

10 most important individuals:
- with highest contribution to total inertia
- with best cos^2 contributions

Individual ID Contribution
---------------------------
       410149      190.979
       124042      187.104
       374406      185.587
       455364      183.284
        45861      179.651
        65746      177.688
       169568      167.529
       108126      157.646
        40562      152.981
       152805      150.560
```

```
Individual ID     cos^2_1      cos^2_2      cos^2_3
-------------------------------------------------------
      410149       0.264        0.298        0.019
      124042       0.132        0.534        0.025
      374406       0.264        0.282        0.028
      455364       0.111        0.540        0.022
       45861       0.300        0.247        0.030
       65746       0.320        0.241        0.014
      169568       0.067        0.577        0.021
      108126       0.015        0.637        0.016
       40562       0.073        0.587        0.034
      152805       0.326        0.172        0.020
```

On en déduit que les 3 premiers axes de l'ACP expliquent environ 65% des données mais qu'il faut en garder au moins 7 pour dépasser les 90%. De plus, on n'a pu analyser que les variables quantitatives. Il semble donc difficile de bien "résumer" notre jeu de données à l'aide de cette ACP.

Graphe de corrélation des variables : le script fournit aussi le graphe de corrélation des variables montré *Figure* 5 qui nous aide à évaluer l'importance de chaque *feature* dans l'ACP.

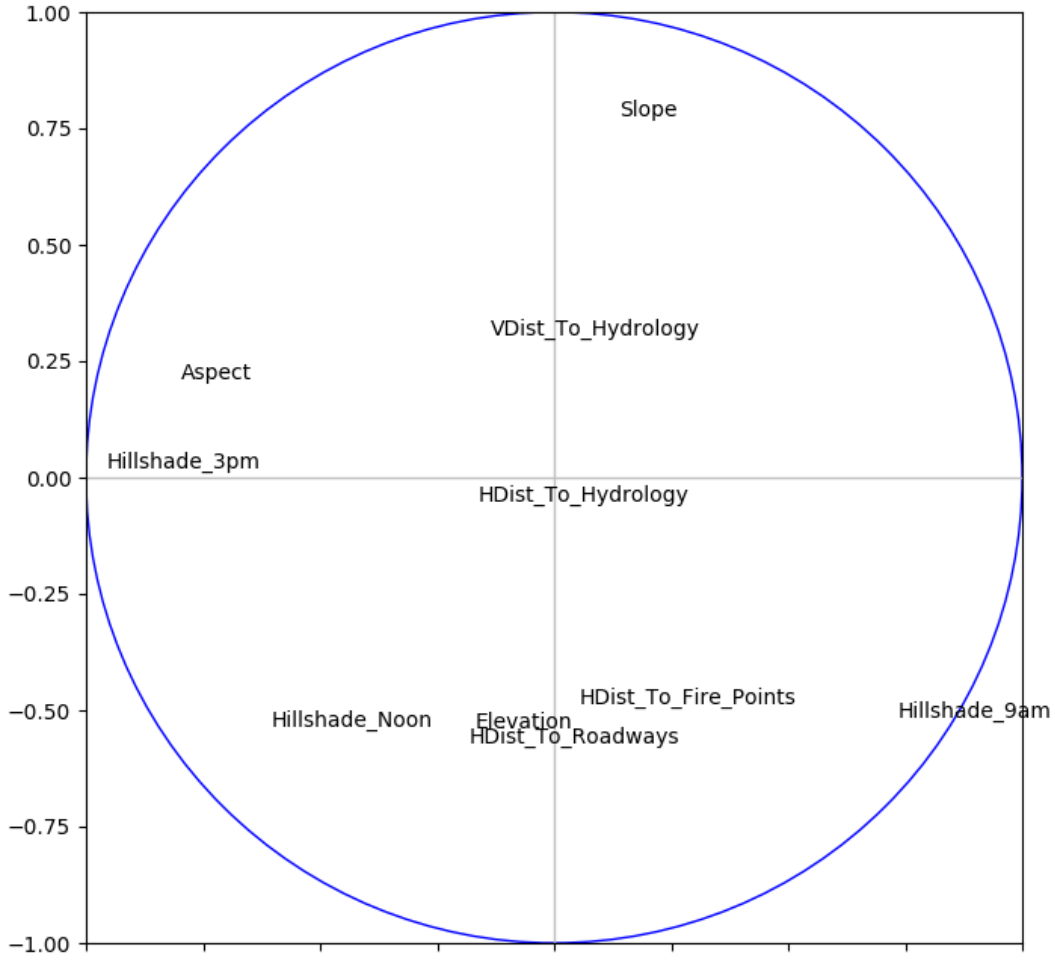


Figure 5 – *Graphe de corrélation des variables de l'ACP*

En particulier, les variables **Hillshade_9am**, **Hillshade_12am** et **Hillshade_3pm** se distinguent encore une fois alors que **HDist_Hydro** ou **VDist_Hydro**, sur l'axe central, sont difficilement interprétables. On retrouve ici la corrélation entre **Hillshade_3pm** et **Aspect**.

Exemple des premiers noeuds d'un arbre de décision (sur un lancer)

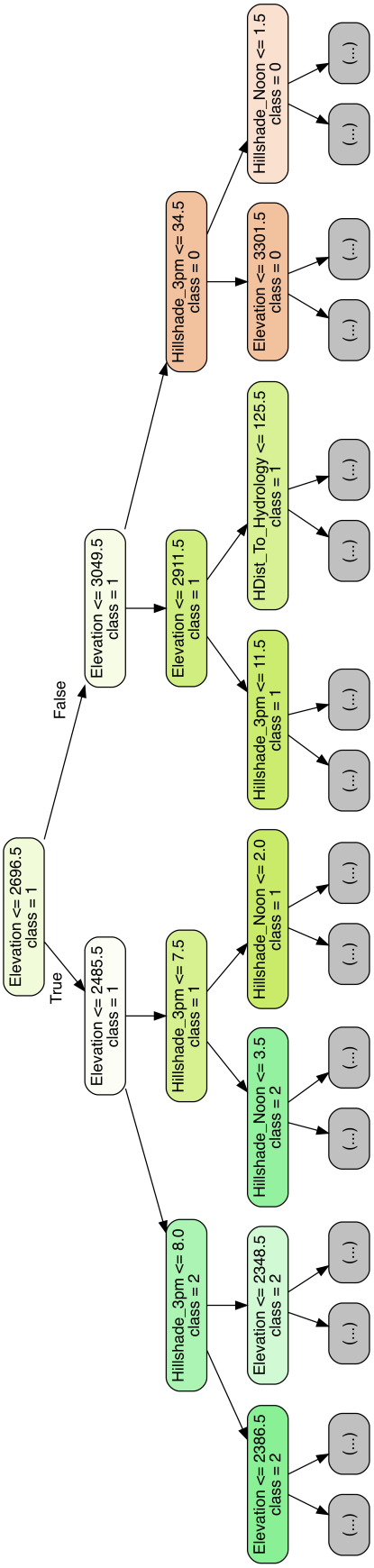


Figure 6 – *Exemple des premiers noeuds d'un arbre de décision (sur un lancer)*

Résultats complets du *gridsearch*

Le script pour l'évaluation *gridsearch* est adapté de celui disponible sur le site officiel de **scikit-learn**.

```
Best parameters for the development set:

{'criterion': 'entropy', 'max_features': None, 'n_estimators': 80}

Grid scores on the development set:

0.961 (+/-0.001) for {'criterion': 'gini', 'max_features': 'auto', 'n_estimators': 30}
0.962 (+/-0.001) for {'criterion': 'gini', 'max_features': 'auto', 'n_estimators': 50}
0.963 (+/-0.001) for {'criterion': 'gini', 'max_features': 'auto', 'n_estimators': 80}
0.964 (+/-0.001) for {'criterion': 'gini', 'max_features': None, 'n_estimators': 30}
0.965 (+/-0.001) for {'criterion': 'gini', 'max_features': None, 'n_estimators': 50}
0.965 (+/-0.001) for {'criterion': 'gini', 'max_features': None, 'n_estimators': 80}
0.961 (+/-0.001) for {'criterion': 'gini', 'max_features': 'log2', 'n_estimators': 30}
0.962 (+/-0.001) for {'criterion': 'gini', 'max_features': 'log2', 'n_estimators': 50}
0.962 (+/-0.001) for {'criterion': 'gini', 'max_features': 'log2', 'n_estimators': 80}
0.963 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'auto', 'n_estimators': 30}
0.964 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'auto', 'n_estimators': 50}
0.965 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'auto', 'n_estimators': 80}
0.965 (+/-0.001) for {'criterion': 'entropy', 'max_features': None, 'n_estimators': 30}
0.967 (+/-0.001) for {'criterion': 'entropy', 'max_features': None, 'n_estimators': 50}
0.967 (+/-0.001) for {'criterion': 'entropy', 'max_features': None, 'n_estimators': 80}
0.962 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'log2', 'n_estimators': 30}
0.964 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'log2', 'n_estimators': 50}
0.965 (+/-0.001) for {'criterion': 'entropy', 'max_features': 'log2', 'n_estimators': 80}

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           1       0.97      0.97      0.97     42288
           2       0.97      0.98      0.97     56845
           3       0.96      0.97      0.96      7124
           4       0.92      0.88      0.90       562
           5       0.92      0.88      0.90      1880
           6       0.95      0.94      0.94      3507
           7       0.97      0.97      0.97      3996

   micro avg       0.97      0.97      0.97    116202
   macro avg       0.95      0.94      0.95    116202
weighted avg       0.97      0.97      0.97    116202
```