

**Université Tunis-Dauphine**

Master Big Data et Intelligence Artificielle

# **Projet : Détection de la Maladie de Parkinson**

**Classification binaire et clustering appliqués à des données biomédicales**

**Réalisé par : Fatma Chahed, Aziz Dhif , Ghada Dhaoui**

**Encadrant : Dr. Sana Louhichi**

**Année universitaire : 2024–2025**

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Exploration et traitement des données</b>	<b>3</b>
2.1	Vue d'ensemble et statistique descriptive . . . . .	3
2.2	Traitement des Valeurs Manquantes . . . . .	6
2.3	Distribution des variables (graphiques) . . . . .	7
2.4	La corrélation entre les variables . . . . .	8
2.5	Traitement des outliers . . . . .	9
2.6	Mise à l'échelle des features . . . . .	10
<b>3</b>	<b>Construction des modèles</b>	<b>11</b>
3.1	Introduction . . . . .	11
3.2	Séparation des données pour l'entraînement et les tests . . . . .	11
3.3	Modélisation avec différents modèles . . . . .	11
3.4	Évaluation des modèles . . . . .	15
3.5	Résumé des modèles testés . . . . .	17
<b>4</b>	<b>Améliorations des modèles</b>	<b>19</b>
4.1	Introduction . . . . .	19
4.2	La performance initiale des modèles . . . . .	19
4.3	Choix des meilleurs hyperparamètres . . . . .	20
4.4	Application de la méthode SMOTE . . . . .	22
4.5	Application de l'ACP . . . . .	23
4.6	Test du Deep Learning . . . . .	25
<b>5</b>	<b>Prédire la gravité de la maladie</b>	<b>29</b>
5.1	Vue d'ensemble et statistique descriptive . . . . .	29
5.2	Distribution des variables (graphiques) . . . . .	31
5.3	Mise à l'échelle des features . . . . .	32
5.4	Construction des Modèles . . . . .	32
5.5	Visualisation des clusters . . . . .	33
5.6	Évaluation des modèles de clustering . . . . .	33
5.7	Quelques statistiques des groupes . . . . .	34
<b>6</b>	<b>Application web de prédiction</b>	<b>36</b>
6.1	Objectif . . . . .	36
6.2	Langage et technologies utilisées . . . . .	36
6.3	Présentation de l'interface utilisateur . . . . .	36
6.4	Étapes de mise en œuvre . . . . .	37

# Chapitre 1

## Introduction

La maladie de Parkinson est une pathologie neurodégénérative chronique qui affecte principalement le système moteur. Elle se manifeste par des tremblements, une rigidité musculaire, des troubles de la posture et une lenteur dans les mouvements. Son diagnostic précoce est essentiel afin d'améliorer la qualité de vie des patients et de ralentir la progression de la maladie grâce à une prise en charge adaptée.

Dans le cadre de ce projet, nous exploitons des données biomédicales pour développer un système d'aide au diagnostic de la maladie de Parkinson à l'aide de méthodes d'intelligence artificielle.

Le premier objectif consiste à résoudre un problème de **classification binaire** : à partir de caractéristiques extraites de patients, il s'agit de prédire la présence ou non de la maladie (étiquettes 0 ou 1). Pour ce faire, nous avons utilisé **des techniques d'apprentissage supervisé** en mettant en œuvre plusieurs modèles de classification.

En complément, un second ensemble de données est analysé dans une perspective non supervisée. Nous y appliquons des **algorithmes de clustering** dans le but d'estimer le niveau de gravité de la maladie pour chaque patient. Cette approche vise à regrouper les patients selon des caractéristiques similaires, permettant ainsi d'identifier des profils types ou des stades d'évolution de la maladie de Parkinson.

Ce rapport présente l'approche méthodologique adoptée, les techniques de prétraitement utilisées, les modèles déployés pour la classification et le clustering, ainsi qu'une analyse critique des résultats obtenus.

# Chapitre 2

## Exploration et traitement des données

### 2.1 Vue d'ensemble et statistique descriptive

La première étape consiste à charger les données depuis le fichier csv donné "parkinson.data" en utilisant la bibliothèque pandas.

```
1 import pandas as pd
2 # Charger les donnees
3 df = pd.read_csv("parkinson.data")
4 # Afficher les premieres lignes
5 print(df.head())
```

	name	MDVP:Fo(Hz)	MDVP:Fhi(Hz)	MDVP:Flo(Hz)	MDVP:Jitter(%)
0	phon_R01_S01_1	119.992	157.302	74.997	0.00784
1	phon_R01_S01_2	122.400	148.650	113.819	0.00968
2	phon_R01_S01_3	116.682	131.111	111.555	0.01050
3	phon_R01_S01_4	116.676	137.871	111.366	0.00997
4	phon_R01_S01_5	116.014	141.781	110.655	0.01284

5 rows × 24 columns

FIGURE 2.1 – visualisation de quelques données

Ensuite, nous affichons la taille du jeu de données qui est égales à (195, 24), c'est à dire 195 lignes et 24 colonnes, ainsi que le nom de toutes les colonnes.

```
1 # dimensions des donnees
2 print("la taille du dataframe est:" ,df1.shape)
3 # affichage des colonnes
4 print("les colonnes du dataframe sont:", df1.columns)
```

Le tableau 2.1 présente les caractéristiques des données ainsi qu'une brève description de chaque variable.

Nom de la colonne	Signification simplifiée
name	Nom du patient et numéro d'enregistrement
MDVP:Fo(Hz)	Fréquence vocale moyenne (en Hz)
MDVP:Fhi(Hz)	Fréquence vocale maximale (en Hz)
MDVP:Flo(Hz)	Fréquence vocale minimale (en Hz)
MDVP:Jitter(%)	Variation relative de la fréquence
MDVP:Jitter(Abs)	Variation absolue de la fréquence
MDVP:RAP	Moyenne des variations rapides de fréquence
MDVP:PPQ	Variation périodique de la fréquence
Jitter:DDP	Moyenne sur 3 périodes de RAP
MDVP:Shimmer	Variation relative de l'amplitude
MDVP:Shimmer(dB)	Variation d'amplitude en décibels
Shimmer:APQ3	Moyenne sur 3 périodes de la variation d'amplitude
Shimmer:APQ5	Moyenne sur 5 périodes de la variation d'amplitude
MDVP:APQ	Moyenne absolue de la variation d'amplitude
Shimmer:DDA	Moyenne de trois APQ3
NHR	Rapport entre le bruit et le son (voix)
HNHR	Rapport harmonique sur bruit (qualité sonore)
status	0 = personne saine, 1 = malade de Parkinson
RPDE	Mesure de complexité du signal vocal
D2	Autre mesure de complexité non linéaire
DFA	Mesure fractale de l'évolution du signal
spread1	Variabilité dans les fréquences vocales
spread2	Autre indicateur de variation vocale
PPE	Degré d'imprécision dans la fréquence vocale

TABLE 2.1 – Description simplifiée des attributs du jeu de données Parkinson

Nous utilisons la méthode `.describe()` pour visualiser plusieurs informations sur la distribution des données, telles que le nombre total de valeurs dans chaque colonne, ainsi que la moyenne, l'écart-type, le minimum, le maximum et les quantiles.

```
1 # decrire les donnees
2 df1.describe()
```

	MDVP:F0(Hz)	MDVP:F1(Hz)	MDVP:F2(Hz)	MDVP:Jitter(%)	MDVP:Jitter(Abs)	MDVP:RAP	MDVP:PPQ	Jitter:DDP
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
mean	154.228641	197.104918	116.324631	0.006220	0.000044	0.003306	0.003446	0.009920
std	41.390065	91.491548	43.521413	0.004848	0.000035	0.002968	0.002759	0.008903
min	88.333000	102.145000	65.476000	0.001680	0.000007	0.000680	0.000920	0.002040
25%	117.572000	134.862500	84.291000	0.003460	0.000020	0.001660	0.001860	0.004985
50%	148.790000	175.829000	104.315000	0.004940	0.000030	0.002500	0.002690	0.007490
75%	182.769000	224.205500	140.018500	0.007365	0.000060	0.003835	0.003955	0.011505
max	260.105000	592.030000	239.170000	0.033160	0.000260	0.021440	0.019580	0.064330

8 rows x 23 columns

FIGURE 2.2 – la description des données

La colonne **name** est l'identifiant unique de chaque ligne qui représente un enregistrement d'une personne.

```
1 # decrire les donnees
2 df1['name'].value_counts()
```

Selon la commande `.info()`, aucune colonne ne présente de valeurs nulles dans le Data-Frame. Par conséquent, nous n'aurons pas besoin de traiter les valeurs manquantes pour cette analyse.

```
1 df1.info()
```

#	Column	Non-Null Count	Dtype
0	name	195 non-null	object
1	MDVP:F0(Hz)	195 non-null	float64
2	MDVP:F1(Hz)	195 non-null	float64
3	MDVP:F2(Hz)	195 non-null	float64
4	MDVP:Jitter(%)	195 non-null	float64
5	MDVP:Jitter(Abs)	195 non-null	float64
6	MDVP:RAP	195 non-null	float64
7	MDVP:PPQ	195 non-null	float64
8	Jitter:DDP	195 non-null	float64
9	MDVP:Shimmer	195 non-null	float64
10	MDVP:Shimmer(dB)	195 non-null	float64
11	Shimmer:APQ3	195 non-null	float64
12	Shimmer:APQ5	195 non-null	float64
13	MDVP:APQ	195 non-null	float64
14	Shimmer:DDA	195 non-null	float64
15	NHR	195 non-null	float64
16	HNR	195 non-null	float64
17	status	195 non-null	int64
18	RPDE	195 non-null	float64
19	DFA	195 non-null	float64
20	spread1	195 non-null	float64
21	spread2	195 non-null	float64
22	D2	195 non-null	float64
23	PPE	195 non-null	float64

dtypes: float64(22), int64(1), object(1)  
memory usage: 36.7+ KB

Nous allons maintenant déterminer le nombre d'enregistrements par personne :

```
1 # Extraire l'identifiant (S01, S02...) a partir de 'name'
2 df1['person_id'] = df1['name'].str.extract(r'(S\d{2})') #
   extrait 'S01', 'S02', etc.
3 count_per_person = df1['person_id'].value_counts()
4 print(count_per_person)
```

person_id	
S21	7
S27	7
S35	7
S01	6
S06	6
S07	6
S04	6
S02	6
S10	6
S13	6
S17	6

FIGURE 2.3 – Nombre d’enregistrements par personne

D’après le résultat obtenu, chaque personne possède de 6 à 7 enregistrements en moyenne.

## Valeur cible

```
1 df1['status'].unique()
```

La colonne `status` est la variable cible que nous cherchons à prédire. Elle prend deux valeurs : 0 ou 1. Il s’agit donc d’un problème de classification binaire, où :

- $y = 1$ , si la personne est malade.
- $y = 0$ , si la personne est non malade.

```
1 df1['status'].value_counts()
```

	count
status	
1	147
0	48

Les données sont réparties comme suit :

- **147** observations où `status` = 1 (malade),
- **48** observations où `status` = 0 (non malade).

## 2.2 Traitement des Valeurs Manquantes

Affichage du nombre de valeurs nulles par colonne :

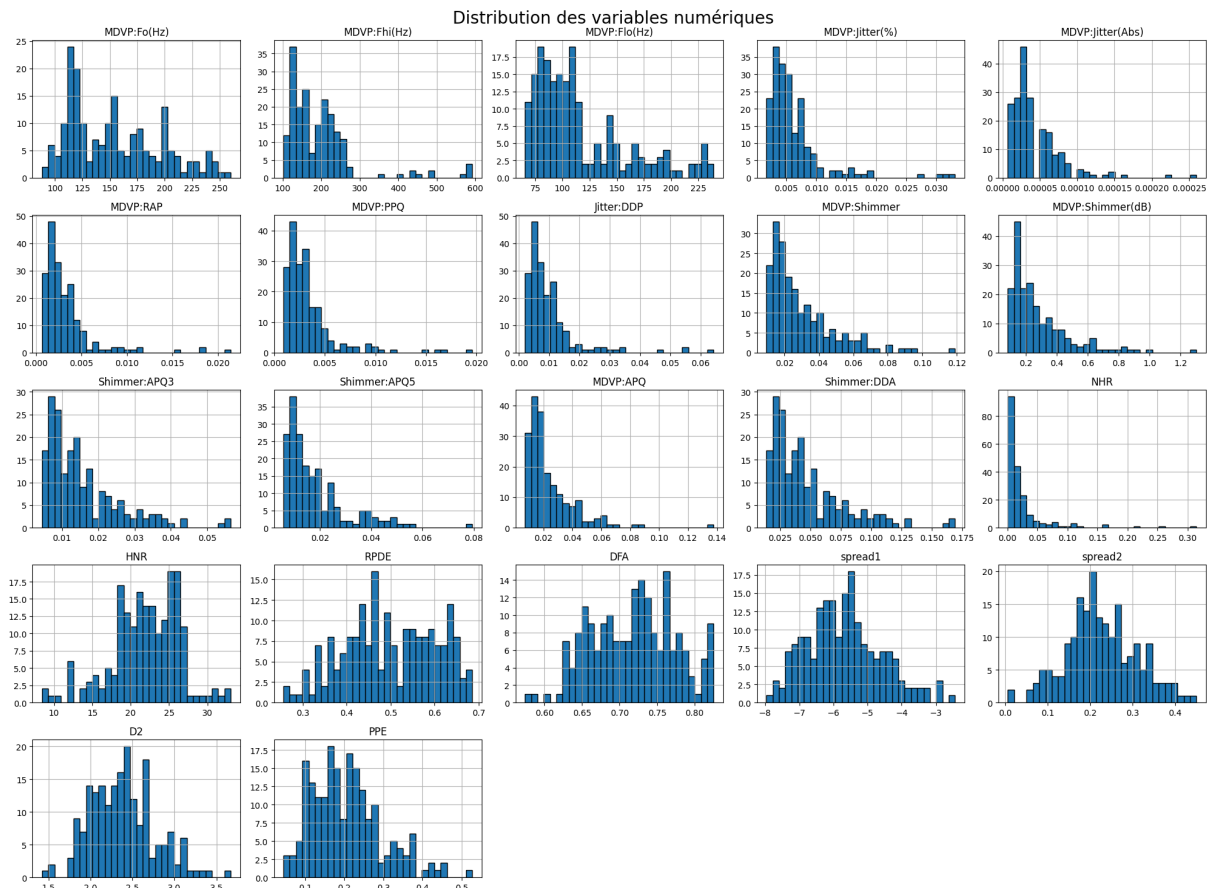
```
1 print(df1.isnull().sum())
```

name	0
MDVP:F0(Hz)	0
MDVP:F1(Hz)	0
MDVP:F2(Hz)	0
MDVP:F3(Hz)	0
MDVP:F4(Hz)	0
MDVP:F5(Hz)	0
MDVP:F6(Hz)	0
MDVP:F7(Hz)	0
MDVP:F8(Hz)	0
MDVP:F9(Hz)	0
MDVP:F10(Hz)	0
MDVP:F11(Hz)	0
MDVP:F12(Hz)	0
MDVP:F13(Hz)	0
MDVP:F14(Hz)	0
MDVP:F15(Hz)	0
MDVP:F16(Hz)	0
MDVP:F17(Hz)	0
MDVP:F18(Hz)	0
MDVP:F19(Hz)	0
MDVP:F20(Hz)	0
MDVP:F21(Hz)	0
MDVP:F22(Hz)	0
MDVP:F23(Hz)	0
MDVP:F24(Hz)	0
MDVP:F25(Hz)	0
MDVP:F26(Hz)	0
MDVP:F27(Hz)	0
MDVP:F28(Hz)	0
MDVP:F29(Hz)	0
MDVP:F30(Hz)	0
MDVP:F31(Hz)	0
MDVP:F32(Hz)	0
MDVP:F33(Hz)	0
MDVP:F34(Hz)	0
MDVP:F35(Hz)	0
MDVP:F36(Hz)	0
MDVP:F37(Hz)	0
MDVP:F38(Hz)	0
MDVP:F39(Hz)	0
MDVP:F40(Hz)	0
MDVP:F41(Hz)	0
MDVP:F42(Hz)	0
MDVP:F43(Hz)	0
MDVP:F44(Hz)	0
MDVP:F45(Hz)	0
MDVP:F46(Hz)	0
MDVP:F47(Hz)	0
MDVP:F48(Hz)	0
MDVP:F49(Hz)	0
MDVP:F50(Hz)	0
MDVP:F51(Hz)	0
MDVP:F52(Hz)	0
MDVP:F53(Hz)	0
MDVP:F54(Hz)	0
MDVP:F55(Hz)	0
MDVP:F56(Hz)	0
MDVP:F57(Hz)	0
MDVP:F58(Hz)	0
MDVP:F59(Hz)	0
MDVP:F60(Hz)	0
MDVP:F61(Hz)	0
MDVP:F62(Hz)	0
MDVP:F63(Hz)	0
MDVP:F64(Hz)	0
MDVP:F65(Hz)	0
MDVP:F66(Hz)	0
MDVP:F67(Hz)	0
MDVP:F68(Hz)	0
MDVP:F69(Hz)	0
MDVP:F70(Hz)	0
MDVP:F71(Hz)	0
MDVP:F72(Hz)	0
MDVP:F73(Hz)	0
MDVP:F74(Hz)	0
MDVP:F75(Hz)	0
MDVP:F76(Hz)	0
MDVP:F77(Hz)	0
MDVP:F78(Hz)	0
MDVP:F79(Hz)	0
MDVP:F80(Hz)	0
MDVP:F81(Hz)	0
MDVP:F82(Hz)	0
MDVP:F83(Hz)	0
MDVP:F84(Hz)	0
MDVP:F85(Hz)	0
MDVP:F86(Hz)	0
MDVP:F87(Hz)	0
MDVP:F88(Hz)	0
MDVP:F89(Hz)	0
MDVP:F90(Hz)	0
MDVP:F91(Hz)	0
MDVP:F92(Hz)	0
MDVP:F93(Hz)	0
MDVP:F94(Hz)	0
MDVP:F95(Hz)	0
MDVP:F96(Hz)	0
MDVP:F97(Hz)	0
MDVP:F98(Hz)	0
MDVP:F99(Hz)	0
MDVP:F100(Hz)	0
MDVP:F101(Hz)	0
MDVP:F102(Hz)	0
MDVP:F103(Hz)	0
MDVP:F104(Hz)	0
MDVP:F105(Hz)	0
MDVP:F106(Hz)	0
MDVP:F107(Hz)	0
MDVP:F108(Hz)	0
MDVP:F109(Hz)	0
MDVP:F110(Hz)	0
MDVP:F111(Hz)	0
MDVP:F112(Hz)	0
MDVP:F113(Hz)	0
MDVP:F114(Hz)	0
MDVP:F115(Hz)	0
MDVP:F116(Hz)	0
MDVP:F117(Hz)	0
MDVP:F118(Hz)	0
MDVP:F119(Hz)	0
MDVP:F120(Hz)	0
MDVP:F121(Hz)	0
MDVP:F122(Hz)	0
MDVP:F123(Hz)	0
MDVP:F124(Hz)	0
MDVP:F125(Hz)	0
MDVP:F126(Hz)	0
MDVP:F127(Hz)	0
MDVP:F128(Hz)	0
MDVP:F129(Hz)	0
MDVP:F130(Hz)	0
MDVP:F131(Hz)	0
MDVP:F132(Hz)	0
MDVP:F133(Hz)	0
MDVP:F134(Hz)	0
MDVP:F135(Hz)	0
MDVP:F136(Hz)	0
MDVP:F137(Hz)	0
MDVP:F138(Hz)	0
MDVP:F139(Hz)	0
MDVP:F140(Hz)	0
MDVP:F141(Hz)	0
MDVP:F142(Hz)	0
MDVP:F143(Hz)	0
MDVP:F144(Hz)	0
MDVP:F145(Hz)	0
MDVP:F146(Hz)	0
MDVP:F147(Hz)	0
MDVP:F148(Hz)	0
MDVP:F149(Hz)	0
MDVP:F150(Hz)	0
MDVP:F151(Hz)	0
MDVP:F152(Hz)	0
MDVP:F153(Hz)	0
MDVP:F154(Hz)	0
MDVP:F155(Hz)	0
MDVP:F156(Hz)	0
MDVP:F157(Hz)	0
MDVP:F158(Hz)	0
MDVP:F159(Hz)	0
MDVP:F160(Hz)	0
MDVP:F161(Hz)	0
MDVP:F162(Hz)	0
MDVP:F163(Hz)	0
MDVP:F164(Hz)	0
MDVP:F165(Hz)	0
MDVP:F166(Hz)	0
MDVP:F167(Hz)	0
MDVP:F168(Hz)	0
MDVP:F169(Hz)	0
MDVP:F170(Hz)	0
MDVP:F171(Hz)	0
MDVP:F172(Hz)	0
MDVP:F173(Hz)	0
MDVP:F174(Hz)	0
MDVP:F175(Hz)	0
MDVP:F176(Hz)	0
MDVP:F177(Hz)	0
MDVP:F178(Hz)	0
MDVP:F179(Hz)	0
MDVP:F180(Hz)	0
MDVP:F181(Hz)	0
MDVP:F182(Hz)	0
MDVP:F183(Hz)	0
MDVP:F184(Hz)	0
MDVP:F185(Hz)	0
MDVP:F186(Hz)	0
MDVP:F187(Hz)	0
MDVP:F188(Hz)	0
MDVP:F189(Hz)	0
MDVP:F190(Hz)	0
MDVP:F191(Hz)	0
MDVP:F192(Hz)	0
MDVP:F193(Hz)	0
MDVP:F194(Hz)	0
MDVP:F195(Hz)	0
MDVP:F196(Hz)	0
MDVP:F197(Hz)	0
MDVP:F198(Hz)	0
MDVP:F199(Hz)	0
MDVP:F200(Hz)	0
MDVP:F201(Hz)	0
MDVP:F202(Hz)	0
MDVP:F203(Hz)	0
MDVP:F204(Hz)	0
MDVP:F205(Hz)	0
MDVP:F206(Hz)	0
MDVP:F207(Hz)	0
MDVP:F208(Hz)	0
MDVP:F209(Hz)	0
MDVP:F210(Hz)	0
MDVP:F211(Hz)	0
MDVP:F212(Hz)	0
MDVP:F213(Hz)	0
MDVP:F214(Hz)	0
MDVP:F215(Hz)	0
MDVP:F216(Hz)	0
MDVP:F217(Hz)	0
MDVP:F218(Hz)	0
MDVP:F219(Hz)	0
MDVP:F220(Hz)	0
MDVP:F221(Hz)	0
MDVP:F222(Hz)	0
MDVP:F223(Hz)	0
MDVP:F224(Hz)	0
MDVP:F225(Hz)	0
MDVP:F226(Hz)	0
MDVP:F227(Hz)	0
MDVP:F228(Hz)	0
MDVP:F229(Hz)	0
MDVP:F230(Hz)	0
MDVP:F231(Hz)	0
MDVP:F232(Hz)	0
MDVP:F233(Hz)	0
MDVP:F234(Hz)	0
MDVP:F235(Hz)	0
MDVP:F236(Hz)	0
MDVP:F237(Hz)	0
MDVP:F238(Hz)	0
MDVP:F239(Hz)	0
MDVP:F240(Hz)	0
MDVP:F241(Hz)	0
MDVP:F242(Hz)	0
MDVP:F243(Hz)	0
MDVP:F244(Hz)	0
MDVP:F245(Hz)	0
MDVP:F246(Hz)	0
MDVP:F247(Hz)	0
MDVP:F248(Hz)	0
MDVP:F249(Hz)	0
MDVP:F250(Hz)	0
MDVP:F251(Hz)	0
MDVP:F252(Hz)	0
MDVP:F253(Hz)	0
MDVP:F254(Hz)	0
MDVP:F255(Hz)	0
MDVP:F256(Hz)	0
MDVP:F257(Hz)	0
MDVP:F258(Hz)	0
MDVP:F259(Hz)	0
MDVP:F260(Hz)	0
MDVP:F261(Hz)	0
MDVP:F262(Hz)	0
MDVP:F263(Hz)	0
MDVP:F264(Hz)	0
MDVP:F265(Hz)	0
MDVP:F266(Hz)	0
MDVP:F267(Hz)	0
MDVP:F268(Hz)	0
MDVP:F269(Hz)	0
MDVP:F270(Hz)	0
MDVP:F271(Hz)	0
MDVP:F272(Hz)	0
MDVP:F273(Hz)	0
MDVP:F274(Hz)	0
MDVP:F275(Hz)	0
MDVP:F276(Hz)	0
MDVP:F277(Hz)	0
MDVP:F278(Hz)	0
MDVP:F279(Hz)	0
MDVP:F280(Hz)	0
MDVP:F281(Hz)	0
MDVP:F282(Hz)	0
MDVP:F283(Hz)	0
MDVP:F284(Hz)	0
MDVP:F285(Hz)	0
MDVP:F286(Hz)	0
MDVP:F287(Hz)	0
MDVP:F288(Hz)	0
MDVP:F289(Hz)	0
MDVP:F290(Hz)	0
MDVP:F291(Hz)	0
MDVP:F292(Hz)	0
MDVP:F293(Hz)	0
MDVP:F294(Hz)	0
MDVP:F295(Hz)	0
MDVP:F296(Hz)	0
MDVP:F297(Hz)	0
MDVP:F298(Hz)	0
MDVP:F299(Hz)	0
MDVP:F300(Hz)	0
MDVP:F301(Hz)	0
MDVP:F302(Hz)	0
MDVP:F303(Hz)	0
MDVP:F304(Hz)	0
MDVP:F305(Hz)	0
MDVP:F306(Hz)	0
MDVP:F307(Hz)	0
MDVP:F308(Hz)	0
MDVP:F309(Hz)	0
MDVP:F310(Hz)	0
MDVP:F311(Hz)	0
MDVP:F312(Hz)	0
MDVP:F313(Hz)	0
MDVP:F314(Hz)	0
MDVP:F315(Hz)	0
MDVP:F316(Hz)	0
MDVP:F317(Hz)	0
MDVP:F318(Hz)	0
MDVP:F319(Hz)	0
MDVP:F320(Hz)	0
MDVP:F321(Hz)	0
MDVP:F322(Hz)	0
MDVP:F323(Hz)	0
MDVP:F324(Hz)	0
MDVP:F325(Hz)	0
MDVP:F326(Hz)	0
MDVP:F327(Hz)	0
MDVP:F328(Hz)	0
MDVP:F329(Hz)	0
MDVP:F330(Hz)	0
MDVP:F331(Hz)	0
MDVP:F332(Hz)	0
MDVP:F333(Hz)	0
MDVP:F334(Hz)	0
MDVP:F335(Hz)	0
MDVP:F336(Hz)	0
MDVP:F337(Hz)	0
MDVP:F338(Hz)	0
MDVP:F339(Hz)	0
MDVP:F340(Hz)	0
MDVP:F341(Hz)	0
MDVP:F342(Hz)	0
MDVP:F343(Hz)	0
MDVP:F344(Hz)	0
MDVP:F345(Hz)	0
MDVP:F346(Hz)	0
MDVP:F347(Hz)	0
MDVP:F348(Hz)	0
MDVP:F349(Hz)	0
MDVP:F350(Hz)	0
MDVP:F351(Hz)	0
MDVP:F352(Hz)	0
MDVP:F353(Hz)	0
MDVP:F354(Hz)	0
MDVP:F355(Hz)	0
MDVP:F356(Hz)	0
MDVP:F357(Hz)	0
MDVP:F358(Hz)	0
MDVP:F359(Hz)	0
MDVP:F360(Hz)	0
MDVP:F361(Hz)	0
MDVP:F362(Hz)	0
MDVP:F363(Hz)	0
MDVP:F364(Hz)	0
MDVP:F365(Hz)	0
MDVP:F366(Hz)	0
MDVP:F367(Hz)	0
MDVP:F368(Hz)	0
MDVP:F369(Hz)	0
MDVP:F370(Hz)	0
MDVP:F371(Hz)	0
MDVP:F372(Hz)	0
MDVP:F373(Hz)	0
MDVP:F374(Hz)	0
MDVP:F375(Hz)	0
MDVP:F376(Hz)	0
MDVP:F377(Hz)	0
MDVP:F378(Hz)	0
MDVP:F379(Hz)	0
MDVP:F380(Hz)	0
MDVP:F381(Hz)	0
MDVP:F382(Hz)	0
MDVP:F383(Hz)	0
MDVP:F384(Hz)	0
MDVP:F385(Hz)	0
MDVP:F386(Hz)	0
MDVP:F387(Hz)	0
MDVP:F388(Hz)	0
MDVP:F389(Hz)	0
MDVP:F390(Hz)	0
MDVP:F391(Hz)	0
MDVP:F392(Hz)	0
MDVP:F393(Hz)	0
MDVP:F394(Hz)	0
MDVP:F395(Hz)	0
MDVP:F396(Hz)	0
MDVP:F397(Hz)	0
MDVP:F398(Hz)	0
MDVP:F399(Hz)	0
MDVP:F400(Hz)	0
MDVP:F401(Hz)	0
MDVP:F402(Hz)	0
MDVP:F403(Hz)	0
MDVP:F404(Hz)	0
MDVP:F405(Hz)	0
MDVP:F406(Hz)	0
MDVP:F407(Hz)	0
MDVP:F408(Hz)	0
MDVP:F409(Hz)	0
MDVP:F410(Hz)	0
MDVP:F411(Hz)	0
MDVP:F412(Hz)	0
MDVP:F413(Hz)	0
MDVP:F414(Hz)	0
MDVP:F415(Hz)	0
MDVP:F416(Hz)	0
MDVP:F417(Hz)	0
MDVP:F418(Hz)	0
MDVP:F419(Hz)	0
MDVP:F420(Hz)	0
MDVP:F421(Hz)	0
MDVP:F422(Hz)	0
MDVP:F423(Hz)	0
MDVP:F424(Hz)	0
MDVP:F425(Hz)	0
MDVP:F426(Hz)	0
MDVP:F427(Hz)	0
MDVP:F428(Hz)	0
MDVP:F429(Hz)	0
MDVP:F430(Hz)	0
MDVP:F431(Hz)	0
MDVP:F432(Hz)	0
MDVP:F433(Hz)	0
MDVP:F434(Hz)	0
MDVP:F435(Hz)	0
MDVP:F436(Hz)	0
MDVP:F437(Hz)	0
MDVP:F438(Hz)	0
MDVP:F439(Hz)	0
MDVP:F440(Hz)	0
MDVP:F441(Hz)	0
MDVP:F442(Hz)	0
MDVP:F443(Hz)	0
MDVP:F444(Hz)	0
MDVP:F445(Hz)	0
MDVP:F446(Hz)	0
MDVP:F447(Hz)	0
MDVP:F448(Hz)	0
MDVP:F449(Hz)	0
MDVP:F450(Hz)	0
MDVP:F451(Hz)	0
MDVP:F452(Hz)	0
MDVP:F453(Hz)	0
MDVP:F454(Hz)	0

## 2.3 Distribution des variables (graphiques)

Dans cette partie, nous visualisons la distribution des données numériques.

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3 # Exclure les colonnes non numériques
4 numeric_cols = df1.select_dtypes(include=['float64', 'int64']).
   columns.drop('status')
5 # Histogrammes
6 df1[numeric_cols].hist(figsize=(20, 15), bins=30, edgecolor='
   black')
7 plt.suptitle("Distribution des variables numériques", fontsize
   =20)
8 plt.tight_layout()
9 plt.show()
```



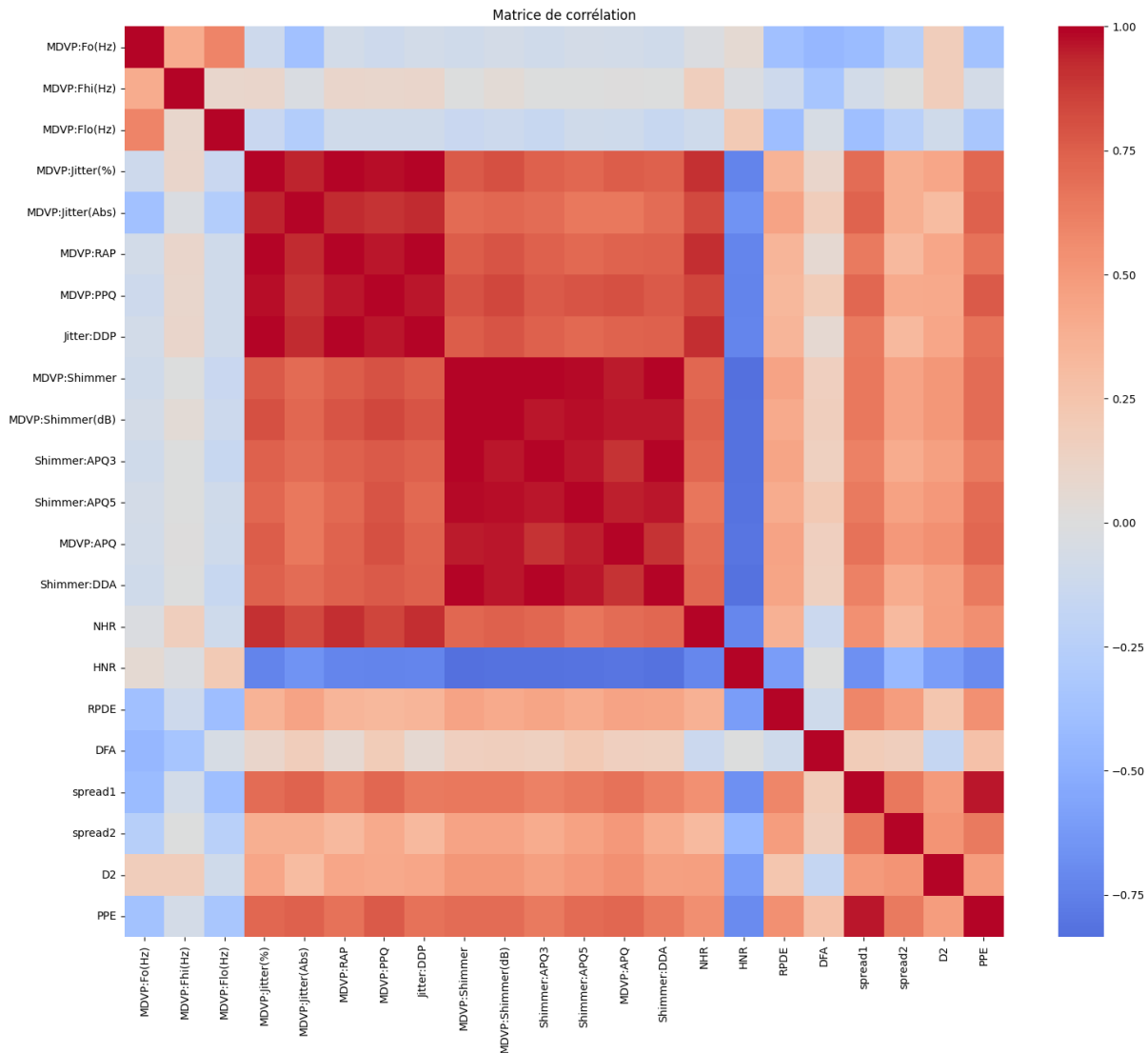
On constate que la majorité des variables suivent une loi normale centrée asymétrique sauf quelques une suivent une loi normale symétriques.



## 2.4 La corrélation entre les variables

Le code ci-dessous trace la matrice de corrélation entre les variables.

```
1 plt.figure(figsize=(18, 15))
2 sns.heatmap(df1[numeric_cols].corr(), annot=False, cmap="
  coolwarm", center=0)
3 plt.title("Matrice de corrélation")
4 plt.show()
```



### Interprétation :

Nous constatons que plusieurs variables sont fortement corrélées :

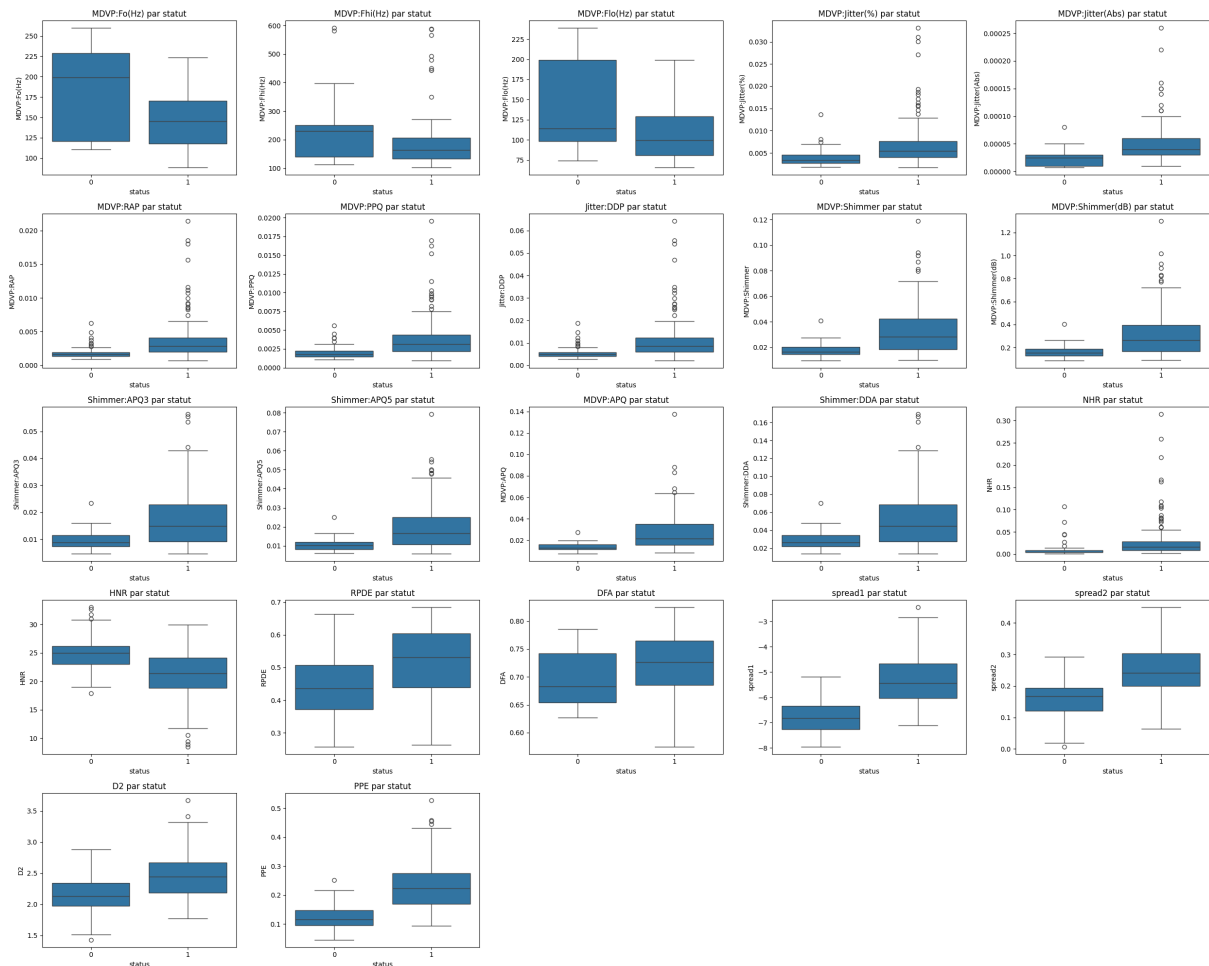
- Les caractéristiques MDVP telles que *Jitter (%)*, *Jitter (Abs)*, *RAP*, *PPQ*, *DDP* et *NHR*.
- De même, les caractéristiques MDVP liées à Shimmer, telles que Shimmer, Shimmer (dB), APQ3, APQ5, APQ et DDA.
- les caractéristiques PPE et speed2.

En revanche, il n'existe qu'une faible corrélation entre les autres variables.

## 2.5 Traitement des outliers

Ce genre de graphe (boxplot) permet de détecter l'existence des outliers dans les jeu de données.

```
1 import math
2 features_to_plot = numeric_cols
3 n = len(features_to_plot)
4 cols = 3
5 rows = math.ceil(n / cols)
6
7 plt.figure(figsize=(5 * cols, 4 * rows))
8 for i, feature in enumerate(features_to_plot):
9     plt.subplot(rows, cols, i + 1)
10    sns.boxplot(data=df1, x='status', y=feature)
11    plt.title(f"{feature} par statut")
12
13 plt.tight_layout()
14 plt.show()
```



```

1 def detect_outliers_iqr(df, column):
2     Q1 = df[column].quantile(0.25)
3     Q3 = df[column].quantile(0.75)
4     IQR = Q3 - Q1
5     lower_bound = Q1 - 3.5 * IQR
6     upper_bound = Q3 + 3.5 * IQR
7     outliers = df[(df[column] < lower_bound) | (df[column] >
8         upper_bound)]
9     return outliers

```

La fonction ci-dessus permet de détecter les outliers (valeurs aberrantes) présents dans le DataFrame. Toutes les lignes contenant des outliers sont retournées.

```

1 for col in df1.select_dtypes(include='number').columns:
2     outliers = detect_outliers_iqr(df1, col)
3     print(f"Nombre d'outliers detectes pour {col} : {len(outliers)}")

```

```

Nombre d'outliers détectés pour MDVP:F0(Hz) : 0
Nombre d'outliers détectés pour MDVP:F1(Hz) : 5
Nombre d'outliers détectés pour MDVP:F1o(Hz) : 0
Nombre d'outliers détectés pour MDVP:Jitter(%) : 4
Nombre d'outliers détectés pour MDVP:Jitter(Abs) : 2

```

FIGURE 2.4 – Nombre d’outliers détectés pour chaque colonne

Étant donné la taille très réduite de l’échantillon, nous avons choisi de ne pas supprimer les outliers afin de conserver un maximum d’informations pour l’analyse.

## 2.6 Mise à l’échelle des features

```

1 # 1. Separation des variables features et target
2 print(df1.head)
3 X=df1.drop(columns=['name', 'person_id','status'])
4 # Features (tout sauf 'status')
5 y = df1['status'] # Target (status)
6
7 # 2. Normalisation des donnees avec StandardScaler
8 scaler = StandardScaler()
9 X_scaled = scaler.fit_transform(X)
10 X_scaled.shape

```

Dans cette étape, nous séparons les variables explicatives (features) de la variable cible (target), puis nous appliquons une normalisation des données à l’aide de la méthode `StandardScaler`, qui permet de centrer et réduire les variables pour faciliter l’apprentissage par les modèles.

# Chapitre 3

## Construction des modèles

### 3.1 Introduction

Dans ce chapitre, nous allons construire et entraîner plusieurs modèles de classification afin de prédire si un patient est atteint de la maladie de Parkinson. La démarche commence par la séparation des données en ensembles d'entraînement et de test. Ensuite, différents algorithmes de machine learning seront implémentés.

Chaque modèle sera évalué à l'aide de métriques classiques telles que la précision, le rappel et le score F1, en comparant ses performances sur les ensembles d'entraînement et de test. L'objectif est d'identifier les modèles les plus robustes et les plus adaptés aux données, tout en veillant à éviter le surapprentissage.

Nous terminerons cette phase par une synthèse comparative des performances des modèles testés, afin de guider le choix final pour une future intégration.

### 3.2 Séparation des données pour l'entraînement et les tests

```
1 from sklearn.model_selection import train_test_split
2 # 1. Séparation des données en entraînement et test
3 X_train, X_test, y_train, y_test = train_test_split(X_scaled, y
    , test_size=0.3, random_state=42)
```

Nous choisissons une séparation de 30 % pour le test et de 70 % pour l'entraînement.

### 3.3 Modélisation avec différents modèles

Dans cette section, nous testons plusieurs algorithmes de classification : la régression logistique, K-Nearest Neighbors (KNN), Random Forest, XGBoost et l'arbre de décision.

#### 1. Régression Logistique

La régression logistique est un modèle linéaire de classification. Elle estime la probabilité qu'une observation appartienne à une classe à l'aide d'une fonction sigmoïde, où  $\beta_i$  sont les coefficients appris par le modèle :

$$P(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \dots + \beta_n x_n)}}$$

```

1 from sklearn.linear_model import LogisticRegression
2
3 log_reg = LogisticRegression(max_iter=10000)
4 log_reg.fit(X_train, y_train)
5 y_pred_log_reg = log_reg.predict(X_test)

```

**Fonction de coût (log-vraisemblance négative)** La régression logistique minimise la fonction de coût suivante, appelée log-loss ou entropie croisée :

$$J(\beta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log \hat{p}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

où  $\hat{p}^{(i)} = \frac{1}{1 + e^{-\beta^\top \mathbf{x}^{(i)}}}$  est la probabilité prédite pour l'exemple  $i$ .

**Minimisation** Cette fonction est minimisée par une méthode itérative de *descente de gradient* ou ses variantes, qui ajuste les coefficients  $\beta$  pour réduire l'erreur sur les données.

## 2. K-Nearest Neighbors (KNN)

KNN est un algorithme non paramétrique qui attribue une classe à un échantillon en fonction de la majorité des classes parmi ses  $k$  plus proches voisins. La distance la plus couramment utilisée est la distance euclidienne :

$$d(x, x') = \sqrt{\sum_{i=1}^n (x_i - x'_i)^2}$$

```

1 from sklearn.neighbors import KNeighborsClassifier
2
3 knn = KNeighborsClassifier(n_neighbors=5)
4 knn.fit(X_train, y_train)
5 y_pred_knn = knn.predict(X_test)

```

**Fonction de coût** KNN ne repose pas explicitement sur une fonction de coût à minimiser, mais classe un point selon la majorité des classes parmi ses  $k$  voisins les plus proches, mesurés par une distance (ex. euclidienne) :

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{j=1}^n (x_j - x'_j)^2}$$

**Minimisation** Il n'y a pas de phase d'optimisation paramétrique. Le choix des voisins et la distance définissent la classification.

### 3. Random Forest

Le modèle Random Forest est un ensemble d'arbres de décision construits sur des sous-échantillons aléatoires des données. Chaque arbre vote pour une classe, et la prédiction finale est basée sur la majorité des votes.

```
1 from sklearn.ensemble import RandomForestClassifier
2
3 rf = RandomForestClassifier(n_estimators=100,
4                             random_state=42)
5 rf.fit(X_train, y_train)
6 y_pred_rf = rf.predict(X_test)
```

**Fonction de coût** Chaque arbre est construit en minimisant une fonction d'impureté locale telle que l'indice de Gini :

$$Gini(t) = 1 - \sum_{i=1}^C p_i^2$$

où  $p_i$  est la proportion d'échantillons de la classe  $i$  dans le nœud  $t$ .

**Minimisation** Les arbres sont construits en sélectionnant les meilleures divisions qui minimisent cette impureté à chaque étape, puis l'ensemble final combine les votes de tous les arbres (bagging) pour la prédiction.

### 4. XGBoost

XGBoost (Extreme Gradient Boosting) est un algorithme de boosting basé sur les arbres, où les modèles sont construits de manière séquentielle. Chaque nouvel arbre corrige les erreurs des arbres précédents. La prédiction s'exprime ainsi :

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), \quad f_k \in \mathcal{F}$$

où  $\mathcal{F}$  est l'espace des arbres de décision.

```

1 from xgboost import XGBClassifier
2
3 xgb = XGBClassifier(use_label_encoder=False, eval_metric=
4     'logloss', random_state=42)
5 xgb.fit(X_train, y_train)
6 y_pred_xgb = xgb.predict(X_test)

```

**Fonction de coût** XGBoost minimise une fonction de coût combinant la perte prédictive (ex. log-loss) et un terme de régularisation pour contrôler la complexité du modèle :

$$Obj = \sum_{i=1}^m l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

avec

$$l(y_i, \hat{y}_i) = -[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

et

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

où  $T$  est le nombre de feuilles dans l'arbre,  $w_j$  le poids de la feuille,  $\gamma$  et  $\lambda$  des hyperparamètres.

**Minimisation** L'optimisation se fait par ajout itératif d'arbres via un algorithme de gradient boosting, corrigeant les erreurs des arbres précédents.

## 5. Arbre de Décision (Decision Tree)

L'arbre de décision divise les données en fonction d'attributs qui maximisent un critère de séparation comme l'entropie ou l'indice de Gini, où  $p_i$  est la proportion d'éléments de la classe  $i$  dans un nœud :

$$Gini = 1 - \sum_{i=1}^C p_i^2 \quad \text{ou} \quad Entropy = - \sum_{i=1}^C p_i \log_2(p_i)$$

```

1 from sklearn.tree import DecisionTreeClassifier
2
3 dtc = DecisionTreeClassifier(random_state=42)
4 dtc.fit(X_train, y_train)
5 y_pred_dtc = dtc.predict(X_test)

```

**Fonction de coût** L'arbre de décision minimise une mesure d'impureté telle que l'indice de Gini (déjà défini) ou l'entropie de Shannon :

$$Entropy(t) = - \sum_{i=1}^C p_i \log_2 p_i$$

**Minimisation** À chaque nœud, la meilleure division est choisie pour réduire au maximum l'impureté (Gini ou entropie), construisant ainsi l'arbre de manière récursive.

## 3.4 Évaluation des modèles

### 1. Régression Logistique

```
1 print("Logistic Regression Evaluation")
2 print(classification_report(y_test, y_pred_log_reg))
3 print(confusion_matrix(y_test, y_pred_log_reg))
```

Logistic Regression Evaluation				
	precision	recall	f1-score	support
0	0.82	0.60	0.69	15
1	0.88	0.95	0.91	44
accuracy			0.86	59
macro avg	0.85	0.78	0.80	59
weighted avg	0.86	0.86	0.86	59
[[ 9 6]				
[ 2 42]]				

FIGURE 3.1 – Performance de la régression logistique

### 2. K-Nearest Neighbors

```
1 print("K-Nearest Neighbors Evaluation")
2 print(classification_report(y_test, y_pred_knn))
3 print(confusion_matrix(y_test, y_pred_knn))
```

K-Nearest Neighbors Evaluation				
	precision	recall	f1-score	support
0	0.91	0.67	0.77	15
1	0.90	0.98	0.93	44
accuracy			0.90	59
macro avg	0.90	0.82	0.85	59
weighted avg	0.90	0.90	0.89	59
[[10 5]				
[ 1 43]]				

FIGURE 3.2 – Performance du modèle KNN



### 3. Random Forest

```
1 print("Random Forest Evaluation")
2 print(classification_report(y_test, y_pred_rf))
3 print(confusion_matrix(y_test, y_pred_rf))
```

Random Forest Evaluation					
		precision	recall	f1-score	support
	0	0.92	0.80	0.86	15
	1	0.93	0.98	0.96	44
accuracy				0.93	59
macro avg		0.93	0.89	0.91	59
weighted avg		0.93	0.93	0.93	59
[[12  3]					
[ 1 43]]					

FIGURE 3.3 – Performance du modèle Random Forest

### 4. XGBoost

```
1 print("XGBoost Classifier Evaluation")
2 print(classification_report(y_test, y_pred_xgb))
3 print(confusion_matrix(y_test, y_pred_xgb))
```

XGBoost Classifier Evaluation					
		precision	recall	f1-score	support
	0	1.00	0.73	0.85	15
	1	0.92	1.00	0.96	44
accuracy				0.93	59
macro avg		0.96	0.87	0.90	59
weighted avg		0.94	0.93	0.93	59
[[11  4]					
[ 0 44]]					

FIGURE 3.4 – Performance du modèle XGBoost


### 5. DecisionTreeClassifier

```
1 print("DecisionTreeClassifier Evaluation")
2 print(classification_report(y_test, y_pred_dtc))
3 print(confusion_matrix(y_test, y_pred_dtc))
```


DecisionTreeClassifier Evaluation				
	precision	recall	f1-score	support
0	0.71	0.80	0.75	15
1	0.93	0.89	0.91	44
accuracy			0.86	59
macro avg	0.82	0.84	0.83	59
weighted avg	0.87	0.86	0.87	59
[[12 3]				
[ 5 39]]				

FIGURE 3.5 – Performance de DecisionTreeClassifier

## 3.5 Résumé des modèles testés

 Performances sur les données d'entraînement :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.882	0.892	0.961	0.925
K-Nearest Neighbors	0.956	0.980	0.961	0.971
Random Forest	1.000	1.000	1.000	1.000
XGBoost Classifier	1.000	1.000	1.000	1.000
DecisionTreeClassifier	1.000	1.000	1.000	1.000

 Performances sur les données de test :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.864	0.875	0.955	0.913
K-Nearest Neighbors	0.898	0.896	0.977	0.935
Random Forest	0.932	0.935	0.977	0.956
XGBoost Classifier	0.932	0.917	1.000	0.957
DecisionTreeClassifier	0.864	0.929	0.886	0.907

### Interprétation

Les performances montrent des résultats très différents entre les données d'entraînement et de test, ce qui indique des problèmes potentiels de surajustement (*overfitting*) pour certains modèles.

— **Régression Logistique :**

- Performance stable entre train (0.882 accuracy) et test (0.864)
- Bon rappel (0.955) mais précision légèrement inférieure (0.875)

→ Modèle le moins performant mais le plus stable (peu d'*overfitting*)

— **K-Nearest Neighbors (KNN) :**

- Forte baisse de performance entre train (0.956) et test (0.898)
- Excellent rappel (0.977) mais précision en baisse (0.896)

→ Signes d'*overfitting* modéré

— **Random Forest :**

- Performance parfaite sur train (1.000) mais baisse sur test (0.932)
- Excellents scores sur test (F1 0.956)

- *Overfitting* présent mais résultats restent très bons
- **XGBoost** :
  - Comme Random Forest : parfait sur train, baisse sur test
  - Rappel parfait sur test (1.000) mais précision légèrement inférieure (0.917)
- Bon compromis entre performance et généralisation
- **Arbre de Décision** :
  - Performance parfaite sur train mais chute importante sur test (0.864)
  - Plus grand écart train-test parmi tous les modèles
- *Overfitting* très prononcé

## Conclusion :

- **Pour la performance pure** : Random Forest et XGBoost offrent les meilleurs résultats sur les données de test (0.932 accuracy).
  - **Pour la stabilité** : La régression logistique, bien que moins performante, montre la meilleure généralisation.
  - **Problèmes identifiés** :
    - Les modèles trop complexes (surtout DecisionTree) souffrent clairement d'*overfitting*
    - Les performances sur train ne garantissent pas les performances sur test
- Les modèles Random Forest et XGBoost se distinguent comme les meilleurs choix, offrant un excellent équilibre entre performance (Accuracy : 0.932) et robustesse. Leur supériorité en précision et en rappel en fait des solutions optimales pour la plupart des cas d'usage, malgré un léger overfitting à contrôler.

# Chapitre 4

## Améliorations des modèles

### 4.1 Introduction

Ce chapitre est principalement dédié à l'application de différentes stratégies d'amélioration des modèles de machine learning. Parmi ces techniques, nous appliquerons notamment la **recherche par grille** (*Grid Search*), la méthode **SMOTE** pour le traitement du déséquilibre des classes, la **réduction de dimension** via L'ACP, ainsi que l'entraînement d'un modèle de **Deep Learning**.

Après l'application de chaque stratégie, nous procéderons à une comparaison rigoureuse des résultats obtenus, suivie d'une analyse critique. L'objectif est de déterminer les bénéfices de chaque méthode et de décider si son intégration dans notre pipeline de modélisation est pertinente ou non.

### 4.2 La performance initiale des modèles

La figure ci-dessous présente les performances initiales des modèles avant l'application des différentes stratégies d'amélioration.

Performances sur les données d'entraînement :				
	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.882	0.892	0.961	0.925
K-Nearest Neighbors	0.956	0.980	0.961	0.971
Random Forest	1.000	1.000	1.000	1.000
XGBoost Classifier	1.000	1.000	1.000	1.000
DecisionTreeClassifier	1.000	1.000	1.000	1.000

Performances sur les données de test :				
	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.864	0.875	0.955	0.913
K-Nearest Neighbors	0.898	0.896	0.977	0.935
Random Forest	0.932	0.935	0.977	0.956
XGBoost Classifier	0.932	0.917	1.000	0.957
DecisionTreeClassifier	0.864	0.929	0.886	0.907

FIGURE 4.1 – La performance des modèles

## 4.3 Choix des meilleurs hyperparamètres

Dans cette partie, nous appliquons la méthode de recherche sur grille (Grid Search) afin d'entraîner les modèles avec les meilleurs hyperparamètres possibles. Cette stratégie permet d'optimiser les performances des algorithmes de machine learning en testant systématiquement différentes combinaisons de paramètres.

Ci-dessous, un exemple de recherche d'hyperparamètres appliquée au modèle *Random Forest* :

```
1 from sklearn.ensemble import RandomForestClassifier
2 from sklearn.model_selection import GridSearchCV
3 from sklearn.metrics import classification_report,
  confusion_matrix
4 # Grille d hyperparamtres pour Random Forest
5 param_grid_rf = {
6     'n_estimators': [100, 200, 300],
7     'max_depth': [None, 5, 10, 20],
8     'min_samples_split': [2, 5, 10],
9     'min_samples_leaf': [1, 2, 4],
10    'bootstrap': [True, False]
11 }
12
13 # Initialisation du mod le
14 rf_clf_resampled = RandomForestClassifier(random_state=42)
15
16 # Recherche par validation crois e (GridSearchCV)
17 grid_rf_resampled = GridSearchCV(estimator=rf_clf_resampled,
18                                  param_grid=param_grid_rf,
19                                  scoring='f1',
20                                  cv=5,
21                                  verbose=2,
22                                  n_jobs=-1)
23
24 # Entra nement sur les donn es r chantillon n es
25 grid_rf_resampled.fit(X_train_resampled, y_train_resampled)
26
27 # Affichage des meilleurs hyperparam tres
28 print("Best parameters for Random Forest:")
29 print(grid_rf_resampled.best_params_)
```

Fitting 5 folds for each of 216 candidates, totalling 1080 fits

Best parameters for Random Forest:

{'bootstrap': False, 'max\_depth': None, 'min\_samples\_leaf': 2, 'min\_samples\_split': 2, 'n\_estimators': 100}

Évaluation sur les données de test :

	precision	recall	f1-score	support
0	0.92	0.80	0.86	15
1	0.93	0.98	0.96	44
accuracy			0.93	59
macro avg	0.93	0.89	0.91	59
weighted avg	0.93	0.93	0.93	59

```
[[12 3]
 [ 1 43]]
```

## Résumé de tous les modèles

Performances sur les données d'entraînement :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.882	0.892	0.961	0.925
Ridge	0.882	0.866	1.000	0.928
Lasso	0.838	0.852	0.951	0.899
K-Nearest Neighbors	1.000	1.000	1.000	1.000
Random Forest	1.000	1.000	1.000	1.000
XGBoost Classifier	1.000	1.000	1.000	1.000
DecisionTreeClassifier	0.993	1.000	0.990	0.995

Performances sur les données de test :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.864	0.875	0.955	0.913
Ridge	0.881	0.863	1.000	0.926
Lasso	0.864	0.860	0.977	0.915
K-Nearest Neighbors	0.966	0.957	1.000	0.978
Random Forest	0.949	0.936	1.000	0.967
XGBoost Classifier	0.949	0.936	1.000	0.967
DecisionTreeClassifier	0.864	0.891	0.932	0.911

FIGURE 4.2 – Les résultats en utilisant grid search

## Analyse

Avant l'optimisation, certains modèles tels que **Random Forest**, **XGBoost** et **Decision Tree** affichaient des performances parfaites sur les données d'entraînement (accuracy, recall et F1-score de 1.000), ce qui indique un **surapprentissage**. Bien que leurs résultats sur les données de test restaient élevés, une légère baisse des métriques montrait une capacité de généralisation limitée.

Après une recherche approfondie des meilleurs paramètres (par validation croisée), les modèles ont été réajustés, et plusieurs améliorations ont été constatées :

- Le modèle **KNN** atteint un **F1-score de 0.978** et un **rappel de 1.000** sur les données de test, ce qui signifie qu'il identifie correctement tous les malades.
- Les modèles **Random Forest** et **XGBoost** maintiennent une performance excellente, avec une meilleure généralisation qu'avant.
- L'introduction des modèles **Ridge** et **Lasso** permet une meilleure gestion du compromis biais-variance. Ridge, par exemple, obtient un **rappel parfait (1.000)** et un F1-score de **0.926**.

## Conclusion

L'optimisation des hyperparamètres a significativement amélioré la robustesse des modèles. Elle a permis :

- de limiter le surapprentissage sur les données d'entraînement ;
- d'augmenter les scores de précision et de F1 sur les données de test ;
- de garantir une **meilleure détection des cas positifs (malades)**, tout en réduisant les faux positifs.

## 4.4 Application de la méthode SMOTE

Comme le montre la Figure 4.3, nos données initiales présentent un déséquilibre important entre les deux classes cibles : seulement **33 patients sains** contre **103 patients atteints de la maladie de Parkinson**. Ce déséquilibre peut fortement biaiser les modèles de classification, en les incitant à prédire majoritairement la classe majoritaire (les malades), au détriment d'une bonne détection des personnes saines.

Afin de pallier ce problème, nous avons appliqué la méthode **SMOTE (Synthetic Minority Over-sampling Technique)**. Cette technique génère artificiellement de nouveaux exemples pour la classe minoritaire (les patients sains) en interpolant des points proches dans l'espace des features.

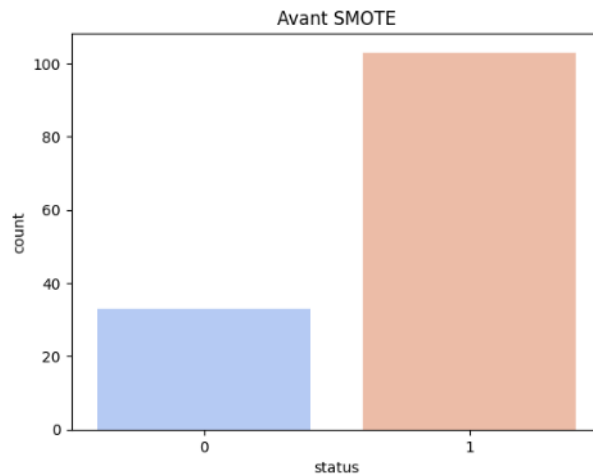


FIGURE 4.3 – Déséquilibre des classes avant SMOTE

Après l'application de SMOTE, le nombre de patients sains a été porté à **133**, équilibrant ainsi les deux classes à un rapport de 1 :1 (Figure 4.4). Cet équilibrage permet aux modèles d'apprentissage supervisé de mieux généraliser et d'améliorer la détection des deux classes de manière équitable.

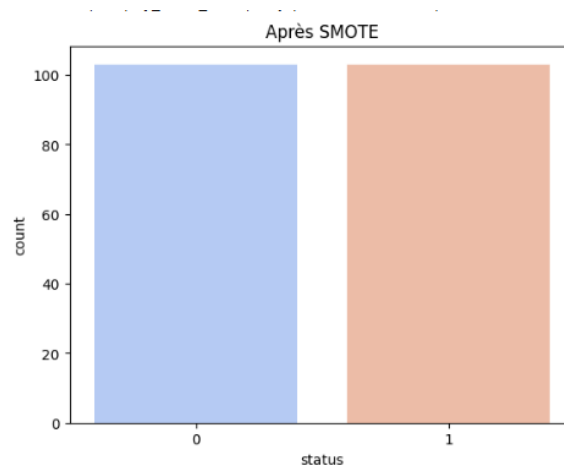


FIGURE 4.4 – Equilibre des classes après SMOTE

## Résumé de tous les modèles

Performances sur les données d'entraînement :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.845	0.874	0.806	0.838
K-Nearest Neighbors	0.932	1.000	0.864	0.927
Random Forest	1.000	1.000	1.000	1.000
XGBoost Classifier	1.000	1.000	1.000	1.000
DecisionTreeClassifier	1.000	1.000	1.000	1.000

Performances sur les données de test :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.864	0.909	0.909	0.909
K-Nearest Neighbors	0.915	1.000	0.886	0.940
Random Forest	0.915	0.933	0.955	0.944
XGBoost Classifier	0.932	0.935	0.977	0.956
DecisionTreeClassifier	0.881	0.930	0.909	0.920

FIGURE 4.5 – Précision des modèles après l'application de SMOTE

## Analyse et conclusion

- **La régression logistique** montre un meilleur équilibre entre précision et rappel après SMOTE.
- **KNN** bénéficie d'une amélioration notable en précision, avec un léger recul du rappel.
- **Random Forest** et **XGBoost** conservent des performances robustes et équilibrées.
- **Decision Tree** devient plus stable, notamment en rappel.

## Conclusion

L'application de **SMOTE** a permis une meilleure représentation des classes minoritaires, réduisant les biais et améliorant les compromis entre rappel et précision. Cela en fait une étape essentielle, notamment dans un contexte médical sensible à la fois aux faux positifs et aux faux négatifs.

## 4.5 Application de l'ACP

Dans cette section, nous appliquons l'Analyse en Composantes Principales (ACP) afin de réduire la dimensionnalité des données. Cette technique permet de transformer les variables initiales corrélées en un ensemble plus restreint de composantes principales non corrélées, tout en conservant un maximum de variance. L'objectif est d'améliorer la performance des modèles ou de faciliter la visualisation des données tout en limitant la perte d'information.

```
1 from sklearn.preprocessing import StandardScaler
2 from sklearn.decomposition import PCA
```



```

3 # 1. PCA      ici on garde 95% de la variance
4 pca = PCA(n_components=0.95, random_state=42)
5 X_train_pca = pca.fit_transform(X_train)
6 X_test_pca = pca.transform(X_test)
7
8 # 3. Afficher le nombre de composantes retenues
9 print(f" Nombre de composantes PCA conserv es : {pca.
    n_components_}")

```

## Résumé de tous les modèles

Performances sur les données d'entraînement :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.882	0.892	0.961	0.925
Ridge	0.882	0.866	1.000	0.928
Lasso	0.875	0.884	0.961	0.921
K-Nearest Neighbors	1.000	1.000	1.000	1.000
Random Forest	0.993	0.990	1.000	0.995
XGBoost Classifier	1.000	1.000	1.000	1.000
DecisionTreeClassifier	0.985	1.000	0.981	0.990

Performances sur les données de test :

	Accuracy	Precision (1)	Recall (1)	F1-score (1)
Logistic Regression	0.881	0.878	0.977	0.925
Ridge	0.864	0.860	0.977	0.915
Lasso	0.881	0.878	0.977	0.925
K-Nearest Neighbors	0.932	0.935	0.977	0.956
Random Forest	0.898	0.896	0.977	0.935
XGBoost Classifier	0.881	0.878	0.977	0.925
DecisionTreeClassifier	0.881	0.930	0.909	0.920

FIGURE 4.6 – La performance des modèles après l'application de l'ACP

## Après l'ACP

- Les performances sont plus équilibrées entre entraînement et test.
- Légère baisse pour Random Forest et XGBoost → réduction du surapprentissage.
- KNN garde la meilleure performance ( $F1 \approx 0,96$ ).
- Introduction de Ridge et Lasso avec de bons résultats ( $F1 \approx 0,92$ ).
- Régression logistique reste stable.

## Conclusion

- L'ACP réduit le surapprentissage et améliore la généralisation.
- Les modèles sont plus stables et les performances restent bonnes.
- KNN reste le modèle le plus robuste avant et après ACP.
- L'ACP est utile pour simplifier les données sans perte majeure de performance.

## 4.6 Test du Deep Learning

### 1. Importation des bibliothèques

```
1 import numpy as np
2 from sklearn.preprocessing import StandardScaler
3 from tensorflow.keras.models import Sequential
4 from tensorflow.keras.layers import Dense, Dropout
5 from tensorflow.keras.optimizers import Adam
6 from tensorflow.keras.metrics import Precision, Recall
7 from sklearn.metrics import classification_report,
  confusion_matrix
```

### 2. Construction du modèle

```
1 model = Sequential([
2     Dense(64, activation='relu', input_shape=(X_train.
3         shape[1],)),
4     Dropout(0.3),
5     Dense(32, activation='relu'),
6     Dropout(0.3),
7     Dense(1, activation='sigmoid') # Classification
8                                     binaire
9 ])
```

Voici une brève explication de la structure du modèle choisi :

- **Nombre total de neurones** :  $64 + 32 + 1 = 97$  neurones
- **Couche d'entrée** : taille = *nombre de colonnes de  $X_{train}$*
- **Nombre de couches cachées** : **2**
  - 1<sup>ère</sup> couche cachée : **64 neurones**, activation ReLU
  - 2<sup>e</sup> couche cachée : **32 neurones**, activation ReLU
- **Couche de sortie** : **1 neurone**, activation sigmoid (pour classification binaire)

### 3. Compilation

```
1 model.compile(
2     optimizer=Adam(learning_rate=0.001),
3     loss='binary_crossentropy',
4     metrics=['accuracy', Precision(), Recall()]
5 )
```

### 4. Entraînement

```
1 history = model.fit(
2     X_train, y_train,
3     epochs=50,
4     batch_size=16,
5     validation_data=(X_test, y_test),
6     verbose=2
7 )
```

## 5. Prédiction et évaluation

```
1 y_pred_nn = model.predict(X_test)
2 y_pred_nn_binary = (y_pred_nn > 0.5).astype(int)
3
4 print(" Rapport de classification du reseau de neurones")
5 print(classification_report(y_test, y_pred_nn_binary))
6 print("Matrice de confusion :")
7 print(confusion_matrix(y_test, y_pred_nn_binary))
```

```
Rapport de classification du réseau de neurones :
              precision    recall  f1-score   support

     0       0.90      0.60      0.72        15
     1       0.88      0.98      0.92        44

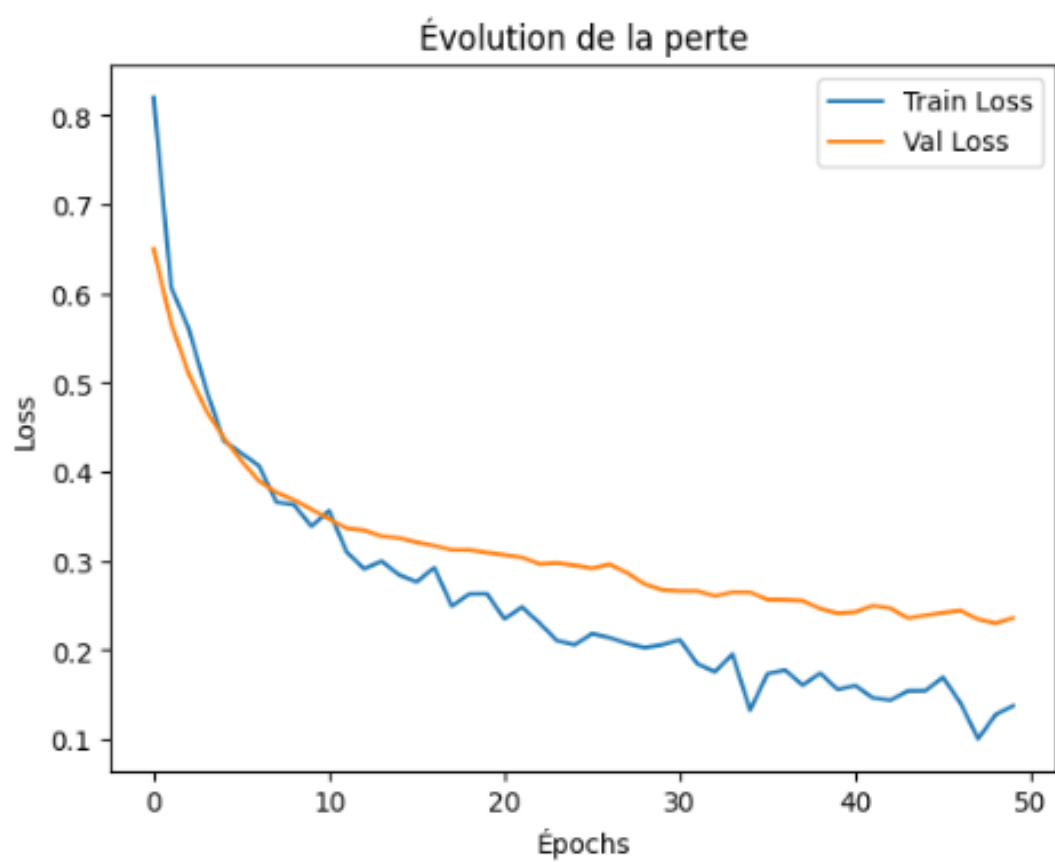
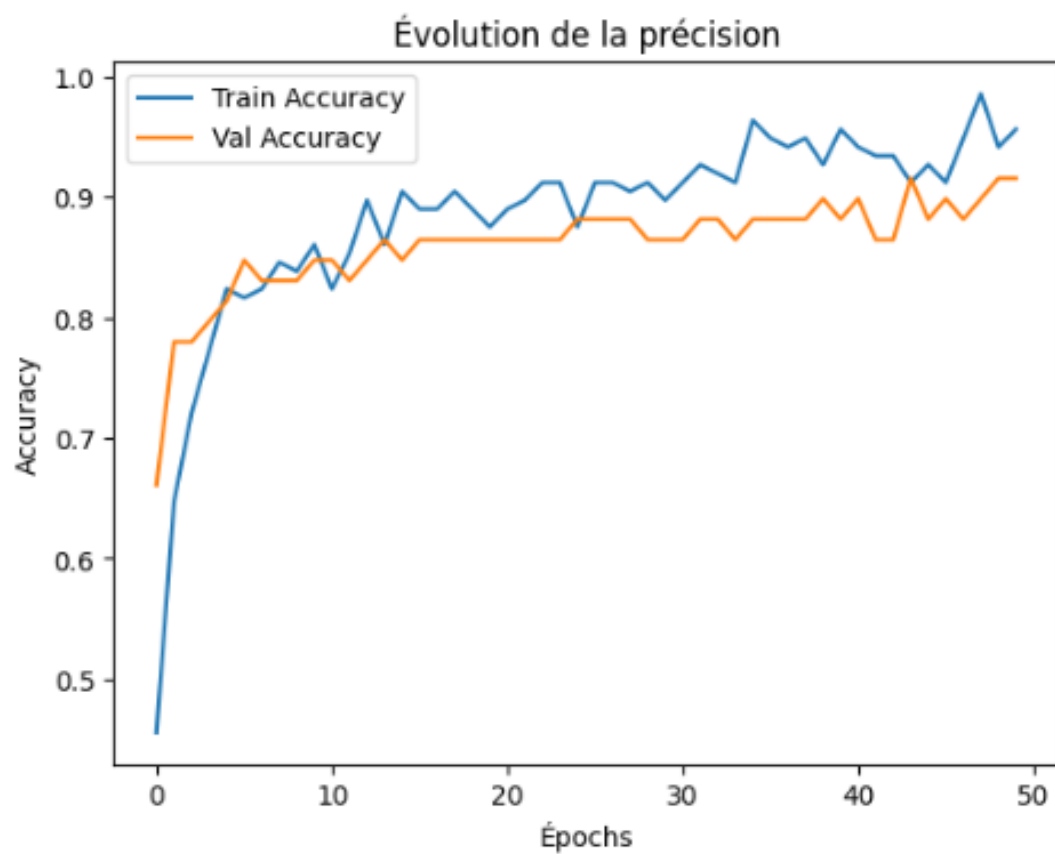
 accuracy          0.88        59
 macro avg          0.89        59
 weighted avg       0.88        59

Matrice de confusion :
[[ 9  6]
 [ 1 43]]
```

FIGURE 4.7 – Performance des réseaux de neurones

## Évolution de la performance du modèle Deep Learning

```
1 import matplotlib.pyplot as plt
2 # Pr cision
3 plt.plot(history.history['accuracy'], label='Train Accuracy')
4 plt.plot(history.history['val_accuracy'], label='Val Accuracy')
5 plt.title(' evolution de la pr cision')
6 plt.xlabel(' epochs ')
7 plt.ylabel('Accuracy')
8 plt.legend()
9 plt.show()
10
11 # Perte
12 plt.plot(history.history['loss'], label='Train Loss')
13 plt.plot(history.history['val_loss'], label='Val Loss')
14 plt.title(' evolution de la perte')
15 plt.xlabel(' epochs ')
16 plt.ylabel('Loss')
17 plt.legend()
18 plt.show()
```



## Interprétation des graphiques

Ces deux graphiques illustrent l'évolution des performances d'un réseau de neurones au cours de l'entraînement sur **50 époques**.

1. Évolution de la précision (*accuracy*)
  - La précision sur l'entraînement augmente rapidement et atteint presque **100%**, ce qui montre que le modèle apprend bien les données d'entraînement.
  - La précision sur la validation augmente également au début mais **se stabilise autour de 90%**, ce qui est un bon signe.
  - L'écart entre les deux courbes reste modéré, ce qui indique qu'il n'y a **pas de surapprentissage (overfitting)** très prononcé.
2. Évolution de la perte (*loss*)
  - La perte d'entraînement diminue régulièrement, ce qui indique que le modèle s'améliore.
  - La perte de validation diminue aussi, mais **se stabilise plus tôt** et reste **plus élevée** que celle de l'entraînement.
  - Cela suggère un **léger surapprentissage** à partir d'un certain point (probablement vers la 20<sup>e</sup> époque), mais rien d'inquiétant.

## Conclusion

Le modèle apprend bien et généralise correctement. Les courbes sont **cohérentes et stables**, ce qui reflète un bon entraînement global. Pour améliorer encore les performances, l'utilisation d'un *early stopping* autour de la 30<sup>e</sup> ou 35<sup>e</sup> époque pourrait être bénéfique pour éviter un surapprentissage éventuel.

# Chapitre 5

## Prédire la gravité de la maladie

### 5.1 Vue d'ensemble et statistique descriptive

Dans cette partie, nous abordons le cas non supervisé. L'objectif est de regrouper les patients atteints de la maladie de Parkinson en fonction du **niveau de gravité de leur état de santé**.

Pour cela, nous utilisons les données issues du second fichier `parkinson.data`.

La première étape consiste à **charger les données** à partir de ce fichier en utilisant la bibliothèque `pandas`.

```
1 import pandas as pd
2 # Charger les donnees
3 df =pd.read_csv("parkinsons_updrs.data")
4 # Afficher les premieres lignes
5 print(df.head())
```

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)
0	1	72	0	5.6431	28.199	34.398	0.00662
1	1	72	0	12.6660	28.447	34.894	0.00300
2	1	72	0	19.6810	28.695	35.389	0.00481
3	1	72	0	25.6470	28.905	35.810	0.00528
4	1	72	0	33.6420	29.187	36.375	0.00335

FIGURE 5.1 – visualisation de quelques données

Ensuite, nous affichons la taille du jeu de données qui est égales à (5875, 22), c'est à dire 5875 lignes et 22 colonnes, ainsi que le nom de toutes les colonnes.

```
1 # dimensions des donnees
2 print("la taille du dataframe est:" ,df1.shape)
3 # affichage des colonnes
4 print("les colonnes du dataframe sont:", df1.columns)
```

Ensuite, nous utilisons la méthode `.describe()` pour visualiser plusieurs informations sur la distribution des données, telles que le nombre total de valeurs dans chaque colonne, ainsi que la moyenne, l'écart-type, le minimum, le maximum et les quantiles.

```

1 # deccrire les donnees
2 df1.describe()

```

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)
count	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000
mean	21.494128	64.804936	0.317787	92.863722	21.296229	29.018942	0.006154
std	12.372279	8.821524	0.465656	53.445602	8.129282	10.700283	0.005624
min	1.000000	36.000000	0.000000	-4.262500	5.037700	7.000000	0.000830
25%	10.000000	58.000000	0.000000	46.847500	15.000000	21.371000	0.003580
50%	22.000000	65.000000	0.000000	91.523000	20.871000	27.576000	0.004900
75%	33.000000	72.000000	1.000000	138.445000	27.596500	36.399000	0.006800
max	42.000000	85.000000	1.000000	215.490000	39.511000	54.992000	0.099990

FIGURE 5.2 – la description des données

La colonne **subject#** est l'identifiant unique de chaque ligne qui représente un enregistrement d'une personne.

```

1 # deccrire les donnees
2 df1['subject#'].value_counts()

```

	count
subject#	
29	168
35	165
41	165
34	161
7	161
24	156
5	156
6	156

FIGURE 5.3 – Nombre d'enregistrements par personne

D'après le résultat obtenu, chaque personne possède environ 150 enregistrements pour le suivi continu de son état.

```

1 df1.info()

```

Selon la commande `.info()`, aucune colonne ne présente de valeurs nulles dans le DataFrame comme le montre la figure ci-dessous. Par conséquent, nous n'aurons pas besoin de traiter les valeurs manquantes pour cette analyse.

```

# Column      Non-Null Count  Dtype
---  -
0  subject#    5875 non-null    int64
1  age         5875 non-null    int64
2  sex         5875 non-null    int64
3  test_time   5875 non-null    float64
4  motor_UPDRS 5875 non-null    float64
5  total_UPDRS 5875 non-null    float64
6  Jitter(%)   5875 non-null    float64
7  Jitter(Abs) 5875 non-null    float64
8  Jitter:RAP   5875 non-null    float64
9  Jitter:PPQ5 5875 non-null    float64
10 Jitter:DDP   5875 non-null    float64
11 Shimmer     5875 non-null    float64
12 Shimmer(dB) 5875 non-null    float64
13 Shimmer:APQ3 5875 non-null    float64
14 Shimmer:APQ5 5875 non-null    float64
15 Shimmer:APQ11 5875 non-null    float64
16 Shimmer:DDA 5875 non-null    float64
17 NHR        5875 non-null    float64
18 HNR        5875 non-null    float64
19 RPDE       5875 non-null    float64
20 DFA        5875 non-null    float64
21 PPE        5875 non-null    float64
dtypes: float64(19), int64(3)
memory usage: 1009.9 KB

```

## 5.2 Distribution des variables (graphiques)

Dans cette partie, nous visualisons la distribution des données numériques. La figure 5.4 illustre ceci :

```

1 # Exclure les colonnes non numériques
2 numeric_cols = df1.select_dtypes(include=['float64', 'int64']).
   columns.drop('subject#')
3 # Histogrammes
4 df1[numeric_cols].hist(figsize=(20, 15), bins=30, edgecolor='
   black')
5 plt.suptitle("Distribution des variables numériques", fontsize
   =20)
6 plt.tight_layout()
7 plt.show()

```

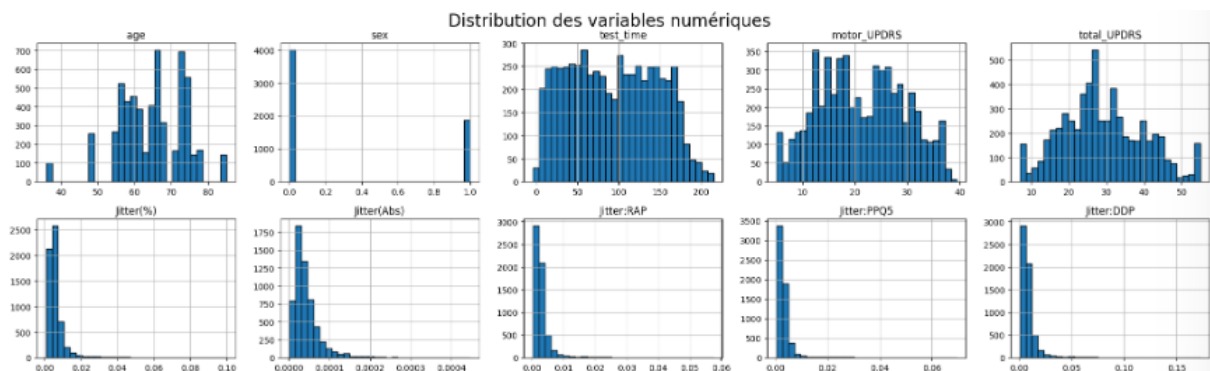


FIGURE 5.4 – La distribution des variables



## 5.3 Mise à l'échelle des features

```
1 from sklearn.preprocessing import StandardScaler
2 # On exclut les colonnes non informatives ou catégorielles
  pour ce type de traitement
3 X = df1.drop(columns=['subject#', 'sex', 'test_time']) #
  optionnel : tu peux encoder 'sex' si pertinent
4 scaler = StandardScaler()
5 X_scaled = scaler.fit_transform(X)
```

Dans cette étape, nous appliquons une normalisation des données à l'aide de la méthode `StandardScaler`, qui permet de centrer et réduire les variables pour faciliter l'apprentissage par les modèles.

## 5.4 Construction des Modèles

Dans cette section, nous testons différents algorithmes de **clustering non supervisé** afin de segmenter les patients en groupes distincts, possiblement représentatifs des stades de la maladie de Parkinson (léger, modéré, avancé). Les modèles suivants ont été appliqués sur les données préalablement normalisées :

- **K-Means** : un algorithme classique de partitionnement basé sur la minimisation de la variance intra-groupe ;
- **Agglomerative Clustering** : une méthode hiérarchique qui fusionne successivement les observations les plus proches ;
- **DBSCAN** : un algorithme basé sur la densité, capable de détecter des structures complexes et des points aberrants (outliers).

### 1. K-means :

```
1 from sklearn.cluster import KMeans
2 # On suppose 3 groupes (léger, modéré, avancé)
3 kmeans = KMeans(n_clusters=3, random_state=42)
4 kmeans_labels = kmeans.fit_predict(X_scaled)
5 df1['cluster_kmeans'] = kmeans_labels
```

### 2. Agglomerative Clustering :

```
1 from sklearn.cluster import AgglomerativeClustering
2 agglo = AgglomerativeClustering(n_clusters=3)
3 agglo_labels = agglo.fit_predict(X_scaled)
4 df1['cluster_agglo'] = agglo_labels
```

### 3. DBSCAN :

```
1 from sklearn.cluster import DBSCAN
2 # eps et min_samples peuvent être ajustés selon le
  dataset
3 dbscan = DBSCAN(eps=1.5, min_samples=5)
4 dbscan_labels = dbscan.fit_predict(X_scaled)
5 df1['cluster_dbscan'] = dbscan_labels
```

## 5.5 Visualisation des clusters

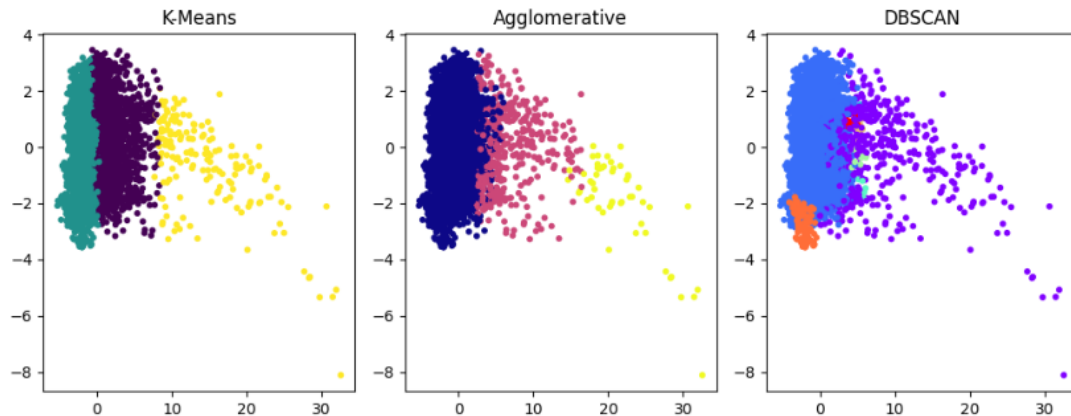


FIGURE 5.5 – Visualisation en 2D des classes

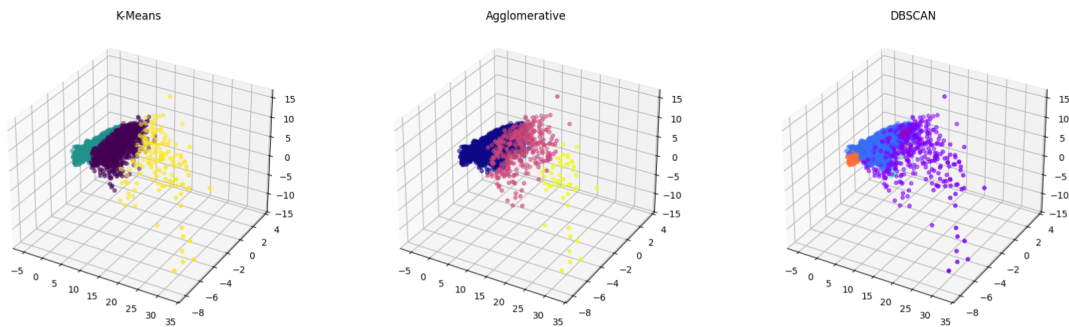


FIGURE 5.6 – Visualisation en 3D des classes

## 5.6 Évaluation des modèles de clustering

Pour évaluer la qualité des regroupements formés par les différents algorithmes de clustering, nous avons utilisé l'indice de silhouette. Cette métrique varie entre -1 et 1 :

- un score proche de 1 indique des clusters bien séparés et denses ;
- un score proche de 0 indique un chevauchement entre les clusters ;
- un score négatif suggère un mauvais regroupement.

Les scores obtenus sont les suivants :

- **KMeans** : 0,26 → qualité de regroupement modérée ;
- **Agglomerative Clustering** : 0,51 → bonne séparation des groupes, meilleur résultat ;
- **DBSCAN** : 0,08 → faible qualité de clustering, structure peu détectée.

Ainsi, l'algorithme **Agglomerative Clustering** semble le plus adapté pour segmenter les patients en groupes distincts dans ce contexte.

## 5.7 Quelques statistiques des groupes

Selon le modèle de clustering le plus performant, **Agglomerative Clustering**, nous avons attribué un label à chaque cluster basé sur les moyennes observées des scores *motor\_UPDRS* et *total\_UPDRS*. Les groupes ont été renommés selon le niveau de sévérité de la maladie : *léger*, *modéré* et *avancé*.

```
1 # Mapping bas sur les moyennes observées
2 label_map = {
3     0: "léger",
4     1: "modéré",
5     2: "avancé"
6 }
7 # Création d'une nouvelle colonne avec les labels
8 df1['severity'] = df1['cluster_agglo'].map(label_map)
9
10 # Affichage des statistiques
11 print(df1['severity'].value_counts())
12 print(df1.groupby('severity')[['motor_UPDRS', 'total_UPDRS']].
      mean())
```

```
severity
léger      5412
modéré     411
avancé      52
Name: count, dtype: int64

      motor_UPDRS  total_UPDRS
severity
avancé      23.172358    30.650692
léger       21.191707    28.940177
modéré      22.435185    29.849664
```

FIGURE 5.7 – Répartition du nombre de personnes dans chaque classe de sévérité

### Analyse des résultats du clustering :

Après avoir appliqué l'algorithme de *clustering agglomératif*, les patients ont été automatiquement regroupés en trois catégories en fonction de la gravité de leurs symptômes.

La répartition des individus dans chaque groupe est la suivante :

- Classe **léger** : 5412 individus
- Classe **modéré** : 411 individus
- Classe **avancé** : 52 individus

Le tableau suivant présente les moyennes des scores `motor_UPDRS` et `total_UPDRS` pour chaque groupe :

Sévérité	<code>motor_UPDRS</code> moyen	<code>total_UPDRS</code> moyen
avancé	23.17	30.65
modéré	22.43	29.85
léger	21.19	28.94

TABLE 5.1 – Moyennes des scores UPDRS selon la sévérité

## Interprétation

Les résultats montrent une progression claire de la gravité de la maladie d'un groupe à l'autre. Les classes ont été bien segmentées :

- La majorité des individus sont dans la classe **léger**, ce qui est cohérent avec une population où la plupart des patients sont à un stade précoce de la maladie.
- Les scores `motor_UPDRS` et `total_UPDRS` augmentent progressivement entre les groupes **léger**, **modéré** et **avancé**, ce qui confirme la pertinence du regroupement effectué par le modèle.

Ainsi, ce clustering permet non seulement de segmenter les patients, mais aussi d'évaluer automatiquement le stade de la maladie sans variable cible initialement définie.

# Chapitre 6

## Application web de prédiction

### 6.1 Objectif

L'objectif de cette application web est d'exploiter le modèle de machine learning ayant obtenu les meilleures performances lors de l'analyse précédente, afin de **prédire si une personne est atteinte de la maladie de Parkinson ou non**.

### 6.2 Langage et technologies utilisées

L'application est développée avec **Streamlit**, un framework Python léger et interactif, conçu pour créer rapidement des interfaces web destinées aux projets de data science et de machine learning.



FIGURE 6.1 – Logo de Streamlit

### 6.3 Présentation de l'interface utilisateur

L'interface permet à l'utilisateur de **saisir les différentes valeurs des caractéristiques** nécessaires à la prédiction. Pour éviter toute saisie incohérente ou aberrante :

- Des **valeurs minimales et maximales** (issues du dataset original) sont imposées pour chaque champ ;
- Un **bouton "Prédire"** déclenche le calcul automatique de la prédiction ;
- Le **résultat s'affiche dynamiquement** sous forme de texte :
  - *Malade* ou *Non malade* avec une probabilité
  - Des recommandations en se basant sur l'état du patient.

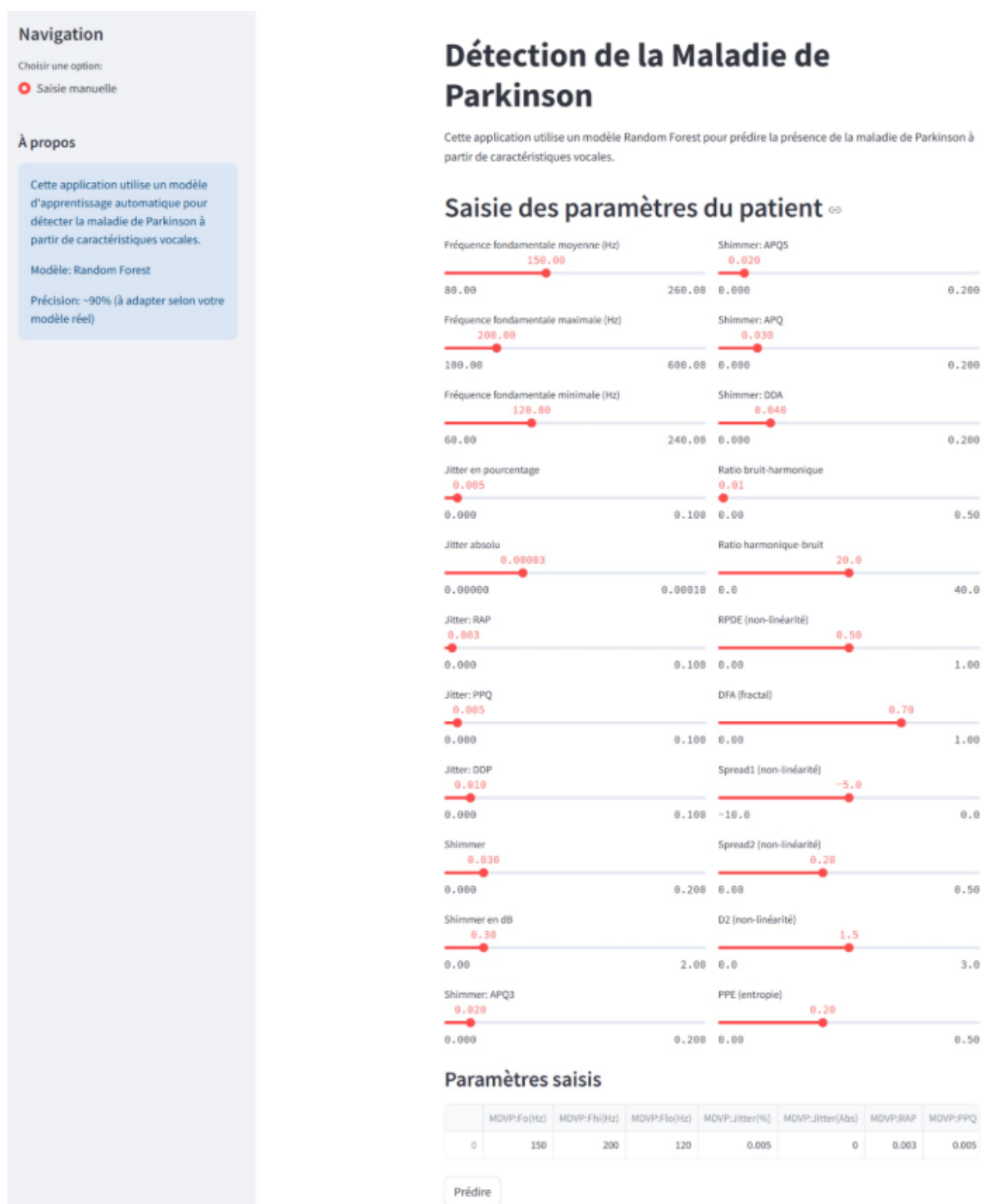


FIGURE 6.2 – Aperçu de l'interface utilisateur de prédiction

## 6.4 Étapes de mise en œuvre

Voici les principales étapes de mise en place du système de prédiction dans l'application :

1. **Choisir le modèle** ayant obtenu les meilleures performances lors de la phase de modélisation (ex : XGBoost ou Random Forest) ;
2. **Enregistrer ce modèle** au format `.pkl` à l'aide de la bibliothèque `pickle`

ainsi que le scaler utilisé lors de l'entraînement du modèle.

```
1 import pickle
2 pickle.dump(scaler, open("scaler.pkl", "wb"))
3 pickle.dump(best_rf_model, open("parkinsons_model.pkl", "
  wb"))
```

### 3. Charger le modèle et le scaler dans l'application Streamlit

```
1 try:
2     scaler = pickle.load(open("scaler.pkl", "rb"))
3     print("Attributs du scaler:", vars(scaler))
4 except Exception as e:
5     print("Erreur lors du chargement:", str(e))
6     # Chargez le mod le
7 model = pickle.load(open('parkinsons_model.pkl', 'rb'))
```

### 4. Définition des champs à saisir

L'utilisateur doit renseigner manuellement les valeurs de certaines variables (ou *features*) nécessaires à la prédiction. Ci-dessous, un extrait de la définition de quelques champs via des curseurs interactifs utilisant la bibliothèque Streamlit :

```
1 inputs['MDVP:Jitter(%)'] = st.slider('Jitter en
  pourcentage', 0.0, 0.1, 0.005, step=0.001, format="%.3
  f")
2 inputs['MDVP:Jitter(Abs)'] = st.slider('Jitter absolu',
  0.0, 0.0001, 0.00003, step=0.00001, format="%.5f")
3 inputs['MDVP:RAP'] = st.slider('Jitter: RAP', 0.0, 0.1,
  0.003, step=0.001, format="%.3f")
4 inputs['MDVP:PPQ'] = st.slider('Jitter: PPQ', 0.0, 0.1,
  0.005, step=0.001, format="%.3f")
5 inputs['Jitter:DDP'] = st.slider('Jitter: DDP', 0.0, 0.1,
  0.01, step=0.001, format="%.3f")
6 inputs['MDVP:Shimmer'] = st.slider('Shimmer', 0.0, 0.2,
  0.03, step=0.001, format="%.3f")
7 inputs['MDVP:Shimmer(dB)'] = st.slider('Shimmer en dB',
  0.0, 2.0, 0.3, step=0.01, format="%.2f")
8 inputs['Shimmer:APQ3'] = st.slider('Shimmer: APQ3', 0.0,
  0.2, 0.02, step=0.001, format="%.3f")
```

Chaque champ est défini avec une valeur minimale, maximale et une valeur initiale par défaut, ce qui permet de limiter les erreurs de saisie de l'utilisateur.

5. Utiliser la méthode `.predict()` pour **générer une prédiction** à partir des valeurs saisies par l'utilisateur

```
1 # Pretraitement des données
2 input_scaled = scaler.transform(input_df)
3
4 # Prediction
5 prediction = model.predict(input_scaled)
6 prediction_proba = model.predict_proba(input_scaled)
7
8 # Affichage des resultats
9 st.subheader('R sultats')
10 status = np.array(['Sain', 'Parkinson'])
11 st.write(f"**Pr diction:** {status[prediction][0]}")
12
13 st.write("**Probabilit s:**")
14 st.write(f"- Sain: {prediction_proba[0][0]*100:.2f}%")
15 st.write(f"- Parkinson: {prediction_proba[0][1]*100:.2f}%")
16 st.write(" ")
```

## Résultats

Prédiction: Parkinson

Probabilités:

- Sain: 0.00%
- Parkinson: 100.00%

FIGURE 6.3 – Pourcentage de la maladie

6. **Afficher le résultat** à l'écran, sous forme de diagnostic : *Malade* ou *Non malade* en pourcentage. Ensuite générer des recommandations pour le patient selon le diagnostic.

```
1 # Interpr tation
2 st.subheader('Interpr tation')
3 if prediction[0] == 1:
4     st.error("Le mod le pr dit que le patient pourrait
5             avoir la maladie de Parkinson.")
6     st.write("""
7     **Recommandations:**
8     - Consulter un neurologue pour un examen approfondi
9     - Effectuer des tests compl mentaires
10    - Surveillance r guli re des sympt mes
11    """)
12 else:
13     st.success("Le mod le pr dit que le patient est
14               probablement sain.")
15     st.write("""
16     **Recommandations:**
17     - Continuer surveiller les sympt mes ventuels
18     - Consulter si des sympt mes apparaissent
19     """)
```



## Interprétation

Le modèle prédit que le patient pourrait avoir la maladie de Parkinson.

FIGURE 6.4 – Résultat de prédiction

### Recommandations:

- Consulter un neurologue pour un examen approfondi
- Effectuer des tests complémentaires
- Surveillance régulière des symptômes

FIGURE 6.5 – Recommandations pour le patient

## 7. Déploiement de l'application avec Streamlit :

- (a) Exécuter le code des modèles et télécharger la version du modèle random forest.
- (b) Ouvrir un terminal (ou une invite de commande).
- (c) Se placer dans le répertoire contenant le fichier Python :

```
1 cd "C:\Users\Fatma CHAHED\Downloads"
```

- (d) Installer Streamlit si ce n'est pas déjà fait :

```
1 pip install streamlit
```

- (e) Exécuter le fichier Streamlit :

```
1 streamlit run Parkinson.py
```

- (f) Un lien sera automatiquement généré dans le terminal. Il suffit de cliquer dessus ou de le copier dans un navigateur pour accéder à l'application.

```
You can now view your Streamlit app in your browser.  
  
Local URL: http://localhost:8501  
Network URL: http://192.168.1.14:8501
```

FIGURE 6.6 – Génération du lien de l'application