ROYAUME DU MAROC
UNIVERSITÉ ABDELMALEK ESSAÂDI
FACULTÉ DES SCIENCES ET TECHNIQUES
TANGER

Master Mobiquité et Big Data

Module: Data Mining & visualisation des données

Rapport du devoir 3:

Réalisé par :

OUAHABI ALHASSANI Bouchra

EL HANI Mariam

EL KRISSI Soukaina

Encadré par :

Pr. AIT KBIR Mohammed

Année universitaire: 2019/2020

Table de matière

Introduction

Machine Learning ou Apprentissage automatique n'est autre que la rencontre des statistiques avec la puissance de calcul disponible aujourd'hui (mémoire, processeurs, cartes graphiques).

Ce domaine a pris toute son importance en raison de la révolution digitale des entreprises qui a conduit à la production de données massives de différentes formes et types, à des rythmes sans cesse en augmentation : le Big Data.

Sur un plan purement mathématique la plupart des algorithmes utilisés ont déjà plusieurs dizaines d'années. Dans ce Devoir on va voir deux algorithmes d'apprentissage automatique KNN et Random forest

Objectifs

Dans ce travaux on s'intéresse de faire Une approche simple mais puissante pour faire des prédictions consiste à utiliser les exemples historiques les plus similaires aux nouvelles données. pour cela on va:

- Utiliser l'algorithme k-Nearest Neighbors, et aussi on va voir comment implémenter l'algorithme Random Forest à partir de zéro .
- La différence entre les arbres de décision ensachés et l'algorithme de forêt aléatoire.
- Comment construire des arbres de décision ensachés avec plus de variance.
- Comment appliquer l'algorithme de forêt aléatoire à un problème de modélisation prédictive.

Outils et langage de programmation

Jupyter

Jupyter est une application web utilisée pour programmer dans plus de 40 langages de programmation, dont: Python, Julia, Ruby, R, ou encore Scala2. Jupyter est une évolution du projet IPython. Jupyter permet de réaliser des calepins ou notebooks, c'est-à-dire des programmes contenant à la fois du texte en markdown et du code en Julia, Python, R... Ces notebooks sont utilisés en science des données pour explorer et analyser des données.

Python :

Python est un langage de programmation interprété, multi-paradigme et multiplateformes. Il favorise la impérative structurée, fonctionnelle et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk et Tcl.

Algorithme K-Nearest Neighbors:

l'algorithme K-NN (K-nearest neighbors) est une méthode d'apprentissage supervisé. Il peut être utilisé aussi bien pour la régression que pour la classification. Son fonctionnement peut être assimilé à l'analogie suivante "dis moi qui sont tes voisins, je te dirais qui tu es…".

Pour effectuer une prédiction, l'algorithme K-NN ne va pas calculer un modèle prédictif à partir d'un Training Set comme c'est le cas pour la régression logistique ou la régression linéaire. En effet, K-NN n'a pas besoin de construire un modèle prédictif. Ainsi, pour K-NN il n'existe pas de phase d'apprentissage proprement dite. C'est pour cela qu'on le catégorise parfois dans le Lazy Learning. Pour pouvoir effectuer une prédiction, K-NN se base sur le jeu de données pour produire un résultat.

Random Forest :

La forêt aléatoire est un algorithme d'apprentissage supervisé qui crée et fusionne au hasard plusieurs arbres de décision en une seule «forêt». L'objectif n'est pas de s'appuyer sur un seul modèle d'apprentissage, mais plutôt sur un ensemble de modèles de décision pour améliorer la précision. La principale différence entre cette approche et les algorithmes d'arbre de décision standard est que les nœuds racine comportent des nœuds de fractionnement sont générés de manière aléatoire.

Réalisation du devoir

Description de dataset

Dans notre devoir nous avons travaillé sur une dataset intitulé **la couverture terrestre** de la source Brian Johnson,johnson '@' iges.or.jp d'apres l'institute for Global Environmental Strategies, Japan

Cet ensemble de données Contient des données de formation et de test et a été dérivé de données géospatiales provenant de deux sources:

- l'imagerie satellitaire Landsat de séries chronologiques des années 2014-2015,
- des polygones géoréférencés en crowdsourcing avec des étiquettes de couverture terrestre

Les polygones externalisés ne couvrent qu'une petite partie de la zone de l'image et sont utilisés pour extraire des données d'apprentissage de l'image pour classer le reste de l'image.Le principal défi de l'ensemble de données est que l'imagerie et les données externalisées contiennent du bruit (en raison de la couverture nuageuse dans les images et d'un étiquetage / numérisation inexact des polygones).

- Nombre d'instances pour les bases de données training et testing nombre d'attributs pour les deux base de données

```
train = pd.read_csv('training.csv')
train.shape
```

```
(10545, 29)
```

```
test = pd.read_csv('testing2.csv')
test.shape
```

```
(299, 29)
```

- Enumération des colonnes

```
print(data.columns)
```

```
Index(['class', 'max_ndvi', '20150720_N', '20150602_N', '20150517_N',
       '20150501_N', '20150415_N', '20150330_N', '20150314_N', '20150226_N',
       '20150210_N', '20150125_N', '20150109_N', '20141117_N', '20141101_N',
       '20141016_N', '20140930_N', '20140813_N', '20140626_N', '20140610_N',
       '20140525_N', '20140509_N', '20140423_N', '20140407_N', '20140322_N',
       '20140218_N', '20140202_N', '20140117_N', '20140101_N'],
      dtype='object')
```

- Type de chaque colonne

```
print(data.dtypes)
```

```
class          object
max_ndvi      float64
20150720_N    float64
20150602_N    float64
20150517_N    float64
20150501_N    float64
20150415_N    float64
20150330_N    float64
20150314_N    float64
20150226_N    float64
20150210_N    float64
20150125_N    float64
20150109_N    float64
20141117_N    float64
20141101_N    float64
20141016_N    float64
20140930_N    float64
20140813_N    float64
20140626_N    float64
20140610_N    float64
20140525_N    float64
20140509_N    float64
20140423_N    float64
20140407_N    float64
20140322_N    float64
20140218_N    float64
20140202_N    float64
20140117_N    float64
20140101_N    float64
dtype: object
```

- description des données

  Certains indicateurs statistiques ne sont valables que pour les variables numériques (ex.moyenne,min, etc. pour max_indiv,...), et inversemment pour les non-numériques (ex. top, freq, etc. pour class), d'où les NaN dans certaines situations

```
print(data.describe(include='all'))
```

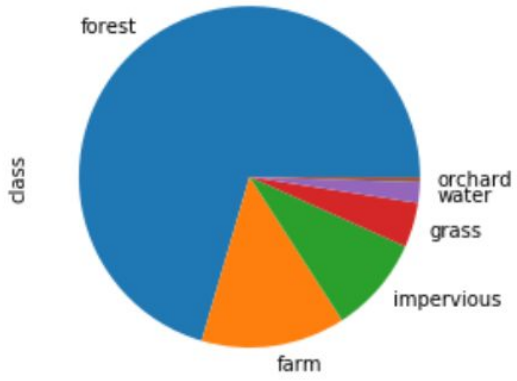
```
         class      max_ndvi    20150720_N    20150602_N    20150517_N  \
count    10545  10545.000000  10545.000000  10545.000000  10545.000000
unique       6           NaN           NaN           NaN           NaN
top     forest           NaN           NaN           NaN           NaN
freq      7431           NaN           NaN           NaN           NaN
mean       NaN   7282.721268   5713.832981   4777.434284   4352.914883
std        NaN   1603.782784   2283.945491   2735.244614   2870.619613
min        NaN    563.444000   -433.735000  -1781.790000  -2939.740000
25%        NaN   7285.310000   4027.570000   2060.600000   1446.940000
50%        NaN   7886.260000   6737.730000   5270.020000   4394.340000
75%        NaN   8121.780000   7589.020000   7484.110000   7317.950000
max        NaN   8650.500000   8377.720000   8566.420000   8650.500000

          20150501_N    20150415_N    20150330_N    20150314_N    20150226_N  \
count   10545.000000  10545.000000  10545.000000  10545.000000  10545.000000
unique           NaN           NaN           NaN           NaN           NaN
top              NaN           NaN           NaN           NaN           NaN
freq             NaN           NaN           NaN           NaN           NaN
mean     5077.372030   2871.423540   4898.348680   3338.303406   4902.600296
std      2512.162084   2675.074079   2578.318759   2421.309390   2691.397266
min     -3536.540000  -1815.630000  -5992.080000  -1677.600000  -2624.640000
25%      2984.370000    526.911000   2456.310000   1017.710000   2321.550000
50%      5584.070000   1584.970000   5638.400000   2872.980000   5672.730000
75%      7440.210000   5460.080000   7245.040000   5516.610000   7395.610000
max      8516.100000   8267.120000   8499.330000   8001.700000   8452.380000

       ...    20140610_N    20140525_N    20140509_N    20140423_N  \
count  ...  10545.000000  10545.000000  10545.000000  10545.000000
unique ...           NaN           NaN           NaN           NaN
top    ...           NaN           NaN           NaN           NaN
freq   ...           NaN           NaN           NaN           NaN
mean   ...   4787.492858   3640.367446   3027.313647   3022.054677
std    ...   2745.333581   2298.281052   2054.223951   2176.307289
min    ...  -3765.860000  -1043.160000  -4869.010000  -1505.780000
25%    ...   2003.930000   1392.390000   1405.020000   1010.180000
50%    ...   5266.930000   3596.680000   2671.400000   2619.180000
75%    ...   7549.430000   5817.750000   4174.010000   4837.610000
max    ...   8489.970000   7981.820000   8445.410000   7919.070000

          20140407_N    20140322_N    20140218_N    20140202_N    20140117_N  \
count   10545.000000  10545.000000  10545.000000  10545.000000  10545.000000
unique           NaN           NaN           NaN           NaN           NaN
top              NaN           NaN           NaN           NaN           NaN
freq             NaN           NaN           NaN           NaN           NaN
mean     2041.609136   2691.604363   2058.300423   6109.309315   2563.511596
std      2020.499263   2408.279935   2212.018257   1944.613487   2336.052498
min     -1445.370000  -4354.630000   -232.292000  -6807.550000  -2139.860000
25%       429.881000    766.451000    494.858000   5646.670000    689.922000
50%      1245.900000   1511.180000    931.713000   6862.060000   1506.570000
75%      3016.520000   4508.510000   2950.880000   7378.020000   4208.730000
max      8206.780000   8235.400000   8247.630000   8410.330000   8418.230000

          20140101_N
count   10545.000000
unique           NaN
top              NaN
freq             NaN
mean     2558.926018
std      2413.851082
min     -4145.250000
25%       685.680000
50%      1458.870000
75%      4112.550000
max      8502.020000

[11 rows x 29 columns]
```

- La classe est la variable de classification cible.

Les classes de couverture terrestre sont: water, forest, impervious, farm, grass, orchard

```
data['class'].value_counts().plot.pie()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xd51b990>
```




```
print(data['class'].value_counts())
```

```
forest        7431
farm          1441
impervious     969
grass          446
water          205
orchard         53
Name: class, dtype: int64
```

- Valeurs d'attribut manquante

notre dataset ne contient pas des valeurs qui sont nuls

```
missing_data=data.isnull()
```

```
missing_data
```

	class	max_ndvi	20150720_N	20150602_N	20150517_N	20150501_N	20150415_N	20150330_N	20150314_N	20150226_N	...	20140610_N	20140525_N
0	False	False	False	False	False	False	False	False	False	False	...	False	False
1	False	False	False	False	False	False	False	False	False	False	...	False	False
2	False	False	False	False	False	False	False	False	False	False	...	False	False
3	False	False	False	False	False	False	False	False	False	False	...	False	False
4	False	False	False	False	False	False	False	False	False	False	...	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...
10540	False	False	False	False	False	False	False	False	False	False	...	False	False
10541	False	False	False	False	False	False	False	False	False	False	...	False	False
10542	False	False	False	False	False	False	False	False	False	False	...	False	False
10543	False	False	False	False	False	False	False	False	False	False	...	False	False
10544	False	False	False	False	False	False	False	False	False	False	...	False	False

10545 rows × 29 columns

Implémentation des algorithmes

Implémentation de l'algorithme K-Nearest Neighbors avec python pure:

Dans cette partie nous allons découvrir l' **algorithme k-Nearest Neighbors,** y compris comment il fonctionne et comment l'implémenter à partir de zéro en Python (*sans bibliothèques*).

pour faire cela on va utiliser ces bibliothèques :

```
from random import seed
from random import randrange
from csv import reader
from math import sqrt
```

```
# chargement de fichier CSV
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row[::-1])
    return dataset
```

```
# Convertir colonne string en float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())
```

Nous pouvons mettre à jour la fonction *str_column_to_int ()* pour imprimer le mappage des noms de classe de chaîne en nombres entiers afin que nous puissions interpréter la prédiction faite par le modèle.

```python
# Convertir colonne string a int
def str_column_to_int(dataset,testData, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  for row in testData:
    row[column] = lookup[row[column]]
  return lookup
```

```python
#Calcule du pourcentage d'accuracy
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

```python
# Evaluation d'algorithm
def evaluate_algorithm(train_set,test_set, algorithm, *args):
  scores = list()
  # print("Train data size: ",len(train_set))
  # print("Test data size:", len(test_set))
  predicted = algorithm(train_set, test_set, *args)
  # actual = [row[-1] for row in fold]
  actual = [row[-1] for row in test_set]
  accuracy = accuracy_metric(actual, predicted)
  scores.append(accuracy)
  return accuracy
```

Étape 1: Calculer de la distance euclidienne

cette étape consiste à calculer la distance entre deux lignes d'un ensemble de données.

Les rangées de données sont principalement constituées de nombres et un moyen facile de calculer la distance entre deux rangées

pour faire ca on va utiliser la mesure de la distance euclidienne:

- Distance euclidienne = sqrt (somme i à N (x1_i - x2_i) ^ 2)

Où *x1* est la première ligne de données, *x2* est la deuxième ligne de données et *i* est l'index d'une colonne spécifique lorsque nous additionnons toutes les colonnes.

Avec la distance euclidienne, plus la valeur est petite, plus les deux enregistrements seront similaires. Une valeur de 0 signifie qu'il n'y a pas de différence entre deux enregistrements.

Ci-dessous se trouve une fonction nommée *euclidean_distance ()* qui implémente cela en Python.

```python
#Calcule de distance euclidienne entre deux vecteurs
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)
```

Etape 2: Obtention des voisins les plus proches

Les voisins d'une nouvelle donnée dans l'ensemble de données sont les *k* instances les plus proches, telles que définies par notre mesure de distance.

Pour localiser les voisins d'une nouvelle donnée dans un ensemble de données, nous devons d'abord calculer la distance entre chaque enregistrement de l'ensemble de données et la nouvelle donnée. Nous pouvons le faire en utilisant notre fonction de distance préparée ci-dessus.

Une fois les distances calculées, nous devons trier tous les enregistrements du jeu de données d'entraînement en fonction de leur distance aux nouvelles données. Nous pouvons alors sélectionner le top *k* pour revenir en tant que voisins les plus similaires.

Nous pouvons le faire en gardant une trace de la distance pour chaque enregistrement dans l'ensemble de données en tant que tuple, trier la liste des tuples par la distance (dans l'ordre décroissant), puis récupérer les voisins.

Ci-dessous se trouve une fonction nommée *get_neighbors ()* qui implémente cela

```python
#location de neighbors les plus similaires
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors
```

Étape 3: Faire des prédictions

Les voisins les plus similaires collectés à partir de l'ensemble de données d'apprentissage peuvent être utilisés pour faire des prédictions.

Dans le cas du classement, on peut renvoyer la classe la plus représentée parmi les voisins.

Nous pouvons y parvenir en exécutant la fonction *max ()* sur la liste des valeurs de sortie des voisins. Étant donné une liste de valeurs de classe observées dans les voisins, la fonction *max ()* prend un ensemble de valeurs de classe uniques et appelle le décompte sur la liste des valeurs de classe pour chaque valeur de classe dans l'ensemble.

Ci-dessous, la fonction nommée *predict_classification ()* qui implémente cela

```
#faire une prediction avec voisinages
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction
```

```
# kNN Algorithme
def k_nearest_neighbors(train, test, num_neighbors):
    predictions = list()
    for row in test:
        output = predict_classification(train, row, num_neighbors)
        predictions.append(output)
    return(predictions)
```

Nous pouvons voir que la précision est d'environ 63,66%

```
trainfilename = 'training.csv' #give train file name
testfilename = 'testing.csv'   #give test file names

train_set=load_csv(trainfilename)[1:]
for i in range(0, len(train_set[0])-1):
  str_column_to_float(train_set, i)

test_set=load_csv(testfilename)[1:]
for i in range(0, len(test_set[0])-1):
  str_column_to_float(test_set, i)

str_column_to_int(train_set,test_set, len(train_set[0])-1)
```

```
{'water': 0, 'impervious': 1, 'grass': 2, 'orchard': 3, 'forest': 4, 'farm': 5}
```

```
# evaluation de notre algorithm
num_neighbors = 9
accuracy = evaluate_algorithm(train_set,test_set, k_nearest_neighbors, num_neighbors)
print('Accuracy: %.3f%%' % (accuracy))
```

```
Accuracy: 63.667%
```

```
row = [7737.77,2275.24,7287.94,6767.88,6235.91,6106.91,4903.47,6543.03,7097.59,7103.79,7122.19,572.959,7737.77,6206.64,2529.75,3
```

```
label = predict_classification(train_set, row, num_neighbors)
print('Data=%s, Predicted: %s' % (row, label))
```

```
Data=[7737.77, 2275.24, 7287.94, 6767.88, 6235.91, 6106.91, 4903.47, 6543.03, 7097.59, 7103.79, 7122.19, 572.959, 7737.77, 620
6.64, 2529.75, 3926.86, 1476.92, 930.467, 6987.75, 5042.33, 2225.12, 683.258, 4078.95, 3916.13, 6543.53, 6466.97, 6902, 2770.
4], Predicted: 4
```

Implémentation de l'algorithme K-Nearest Neighbors avec sklearn:

+Importation de bibliothèque et entraînement:

Classificateur implémentant le vote des voisins k les plus proches.

Permet de démarrer l'algorithme avec k = 7 pour l'instant:

```
from sklearn.neighbors import KNeighborsClassifier
```

```
k = 7
#Train Model and Predict
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=7, p=2,
                     weights='uniform')
```

+Prediction:

Nous pouvons maintenant utiliser le modèle pour prédire l'ensemble de test:

```
yhat = neigh.predict(X_test)
yhat[0:5]
```

```
array([2, 2, 4, 2, 2], dtype=int64)
```

+Évaluation de la précision

Dans la classification multi-étiquettes, le score de classification d'exactitude est une fonction qui calcule la précision d'un sous-ensemble. Cette fonction est égale à la fonction jaccard_similarity_score. Essentiellement, il calcule la correspondance entre les étiquettes réelles et les étiquettes prévues dans l'ensemble de test

```
from sklearn import metrics
print("Test set Accuracy: ", metrics.accuracy_score(y_test, yhat))
```

```
Test set Accuracy:  0.6233333333333333
```

+Et les autres K?

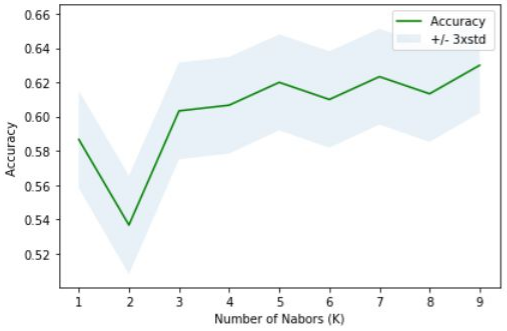
K dans KNN, est le nombre de voisins les plus proches à examiner. Il est censé être spécifié par l'utilisateur. Alors, comment pouvons-nous choisir la bonne valeur pour K? La solution générale consiste à réserver une partie de nos données pour tester la précision du modèle. Choisissons ensuite k = 1, utilisons la partie d'apprentissage pour la modélisation et calculons la précision de la prédiction à l'aide de tous les échantillons de notre ensemble de tests. Répétons ce processus en augmentant le k et voyons quel k est le meilleur pour notre modèle.

Nous pouvons calculer la précision de KNN pour différents Ks.

```
Ks = 10
mean_acc = np.zeros((Ks-1))
std_acc = np.zeros((Ks-1))
ConfustionMx = [];
for n in range(1,Ks):

    #Train Model and Predict
    neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train,y_train)
    yhat=neigh.predict(X_test)
    mean_acc[n-1] = metrics.accuracy_score(y_test, yhat)


    std_acc[n-1]=np.std(yhat==y_test)/np.sqrt(yhat.shape[0])

mean_acc
```

Graphe de précision du modèle pour un nombre différent de voisins.

```
plt.plot(range(1,Ks),mean_acc,'g')
plt.fill_between(range(1,Ks),mean_acc - 1 * std_acc,mean_acc + 1 * std_acc, alpha=0.10)
plt.legend(('Accuracy ', '+/- 3xstd'))
plt.ylabel('Accuracy ')
plt.xlabel('Number of Nabors (K)')
plt.tight_layout()
plt.show()
```




```
print( "The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1)
```

```
The best accuracy was with 0.63 with k= 9
```

Implémentation de l'algorithme Random Forests avec python pure:

Les arbres de décision peuvent souffrir d'une variance élevée qui rend leurs résultats fragiles aux données de formation spécifiques utilisées.

La création de plusieurs modèles à partir d'échantillons de nos données d'entraînement, appelés ensachage, peut réduire cette variance, mais les arbres sont fortement corrélés.

Random Forest est une extension de l'ensachage qui, en plus de construire des arbres sur la base de plusieurs échantillons de vos données d'entraînement, limite également les fonctionnalités qui peuvent être utilisées pour construire les arbres, forçant les arbres à être différents. Ceci, à son tour, peut donner un coup de fouet aux performances.

Alors dans cette partie on va découvrire comment implémenter l'algorithme Random Forest à partir de zéro en Python

Nous utiliserons également une implémentation de l'algorithme CART (Classification and Regression Trees) adapté pour l'ensachage, y compris les fonctions d'assistance test_split () pour diviser un ensemble de données en groupes

```
# split de  data basant sur un attribut et une valeur d'attribut
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right
```

Et qui contient aussi gini_index () pour évaluer un point de division

```
#calcule de gini index pour une dataset deja divisé
def gini_index(groups, classes):
    # count all samples at split point
    n_instances = float(sum([len(group) for group in groups]))
    # sum weighted Gini index for each group
    gini = 0.0
    for group in groups:
        size = float(len(group))
        # avoid divide by zero
        if size == 0:
            continue
        score = 0.0
        # score the group based on the score for each class
        for class_val in classes:
            p = [row[-1] for row in group].count(class_val) / size
            score += p * p
        # weight the group score by its relative size
        gini += (1.0 - score) * (size / n_instances)
    return gini
```

notre fonction get_split () modifiée discutée dans l'étape précédente,

```
#selectionment de la meilleur point de division pour une dataset
def get_split(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

to_terminal (), split () et build_tree () utilisées pour créer un seul arbre de décision

```
#creation d'une valeur de feuille
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count)
```

```
#creation de division enfants pour une noeud ou creation d'une feuille
def split(node, max_depth, min_size, n_features, depth):
    left, right = node['groups']
    del(node['groups'])
    # check for a no split
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return
    # check for max depth
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    # process left child
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left, n_features)
        split(node['left'], max_depth, min_size, n_features, depth+1)
    # process right child
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right, n_features)
        split(node['right'], max_depth, min_size, n_features, depth+1)
```

prédite () pour faire une prédiction avec un arbre de décision, sous-échantillon () pour faire un sous-échantillon de l'ensemble de données d'apprentissage

```
#faire une prediction avec l'arbre de decision
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

```
#creation d'une subsample aleatoir depuis un dataset
def subsample(dataset, ratio):
    sample = list()
    n_sample = round(len(dataset) * ratio)
    while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample
```

bagging_predict () pour faire une prédiction avec une liste d'arbres de décision.

```
#faire une prediction avec une liste de arbres bagged
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree in trees]
    return max(set(predictions), key=predictions.count)
```

Un nouveau nom de fonction random_forest () est développé qui crée d'abord une liste d'arbres de décision à partir de sous-échantillons de l'ensemble de données d'apprentissage, puis les utilise pour faire des prédictions.

```python
# Random Forest Algorithme
def random_forest(train, test, max_depth, min_size, sample_size, n_trees, n_features):
    trees = list()
    for i in range(n_trees):
        sample = subsample(train, sample_size)
        tree = build_tree(sample, max_depth, min_size, n_features)
        trees.append(tree)
    predictions = [bagging_predict(trees, row) for row in test]
    return(predictions)
```

Dans cette section, nous appliquerons l'algorithme Random Forest au jeu de notre dataset.

```python
print("################# My Random Forest Classifier Implementation ##############")
# teste de notre algorithme

###################### preparation et chargement du data
trainfilename = 'training.csv' #train file
testfilename = 'testing.csv'   #test file

train_set=load_csv(trainfilename)[1:]
for i in range(0, len(train_set[0])-1):
  str_column_to_float(train_set, i)

test_set=load_csv(testfilename)[1:]
for i in range(0, len(test_set[0])-1):
  str_column_to_float(test_set, i)

str_column_to_int(train_set,test_set, len(train_set[0])-1)
```

```
################# My Random Forest Classifier Implementation ##############
{'orchard': 0, 'forest': 1, 'water': 2, 'farm': 3, 'grass': 4, 'impervious': 5}
```

Des arbres profonds ont été construits avec une profondeur maximale de 10 et un nombre minimum de lignes d'apprentissage à chaque nœud de 1. Des échantillons de l'ensemble de données d'apprentissage ont été créés avec la même taille que l'ensemble de données d'origine, ce qui est une attente par défaut pour l'algorithme Random Forest.

Le nombre d'entités prises en compte à chaque point de partage a été défini sur sqrt (num_features)

une suite de 3 nombres d'arbres différents a été évaluée à des fins de comparaison, montrant la compétence croissante à mesure que de nouveaux arbres sont ajoutés.

L'exécution de l'exemple imprime les scores pour chaque pli et le score moyen pour chaque configuration.

```
max_depth = 10        #Maximum profondeur pour une arbre
min_size = 1          #taille minimal pour une noeud
sample_size = 0.1     #pendre 10% samples aleatoirement with replacement dans chaque arbree
n_features = int(sqrt(len(train_set[0])-1)) #nombre de features pour creation d'une arbre (Aleatoirement)
number_of_trees = [1, 5, 20] # nombre de arbres dans notre foret

for n_trees in number_of_trees:
    accuracy = evaluate_algorithm(train_set,test_set, random_forest, max_depth, min_size, sample_size, n_trees, n_features)
    print('Trees: %d' % n_trees)
    print('Accuracy: ',accuracy)
```

```
Trees: 1
Accuracy:  51.0
Trees: 5
Accuracy:  59.333333333333336
Trees: 20
Accuracy:  62.0
```

Implémentation de l'algorithme Random Forests avec sklearn:

+Importation de bibliothèque et entraînement:

```
from sklearn.ensemble import RandomForestClassifier
```

```
clf=RandomForestClassifier(n_estimators=100)
```

+Prediction:

Nous pouvons maintenant utiliser le modèle pour prédire l'ensemble de test:

```
y_pred=clf.predict(X_test)
```

+Évaluation de la précision

```
from sklearn import metrics
```

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6233333333333333
```

Résultats:

Après l'implémentation des algorithmes avec les deux méthodes (avec python pure et la bibliothèque sklearn) nous avons obtenu les résultat suivants:

_ En comparant les résultats obtenu avec l'algorithm de KNN on trouve que la précision avec python pure (63,66 %) est la même que celle obtenu en utilisant Sklearn (62, 33 %).

_ Néanmoins pour l'algorithm de Random forest en trouve des petits différence qu'on peut envisager comme négligeables (62 % avec python pure et 62,3 % avec la bibliothèque sklearn) .

Finalement en comparant les résultat entre les deux algorithm on trouve que la meilleur résultat obtenu avec KNN est 63% tant que avec Random Forest c'est 62%, la différence est de un peut prés 1% mais on peut dire qu'en terme de précision KNN donne un meilleur performance , cela en terme de vitesse aussi, nous avons remarqué que KNN fonctionne mieux en termes de vitesse d'apprentissage du modèle.

Conclusion

Les arbres de décision répondent simplement à un problème de discrimination, c'est une des rares méthodes que l'on peut présenter assez rapidement à un public non spécialiste du traitement des données sans se perdre dans des formulations mathématiques délicates à appréhender. Dans cette présentation de mini projet, nous avons voulu mettre l'accent sur les éléments clés de leur construction à partir d'un ensemble de données, puis nous avons présenté l'algorithme KNN et random forest qui permettent de répondre à ces spécifications.

Références :

https://mrmint.fr/introduction-k-nearest-neighbors
https://openclassrooms.com/fr/courses/4011851-initiez-vous-au-machine-learning/4022441-entrainez-votre-premier-k-nn
https://machinelearningmastery.com/implement-random-forest-scratch-python/
https://www.youtube.com/watch?v=WvmPnGmCaIM
https://towardsdatascience.com/random-forests-and-decision-trees-from-scratch-in-python-3e4fa5ae4249
https://stackabuse.com/random-forest-algorithm-with-python-and-scikit-learn/