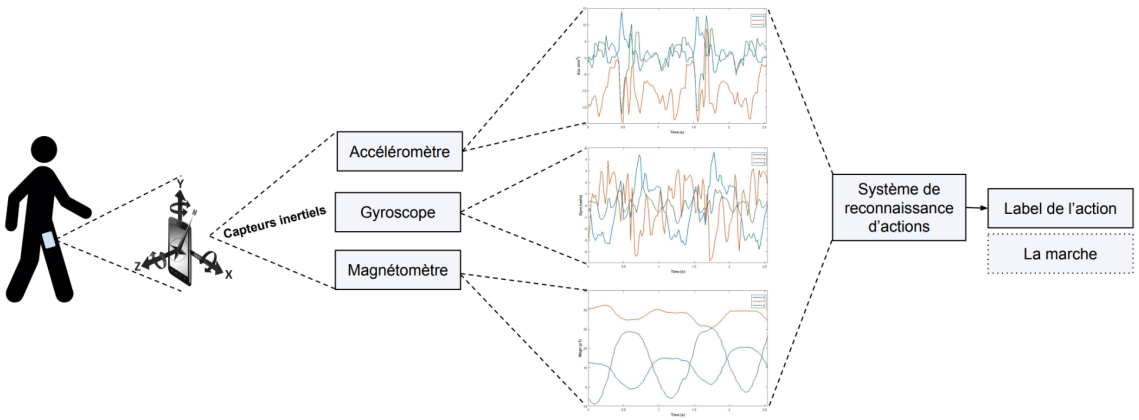
IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

Compte-rendu de projet :

Reconnaissance des actions humaines à l'aide de capteurs inertiels



Réalisé par : Dorian Macquet et Ombeline Moineau
Dans le cadre de l'UV : UV-HY-AADA
FISE 2024
Année 2022-2023

Table des matières

Introduction

Dans le cadre de l'UV hybride AADA, nous sommes amenés à réaliser un projet d'apprentissage automatique, nous permettant de mettre en œuvre les compétences acquises durant l'UV. L'objectif de ce projet est de réaliser un système de reconnaissance des actions humaines à l'aide de capteurs inertiels. Ces capteurs inertiels embarqués à l'intérieur du smartphone porté par la personne, à savoir, l'accéléromètre, le gyroscope et le magnétomètre permettent de collecter les signaux inertiels. Ces derniers seront ensuite mis à l'entrée d'un système de reconnaissance d'actions afin d'identifier l'action exécutée en temps réel.

Pour mener à bien ce projet, nous proposons d'utiliser le langage Python, un langage de programmation très populaire en intelligence artificielle et en apprentissage automatique.

La base de données utilisée dans ce projet contient 27 différentes actions réalisées par 8 sujets, dont 4 hommes et 4 femmes. Chaque sujet a répété chaque action 4 fois. Après avoir supprimé trois séquences corrompues, la base de données comprend en total 861 séquences. Les données ont été stockées en utilisant l'environnement de développement MATLAB.

1. Chargement des données

La première étape de notre projet consiste à charger nos données sur python. Après avoir téléchargé la base de données, nous devons créer une fonction "load_data" qui nous permettra d'obtenir un dataframe avec tous les fichiers de notre base de données. Notre dataframe sera organisé comme suite :

- Colonnes 0-2 : contiennent les données de l'accéléromètre suivant les trois axes
- Colonnes 3-5 : contiennent les données du gyroscope suivant les trois axes
- Colonne 6 : contient l'identifiant du sujet (1 à 8)
- Colonne 7 : contient l'identifiant de l'essai (1 à 4)

Afin de créer notre fonction "load_data", on crée une fonction "load_temp" qui permet de récupérer les indices dans le nom des fichiers. En effet, on sait que les fichiers sont nommés de la manière suivante "ai_sj_tk_inertial.mat", où "ai" représente l'action numéro i, "sj" représente le sujet numéro j et "tk" signifie l'essai numéro k. Avec cette fonction, on va créer un dataframe à partir d'un fichier en paramètre.

Pour les données de l'accéléromètre et du gyroscope (les 6 premières colonnes), on a simplement à les récupérer du fichier MATLAB pour les mettre dans notre dataframe. Par contre, les données Sujet, Essai et Action se trouvent dans le titre du fichier. Pour récupérer ces informations, on utilise la méthode "regex". On sépare le nom du fichier avec les "_" et ensuite on récupère le numéro qui se situe avec les lettres. Ensuite on ajoute les numéros qu'on a récupéré dans le titre du fichier, dans la colonne correspondante du dataframe.

```
def load_temp(path):
  data = sio.loadmat(path)
  df = pd.DataFrame(data['d_iner'], columns=['Accélèromètre_x','Accélèromètre_y', 'Accélèromètre_z', 'Gyroscope_x', 'Gyroscope_y', 'Gyroscope_z
  x = re.split("/", path)
  x = x[-1]
  x = re.split("_", x)
  action = re.split("^a", x[0])
  action = action[1]
  sujet = re.split("^s", x[1])
  sujet = sujet[1]
  essai = re.split("^t", x[2])
  essai = essai[1]
  df = df.assign(Sujet=int(sujet))
  df = df.assign(Essai=int(essai))
  df = df.assign(Action=int(action))
  return df
```

On crée ensuite notre fonction “load_data’” qui prend en paramètre notre dossier contenant nos données. Avec la fonction “listdir”, on récupère les noms des fichiers du dossier dans une liste. Ensuite avec cette liste, on fait une boucle qui va créer un dataframe par fichier avec la fonction “load_temp” créée auparavant. Avec cette boucle, on va mettre tous ces dataframes dans une liste. Ensuite on va concaténer tous ces dataframes dans un dataframe global avec la fonction “pd.concat”.

```
# Fonction qui permet de mettre tous les fichiers d'un dossier dans un dataframe dataset
def load_data(path) :
  frames = []
  dir = os.listdir(path)  # dir est une liste
  for i in range(len(dir)):
    new_path=0
    new_path = path + dir[i]
    if (new_path != '/content/IMU/.ipynb_checkpoints' ):  # fichier qui apparait à cause de google collab donc on l'ignore
      df_temp = load_temp(new_path)
      frames.append(df_temp)
  dataset=pd.concat(frames)
  return dataset
```

On obtient ceci comme sortie :

```
# Création de notre dataframe, chargement des données IMU
df = pd.DataFrame(columns=['Accélèromètre_x','Accélèromètre_y', 'Accélèromètre_z', 'Gyroscope_x', 'Gyroscope_y', 'Gyroscope_z', 'Sujet', 'Essai
df=load_data('/content/IMU/')
df.head()
```

	Accélèromètre_x	Accélèromètre_y	Accélèromètre_z	Gyroscope_x	Gyroscope_y	Gyroscope_z	Sujet	Essai	Action
0	-1.031738	-0.171143	0.261963	2.381679	9.557252	-4.274809	2	3	5
1	-1.075928	-0.173584	0.235840	8.854962	15.633588	-7.633588	2	3	5
2	-1.106689	-0.256348	0.179443	20.763359	24.061069	-12.763359	2	3	5
3	-1.135986	-0.336182	0.146729	25.648855	35.541985	-20.641221	2	3	5
4	-1.126709	-0.444092	0.065918	12.152672	49.679389	-24.122137	2	3	5

2. Traçage des 3 signaux (x,y,z)

On cherche maintenant à créer une fonction “tracer_signal” qui nous permettra de tracer les trois signaux (x, y, z) d’un capteur correspondant à une action en particulier. Cette fonction prendra donc 5 entrées : le dataframe, le capteur (1 : accéléromètre, 2 : gyroscope), le numéro de l’action, le numéro du sujet et le numéro de l’essai.

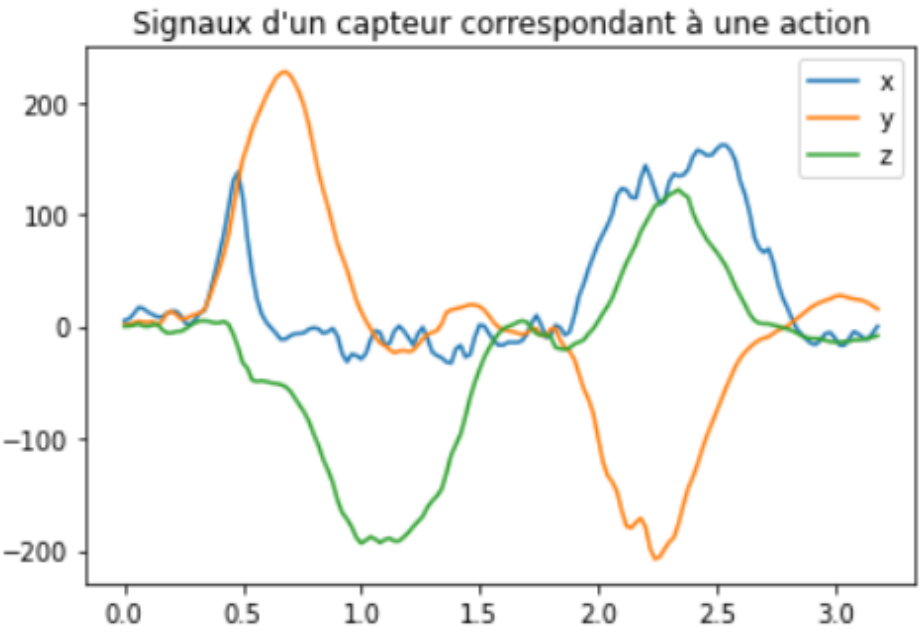
Dans cette fonction, on crée trois listes x, y et z, dans lesquelles on va stocker les valeurs de nos capteurs par la suite. Puis, on différencie le cas du capteur 1 du 2, avec une condition “if”. Ensuite, on crée un dataframe ne comportant uniquement les données du sujet, de l’action et de l’essai correspondant. Les listes x, y et z n’ont plus qu’à prendre les valeurs de la colonne correspondante de notre nouveau dataframe. Puis on trace les courbes avec une fréquence d'échantillonnage de 50 Hz comme voulue dans le sujet.

```
[3]  # Fonction qui permet de tracer les trois signaux
     def tracer_signal (df, capteur, action, sujet, essai):
       x = []
       y = []
       z = []
       if (capteur==1):
         res = df[(df['Sujet'] == sujet) & (df['Essai'] == essai) & (df['Action'] == action) ]
         x = res.Accélèromètre_x
         y = res.Accélèromètre_y
         z = res.Accélèromètre_z
       else :
         res = df[(df['Sujet'] == sujet) & (df['Essai'] == essai) & (df['Action'] == action) ]
         x = res.Gyroscope_x
         y = res.Gyroscope_y
         z = res.Gyroscope_z

       t = np.arange(0,len(x)*(1/50),1/50)
       plt.plot(t,x, label= 'x')
       plt.plot(t,y, label = 'y')
       plt.plot(t,z, label = 'z')
       plt.legend(loc = 'upper right')
       plt.title("Signaux d'un capteur correspondant à une action")
       plt.show()

     # On crée un graphique avec les 3 signaux pour l'action 1, du sujet 1, pour son essai 1
     tracer_signal(df, 2, 1, 1, 1)
```

Voici un exemple du traçage des trois signaux pour le capteur 2 (gyroscope), pour l'action 1 du sujet 1, lors de son 1er essai :



3. Extraction des attributs

Pour décrire chaque action humaine, une liste d'attributs est extraite des signaux. Dans ce projet, nous allons opter pour l'extraction d'attributs statistiques, nous avons choisi d'extraire l'écart-type et la moyenne pour les trois axes des deux capteurs. On cherche maintenant à créer une fonction "feature_extraction" qui prend en entrée le "dataframe" créé. Pour chaque action, nous créons une liste d'attributs. Ensuite, nous ajoutons ces listes dans une liste globale qu'on appelle "vecteurs".

```
# Fonction qui calcule pour chaque action un vecteur d'attributs (moyenne et écart-type)
def feature_extraction (df):
  vecteurs = []
  for i in range (1,28):
    df_temp = df[(df['Action'] == i)]
    Ax_mean = df_temp['Accélèromètre_x'].mean()
    Ax_std = df_temp['Accélèromètre_x'].std()
    Ay_mean = df_temp['Accélèromètre_y'].mean()
    Ay_std = df_temp['Accélèromètre_y'].std()
    Az_mean = df_temp['Accélèromètre_z'].mean()
    Az_std = df_temp['Accélèromètre_z'].std()
    Gx_mean = df_temp['Gyroscope_x'].mean()
    Gx_std = df_temp['Gyroscope_x'].std()
    Gy_mean = df_temp['Gyroscope_y'].mean()
    Gy_std = df_temp['Gyroscope_y'].std()
    Gz_mean = df_temp['Gyroscope_z'].mean()
    Gz_std = df_temp['Gyroscope_z'].std()
    V = [Ax_mean, Ax_std, Ay_mean,Ay_std, Az_mean,Az_std, Gx_mean, Gx_std, Gy_mean,  Gy_std, Gz_mean, Gz_std]
    vecteurs.append(V)
  return vecteurs
```

4. Préparation des données

Afin de préparer les données (attributs calculés) pour l'étape de classification, ces dernières doivent être divisées en deux groupes : les données d'apprentissage et les données de test. Dans ce projet, les données d'apprentissage doivent contenir les attributs calculés relatifs aux sujets [1, 3, 5, 7] et les données de test aux sujets [2, 4, 6, 8].

Pour ce faire, nous devons d'abord réaliser une normalisation des données. Pour appliquer cette dernière, il suffit de calculer le vecteur des moyennes et des l'écart-types sur les attributs d'apprentissage. Puis ensuite soustraire le vecteur moyenne des données et diviser sur l'écart-type. Nous avons décidé de normaliser selon chaque action.
Nous avons donc créé une fonction "preparation". Elle prend en entrée un dataframe et retourne les données d'apprentissage, les données tests, les labels d'apprentissage et les labels de test. Il y a d'abord l'étape de la normalisation. Pour chaque action (1 à 27), on crée un dataframe temporaire des données "data_temp" et une copie de ce dataframe qu'on nomme "data". Puis, on enlève les colonnes "Sujet", "Action" et "Essai" du dataframe "data", puis on le normalise. Ensuite, on lui rajoute les colonnes "Sujet", "Action" et "Essai" qu'on va chercher dans "data_temp". Puis on crée une liste "L" dans laquelle on ajoute à chaque fois, pour chaque action différente, le dataframe normalisé "data". À la fin, on crée notre dataframe final normalisé "dataset", en concaténant notre liste "L".

Nos données sont maintenant normalisées. Pour séparer nos données d'apprentissage et de test, on crée deux nouveaux dataframes "df_train" et "df_test" dans lesquels on met les données de notre "dataset" uniquement pour les sujets dont on a besoin. On stocke les classes, donc les numéros d'actions, dans deux autres dataframes "label_train" et "label_test". Puis on retire les colonnes des classes, donc "Action", dans chacun de nos jeux de données, "df_train" et "df_test". On obtient donc nos données d'apprentissage, données tests, labels d'apprentissage et labels de test.

```
# Fonction qui normalise nos données et qui split nos données test et train
def preparation (df):
  L = []
  dataset = pd.DataFrame()
  # On normalise les données
  for i in range(28):
    df_temp = df[(df['Action'] == i)]
    data = df_temp.copy()
    data.drop(columns=['Sujet','Essai','Action'])
    data = (data - data.mean())/data.std()
    data['Sujet']=df_temp['Sujet']
    data['Essai']=df_temp['Essai']
    data['Action']=df_temp['Action']
    L.append(data)
  dataset=pd.concat(L)
  # On split nos données test et train
  df_train = dataset[(dataset['Sujet'] == 1) | (dataset['Sujet'] == 3)| (dataset['Sujet'] == 5)| (dataset['Sujet'] == 7 )]
  df_test = dataset[(dataset['Sujet'] == 2) | (dataset['Sujet'] == 4)| (dataset['Sujet'] == 6)| (dataset['Sujet'] == 8 )]
  # On stocke à part nos classes
  label_train=df_train['Action']
  label_test=df_test['Action']
  # On supprime nos classes du jeu de données
  df_train = df_train.drop(columns=['Action'],axis=1)
  df_test = df_test.drop(columns=['Action'], axis=1)
  return df_train, df_test, label_train, label_test

# On crée nos données train et test
# X correspond au jeu de données et Y aux classes
X_train, X_test, Y_train, Y_test = preparation(df)
```

5. Entraînement d'un modèle de classification

Afin d'entamer la classification des actions humaines, nous allons d'abord entraîner un modèle sur les données d'apprentissage. Pour cela, nous allons utiliser "Scikit-learn", une bibliothèque libre de Python destinée à l'apprentissage automatique. Dans ce projet, nous avons utilisé 4 classifieurs :

- **Naïve-Bayes**

Naïve Bayes est un algorithme de classification probabiliste basé sur l'application du théorème de Bayes avec des hypothèses naïves sur les variables indépendantes. Il est rapide et efficace, mais suppose que les variables indépendantes sont indépendantes, ce qui peut entraîner des résultats inexacts dans certaines situations.

```
# On entraîne notre modèle avec les classes
clf = GaussianNB()
clf.fit(X_train, Y_train)
```

- **K-plus proches voisins**

KNN (k-nearest neighbors) est un algorithme de classification basé sur l'apprentissage par voisinage. Il est simple à comprendre et à mettre en œuvre, mais peut être peu performant avec de grandes quantités de données. Il est également sensible aux bruits dans les données d'entraînement.

```
[11] neigh = KNeighborsClassifier()
     neigh.fit(X_train,Y_train)
```

- **Random Forest**

Random Forest est un ensemble d'arbres de décision. Il est généralement plus robuste et plus performant que les arbres de décision individuels. Il est également capable de gérer les variables catégoriques et numériques, et de gérer les données manquantes.

```
[16] rf = RandomForestClassifier()
     rf.fit(X_train,Y_train)
```

- **Arbre de décision**

L'arbre de décision est un algorithme de classification basé sur un arbre de décision. Il est facile à comprendre et à visualiser, mais peut être sujet à l'overfitting. Il est également sensible aux variations dans les données d'entraînement.

```
1 arbre = tree.DecisionTreeClassifier()
2 arbre.fit(X_train, Y_train)
```

Test du modèle et analyse des données

Afin d'estimer les performances de reconnaissances globales de notre modèle, des mesures peuvent être calculées, nous avons choisi d'évaluer la précision et l'exactitude (accuracy en anglais).

- **Naïve-Bayes**

```
1 training_data_accuracy = accuracy_score(X_train_pred, Y_train)
2 test_data_accuracy = accuracy_score(X_test_pred, Y_test)
3 print("Accuracy score du training data : ", round(training_data_accuracy*100,2), " %")
4 print("Accuracy score du test data : ", round(test_data_accuracy*100,2)," %")
```

```
Accuracy score du training data :  6.71  %
Accuracy score du test data :  3.27  %
```

Classification report du modèle de Naïve Bayes :

```
class_report = classification_report(Y_test, X_test_pred)
print(class_report)
```

	precision	recall	f1-score	support
1	0.01	0.00	0.00	2643
2	0.00	0.00	0.00	2555
3	0.09	0.00	0.01	2787
4	0.00	0.00	0.00	2633
5	0.00	0.00	0.00	2416
6	0.02	0.00	0.00	2639
7	0.02	0.06	0.03	2393
8	0.01	0.01	0.01	2866
9	0.05	0.01	0.01	3114
10	0.02	0.01	0.01	3146
11	0.06	0.06	0.06	3277
12	0.02	0.02	0.02	3314
13	0.01	0.00	0.01	2839
14	0.00	0.00	0.00	3214
15	0.04	0.03	0.03	2870
16	0.05	0.01	0.02	2549
17	0.03	0.03	0.03	3006
18	0.00	0.00	0.00	2690
19	0.01	0.00	0.00	3033
20	0.00	0.00	0.00	2445
21	0.01	0.02	0.01	3909
22	0.05	0.02	0.03	2618
23	0.04	0.01	0.01	3014
24	0.02	0.02	0.02	2437
25	0.01	0.00	0.00	2608
26	0.02	0.04	0.02	3665
27	0.04	0.48	0.08	3171
accuracy			0.03	77851
macro avg	0.02	0.03	0.02	77851
weighted avg	0.02	0.03	0.02	77851

Pour le modèle de Naïve Bayes, on obtient sur nos données de test un accuracy score de 3.27% et une précision moyenne de 2%. Ce modèle est donc peu fiable et ne prédit quasiment rien d'exact, il est "underfitted".

- **k-plus proches voisins**

```
training_data_accuracy = accuracy_score(X_train_pred, Y_train)
test_data_accuracy = accuracy_score(X_test_pred, Y_test)
print("Accuracy score du training data : ", round(training_data_accuracy*100,2), " %")
print("Accuracy score du test data : ", round(test_data_accuracy*100,2)," %")
```

```
Accuracy score du training data :  90.4  %
Accuracy score du test data :  23.53  %
```

```
1 class_report = classification_report(Y_test, X_test_pred)
2 print(class_report)
```

	precision	recall	f1-score	support
1	0.19	0.23	0.21	2643
2	0.21	0.25	0.23	2555
3	0.20	0.21	0.20	2787
4	0.22	0.29	0.25	2633
5	0.18	0.23	0.20	2416
6	0.31	0.33	0.32	2639
7	0.16	0.17	0.16	2393
8	0.22	0.25	0.23	2866
9	0.31	0.30	0.31	3114
10	0.18	0.16	0.17	3146
11	0.15	0.21	0.18	3277
12	0.24	0.36	0.29	3314
13	0.10	0.10	0.10	2839
14	0.21	0.22	0.21	3214
15	0.19	0.23	0.21	2870
16	0.43	0.38	0.40	2549
17	0.18	0.15	0.16	3006
18	0.19	0.23	0.21	2690
19	0.24	0.14	0.18	3033
20	0.12	0.08	0.10	2445
21	0.24	0.16	0.19	3909
22	0.30	0.27	0.28	2618
23	0.21	0.15	0.18	3014
24	0.55	0.47	0.50	2437
25	0.53	0.41	0.46	2608
26	0.24	0.16	0.19	3665
27	0.32	0.29	0.30	3171
accuracy			0.24	77851
macro avg	0.24	0.24	0.24	77851
weighted avg	0.24	0.24	0.24	77851

Pour le modèle des k-plus proches voisins, on obtient sur nos données de test un accuracy score de 23.53% et une précision moyenne de 24%. Ce modèle est donc un peu plus fiable que le précédent mais il reste “underfitted”.

- **Random Forest**

```
1 training_data_accuracy = accuracy_score(X_train_pred, Y_train)
2 test_data_accuracy = accuracy_score(X_test_pred, Y_test)
3 print("Accuracy score du training data : ", round(training_data_accuracy*100,2), " %")
4 print("Accuracy score du test data : ", round(test_data_accuracy*100,2)," %")
```

```
Accuracy score du training data :  100.0  %
Accuracy score du test data :  33.06  %
```

```
class_report = classification_report(Y_test, X_test_pred)
print(class_report)
```

```
              precision    recall  f1-score   support

           1       0.35      0.28      0.31      2643
           2       0.31      0.27      0.29      2555
           3       0.30      0.25      0.27      2787
           4       0.32      0.34      0.33      2633
           5       0.32      0.26      0.29      2416
           6       0.46      0.44      0.45      2639
           7       0.26      0.22      0.24      2393
           8       0.31      0.24      0.27      2866
           9       0.41      0.39      0.40      3114
          10       0.28      0.25      0.26      3146
          11       0.25      0.27      0.26      3277
          12       0.32      0.48      0.39      3314
          13       0.20      0.21      0.20      2839
          14       0.25      0.23      0.24      3214
          15       0.31      0.30      0.30      2870
          16       0.60      0.49      0.54      2549
          17       0.31      0.28      0.29      3006
          18       0.28      0.36      0.32      2690
          19       0.34      0.24      0.28      3033
          20       0.28      0.18      0.21      2445
          21       0.35      0.31      0.33      3909
          22       0.23      0.32      0.27      2618
          23       0.22      0.29      0.25      3014
          24       0.72      0.65      0.68      2437
          25       0.66      0.53      0.59      2608
          26       0.24      0.36      0.29      3665
          27       0.42      0.51      0.46      3171

    accuracy                           0.33     77851
   macro avg       0.34      0.33      0.33     77851
weighted avg       0.34      0.33      0.33     77851
```

Pour le modèle Random Forest, on obtient sur nos données de test un accuracy score de 33.06% et une précision moyenne de 34%. Ce modèle est donc encore un peu plus fiable que le précédent mais il reste “underfitted”.

- **Arbre de décision**

Quand nous appliquons le classifieur de l’arbre de décision à notre jeu de données on obtient les règles ci-dessous. Les “feature” représentent les 7 premières colonnes de notre dataset, soit l'accéléromètre en x, y, z, le gyromètre en x, y, z et le numéro du sujet. Et le “truncated branch of depth” représente le numéro de l’action en sortie (de 1 à 27). On remarque donc que l’élément le plus discriminant selon l'algorithme de l’arbre de décision est le “feature_0”, qui représente la colonne 0 de notre dataset, soit l'accéléromètre en x.

```
1 r = tree.export_text(arbre)
2 print(r)
```

```
|--- feature_0 <= -0.53
|   |--- feature_1 <= 0.31
|   |   |--- feature_0 <= -0.72
|   |   |   |--- feature_2 <= 0.50
|   |   |   |   |--- feature_0 <= -1.17
|   |   |   |   |   |--- feature_6 <= 6.00
|   |   |   |   |   |   |--- feature_5 <= 1.32
|   |   |   |   |   |   |   |--- feature_5 <= -0.69
|   |   |   |   |   |   |   |   |--- feature_1 <= -3.42
|   |   |   |   |   |   |   |   |   |--- feature_0 <= -1.77
|   |   |   |   |   |   |   |   |   |   |--- feature_4 <= 0.12
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- feature_4 >  0.12
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 3
|   |   |   |   |   |   |   |   |   |--- feature_0 >  -1.77
|   |   |   |   |   |   |   |   |   |   |--- feature_4 <= 1.46
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 3
|   |   |   |   |   |   |   |   |   |   |--- feature_4 >  1.46
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |--- feature_1 >  -3.42
|   |   |   |   |   |   |   |   |   |--- feature_5 <= -1.56
|   |   |   |   |   |   |   |   |   |   |--- feature_6 <= 2.00
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 6
|   |   |   |   |   |   |   |   |   |   |--- feature_6 >  2.00
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 12
|   |   |   |   |   |   |   |   |   |--- feature_5 >  -1.56
|   |   |   |   |   |   |   |   |   |   |--- feature_3 <= -0.92
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 9
|   |   |   |   |   |   |   |   |   |   |--- feature_3 >  -0.92
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 13
|   |   |   |   |   |   |   |--- feature_5 >  -0.69
```

```
1 training_data_accuracy = accuracy_score(X_train_pred, Y_train)
2 test_data_accuracy = accuracy_score(X_test_pred, Y_test)
3 print("Accuracy score du training data : ", round(training_data_accuracy*100,2), " %")
4 print("Accuracy score du test data : ", round(test_data_accuracy*100,2)," %")
```

```
Accuracy score du training data :  100.0  %
Accuracy score du test data :  24.75  %
```

Pour le modèle de l'Arbre de décision, on obtient sur nos données de test un accuracy score de 24.75 %.

Conclusion

Il n'y a pas de réponse unique à la question quel est le meilleur algorithme de classification en machine learning, car cela dépend des données et des objectifs spécifiques du projet. KNN aurait pu être un bon choix, malheureusement nos données ici sont de trop grande dimension. L'arbre de décision aurait pu être un bon choix également, mais dans ce projet nous avons beaucoup trop de colonnes et de valeurs dans notre dataset.
Cependant, après avoir étudié les résultats précédents, on observe que le modèle avec Random Forest paraît comme plus performant pour la classification en raison de sa robustesse.