Implémentation à partir de zéro de l'algorithme RandomForest

Julien Nique
Dan-Azoumi Lawan
Pap Diouf
Brice Derrien
Fatima ezzahraa BELMEKKI

26 octobre 2020

Dans ce projet, nous implémentons l'algorithme d'apprentissage supervisé Foret aléatoire dans le langage Python puis nous le testons sur des jeux de données et comparons les résultats avec d'autres modèles.

Contents

1 Introduction

Pour ce projet de Machine Learning nous avons décidé de travailler sur la méthode des forêts aléatoires (RandomForest). Ce modèle, combinant bagging et arbres de décision, a été proposé par Leo Breiman et Adèle Cutler en 2001.
Comme dit précédemment, cette méthode est une extension des arbres de décisions. L'intention de celle ci va être d'ajouter de l'aléatoire dans la production d'un arbre. L'idée est de tirer aléatoirement un sous-ensemble de variables qui sera candidat à la division en branches.
Notre groupe a décidé de travailler sur ce sujet car nous trouvions que c'était une méthode moderne et largement utilisée. Nous voulions donc l'étudier en profondeur. De ce fait nous avons implémenté cette méthode. Pour effectuer cette tâche notre choix s'est porté sur le langage Python. Il était pour nous le plus approprié et nous disposions de suffisamment de ressources sur internet pour nous aider.
Nous avons par la suite testé notre programme sur différent jeux de données. Pour finir, afin de prendre du recul sur notre travail nous avons comparé nos résultats avec ceux d'autres modèles existant.

2 Les arbres de décision et les forêts aléatoires

2.1 Les arbres de décision

Le principe d'arbre de décision est à la racine même du concept de forêt aléatoire. Nous allons donc brièvement rappeler cette méthode.
Tout d'abord un arbre de décision est une structure en forme d'arbre qui représente un ensemble de règles de décision. Les classes se trouvent en bout d'arbre et se nomment les feuilles. L'arbre est aussi composé d'une racine, de branches et de noeuds. La racine va être le point de départ (les X de notre échantillon), tandis que les noeuds seront quand à eux des sous-espaces de X, qui décriront un sous-ensemble de X, issus d'un test logique représenté sur une branche. Pour séparer l'arbre en plusieurs branches, on va regarder la qualité de cette séparation. Pour cela on va utiliser l'entropie de Shannon (on peut aussi utiliser l'indice de Gini). Sa formule est la suivante :

$$ent(p) = -p\log_2(p) - (1-p)\log_2(1-p)$$

. On divise donc avec la variable ayant la plus petite entropie. On effectue cette tâche à chaque nœud en cherchant localement la qualité d'une nouvelle division et ce jusqu'a ce que l'on obtienne une séparation pure.
Un des principaux problèmes des arbres de décision est sa variance. Et le moindre changement de données d'apprentissage peu changer complètement l'arbre. Pour pallier à ce problème on introduit donc les forêts aléatoires. Les ad sont fortement variables (changer les données d'entraînement peut changer drastiquement un ad) et donc le principe de "moyenner" plusieurs arbres sous-jacents au bagging permet de réduire la variance (et donc améliorer en principe l'erreur en généralisation) tout en maintenant un faible biais.

2.2 Les forets aléatoire

Le principe des forêts aléatoire repose sur les arbres de décision. Il est donc dans son ensemble comparable à ceux-ci. La variance des arbres de décision étant le principal problème, celui est réglé ici grâce au Bagging. On doit donc avoir plusieurs arbres de décision que l'on va moyenner avec le bagging. Cette étape va permettre de réduire la variance.
Pour former les différent arbres On va prendre au hasard un sous-ensemble de variables qui seront les candidats à la division.

3 Implémentation du modèle de classification RandomForest

Le langage de programmation choisi pour implémenter l'algorithme de classification RadomForest est Python. Sa popularité en Data Science, notamment grâce à des librairies comme Scikit-learn, n'est plus à démontrer. Nous n'utiliserons toutefois que les libraires pandas, pour la gestion des données, et numpy pour réaliser notre projet. Par définition, une forêt aléatoire est un ensemble d'arbres de décision (dans notre cas). Nous avons fait le choix d'implémenter chaque arbre de décision sous la forme d'un ensemble de noeuds imbriqués dans un noeud racine et ayant tous la classe **Node**.

3.1 La classe Node

La classe **Node**, dont le code suit, est instanciée à l'aide d'une structure de données **data** de type pandas dataframe dont la dernière colonne contient la variable qualitative à prédire.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.classeMaj = data.iloc[:,-1].value_counts().idxmax()
        self.var = (data.columns.delete(-1)).tolist()
        self.parent = None
        self.child = []
        self.split = []
        self.leaf = False

    def __str__(self):
        return str(self.data)
```

Chaque noeud créé par l'appel au constructeur Node(data) contient donc les attributs suivants :

1. data : les données du noeud qui ont servi à sa construction ;
2. classeMAj : la classe majoritaire de la variable cible ;
3. child : une liste contenant les éventuels noeuds descendants ;
4. parent : le noeud parent (sauf pour la racine) ;
5. var : une liste des noms des variables discriminantes susceptibles de diviser le noeud (utilisée pour créer une RandomForest) ;
6. split : une liste contenant le nom de la variable qui divise le noeud ainsi que le seuil pour une variable numérique ou les modalités pour une variable qualitative.
7. leaf : un booléen indiquant si le noeud est une feuille ou non.

Chaque arbre de classification est ensuite créé récursivement avec la fonction **GenerationArbre**, à partir du noeud initial qui sera la racine de l'arbre.

3.2 Fonction GenerationArbre

La fonction **GenerationArbre** prend deux arguments initiaux en entrée :

1. un **noeud** racine, instancié avec les données initiales ;
2. un hyperparamètre **seuil** égal à l'entropie minimale souhaitée pour effectuer une division.

Elle renvoie le **noeud** racine avec tous ses descendants, c'est à dire l'arbre de classification entraîné sur le jeu de données initiales contenu dans **noeud.data**. A chaque appel de la fonction **GenerationArbre**, on teste si le jeu de données du noeud courant peut être divisé grâce à la fonction **DivisionAttribut**. Le cas échéant :

1. des noeuds descendants sont créés à partir des sous-ensembles de données issus de la division et ajoutés à l'attribut **child** du noeud courant ;
2. leur attribut **parent** prend la valeur du noeud courant ;
3. l'attribut split du noeud courant est mis à jour ;

Sinon le noeud courant est considéré comme une feuille et son attribut **leaf** prend la valeur **True**.

3.3 Fonction DivisionAttribut

La fonction **DivisionAttribut** prend un argument le jeu de données d'un noeud de l'arbre qui n'a pas encore été divisé. Elle en détermine la meilleure division possible, c'est à dire celle qui minimise l'entropie, en veillant à ne pas dépasser le seuil du modèle. Elle retourne une liste à trois éléments :

1. j : le nom de la variable à utiliser pour la prochaine division ou une liste vide si aucune division n'est possible ;
2. so : un seuil numérique pour une variable quantitative, *nn* pour une variable non numérique ;
3. MinE : la valeur de l'entropie minimale possible.

3.4 Prédictions et premiers tests

La fonction **Prediction** prend en argument un arbre de classification entraîné à partir d'un noeud racine (contenant un jeu de données initiales *data_train*) et un jeu de données test X (*data_test*). Elle renvoie sous forme de liste les prédictions de classe pour chaque ligne de X. Le code suivant réalise un test sur le jeu de données **Iris** et montre les différents appels aux fonctions décrites précédemment.

```
"""Récupération des données et création du dataframe data"""
from sklearn.datasets import load_iris
iris = load_iris()
target = (iris.target).reshape(len(iris.target),1)
data = pd.DataFrame(np.concatenate((iris.data, target), axis = 1),
                    columns = ['sepl', 'sepw', 'petl', 'petw', 'spec'])

"""Création d'un jeu de données d'entrainement et d'un jeu de données pour le test"""
data_train, data_test = train_test_split(data, test_size=0.3)

"""Entraînement du modèle"""
Racine = Node(data_train)
arbre = GenerationArbre(Racine, 1)

"""Prédictions et score"""
ypred = Prediction(arbre,data_test)
score = sum(ypred == data_test['spec'])/len(ypred)
print('Le score est de :', score)
```

	Taille du data_set	Score moyen (N = 10)
1	10 %	0.953
2	20 %	0.95
3	30 %	0.951
4	40 %	0.928
5	50 %	0.929
6	60 %	0.929
7	70 %	0.929
8	80 %	0.900
9	90 %	0.85
10	95 %	0.64

Table 1: Scores moyen sur le jeu de données Iris

On constate que les résultats sur ce dataset sont excellents en généralisation.

4 RandomForest

4.1 Implémentation

Pour créer une forêt aléatoire, on adapte les algorithmes et codes précédents de façon à intégrer le bootstrap sur le jeu de données d'entraînement et la sélection aléatoire de variables à chaque division. Les fonctions **RFGenerationArbre** et **RFDivisionAttribut** prennent l'argument supplémentaire p qui donnent le nombre maximal de variables à utiliser à chaque nouvelle division de l'arbre. L'appel à la fonction **RandomForest(Noeud, seuil, n, p)** renvoie une liste de n arbres. Chaque arbre est initialisé à partir des données contenues dans le dataframe Noeud.data sur lesquelles on effectue un bootstrap, puis entraîné avec la fonction **RFGenerationArbre**.

4.2 Prédictions et premiers tests

La fonction **RFPrediction** prend en argument une forêt aléatoire entraînée à partir d'un noeud racine (contenant un jeu de données initiales *data_train*) et un jeu de données test X (*data_test*). Elle renvoie sous forme de liste les prédictions de classe, issues d'un vote majoritaire parmi les arbres de la forêt, pour chaque ligne de X. Le code suivant réalise un test sur le jeu de données **Titanic** et montre les différents appels aux fonctions décrites précédemment.

```
data = pd.read_csv("train.csv", sep=";")
data.dropna(how='all', inplace = True)
data = data[['Age','Pclass','Sex','Embarked','Survived']]
data_train, data_test = train_test_split(data, test_size=0.30)

noeud = Node(data_train)
forest = RandomForest(noeud, 1, 2, 2)

ypred = RFPrediction(forest, data_test)
score = sum(ypred == data_test['Survived'])/len(ypred)
print('Le score est de :', score)
```

	Taille du data_set	nombre d'arbres	hyperparamètre p	Score moyen (N = 10)
1	30 %	2	2	0.774
2	30 %	5	2	0.803
3	30 %	10	2	0.816
4	30 %	2	4	0.767
5	30 %	5	4	0.763
6	30 %	10	4	0.787

Table 2: Scores moyen sur le jeu de données Titanic

On constate que le score moyen augmente quand le nombre d'arbres de la forêt augmente et que l'hyperparamètre p améliore les performances pour une valeur égale à 2, soit la moitié du nombre total de variables explicatives.

4.3 Comparaison avec les modèles déjà implémentés

Dans un premier temps, nous testons la forêt avec un nombre d'arbres différents, le but étant de voir si ce facteur est important pour la prédiction de nos données. Ainsi sur cette dataset, on s'aperçoit que pour une foret aléatoire, plus le nombre d'arbres augmentent, plus le modèle est performant, autrement dit plus la foret est dense, meilleure est la prédiction. Comme l'illustre ce schéma. Par ailleurs, à partir d'un certain seuil, un nombre très important d'arbres ne joue aucun effet sur le modèle, et peut au contraire entraîner du sur-apprentissage.

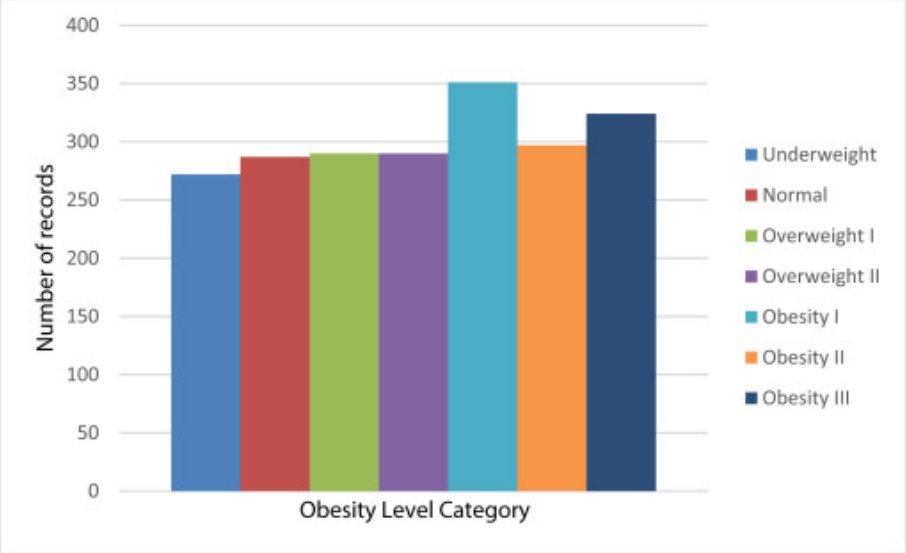
	Taille du data_set	nombre d'arbres	Score
1	10 %	2	0.911
2	20 %	3	0.911
3	30 %	5	0.911
4	40 %	10	0.955
5	50 %	20	0.955

Table 3: Scores sur le jeu de données Iris

4.4 Présentation de la dataset utilisée

Cet ensemble de données est une estimation des niveaux d'obésité chez les personnes des pays du Mexique, du Pérou et de la Colombie, en fonction de leurs habitudes alimentaires et de leur condition physique. Les données contiennent 17 variables explicatives et 2111 enregistrements, les enregistrements sont étiquetés avec la variable de classe NObesity (qui comprend 7 labels distincts) correspondant au niveau d'obésité d'une personne. Nous entraînerons nos modèles sur cette dataset pour une tâche de classification en utilisant les variables : poids insuffisant, poids normal, surpoids niveau I, surpoids niveau II, obésité type I , Obésité de type II et obésité de type III . 77% des données ont été générées de manière synthétique à l'aide de l'outil Weka et du filtre SMOTE, 23% des données ont été collectées directement auprès des utilisateurs via une plateforme web.

Une distribution de données suivant les 7 classes



4.5 Qui est le gagnant entre notre modèle et celui de Scikit-learn?

Nos métriques de comparaisons entre deux modèles seront la matrice de confusion et le score d'un classifieur. Nous comparerons notre Arbre avec notre forêt aléatoire que nous avons implémentée afin de montrer l'intérêt des forêts aléatoires, la deuxième traite d'une comparaison entre l'arbre de Sklearn, et celui que nous avons implémenté puis notre forêt aléatoire par rapport à celui implémenté dans Scikit-learn. Cette comparaison sur notre dataset, qui contient 2111 échantillons et 17 variables dont 16 sont

prédictives quantitatives et qualitatives,et une variable à prédire qui contient 6 labels, nous sommes alors dans un cas de classification multiclasse.

Taille du data_set	Labels
Obesity Type I	0
Obesity Type II	1
Obesity Type III	2
Overweight LevelI	3
Overweight LevelII	4
Insufficient Weight	5
Normal Weight	6

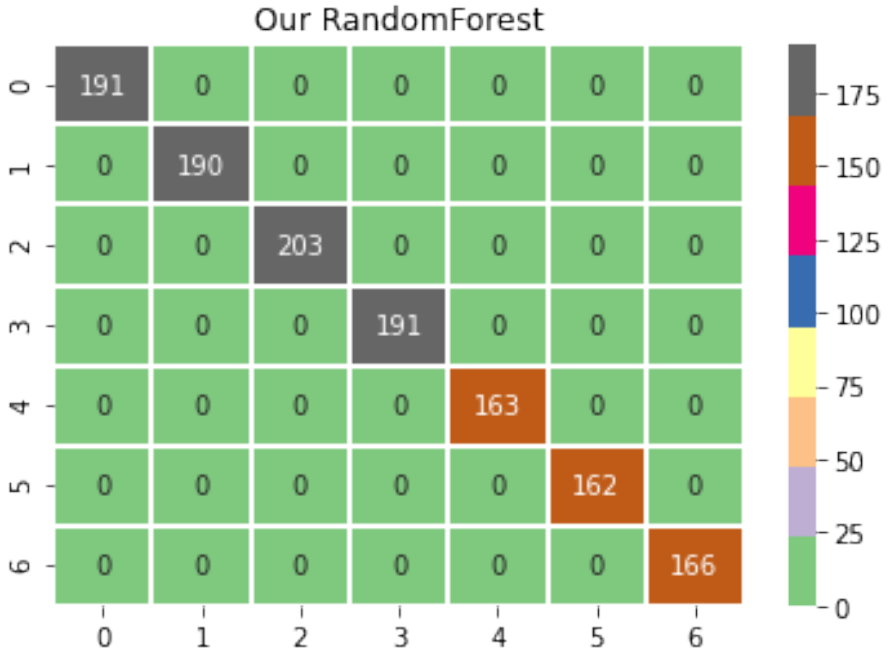
4.5.1 Notre Arbre vs notre Forêt aléatoire

Une illustration de la matrice de confusion obtenue par notre arbre

Nous pouvons clairement affirmer qu'il y a match nul entre l'arbre et la forêt que nous avons implémentée sur ce jeux de données; puisqu'on a un arbre qui fonctionne bien, on imagine que la forêt aléatoire devrait aussi bien fonctionner, puisque la forêt aléatoire se sert de plusieurs arbres pour faire sa prédiction.

Modèles	Données d'entraînement	Données de test	Score
Notre Arbre	1266	845	0.9925
Notre foret aléatoire	1266	845	1

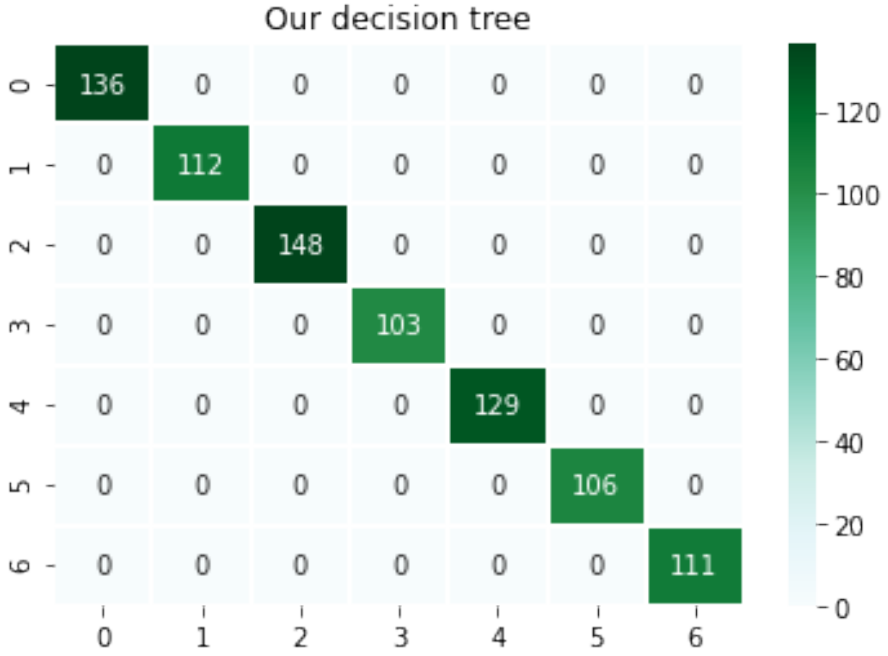
Une illustration de la matrice de confusion obtenue par notre Forêt aléatoire



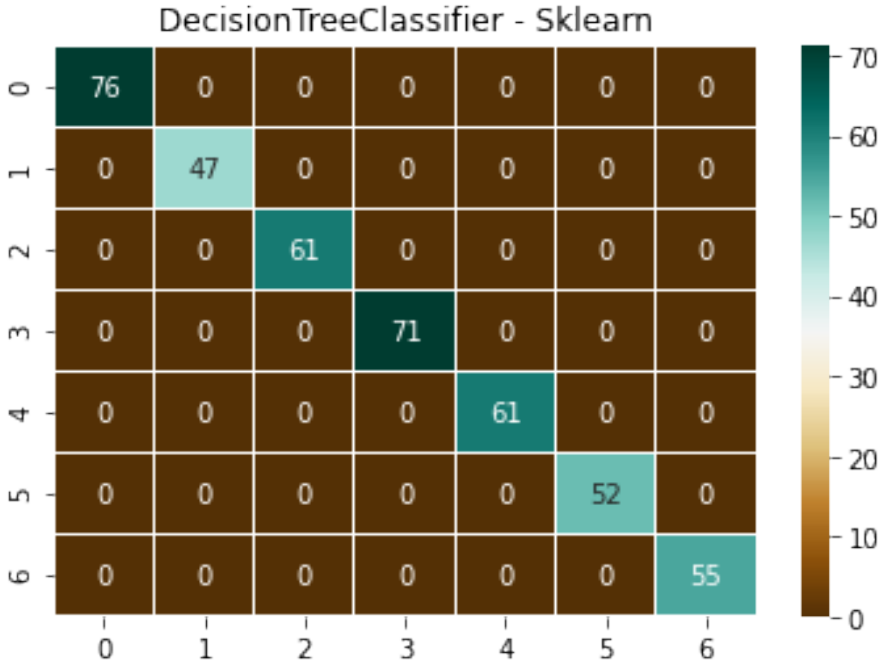
4.5.2 Arbre vs l'arbre implémenté sur Sklearn

Notre classifieur Arbre de décision ne se trompe pas sur le jeux de données test, et fait aussi mieux que celui implémenté par Scikit-learn. On a une belle matrice de confusion, avec des zéros partout et des valeurs sur la diagonale.

La matrice de confusion de notre modèle arbre de décision.



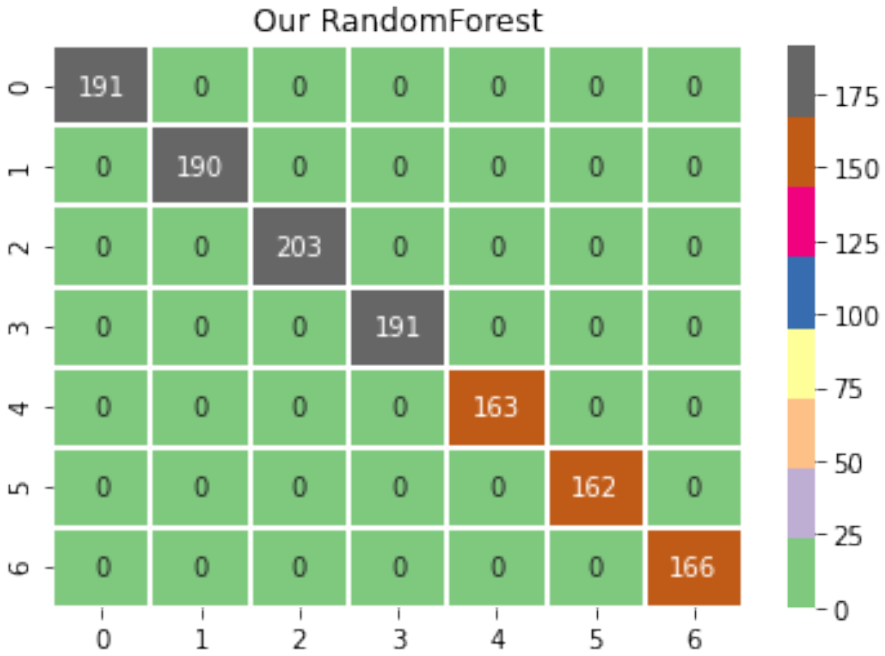
La matrice de confusion du modèle Sklearn.



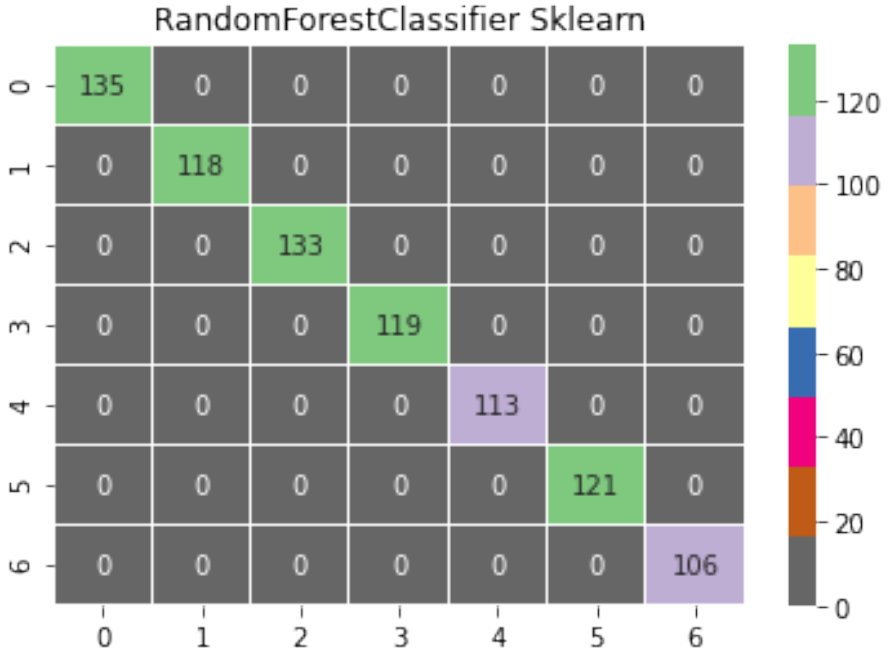
4.5.3 Forêt vs Forêt implémentée

Notre forêt rivalise bien avec celle de Scikik-learn, avec une matrice de confusion qui ne contient aucun faux positif et aucun faux négatif; notre modèle malgré le peu d'hyper-paramètres, fait aussi mieux.

La matrice de confusion de notre modèle foret aléatoire.



La matrice de confusion de la foret aléatoire de Sklearn.



5 Conclusion

L'objectif fixé dans notre projet était de faire une étude approfondie des arbres de décison et du RandomForest, ce qui nous a amené à créer des modèles prédictifs sans utiliser les bibliothèques déjà mises en place dans Scikit-learn.
L'étude a été enrichissante car malgré les difficultés rencontrées lors de la mise en place du code, cela nous a permis de mieux comprendre en détails le fonctionnement des arbres de décision et des forêts aléatoires.
En réponse du travail qui nous est demandé, nous avons pu mettre en place deux modèles d'arbre de décision et de RandomForest assez performants qui fournissent de bons scores avec les données Iris :

1. 85% avec le modèle arbre
2. 95.5% avec le modèle RandomForest

Cette différence de score est logique car le modèle RandomForest est une amélioration du modèle Arbre. Les forets aléatoires ont une grande capacité de généralisation et de stabilité dans les données tests, ce qui n'est pas le cas pour un arbre.
On a également testé nos modèles avec un autre dataset comprenant 2111 observations et 16 variables explicatives. Le résultat obtenu confirme davantage la performance de nos modèles :

1. 99.25% avec le modèle arbre
2. 100% avec le modèle RandomForest

En comparaison avec les modèles Arbre et RandomForest déjà implémentés dans Scikit-Learn, on obtient un score bien meilleur avec d'autres modèles créés from scratch.
Cependant, nous sommes conscients que notre modèle peut être amélioré davantage en utilisant plus d'hyper-paramètres afin de mieux l'optimiser, mais aussi de le tester sur un dataset beaucoup plus intéressante pour mieux évaluer sa généralisation.

Appendix A Code des fonctions

A.1 Fonction GenerationArbre

```
def GenerationArbre(Noeud, seuil):
    #on initialise les data avec celles du noeud passé en argument
    data = Noeud.data
    #on récupère la variable attr pour la prochaine division, le seuil so si cette variable est numé
    #et l'entropie MinE
    [attr, so, MinE] = DivisionAttribut2(data)
    #on teste que l'entropie est inférieure au seuil fixé pour l'arbre
    if(MinE <= seuil):
        #on met à jour les informations du node Noeud et on crée les noeuds descendants
        Noeud.split = [attr, so]
        #pour une variable numérique
        if  data[attr].dtypes == 'float64':
            noeud = Node(data.loc[data[attr] <= so])
            Noeud.child.append(noeud)
            noeud.parent = Noeud

            noeud = Node(data.loc[data[attr] > so])
            Noeud.child.append(noeud)
            noeud.parent = Noeud
        #pour une variable qualitative
        else:
            Noeud.split.pop(1)
            if len(np.unique(data[attr])) >= 2:
                #print('creation de child')
                for val in data[attr].value_counts().index:
                    Noeud.split.append(val)
                    noeud = Node(data.loc[data[attr] == val])
                    Noeud.child.append(noeud)
                    noeud.parent = Noeud

        #on appelle récursivement la fonction pour créer les noeuds suivants
        for child in Noeud.child:
            GenerationArbre(child, seuil)
    #si le noeud courant n'a pas de descendants, c'est une feuille par définition
    else:
        Noeud.leaf = True

    return Noeud
```

A.2 Fonction DivisionAttribut

```
def DivisionAttribut(data):
    global Einf, Esup, E
    target = data.columns[-1] #target contient la variable cible
    ncol = data.shape[1] #le nombre de colonnes dans le dataframe data
    j = [] ; MinE = 10 ; so = [] #initialisation des valeurs de sortie
    #on teste si il y a au moins deux modalités dans la target
    if len(np.unique(data.iloc[:,-1])) >= 2:
        #on parcourt toutes les variables descriptives
        for col_index in range(ncol-1):
          #on récupère les modalités/valeurs de la variable courante
          unique_values = np.unique(data.iloc[:,col_index])
          #on teste si la variable est numérique
          if data.iloc[:,col_index].dtypes == "float64":
            #on parcourt alors les valeurs par ordre croissant
            for i in range(1,len(unique_values)):
              #chaque seuil correspond à la moyenne de deux valeurs consécutives
              seuil = (unique_values[i] + unique_values[i-1])/2
              #on divise le dataframe data en deux et on calcule l'entropie pour la cible target
              dataSup = data[data.iloc[:,col_index]>seuil]
              targetSup = dataSup.iloc[:,-1]
              dataInf = data[data.iloc[:,col_index]<=seuil]
              targetInf = dataInf.iloc[:,-1]
              tinf = targetInf.value_counts()
              tsup = targetSup.value_counts()
              Ninf = np.sum(tinf) ; Nsup = np.sum(tsup) ; N = Ninf + Nsup
              Einf = 0
              for n in tinf:
                Einf = Einf - n/Ninf*np.log2(n/Ninf)
              Esup = 0
              for n in tsup:
                Esup = Esup - n/Nsup*np.log2(n/Nsup)
              E = Ninf/N*Einf + Nsup/N*Esup
              #on teste si l'entropie est meilleure que la précédente calculée
              if (E < MinE):
                #on met à jour la variable j qui contient le nom de la variable qui réalise le meill
                #la valeur de l'entropie correspondante et le seuil
                MinE = E ; j = data.columns[col_index] ; so = seuil


          else: #la variable est qualitative
            attr = data.columns[col_index]
            #on teste que la variable contient au moins deux modalités
            if (len(np.unique(data[attr])) >= 2):
              E = 0
              N = len(data[attr])
              for s in data[attr].value_counts().index:
                t = data.loc[data[attr] == s, target].value_counts()
                Nt = np.sum(t)
                for n in t:
                  E = E -Nt/N*(n/Nt*np.log2(n/Nt))
              if (E < MinE):
                MinE = E ; j = attr ; so = 'nn' #on met à jour la variable j

    return [j, so, MinE]
```

A.3 Fonction RFGenerationArbre

```
def RFGenerationArbre(Noeud, seuil, p):
    #on initialise les data avec celles du noeud passé en argument
    data = Noeud.data
    #on récupère la variable attr pour la prochaine division, le seuil so si cette variable est numé
    #et l'entropie MinE
    [attr, so, MinE] = RFDivisionAttribut(Noeud, p)
    #on teste que l'entropie est inférieure au seuil fixé pour l'arbre
    if(MinE <= seuil and MinE != 10):
        Noeud.split = [attr, so]
        if  (data[attr].dtypes == 'float64' or data[attr].dtypes == 'int64'):
            noeud = Node(data.loc[data[attr] <= so])
            Noeud.child.append(noeud)
            noeud.parent = Noeud

            noeud = Node(data.loc[data[attr] > so])
            Noeud.child.append(noeud)
            noeud.parent = Noeud
        else:
            Noeud.split.pop(1)
            Noeud.var.remove(attr)
            if len(np.unique(data[attr])) >= 2:
                for val in data[attr].value_counts().index:
                    Noeud.split.append(val)
                    noeud = Node(data.loc[data[attr] == val])
                    Noeud.child.append(noeud)
                    noeud.parent = Noeud

        for child in Noeud.child:
            RFGenerationArbre(child, seuil, p)
    else:
        Noeud.leaf = True

    return Noeud
```

A.4 Fonction RFDivisionAttribut

```
def RFDivisionAttribut(Noeud, p):
    global Einf, Esup, E, j
    data = Noeud.data
    target = data.columns[-1]
    j = []; MinE = 10 ; so = []
    if len(np.unique(data.iloc[:,-1])) >= 2:
        #on choisit aléatoirement p variables descriptives
        for attr in np.unique(random.choices(Noeud.var, k = p)):
            if (data[attr].dtypes == 'float64' or data[attr].dtypes == 'int64'):
                unique_values = np.unique(data[attr])
                for i in range(1,len(unique_values)):
                    s = (unique_values[i] + unique_values[i-1])/2
                    targetSup = data.loc[data[attr] > s, target]
                    targetInf = data.loc[data[attr] <= s, target]
                    tinf = targetInf.value_counts()
                    tsup = targetSup.value_counts()
                    #calcul de l'entropie associée au seuil s
                    Ninf = np.sum(tinf) ; Nsup = np.sum(tsup) ; N = Ninf + Nsup
                    if (Ninf != 0):
                        Einf = 0
                        for n in tinf:
                            Einf = Einf - n/Ninf*np.log2(n/Ninf)
                    if (Nsup != 0):
                        Esup = 0
                        for n in tsup:
                            Esup = Esup - n/Nsup*np.log2(n/Nsup)
                    E = Ninf/N*Einf + Nsup/N*Esup
                    if (E < MinE):
                        MinE = E ; j = attr ; so = s
            else:
                if (len(np.unique(data[attr])) >= 2):
                    E = 0
                    N = len(data[attr])
                    for s in data[attr].value_counts().index:
                        t = data.loc[data[attr] == s, target].value_counts()
                        Nt = np.sum(t)
                        for n in t:
                            E = E -Nt/N*(n/Nt*np.log2(n/Nt))
                    if (E < MinE):
                        MinE = E ; j = attr ; so = 'nn'

    return [j, so, MinE]
```

A.5 Fonction RandomForest

```
#En arguments de la fonction RandomForest, n est le nombre d'arbres,
#p est le nombre de variables choisies aléatoirement à chaque division.
def RandomForest(Noeud, seuil, n, p):
    forest = []
    #pour chaque arbre de la forêt
    for k in range(0,n):
        nrow = (Noeud.data).shape[0]
        #on effectue un bootstrap sur le jeu de données Noeud.data
        rows = random.choices(range(0,nrow), k = nrow)
        dataBS = Noeud.data.iloc[rows,:]
        #on crée la racine de l'arbre avec le bootstrap
        racine = Node(dataBS)
        #on entraîne l'arbre de classification
        arbre = RFGenerationArbre(racine, seuil, p)
        #on ajoute l'arbre à la forêt
        forest.append(arbre)
    return forest
```

References

https://towardsdatascience.com/understanding-decision-trees-once-and-for-all-2d891b1be579
https://towardsai.net/p/programming/decision-trees-explained-with-a-practical-example-fe47872d3b53
https://scikit-learn.org/stable/autoexamples/tree/plotunveiltreestructure.html
https://machinelearningmastery.com/cross-entropy-for-machine-learning/
https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e
https://medium.com/data-science-bootcamp/understand-cross-entropy-loss-in-minutes-9fb263caee9a
https://scikit-learn.org/0.15/modules/generated/sklearn.ensemble.RandomForestClassifier.html
https://www.datacamp.com/community/tutorials/random-forests-classifier-python
https://medium.com/machine-learning-101/chapter-5-random-forest-classifier-56dc7425c3e1
https://www.udemy.com/course/random-forest-algorithm-in-machine-learning/
https://www.udemy.com/tutorial/data-science-and-machine-learning-using-python-bootcamp-qazi/scikit-learn-decision-tree-and-random-forests-part-1/
https://www.udacity.com/course/classification-models–ud978
https://blog.udacity.com/2020/05/machine-learning-with-python-explained.html
https://www.coursera.org/lecture/python-machine-learning/random-forests-lF9QN
https://www.coursera.org/specializations/data-science-statistics-machine-learning
https://ui.adsabs.harvard.edu/abs/2012JPRS...67...93R/abstract
https://openclassrooms.com/fr/courses/4011851-initiez-vous-au-machine-learning
https://machinelearningmastery.com/implement-random-forest-scratch-python/
https://towardsdatascience.com/random-forests-and-decision-trees-from-scratch-in-python-3e4fa5ae4249
https://towardsdatascience.com/building-a-random-forest-classifier-c73a4cae6781
https://medium.com/@amaarora/implementing-random-forests-from-scratch-using-object-oriented-programming-in-python-in-5-simple-2f788afb9dbb
https://www.analyticsvidhya.com/blog/2018/12/building-a-random-forest-from-scratch-understanding-real-world-data-products-ml-for-programmers-part-3/
https://carbonati.github.io/posts/random-forests-from-scratch/
https://stackabuse.com/random-forest-algorithm-with-python-and-scikit-learn/
https://www.udemy.com/course/decision-tree-and-random-forest-python-from-zero-to-hero/ https://www.kdnuggets.com/2019/03/random-forest-python.html https://www.geeksforgeeks.org/random-forest-regression-in-python/
https://stackoverflow.com/questions/60208889/how-to-implement-random-forest-from-scratch-for-text-classification-using-fastte https://www.researchgate.net/post/Randomforestimplementation
https://onestopdataanalysis.com/python-random-forest-regression/ https://www.coursera.org/lecture/machine-learning-data-analysis/building-a-random-forest-with-python-eTO92 https://intellipaat.com/blog/what-is-random-forest-algorithm-in-python/