

Classification binaire avec Python

-Prédiction de la classe “ Class(0,1) “-

-problème supervisé-

Introduction

- **Objectif** : Entraîner un modèle de classification pour prédire si une instance appartient à la classe 0 ou 1 .
- **Données utilisées** : Fichier Excel data.xlsx.
- **Métrique d'évaluation** : F1 Score (pour gérer le déséquilibre éventuel des classes).
- **Outils** : Python, Pandas, Scikit-learn, Matplotlib/Seaborn.

Démarche Suivie

- 1. Exploration et visualisation des données**
- 2. Prétraitement des données (Nettoyage, Imputation, Sélection)**
- 3. Séparation des données (Entraînement, Test)**
- 4. Modélisation avec différents algorithmes**
- 5. Évaluation et comparaison par F1-score**

1. Analyse Exploratoire

un aperçu du dataset :

- Nombre d'échantillons : 260
- Colonnes : 39
- Target : Class (0 ou 1)
- Caractéristiques :

.Variables démographiques :

Gender, Nationality, Major, Level

.Variables d'évaluation ou compétences :

IE1, RAS1, SMSK1, ..., IM6

.Employed : information sur l'emploi

	ID	Gender	Nationality	Major	Level	IE1	SMSK3	RAS1	RAS2	SMSK1	...	\
0	1	0	1	0	2	4	3	4	4	2	...	
1	2	0	0	0	2	4	4	4	4	3	...	
2	3	0	1	0	1	3	2	3	4	3	...	
3	4	1	1	1	3	4	4	4	4	4	...	
4	5	0	0	0	2	4	3	3	4	3	...	

	IM3	IM4	IM5	IM6	W1	W2	W3	Employed	Score	Class
0	4	4	4	4	4.0	4.0	4.0	1.0	3.20	1
1	4	3	4	4	3.0	4.0	3.0	0.0	3.82	1
2	3	4	4	3	4.0	4.0	2.0	1.0	3.75	1
3	4	4	4	4	3.0	4.0	4.0	1.0	3.70	1
4	4	3	3	3	4.0	4.0	2.0	1.0	3.82	0

[5 rows x 39 columns]

Dimensions du dataset : (260, 39)

Colonnes du dataset : Index(['ID', 'Gender', 'Nationality', 'Major', 'Level', 'IE1', 'RAS2', 'SMSK1', 'SMSK4', 'IE2', 'TL1', 'RAS3', 'IE3', 'RAS4', 'RAS5', 'IE4', 'SMSK2', 'TL2', 'TL3', 'PSD1', 'PSD2', 'PSD3', 'IE5', 'PSD4', 'PSD5', 'IM1', 'IM2', 'IM3', 'IM4', 'IM5', 'IM6', 'W1', 'W2', 'W3', 'Employed', 'Score', 'Class'],
dtype='object')

Détection de redondances (duplication)

Avant de detecter les redondances on a Supprimé la colonne ID car elle est non pertinente pour la prédiction

```
# Supprimer ID car il n'a pas de rôle dans la classification
df.drop(columns=['ID'], inplace=True, errors='ignore')
print(df.columns)
# Afficher les lignes dupliquées
duplicates = df[df.duplicated()]
print("Lignes dupliquées :")
print(duplicates)
✓ 0.0s
```

14 lignes redondantes

→ suppression des lignes dupliquées

Lignes dupliquées :																					
	Gender	Nationality	Major	Level	IE1	SMSK3	RAS1	RAS2	SMSK1	SMSK4	...	IM3	IM4	IM5	IM6	W1	W2	W3	Employed	Score	Class
216	0	0	1	3	4	4	4	4	4	4	...	4	3	4	4	4.0	4.0	NaN	1.0	3.51	0
217	0	0	1	3	4	4	4	4	4	4	...	4	3	3	3	4.0	4.0	1.0	1.0	3.51	0
218	0	0	1	3	3	4	4	3	3	3	...	4	4	3	3	4.0	4.0	1.0	4	4	3
219	0	0	1	3	4	4	4	4	4	4	...	4	4	4	4	4.0	4.0	1.0	4	4	4
220	0	0	1	3	4	4	4	4	4	4	...	4	4	4	4	4.0	4.0	2	4	2	4
235	1	0	1	2	4	3	3	3	4	3	...	4	3	3	3	4.0	3	4	3	4	4
236	1	1	1	2	3	3	3	3	3	3	...	2	2	2	2	3.0	3	3	2	2	2
237	1	0	1	2	3	3	3	4	4	4	...	3	3	3	3	4.0	4	4	3	3	3
240	1	0	1	2	4	3	3	4	4	4	...	2	2	2	2	4.0	4	4	2	2	3
241	1	0	1	2	4	4	4	4	4	4	...	4	4	4	4	4.0	4	4	4	4	4
246	0	1	0	2	4	4	4	3	3	3	...	3	3	3	3	4.0	3	4	3	3	3
247	1	1	0	2	4	4	4	4	4	4	...	3	3	3	3	4.0	4	4	3	3	3
248	1	1	0	2	3	3	3	4	4	4	...	2	2	2	2	4.0	4	4	2	2	3
249	0	1	0	2	3	3	3	3	4	4	...	2	2	2	2	4.0	4	4	2	2	2

Types des données

```
Types de données distincts : [dtype('int64') dtype('float64')] ...
```

Le dataset ne contient que des colonnes numériques (entiers et réels) donc :

- Pas besoin d'encodage car il n'y a aucune variable catégorielle.

→ standardisation parfois nécessaire

Détection des outliers

Affichage des valeurs distinctes pour chaque colonnes

Colonne	Valeurs Distinctes
Gender	[0, 1]
Nationality	[1, 0]
Major	[0, 1]
Level	[2, 1, 3]
IE1	[4, 3, 2, 1]
SMSK3	[3, 4, 2, 1]
RAS1	[4, 3, 2, 1]
RAS2	[4, 3, 2, 1]

Colonne	Valeurs Distinctes
SMSK1	[2, 3, 4, 1]
SMSK4	[3, 4, 2, 1]
IE2	[3, 4, 2, 1]
TL1	[3, 4, 2, 1]
RAS3	[4, 3, 2, 1]
IE3	[3, 4, 2, 1]
RAS4	[3, 4, 2, 1]
RAS5	[2, 3, 4, 1]

Colonne	Valeurs Distinctes
IE4	[4, 3, 2, 1]
SMSK2	[3, 4, 2, 1]
TL2	[3, 4, 2, 1]
TL3	[4, 3, 2, 1]
PSD1	[3, 4, 2, 1]
PSD2	[3, 4, 2, 1]
PSD3	[4, 3, 2, 1]
IE5	[3, 4, 2, 1]

```
for col in df.columns:  
    print(f"--- {col} ---")  
    print(df[col].unique())  
    print("\n")
```

Colonne	Valeurs Distinctes
PSD4	[3, 4, 2, 1]
PSD5	[3, 4, 2, 1]
IM1	[4, 3, 2, 1]
IM2	[4, 3, 2, 1]
IM3	[4, 3, 1, 2]
IM4	[4, 3, 2, 1]
IM5	[4, 3, 2, 1]
IM6	[4, 3, 2, 1]
W1	[4.0, 3.0, 2.0, 1.0]
W2	[4.0, 3.0, 2.0, 1.0]
W3	[4.0, 3.0, 2.0, 1.0]
Employed	[1.0, 0.0]
Score	74 valeurs distinctes [0 4]
Class	[1, 0]

--> pas des valeurs bruitées
sauf dans la colonne de score est très probablement contient des outliers.

Détection des outliers

Outliers dans la colonne score

```
Q1 (25%): 3.405  
Q3 (75%): 3.75  
IQR: 0.3450000000000002  
Borne inférieure: 2.8874999999999993  
Borne supérieure: 4.2675  
Nombre d'outliers détectés dans 'Score' : 4
```

- 4 outliers sont détectés

→ Suppression des lignes
contenant ces valeurs
bruitées

```
# Extraire la colonne Score  
score_col = df['Score']  
  
# Calcul des quartiles  
Q1 = score_col.quantile(0.25)  
Q3 = score_col.quantile(0.75)  
IQR = Q3 - Q1  
  
# Bornes pour détecter les outliers  
lower_bound = Q1 - 1.5 * IQR  
upper_bound = Q3 + 1.5 * IQR  
  
print(f"Q1 (25%): {Q1}")  
print(f"Q3 (75%): {Q3}")  
print(f"IQR: {IQR}")  
print(f"Borne inférieure: {lower_bound}")  
print(f"Borne supérieure: {upper_bound}")  
  
# Identifier les outliers  
outliers = df[(df['Score'] < lower_bound) | (df['Score'] > upper_bound)]  
print(f"Nombre d'outliers détectés dans 'Score' : {len(outliers)}")
```

les valeurs manquantes

Valeurs manquantes :

- W1: 19 valeurs manquantes (~7% du total)
- W2: 14 valeurs manquantes (~5%)
- W3: 45 valeurs manquantes (~18%)
- Employed: 2 valeurs manquantes

→ gestion des valeurs manquantes en tenant compte à la fois de type de données de chaque colonnes et sa proportion de valeurs manquantes

```
# Vérifier les valeurs manquantes par colonne
valeurs_nulles = df.isnull().sum()

# Filtrer uniquement les colonnes qui ont au moins une valeur manquante
colonnes_avec_nulles = valeurs_nulles[valeurs_nulles > 0]

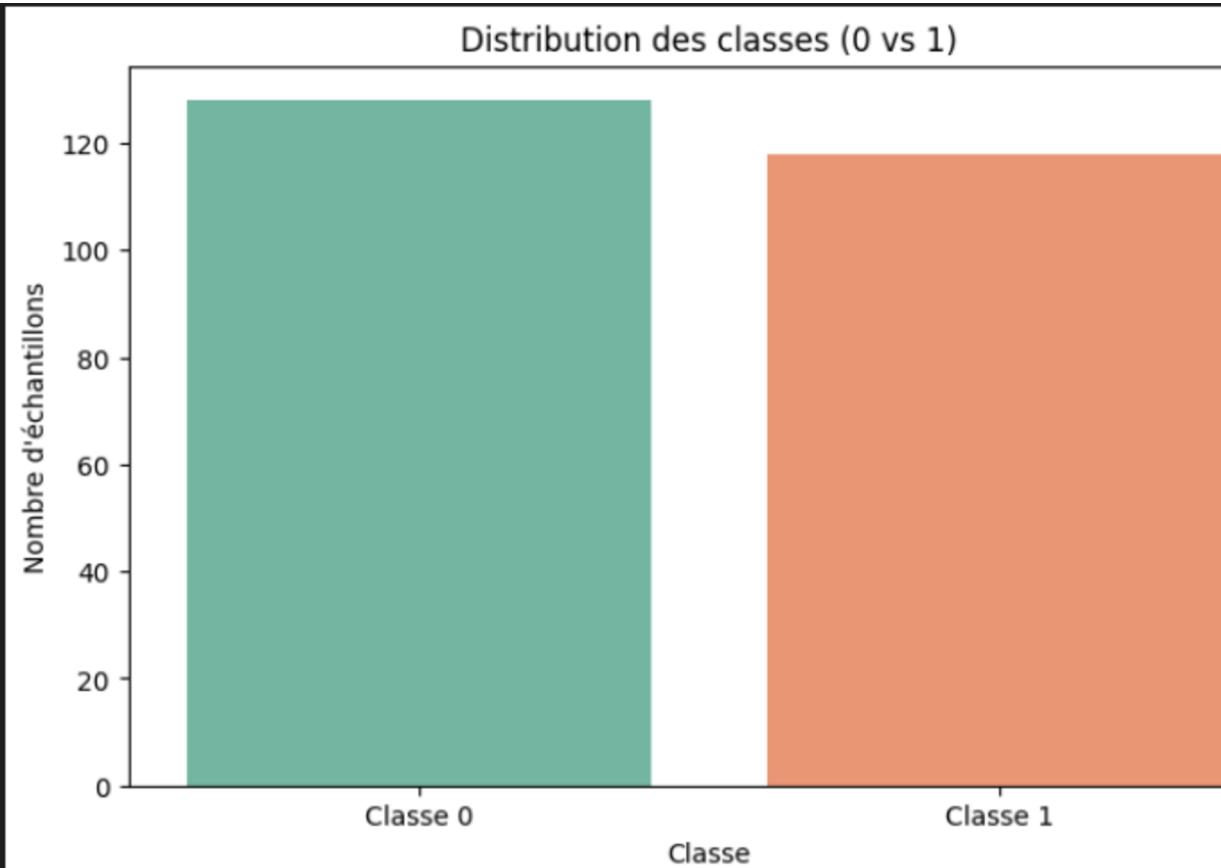
# Afficher le résultat
print("Colonnes avec des valeurs manquantes :")
print(colonnes_avec_nulles)

[✓] 0.0s
```

Colonnes avec des valeurs manquantes :

W1	19
W2	14
W3	45
Employed	2
	dtype: int64

Dataset équilibré ?



```
Distribution des classes :  
Class  
0    128  
1    118  
Name: count, dtype: int64  
  
Pourcentage par classe :  
Class  
0    52.03  
1    47.97
```

```
# Vérifier la distribution des classes  
class_distribution = df['Class'].value_counts()  
print("Distribution des classes :")  
print(class_distribution)  
  
# Afficher les pourcentages  
class_percentage = df['Class'].value_counts(normalize=True) * 100  
print("\nPourcentage par classe :")  
print(class_percentage.round(2))  
  
# Visualisation sous forme de graphique à barres  
plt.figure(figsize=(8, 5))  
sns.countplot(x='Class', data=df, palette='Set2')  
plt.title("Distribution des classes (0 vs 1)")  
plt.xlabel("Classe")  
plt.ylabel("Nombre d'échantillons")  
plt.xticks([0, 1], ['Classe 0', 'Classe 1'])  
plt.show()  
  
# Affichage conditionnel : Équilibré ou Déséquilibré ?  
if abs(class_percentage[0] - class_percentage[1]) < 10:  
    print("\n Le dataset est approximativement équilibré.")  
else:  
    print("\n Le dataset est déséquilibré.")
```

--> le dataset est approximativement équilibré avec une légère majorité pour la classe 0

→ Le F1 Score ne favorise pas la classe majoritaire .
permet de comparer objectivement plusieurs modèles,

2. Prétraitement des données :

Suppression des duplications :

```
# supprimer les lignes dupliquées  
df= df.drop_duplicates()
```

Suppression des outliers :

```
# Supprimer les outliers du DataFrame  
df_clean = df[~((df['Score'] < lower_bound) | (df['Score'] > upper_bound))].copy()  
print(f"Nombre de lignes après suppression : {len(df_clean)}")
```

```
Nombre d'outliers détectés dans 'Score' : 4  
Nombre de lignes après suppression : 242
```

traitement de données maquantes:

- W1 et W2 : Variables numériques valeurs manquantes <7%.

→ Recommandation : remplacer par la médiane pour préserver la distribution sans être influencé par des valeurs extrêmes.

- W3 : Variable numérique avec 45 valeurs manquantes (~18%).

→ Recommandation : remplacer par la médiane ou le mode.

→ Si la colonne est faiblement corrélée avec Class, une suppression pourrait être envisagée pour simplifier le modèle.

- Employed : Variable binaire avec 2 valeurs manquantes.

→ Recommandation : remplacer par le mode, car les valeurs manquantes sont minimes.

```
# Remplacer W1, W2, W3 par la médiane
df['W1'].fillna(df['W1'].median(), inplace=True)
df['W2'].fillna(df['W2'].median(), inplace=True)
df['W3'].fillna(df['W3'].median(), inplace=True)

# Remplacer Employed par le mode
df['Employed'].fillna(df['Employed'].mode()[0], inplace=True)

# Vérifier après traitement
print("\nValeurs manquantes après traitement :")
print(df[['W1', 'W2', 'W3', 'Employed']].isnull().sum())

✓ 0.0s

Valeurs manquantes après traitement :
W1      0
W2      0
W3      0
Employed      0
dtype: int64
```

Filtrage des colonnes basé sur la corrélation avec target

Corrélation très forte (≥ 0.5)

Variable	Corrélation
IM5	0.662345
IM2	0.631967
IM6	0.631752
IM1	0.565969
IM4	0.562259

Corrélation négative

Variable	Corrélation
Major	-0.127969
Level	-0.193994

Corrélation modérée (entre 0.3 et 0.5)

Variable	Corrélation
PSD4	0.489854
IE3	0.463506
IM3	0.458289
IE5	0.441001
PSD3	0.389568
TL2	0.379572
IE4	0.379088
PSD5	0.370032
SMSK2	0.365342
W2	0.360766
SMSK1	0.352622
IE1	0.342317
SMSK3	0.342079
RAS3	0.336907
PSD2	0.333160
TL1	0.309792
RAS4	0.307397

Corrélation faible (entre 0 et 0.3)

Variable	Corrélation
RAS1	0.293767
PSD1	0.291010
SMSK4	0.290698
TL3	0.289496
IE2	0.234836
RAS5	0.212977
RAS2	0.200330
W1	0.155886
Employed	0.111538
Nationality	0.079912
Score	0.068210
Gender	0.057801

sélection des colonnes pertinentes

```
# Filtrer les colonnes corrélées avec Class (> 0.3)
threshold = 0.3
important_columns = correlation_with_class[abs(correlation_with_class) > threshold].index.tolist()

# Supprimer 'Class' de cette liste si elle y est
if 'Class' in important_columns:
    important_columns.remove('Class')

# Ajouter 'major' et 'level' à la liste des colonnes importantes
important_columns.extend(['Major', 'Level'])

# Afficher les colonnes sélectionnées
print("\nColonnes sélectionnées pour le modèle :")
print(important_columns)

# Créer le nouveau DataFrame
X = df[important_columns]
y = df['Class']
```

Les 23 colonnes sélectionnées sont :

**['IM5', 'IM2', 'IM6', 'IM1', 'IM4',
'PSD4', 'IE3', 'IM3', 'IE5', 'PSD3',
'TL2',
'IE4', 'PSD5', 'SMSK2', 'W2',
'SMSK1', 'IE1', 'SMSK3', 'RAS3',
'PSD2',
'TL1', 'RAS4', 'Major', 'Level']**

On fixe un seuil à 0.3 : seules les colonnes ayant une corrélation absolue supérieure à 0.3 avec Class seront retenues en conservant Major et Level, on prend en compte des facteurs contextuels qui peuvent influencer indirectement la réussite, même si leur corrélation linéaire est faible ou négative.

Séparation des caractéristiques et target

```
# Créer le nouveau DataFrame  
X = df[important_columns]  
y = df['Class']
```

X : un tableau de features (variables indépendantes) → une matrice NumPy.
y : un tableau de labels (variable dépendante) → un tableau de 1 et 0

3. Modélisation

Séparation de l'apprentissage de l'évaluation

```
# Division train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- **80 % (données d'entraînement)** : servent à entraîner le modèle.
- **20 % (données de test)** : servent à évaluer la performance du modèle sur des données nouvelles.

But:

- . Prévenir le sur-apprentissage (overfitting)
- Mesurer la capacité de généralisation

```

# Définir les modèles
models = {
    "Random Forest": RandomForestClassifier(random_state=42),
    "SVM": SVC(kernel='rbf', random_state=42),
    "KNN": KNeighborsClassifier(n_neighbors=5),
    "Logistic Regression": LogisticRegression(penalty='l2', solver='liblinear', random_state=42),
    "Neural Network (MLP)": MLPClassifier(hidden_layer_sizes=(64, 32), max_iter=1000, random_state=42),
    "XGBoost": XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)
}

# Normalisation (important pour SVM, KNN et MLP)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Entrainer et évaluer chaque modèle
results = {}

for name, model in models.items():
    print(f"\nEntraînement du modèle : {name}")

    # Pas besoin de normaliser pour Random Forest, Logistic Regression et XGBoost
    if name in ["Random Forest", "Logistic Regression", "XGBoost"]:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
    else:
        model.fit(X_train_scaled, y_train)
        y_pred = model.predict(X_test_scaled)

    f1 = f1_score(y_test, y_pred, average='macro')
    results[name] = f1
    print(f"{name} - F1 Score (Macro): {f1:.4f}")

```

Modèles testés :

- **Random Forest**
- **SVM**
- **KNN**
- **Régression Logistique (L2)**
- **Réseau de Neurones (MLP)**
- **XGBoost**

NB: Standardisation appliquée pour les modèles basés sur la distance d'approximité

Pourquoi ces modèles

Random Forest

- Gère bien les données hétérogènes
- Robuste au surapprentissage grâce à l'agrégation d'arbres
- Donne une bonne précision même sans réglages complexes

Logistic Regression

- Facile à interpréter
- Rapide à entraîner

K-Nearest Neighbors (KNN)

- Simple et intuitif
- Pas d'entraînement : juste du calcul de distance à l'inférence

Support Vector Machine (SVM)

- Performant surtout dans des espaces de petite ou moyenne dimension
- Capable de gérer des frontières complexes avec un noyau (kernel)

XGBoost / LightGBM (Gradient Boosting)

- Très performant pour les données tabulaires
- Excellente précision avec un bon réglage
- Supporte le traitement des valeurs manquantes

Comparaison en terme de f1 score:

$$F1 = \frac{2 * (\text{précision} * \text{rappel})}{\text{précision} + \text{rappel}}$$

Le F1-score est utilisé car :

- Il évalue chaque classe indépendamment, puis fait la moyenne.
- Cela permet de ne pas favoriser une classe par rapport à une autre.
- Il offre une vision plus fine que la simple accuracy, en tenant compte à la fois des faux positifs et des faux négatifs.

Résultat

```
import matplotlib.pyplot as plt

# Afficher les scores F1 obtenus
plt.figure(figsize=(10, 6))
plt.bar(results.keys(), results.values(), color=['skyblue', 'salmon', 'lightgreen', 'orange', 'red', 'blue'])
plt.title("Comparaison des modèles par F1 Score (Macro)")
plt.ylabel("F1 Score")
plt.ylim(0.7, 1.0)
plt.grid(True)
plt.show()
```

Entraînement du modèle : Random Forest
Random Forest - F1 Score (Macro): 0.9600

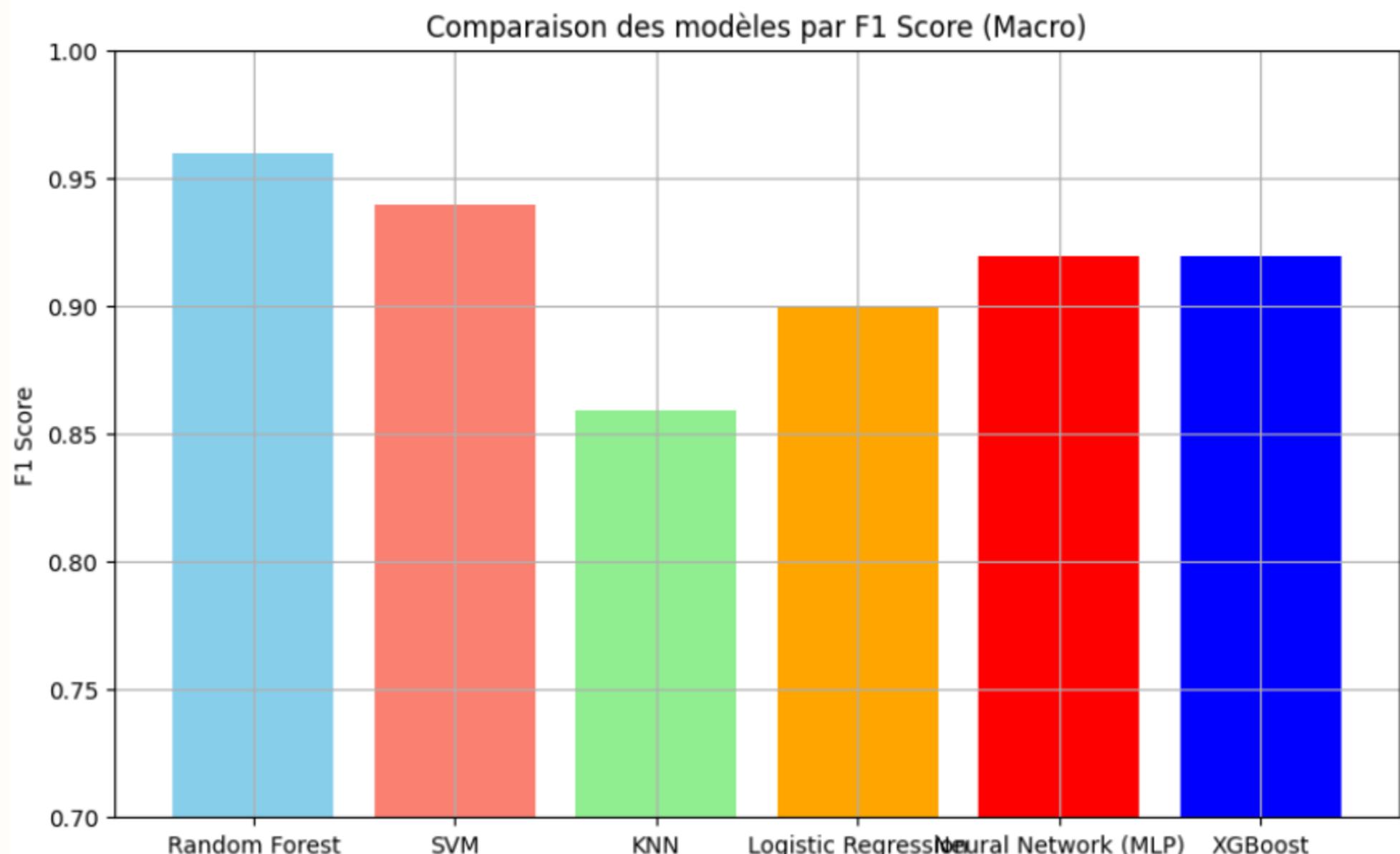
Entraînement du modèle : SVM
SVM - F1 Score (Macro): 0.9400

Entraînement du modèle : KNN
KNN - F1 Score (Macro): 0.8595

Entraînement du modèle : Logistic Regression
Logistic Regression - F1 Score (Macro): 0.9000

Entraînement du modèle : Neural Network (MLP)
Neural Network (MLP) - F1 Score (Macro): 0.9199

Entraînement du modèle : XGBoost
XGBoost - F1 Score (Macro): 0.9199



Analyse des Résultats

- Random Forest est le meilleur modèle avec un F1 Score de 0.96, car il gère bien la complexité et les petites données.
- SVM (0.94), XGBoost (0.92) sont aussi performants
- MLP (0.92) performant mais demande plus de réglages et temps d'entraînement.
- Logistic Regression (0.90) est simple mais efficace.
- KNN (0.86) est le moins bon, sensible aux données bruitées.

Conclusion

- **Random Forest recommandé pour ce type de données**
- **Pipeline complet : Nettoyage → Sélection → Modélisation → Évaluation**
- **F1-score comme métrique pertinente pour évaluer le modèle**