

Online Signature Verification Challenge

Adnane Mehdaoui

Abdelghaffour Mouhsine

Yendoubouam Alexandre Lalle

Karim jtit

youssef aderdar

Rachid Baiou

Entrée [1]:

```

1 # importation des bibliothéques nécessaires
2 import pandas as pd
3 import math
4 import numpy as np
5 import os
6 from sklearn.neighbors import KNeighborsClassifier
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import accuracy_score
9 from sklearn.impute import SimpleImputer
10 from scipy.spatial.distance import euclidean
11 from itertools import combinations
12 import random
13 import matplotlib.pyplot as plt
14 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, precision_score, recall_score,
15 from sklearn.calibration import label_binarize
16 from sklearn.metrics import precision_recall_curve, roc_curve, auc
17 import seaborn as sn
18 import joblib

```

chargement des fichiers de signature

Entrée [2]:

```

1 def load_signature_data(file_path):
2     # Définir les noms de colonnes
3     column_names = ['x', 'y', 'timestamp', 'button', 'azimuth', 'altitude', 'pressure']
4
5     # Charger le fichier CSV dans un DataFrame Pandas en utilisant les noms de colonnes
6     df = pd.read_csv(file_path, sep=' ', index_col=False, names=column_names)
7
8     # Supprimer la première ligne qui contient le nombre de lignes
9     df = df.iloc[1:]
10
11    # Réinitialiser les index du DataFrame
12    df = df.reset_index(drop=True)
13
14    return df

```

calculer différentes caractéristiques de la signature à partir des coordonnées (x, y), de l'altitude et de l'azimuth de chaque point

Entrée [3]:

```

1 def CalculeFeature(df):
2     epsilon = 1e-8 # Petite valeur epsilon pour éviter la division par zéro
3     # Calculer la différence de coordonnées à chaque instant
4     delta_x = df['x'].diff()
5     delta_y = df['y'].diff()
6
7     # Calculer la norme de la vitesse à chaque instant
8     delta_t = df['timestamp'].diff() + epsilon
9     vitesse = np.sqrt(delta_x**2 + delta_y**2) / delta_t
10    df['vitesse'] = vitesse
11
12    # Calculer la norme de l'accélération à chaque instant
13    acceleration = vitesse.diff() / delta_t
14    df['acceleration'] = acceleration
15
16    # Calculer l'inclinaison du stylo
17    pen_incl = np.arctan2(df['altitude'], df['azimuth'])
18    df['pen_incl'] = pen_incl
19
20    # Calculer les dérivées secondes des coordonnées à chaque point
21
22    dx = np.gradient(df['x'])
23    dy = np.gradient(df['y'])
24
25    d2x = np.gradient(dx)
26    d2y = np.gradient(dy)
27
28    denominator = np.power(dx*dx + dy*dy, 1.5) + epsilon
29    curvature = np.abs(d2x*dy - dx*d2y) / denominator
30
31    df['curvature'] = curvature
32    if df.isnull().any().any(): # Vérifier si une colonne a des valeurs nulles
33        df = df.fillna(df.mean())
34
35    return df

```

ajouter un champ label aux signatures de chaque utilisateur

Entrée [4]:

```

1 def DonnerClasse(directory):
2     # Initialiser une liste pour stocker les DataFrames de chaque utilisateur
3     dfs = []
4     # Parcourir tous les fichiers du répertoire
5     for filename in os.listdir(directory):
6         if filename.endswith('.TXT'):
7             # Joindre le chemin du fichier avec le répertoire de travail actuel
8             file_path = os.path.join(directory, filename)
9             df = CalculeFeature(load_signature_data(file_path))
10            # extraire l'id de l'utilisateur
11            user_id, signature_id = filename.split('S')
12            user_id = int(user_id[1:])
13            signature_id = int(signature_id[:-4])
14
15            df['classe'] = user_id
16
17            # Ajouter le DataFrame à la liste
18            dfs.append(df)
19
20    # Concaténer tous les DataFrames en un seul DataFrame
21    result = pd.concat(dfs, axis=0, ignore_index=True)
22
23    return result

```

L'exploration des données

Entrée [5]:

```

1 # Définir le répertoire contenant les fichiers texte
2 directory = './data/Task1/'
3
4 data = DonnerClasse(directory)
5 data.head()

```

Out[5]:

	x	y	timestamp	button	azimuth	altitude	pressure	vitesse	acceleration	pen_incl	curvature	classe
0	3236	5028.0	17638839,0	0,0	1260,0	400,0	351,0	15.700367	0,011582	0,307397	0,000990	
1	3428	4998,0	17638849,0	1,0	1260,0	400,0	382,0	19,432962	0,011582	0,307397	0,002234	
2	3541	5056,0	17638859,0	1,0	1260,0	400,0	394,0	12,701575	-0,673139	0,307397	0,001014	
3	3665	5135,0	17638869,0	1,0	1310,0	400,0	397,0	14,702721	0,200115	0,296352	0,000052	
4	3945	5229,0	17638879,0	1,0	1370,0	390,0	405,0	29,535741	1,483302	0,277335	0,000189	

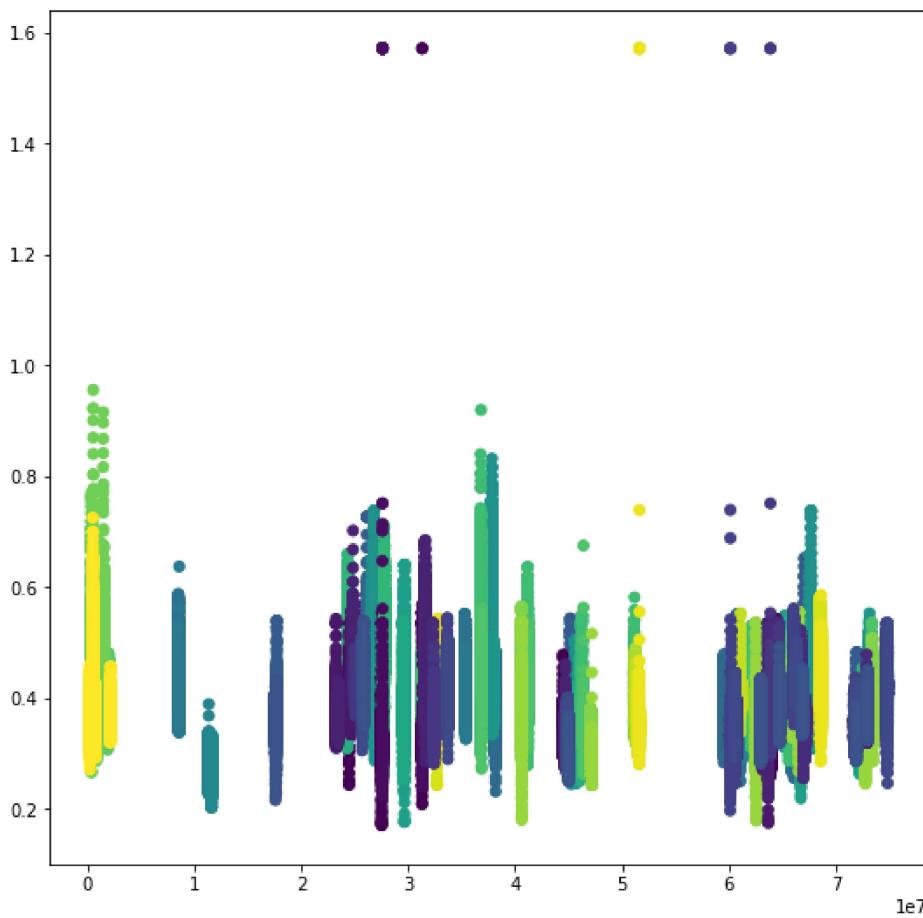
visualisation de la distribution des données

Entrée [9]:

```
1 plt.figure(figsize=(9,9))
2 plt.scatter(data["timestamp"], data["pen_incl"], c=data["classe"])
```

Out[9]:

<matplotlib.collections.PathCollection at 0x1374d37c250>



visualisation de la distribution des données en 3D

Entrée [10]:

```
1 # visualisation 3D
2
3 %matplotlib
4 from mpl_toolkits.mplot3d import Axes3D
5
6 ax = plt.axes(projection='3d')
7 ax.scatter(data["x"], data["y"], data["timestamp"], c=data["classe"])
8 plt.xlabel("x")
9 plt.ylabel("y")
10 plt.show()
```

Using matplotlib backend: <object object at 0x000001374916D810>

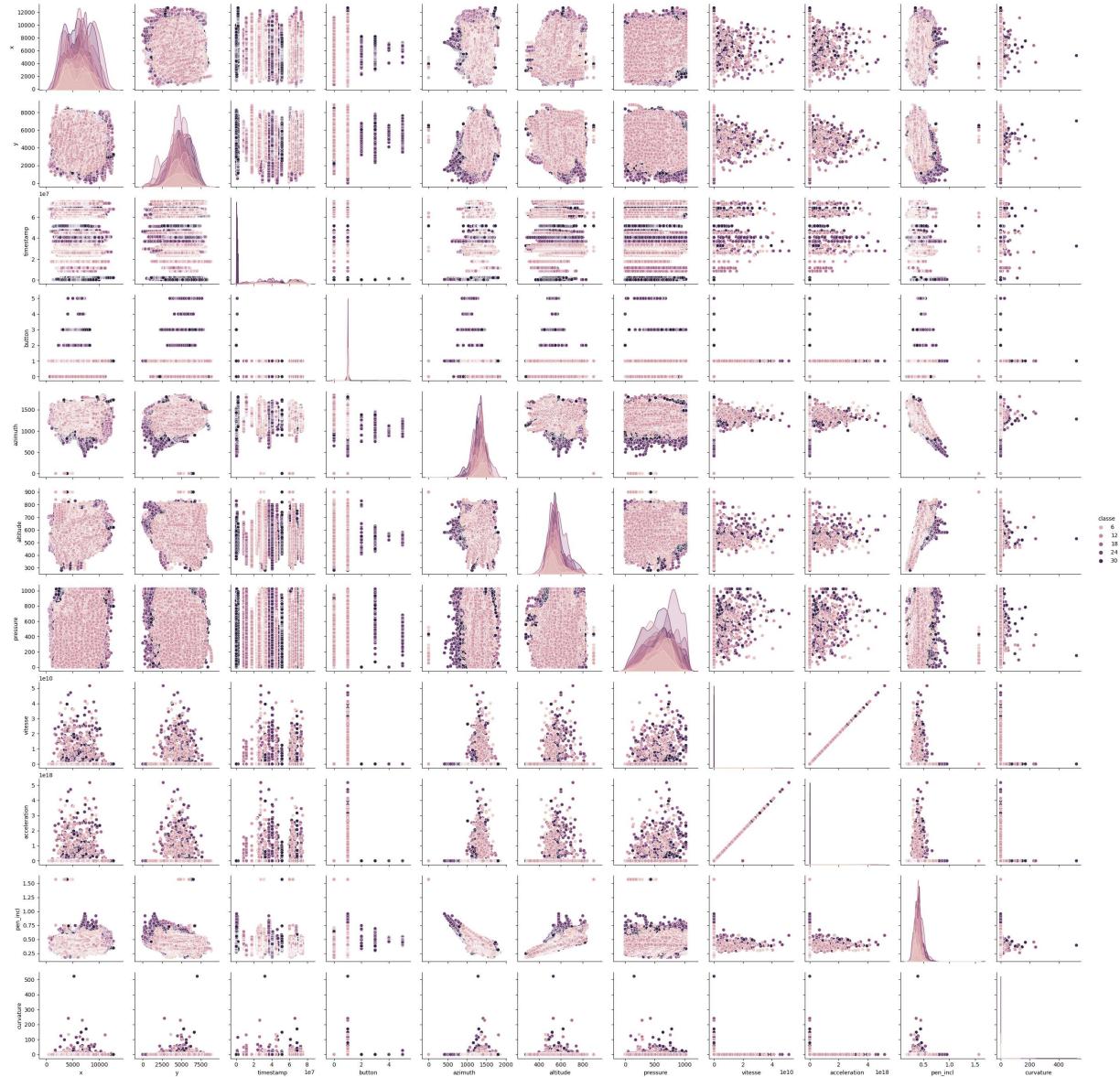
visualisation de la distribution des données pour chaque paire de caractéristiques

Entrée [16]:

```
1 sn.pairplot(data, hue='classe')
```

Out[16]:

```
<seaborn.axisgrid.PairGrid at 0x18d32609e80>
```



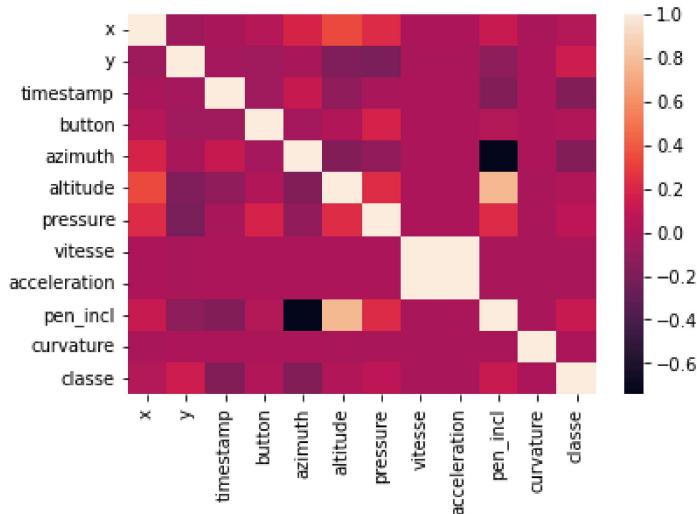
visualisation de la corrélation entre chaque paire de variables

Entrée [6]:

```
1 sn.heatmap(data.corr())
```

Out[6]:

<AxesSubplot:>



Entrée [7]:

```
1 data.corr()
```

Out[7]:

	x	y	timestamp	button	azimuth	altitude	pressure	vitesse	acceleration
x	1.000000	-0.051062	-0.008948	0.030969	0.189916	0.333720	0.218019	0.001062	0.001007
y	-0.051062	1.000000	-0.035070	-0.041903	-0.018004	-0.179113	-0.204595	-0.004316	-0.004351
timestamp	-0.008948	-0.035070	1.000000	-0.043829	0.111499	-0.114635	-0.014257	0.003494	0.003361
button	0.030969	-0.041903	-0.043829	1.000000	-0.028973	0.009746	0.182320	0.003157	0.003165
azimuth	0.189916	-0.018004	0.111499	-0.028973	1.000000	-0.159886	-0.104047	0.005446	0.005507
altitude	0.333720	-0.179113	-0.114635	0.009746	-0.159886	1.000000	0.230342	0.003790	0.003918
pressure	0.218019	-0.204595	-0.014257	0.182320	-0.104047	0.230342	1.000000	0.004121	0.004105
vitesse	0.001062	-0.004316	0.003494	0.003157	0.005446	0.003790	0.004121	1.000000	0.998113
acceleration	0.001007	-0.004351	0.003361	0.003165	0.005507	0.003918	0.004105	0.998113	1.000000
pen_incl	0.119879	-0.124486	-0.165561	0.025935	-0.742125	0.763034	0.223449	-0.001113	-0.001055
curvature	-0.000419	0.004737	-0.000145	0.004584	0.004245	-0.001731	-0.010539	-0.000860	-0.000860
classe	0.021748	0.139025	-0.161620	0.008738	-0.176955	0.019799	0.062436	-0.001975	-0.002052

Selectionner les caractéristiques pertinante sur lesquelles ont se base pour mesurer la simularité . (une correlation avec le target ("classe") élevée)

Entrée [6]:

```
1 def selectFeatures(df):
2     nouvelledf = df[['x', 'y', 'timestamp', 'azimuth', 'altitude', 'pressure', 'pen_incl']]
3     return nouvelledf
```

Séparer les données en ensembles d'entraînement et de test

Entrée [7]:

```

1 X = selectFeatures(data)
2 y = data['classe']
3 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
4
5 # Remplacer les valeurs infinies ou trop grandes par NaN
6 X_train[~np.isfinite(X_train)] = 0
7 X_test[~np.isfinite(X_test)] = 0
8
9 # Imputer les valeurs manquantes par la moyenne de chaque colonne
10 imputer = SimpleImputer(strategy='mean')
11 X_train = imputer.fit_transform(X_train)
12 X_test = imputer.transform(X_test)
13

```

Definition et entrainement de Model avec la méthode KNN

Entrée [8]:

```

1 from sklearn.model_selection import GridSearchCV

```

Entrée [9]:

```

1 def KNN(X_train, X_test, y_train, y_test):
2     # Trouver les meilleurs hyperparamètres pour le modèle
3     param_grid = {'n_neighbors': np.arange(1,5), 'metric': ['euclidean', 'manhattan']}
4     grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=3)
5
6     # Entrainement de model
7     grid.fit(X_train, y_train)
8
9     # Prédire les classes pour les données de test
10    y_pred = grid.predict(X_test)
11
12    print(f"les meilleurs paramètres sont : {grid.best_params_}")
13    print(f"La performance du modèle avec ces paramètres : {grid.best_score_}")
14
15    # Calculer l'accuracy
16    accuracy = accuracy_score(y_test, y_pred)
17    print('knn Accuracy:', accuracy)
18
19    return grid

```

Entrée [11]:

```

1 def training(X_train, X_test, y_train, y_test):
2     knn=KNN(X_train, X_test, y_train, y_test)
3     return knn

```

Entrée [12]:

```

1 knn = training(X_train, X_test, y_train, y_test)

```

les meilleurs paramètres sont : {'metric': 'manhattan', 'n_neighbors': 1}
La performance du modèle avec ces paramètres : 0.9999542480015645
knn Accuracy: 0.9998838883470345

Enregistrement de Model en disque dur

Entrée [13]:

```
1 joblib.dump(knn, "Model_Signature_Verification.joblib")
```

Out[13]:

```
['Model_Signature_Verification.joblib']
```

Chargement de Model

Entrée [14]:

```
1 model = joblib.load("./Model_Signature_Verification.joblib")
```

1 Les signatures d'un même utilisateur sont considérées comme similaire et les signatures d'utilisateurs différents sont considérées comme non similaire.

Entrée [15]:

```
1 def compare_signatures(signature_path1, signature_path2):
2
3     model = joblib.load("./Model_Signature_Verification.joblib")
4     signature1 = selectFeatures(CalculeFeature(load_signature_data(signature_path1)))
5     signature2 = selectFeatures(CalculeFeature(load_signature_data(signature_path2)))
6
7     minN = min(len(signature1), len(signature2))
8
9     y1=model.predict(signature1.iloc[:minN,:])
10    y2=model.predict(signature2.iloc[:minN,:])
11
12    accuracy = accuracy_score(y1, y2)
13
14    if accuracy>0.50 :
15        similarity = True
16    else :
17        similarity = False
18    return accuracy , similarity
19
```

tester le modèle

Entrée [17]:

```
1
2 acc,sm = compare_signatures("./data/Test/U10S11.TXT", "./data/Test/U10S16.TXT")
3 print("accuracy : ",acc)
4 print("similarity : ",sm)
```

accuracy : 1.0
similarity : True

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\base.py:443: UserWarning: X has feature names, but KNeighborsClassifier was fitted without feature names
warnings.warn(
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\base.py:443: UserWarning: X has feature names, but KNeighborsClassifier was fitted without feature names
warnings.warn(

Ecrire les résultats dans un fichier CSV chaque fois qu'on fait des combinaisons entre deux signatures aléatoirement choisies.

Entrée [18]:

```

1 test_folder_path = './data/Test/'
2 train_folder_path = './data/Task1/'
3 train_files = os.listdir(train_folder_path)
4
5 files_paths = [os.path.join(test_folder_path, filename) for filename in os.listdir(test_folder_path)]
6
7 random_files_paths = random.sample(files_paths, 30)

```

Entrée [19]:

```

1 import warnings
2 warnings.filterwarnings("ignore", category=UserWarning)
3
4 file_info = [{'file_name': os.path.basename(file_path), 'file_path': file_path} for file_path in random_files_paths]
5
6 file_combinations = combinations(file_info, 2)
7 results = []
8
9 results_df = pd.DataFrame(columns=['File 1', 'File 2', 'Similarity'])
10
11 for file_pair in file_combinations:
12     file1, file2 = file_pair
13     acc, sm = compare_signatures(file1['file_path'], file2['file_path'])
14     if sm:
15         label=0
16     else:
17         label=1
18     results.append({'File 1': file1['file_name'], 'File 2': file2['file_name'], 'Similarity': label})
19
20 results_df = pd.concat([results_df, pd.DataFrame(results)], ignore_index=True)
21 results_df.to_csv('comparison_results.csv', index=False)
22

```

Evaluation

1 Le but de ce code est de calculer la précision, le rappel et le score F1 pour évaluer la performance du modèle de classification des signatures entraîné en utilisant l'algorithme KNN, ainsi que la matrice de confusion

Entrée [22]:

```

1 # Calculer la précision, le rappel et le F1-score
2 y_pred = model.predict(X_test)
3 precision = precision_score(y_test, y_pred, average='macro')
4 recall = recall_score(y_test, y_pred, average='macro')
5 f1 = f1_score(y_test, y_pred, average='macro')
6 print("Precision:", precision)
7 print("Recall:", recall)
8 print("F1-score:", f1)

```

Precision: 0.9998511578442126
 Recall: 0.9998662387989318
 F1-score: 0.9998586766676647

Entrée [23]:

```

1 ##### Matrice de confusion #####
2
3 # Calculer la matrice de confusion
4 cm = confusion_matrix(y_test, y_pred, labels=np.unique(y_test))
5
6 # Normaliser la matrice de confusion pour montrer la proportion de prédictions correctes pour chaque
7 cm_norm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
8
9 # Afficher la matrice de confusion
10 #np.set_printoptions(threshold=np.inf)
11 print("\nConfusion Matrix:\n", cm)
12
13
14 plt.figure(figsize=(7, 7))
15 plt.imshow(cm_norm, interpolation='nearest', cmap=plt.cm.Blues)
16 plt.colorbar()
17 tick_marks = np.arange(len(np.unique(y_test)))
18 plt.xticks(tick_marks, np.unique(y_pred), rotation=90)
19 plt.yticks(tick_marks, np.unique(y_test))
20 plt.tight_layout()
21 plt.xlabel('Valeur prédictive')
22 plt.ylabel('Valeur réelle')
23 plt.show()

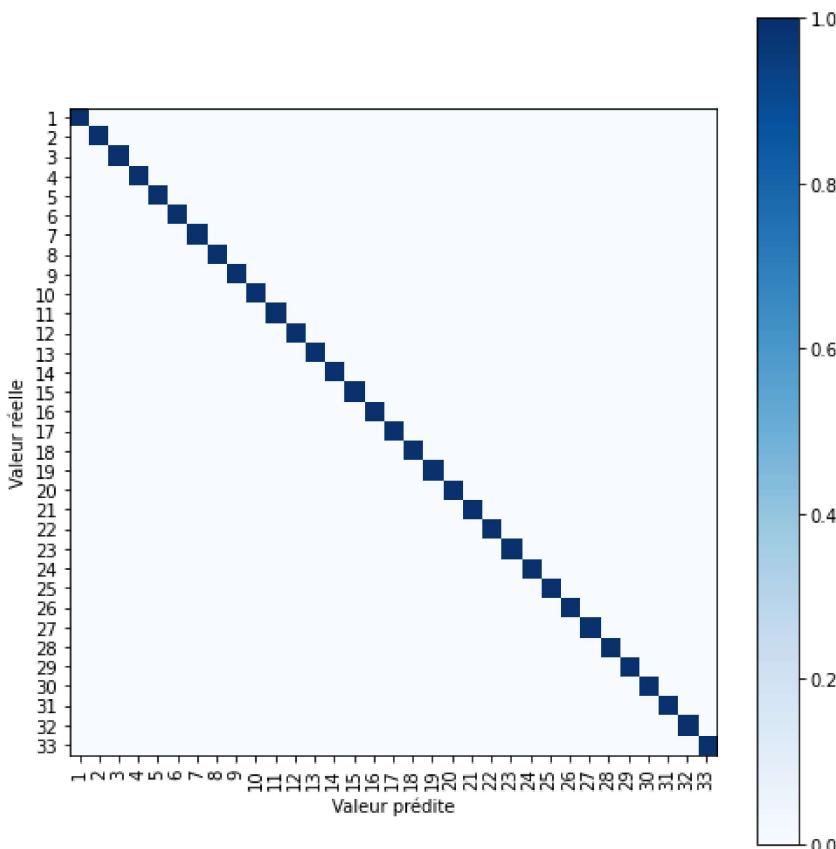
```

Confusion Matrix:

```

[[1460     0     0 ...     0     0     0]
 [  0 1655     0 ...     0     0     0]
 [  0     0 1601 ...     0     0     0]
 ...
 [  0     0     0 ... 2251     0     0]
 [  0     0     0 ...     0 2324     0]
 [  0     0     0 ...     0     0 2174]]

```



courbes de précision-rappel (precision-recall)

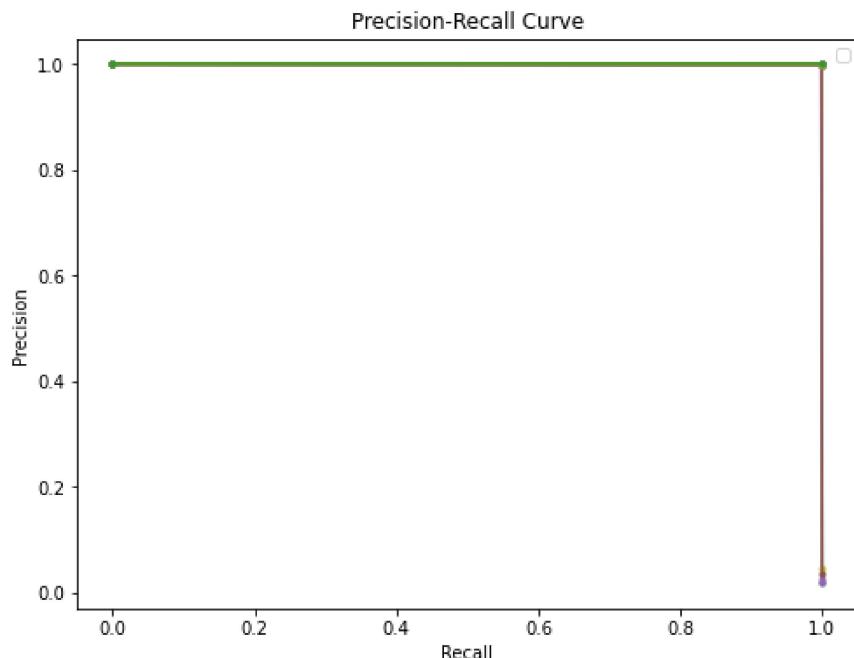
Entrée [24]:

```

1 # Convertir les étiquettes multiclasse en étiquettes binaires
2 y_test_bin = label_binarize(y_test, classes=np.unique(y_test))
3
4 # Calculer les probabilités pour chaque classe en utilisant predict_proba
5 y_scores = knn.predict_proba(X_test)
6
7 # Calculer la précision et le rappel pour chaque classe
8 precision_curve = dict()
9 recall_curve = dict()
10 for i in range(len(np.unique(y_test))):
11     precision_curve[i], recall_curve[i], _ = precision_recall_curve(y_test_bin[:, i], y_scores[:, i])
12
13 # Tracer les courbes de précision-rappel pour chaque classe
14 plt.figure(figsize=(8, 6))
15 for i in range(len(np.unique(y_test))):
16     plt.plot(recall_curve[i], precision_curve[i], marker='.', label=f'Class {i}')
17     plt.plot(recall_curve[i], precision_curve[i], marker='.')
18 plt.xlabel('Recall')
19 plt.ylabel('Precision')
20 plt.title('Precision-Recall Curve')
21 plt.legend()
22 plt.show()

```

No artists with labels found to put in legend. Note that artists whose label start with a n underscore are ignored when legend() is called with no argument.



courbes ROC

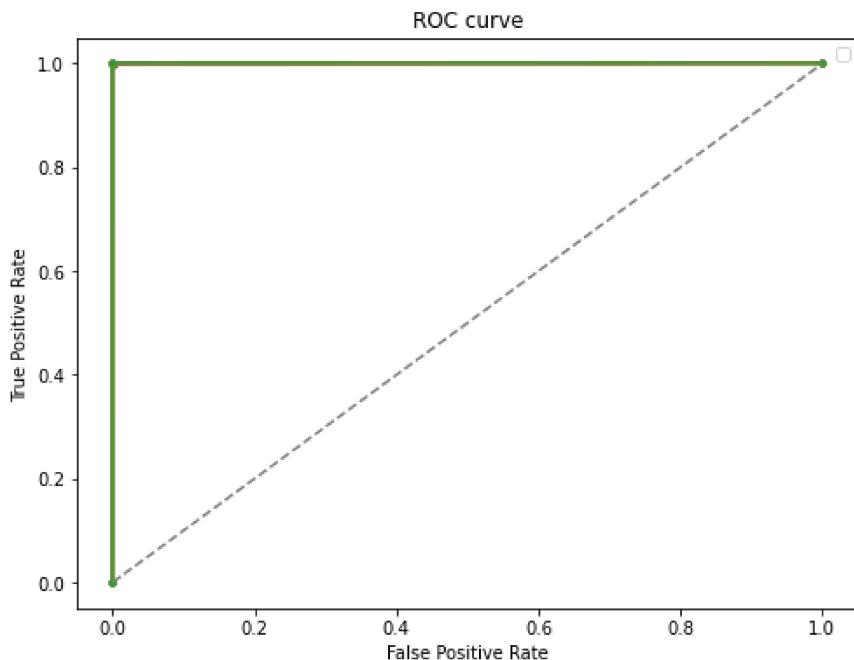
Entrée [25]:

```

1 # Calculer le taux de faux positifs, le taux de vrais positifs et l'aire sous la courbe ROC (AUC) pour
2 fpr = dict()
3 tpr = dict()
4 # roc_auc = dict()
5 for i in range(len(np.unique(y_test))):
6     fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_scores[:, i])
7     # roc_auc[i] = auc(fpr[i], tpr[i])
8
9 # Tracer les courbes ROC pour chaque classe
10 plt.figure(figsize=(8, 6))
11 for i in range(len(np.unique(y_test))):
12     # plt.plot(fpr[i], tpr[i], marker='.', label=f'Class {i} (AUC = {roc_auc[i]:.2f})')
13     plt.plot(fpr[i], tpr[i], marker='.')
14
15 plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
16 # plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Random')
17 plt.xlabel('False Positive Rate')
18 plt.ylabel('True Positive Rate')
19 plt.title('ROC curve')
20 plt.legend()
21 plt.show()

```

No artists with labels found to put in legend. Note that artists whose label start with a `n underscore` are ignored when `legend()` is called with no argument.



Entrée []:

1