Détection de la maladie de Parkinson

Aicha Ettriki et Abdelhedi Youssef

May 2024

Contents

1 Introduction

La maladie de Parkinson (MP) est un trouble neurologique progressif qui affecte le mouvement et peut provoquer des tremblements, une raideur et des difficultés d'équilibre et de coordination. La détection précoce et le diagnostic précis de la MP sont cruciaux pour une prise en charge et un traitement efficaces de la maladie. Ces dernières années, les progrès des techniques d'apprentissage automatique se sont révélés prometteurs pour faciliter le diagnostic de la MP à l'aide de mesures vocales biomédicales.
Dans ce projet, nous visons à utiliser divers algorithmes d'apprentissage automatique, notamment les k-Nearest Neighbours (KNN), la régression logistique, le classificateur Naive Bayes et les réseaux de neurones pour analyser un ensemble de données comprenant des mesures vocales biomédicales. L'objectif principal de cette étude est de développer des modèles de classification robustes capables de faire la distinction entre les individus en bonne santé et ceux atteints de la maladie de Parkinson sur la base des mesures vocales fournies.

2 Prétraitement des données

Dans le cadre de ce rapport, il est impératif de souligner l'importance de procéder à une analyse de données préalable. L'analyse de données constitue une étape essentielle pour obtenir une compréhension approfondie des informations disponibles et pour prendre des décisions éclairées.Notre base de données est composée de ces colonnes :

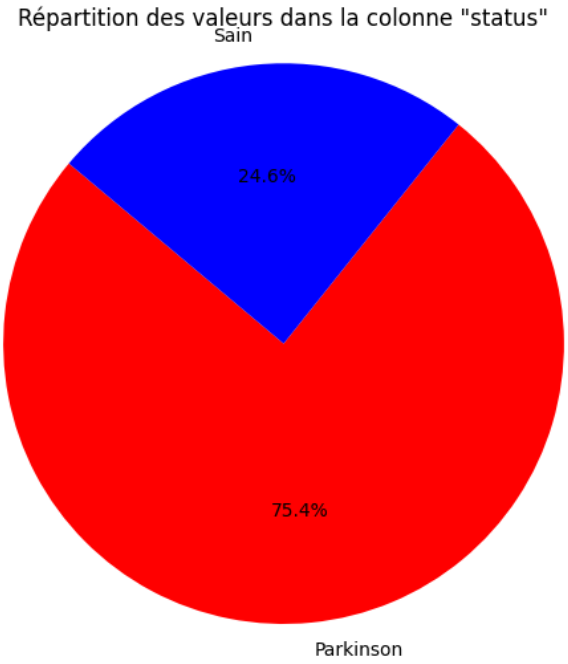
- **nom** - nom du sujet ASCII et numéro d'enregistrement
- **MDVP:Fo(Hz)** - Fréquence fondamentale vocale moyenne
- **MDVP:Fhi(Hz)** - Fréquence fondamentale vocale maximale
- **MDVP:Flo(Hz)** - Fréquence fondamentale vocale minimale
- **MDVP : Gigue (%)**, **MDVP : Gigue (Abs)**, **MDVP : RAP**, **MDVP : PPQ**, **Gigue : DDP** - Plusieurs mesures de variation de la fréquence fondamentale
- **MDVP:Shimmer**, **MDVP:Shimmer(dB)**, **Shimmer:APQ3**, **Shimmer:APQ5**, **MDVP:APQ**, **Shimmer:DDA** - Plusieurs mesures de variation d'amplitude
- **NHR**, **HNR** - Deux mesures du rapport entre le bruit et les composantes tonales de la voix
- **statut** - État de santé du sujet (1) - Parkinson, (0) - en bonne santé
- **RPDE**, **D2** - Deux mesures de complexité dynamique non linéaire
- **DFA** – Exposant de mise à l'échelle fractale du signal
- **spread1**, **spread2**, **PPE** - Trois mesures non linéaires de la variation de fréquence fondamentale

Nous avons que toutes les variables sont des variables numériques sauf pour noms, notre cible qui est 'statut' est un entier composé de 0 et 1 (puisqu'elle est binaire) et les autres sont des floats.

2.1 Attribut cible

Analysons maintenant notre cible nous pouvons voir que les valeurs dans status sont composée de 147 malades et 48 non malades :

```
status
1    147
0     48
Name: count, dtype: int64
```



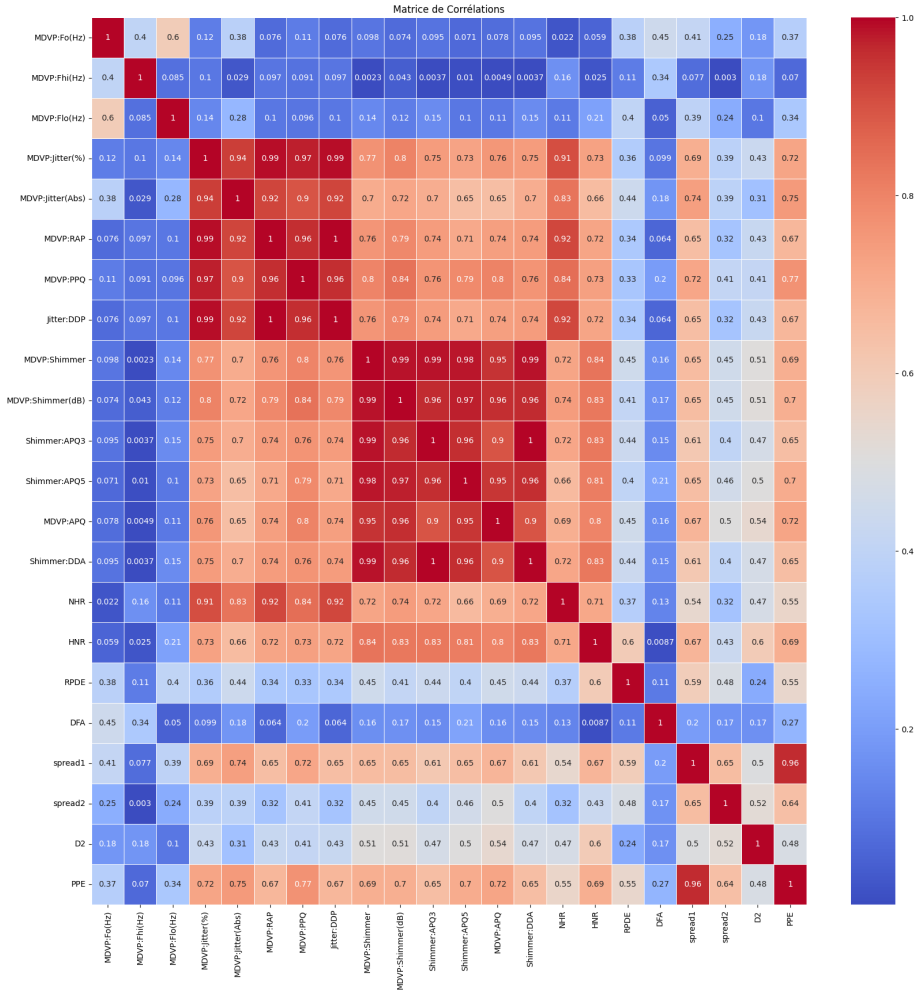
Il y a donc une sur-représentation des malades en comparaison de ceux qui ne le sont pas. La méthode la plus utilisée dans ce cas serait l'oversampling, autrement dit une création artificielle d'individus dans la classe minoritaire. Une des techniques d'oversampling, le SMOTE, consiste à choisir K voisins et à prendre une combinaison linéaire des caractéristiques de la classe minoritaire pour créer un nouvel individu. Ce procédé se répète jusqu'à obtenir une data équilibrée. Cependant, nous avons fait le choix de garder ce déséquilibre pour savoir si cela nous permet quand même d'obtenir des résultats précis.

2.2 Séléction des attributs

Tout d'abord, nous devons créer la matrice de corrélation :

```
df = df.drop(columns=['name'])
X = df.drop(columns=['status'])
```

```
y = df['status']
matrice_correlation = X.corr()
```



Cette matrice nous permet de voir qu'il y a beaucoup de variables fortement corrélées. Les retirer diminuerait des redondances tout en diminuant le risque de sur-apprentissage en raison d'un nombre moindre d'attributs. De plus, pour des modèles d'apprentissage tel que le Naive Bayesian Classifier, l'hypothèse naive est l'independance conditionnelle des attributs. Une forte corrélation implique généralement une forte dépendance, c'est donc également pour cette raison que nous les retirons.

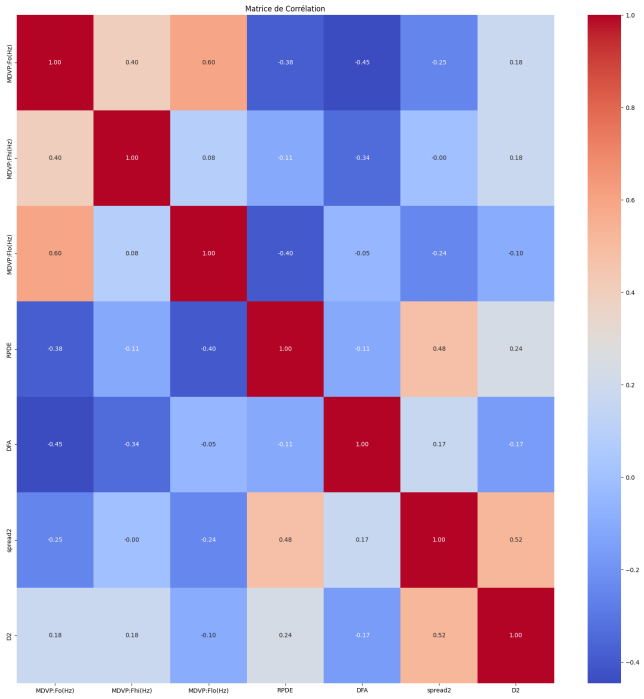
Nous décidons qu'un attribut est fortement corrélé à un autre si leur correlation est supérieure à 0.65.

```
matrice_correlation=matrice_correlation.abs()
groupes_correles = []
for colonne in matrice_correlation.columns:
    if not into(colonne,groupes_correles):
        groupe_correle = {colonne}
        for autre_colonne in matrice_correlation.columns:
            if colonne != autre_colonne and
                matrice_correlation[colonne][autre_colonne] >
                 0.65:
                groupe_correle.add(autre_colonne)
        if len(groupe_correle) > 1:
            groupes_correles.append(groupe_correle)
```

Nous mettons les groupes d'attributs corrélés entre eux dans une liste, et nous choisissons l'attribut à garder en fonction de son coefficient de Ridge. les coefficients de Ridge étant : $\hat{\beta}^{\text{Ridge}} = \left(\mathbf{X^T X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X^T y}$

```
model = Ridge(alpha=1)
for groupe in groupes_finaux:
    print(groupes_finaux)
    X0 = df[list(groupe)]  # Convertir l'ensemble en liste
       pour l'acc s aux colonnes du dataframe
    y0 = df[list(groupe)[0]]
    model.fit(X0, y0)
    coefficients = model.coef_
    attribut_principal = list(groupe)[np.abs(coefficients).
       argmax()]
    groupe.remove(attribut_principal)
    for l in groupe:
        try:
            X = X.drop(columns=[l])
        except KeyError:
            print("Certaines colonnes n'existent pas dans le
                DataFrame X.")
```

La nouvelle matrice devient :



2.3 Normalisation des données

Pour normaliser nos données, nous utilisons un objet `StandardScaler` de la bibliothèque scikit-learn. La normalisation est une étape importante dans le prétraitement des données, car elle permet de mettre à l'échelle les caractéristiques pour qu'elles aient une moyenne de 0 et un écart-type de 1, ce qui est bénéfique pour de nombreux algorithmes d'apprentissage automatique.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

2.4 Division des données

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
```

Dimensions de l'ensemble d'entraînement (X_train) : $(156, 7)$
Dimensions de l'ensemble de test (X_test) : $(39, 7)$
Dimensions de l'ensemble d'entraînement (y_train) : $(156,)$
Dimensions de l'ensemble de test (y_test) : $(39,)$

Cela garantit que les données de test sont normalisées de la même manière que les données d'entraînement, ce qui est important pour obtenir des résultats précis lors de l'évaluation du modèle.

3 Modélisation et évaluation du KNN

3.1 Définition

Le K plus proches voisins (KNN) est un algorithme d'apprentissage supervisé utilisé pour la classification et la régression. Son principe est simple : il classe ou prédit une observation en se basant sur les étiquettes des k observations les plus similaires dans l'ensemble de données d'entraînement. La similarité est souvent mesurée par la distance euclidienne entre les caractéristiques des observations. Le choix de la valeur de k est crucial, car elle influence la robustesse du modèle face au bruit et à la complexité des données.

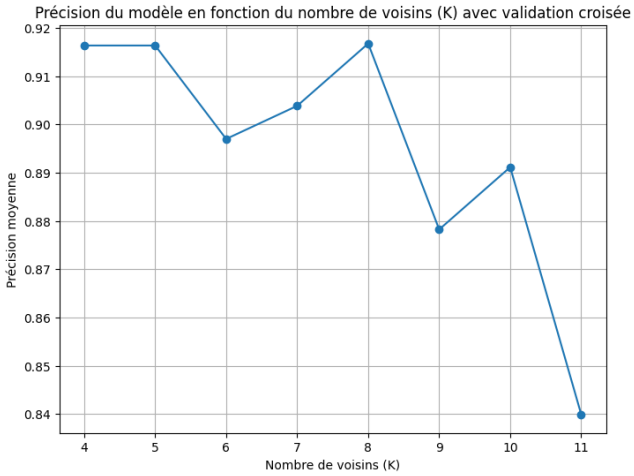
3.2 Modélisation

Pour sélectionner le meilleur nombre de voisins (k) pour un modèle de classification KNN, nous utilisons la validation croisée avec l'objet `cross_val_score` de la bibliothèque scikit-learn. La validation croisée est une technique qui permet d'estimer la performance du modèle sur des données non vues.

```
from sklearn.model_selection import cross_val_score
import numpy as np

def select_best_k(X_train, y_train, k_values, cv=5):
    mean_scores = []
    for k in k_values:
        knn = KNeighborsClassifier(n_neighbors=k)
        scores = cross_val_score(knn, X_train, y_train, cv=
            cv, scoring='accuracy')
        mean_score = np.mean(scores)
        mean_scores.append(mean_score)
    best_k = k_values[np.argmax(mean_scores)]
    return best_k
k_values = [4, 5, 6, 7, 8, 9, 10]
best_k = select_best_k(X_train_scaled, y_train, k_values)
```

Dans ce code, nous utilisons la fonction `select_best_k` pour itérer à travers différentes valeurs de k et sélectionner celle qui donne le meilleur score de validation croisée. Cette fonction utilise la moyenne des scores de validation croisée pour chaque valeur de k afin de sélectionner le meilleur k.



Dans notre cas , le meilleur k c'est 8, Nous appliquons alors notre modèle :

```
knn = KNeighborsClassifier(n_neighbors=8)
knn.fit(X_train_scaled, y_train)
y_pred = knn.predict(X_test_scaled)
```

3.3 Évaluation

```
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
```

Matrice de confusion :

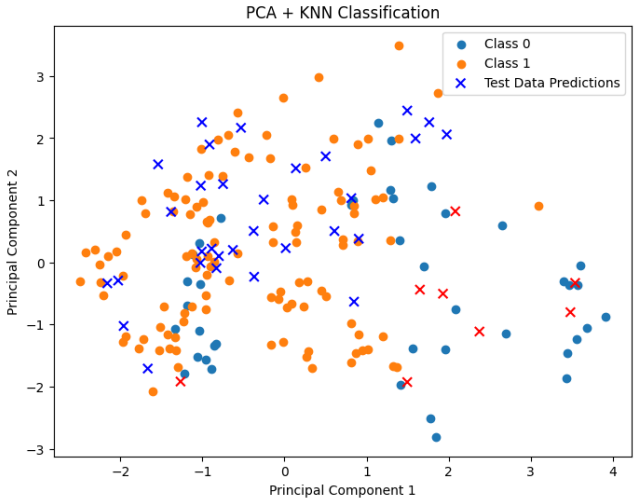
$$\begin{bmatrix} 6 & 1 \\ 3 & 29 \end{bmatrix}$$

Accuracy : 0.8974358974358975
F1-score : 0.9354838709677419

Vrai Positif (VP) : 29
Faux Positif (FP) : 1
Faux Négatif (FN) : 3
Vrai Négatif (VN) : 6

- **Vrai Positif (VP)** : Il y a 29 cas où le modèle a correctement prédit la classe positive.
- **Faux Positif (FP)** : Il y a 1 cas où le modèle a prédit positivement à tort (faux alarmes).

- **Faux Négatif (FN)** : Il y a 3 cas où le modèle a manqué de prédire positivement (faux négatifs).
- **Vrai Négatif (VN)** : Il y a 6 cas où le modèle a correctement prédit la classe négative.
- **Accuracy** :La précision globale du modèle est de 89.7 %. Cela signifie que le modèle est correct dans environ 89.7% des cas.
- **F1-score** : Le F1-score de 0.94 montre que le modèle a un équilibre entre la précision et le rappel.



4 Modélisation et évaluation de la Regression logistique

4.1 Définition

La régression logistique est une technique d'apprentissage supervisé utilisée pour la classification binaire. Elle modélise la relation entre les variables d'entrée et une variable cible binaire en utilisant une fonction sigmoïde, qui produit des prédictions de probabilité. Pendant l'entraînement, les paramètres du modèle sont ajustés pour minimiser l'écart entre les valeurs prédites et les valeurs réelles. Une fois entraîné, le modèle peut classer de nouvelles observations en fonction de leur probabilité estimée, généralement en utilisant un seuil de 0,5 pour la classification binaire.

4.2 Modélisation

Au début, nous avons utilisé la bibliothèque scikit-learn pour modéliser la régression logistique. Cette approche nous a permis de bénéficier de l'efficacité

et de la facilité d'utilisation de la mise en œuvre fournie par scikit-learn, ce qui nous a permis de rapidement construire et évaluer des modèles de régression logistique.
Cependant,nous avons essayer de créer notre propre implémentation de la régression logistique en utilisant des concepts tels que la fonction sigmoïde pour la transformation des valeurs de sortie en probabilités et la descente de gradient pour ajuster les paramètres du modèle. Cette approche manuelle nous a permis d'explorer en détail le fonctionnement de la régression logistique et de mieux comprendre son comportement dans différentes situations.

4.2.1 Régression logistique avec scikit-learn

La descente de gradient est utilisée automatiquement dans scikit-learn.

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train_scaled, y_train)
y_pred_log_reg = log_reg.predict(X_test_scaled)
```

Évaluation

```
accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
f1_log_reg = f1_score(y_test, y_pred_log_reg)
log_loss(y_test, y_pred_log_reg)
```

Les métriques suivantes ont été obtenues après l'entraînement du modèle de régression logistique :

Accuracy : 0.8717948717948718
F1-score : 0.9253731343283582
Log Loss : 4.620981203732968

Interprétation

- **Accuracy** : Avec une précision de 87.2%, le modèle a correctement classé environ 87.2% des exemples de test. Cela indique que le modèle peut distinguer entre les classes.
- **F1-score** : Un F1-score de 0.93 montre que le modèle a un équilibre entre la précision et le rappel. Cela signifie qu'il est bon pour identifier les vrais positifs tout en minimisant les faux positifs et les faux négatifs.
- **Log Loss** : Un log loss de 4.62 indique une pénalité trés élevée pour les prédictions incorrectes. Cela suggère que les prédictions de probabilité du modèle ne sont pas très fiables, ce qui peut nécessiter une calibration ou des ajustements supplémentaires.

Vu que le log loss est trop élevée nous avons décidé de créer un calibrateur de probabilité :

```
from sklearn.calibration import CalibratedClassifierCV
calibrated_lr = CalibratedClassifierCV(log_reg, method='
    sigmoid', cv='prefit')
calibrated_lr.fit(X_train_scaled, y_train)
calibrated_probs = calibrated_lr.predict_proba(X_test_scaled
    )
logloss = log_loss(y_test, calibrated_probs)
y_pred_calibrated = calibrated_lr.predict(X_test_scaled)
```

- **Log Loss après calibration :** 0.3983359823560553
  Cela signifie que le log loss après la calibration du modèle est de 0.398, ce qui indique une bonne performance en termes de précision des probabilités prédites par rapport aux valeurs réelles.
- **Accuracy après calibration :** 0.8717948717948718
- **F1-score après calibration :** 0.8571826183766483

Ces valeurs montrent que, après la calibration du modèle, l'accuracy est de 0.872 et le F1-score est de 0.857. Cela indique que le modèle a une précision élevée et un bon équilibre entre la précision et le rappel dans ses prédictions.

4.2.2 Implémentation personnalisée de la régression logistique utilisant la descente de gradient

Le modèle de régression logistique est de la forme :

$$P(Y = 1|X = x) = \sigma(w^T x + b)$$

Fonction sigmoïde:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Fonction de coût:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

La descente de gradient est un algorithme d'optimisation utilisé pour trouver les valeurs optimales des paramètres d'un modèle en minimisant une fonction de coût. Pour une fonction de coût $J(\theta)$, la mise à jour des paramètres θ à chaque itération est donnée par :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

où α est le taux d'apprentissage et j est l'indice de l'itération. Le terme $\frac{\partial}{\partial \theta_j} J(\theta)$ représente le gradient de la fonction de coût par rapport aux paramètres θ.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
def fonction_cout(X, y, theta, lambda_, epsilon=1e-5):
    m = len(y)
    h = sigmoid(X.dot(theta))
    h = np.clip(h, epsilon, 1 - epsilon)
    cost = (-1/m) * (y.T.dot(np.log(h)) + (1 - y.T).dot(np.
        log(1 - h))) + (lambda_/(2*m)) * np.sum(np.square(
        theta[1:]))
    return cost
```

```
def gradient_descent(X, y, theta, alpha, num_iterations,
    epsilon=1e-5):
    m = len(y)
    cost_history = []
    for i in range(num_iterations):
        h = sigmoid(X.dot(theta))
        gradient = (1/m) * X.T.dot(h - y)
        theta = theta - alpha * gradient
        cost = compute_cost(X, y, theta,epsilon)
        cost_history.append(cost)

    return theta, cost_history
```

Nous avons entraîné notre modèle, mais comme tout à l'heure, nous avons eu une perte logarithmique élevée. Nous avons donc décidé d'appliquer une régularisation Ridge avec un terme de biais lors de la descente de gradient.

```
X_train_scaled_bias = np.c_[np.ones((X_train_scaled.shape
    [0], 1)), X_train_scaled]
X_test_scaled_bias = np.c_[np.ones((X_test_scaled.shape[0],
    1)), X_test_scaled]
theta = np.zeros(X_train_scaled_bias.shape[1])
alpha = 0.1
num_iterations = 1000
lambda_ = 0.1
theta_final, cost_history = gradient_descent(
    X_train_scaled_bias, y_train, theta, alpha,
    num_iterations, lambda_)
y_pred_proba = sigmoid(X_test_scaled_bias.dot(theta_final))
```

Le prédicteur de la régression logistique est de la forme :

$$\hat{y}(x) = \begin{cases} 1 & \text{si } \sigma(w^T x + b) \geq 0.5 \\ 0 & \text{sinon} \end{cases}$$

```
y_pred = (y_pred_proba >= 0.5).astype(int)
```

Résultats de l'évaluation du modèle

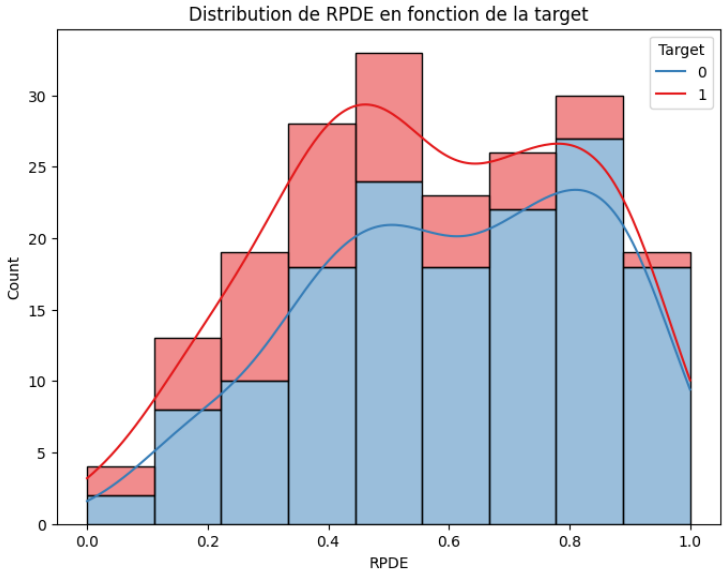
Métrique	Valeur
Accuracy	0.8717948717948718
F1-score	0.9253731343283582
Log Loss	0.38440697994463335

Ces résultats indiquent une performance satisfaisante du modèle.

5 Modélisation et évaluation du Naive Bayesian Classifier

5.1 Définition

Le théorème de Bayes est une formule mathématique qui permet de calculer la probabilité d'un événement conditionnellement à l'observation de certains faits. Autrement dit : $P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$ Dans le contexte du classificateur naïf bayésien, nous cherchons à prédire la classe C d'une observation donnée X. Pour cela, nous calculons la probabilité $P(C = 0|X)$ et $P(C = 1|X)$, la probabilité qui est supérieure détermine la classe. Pour pouvoir calculer ces probabilités, nous nous basons sur une hypothèse dite naive étant l'indépendance conditionnelle des attributs. Nous aurons donc : $P(C|X) = \frac{P(X_1|C) \cdot P(X_2|C) \cdot \ldots \cdot P(X_n|C) \cdot P(C)}{P(X)}$ Cela peut être effectué dans le cas de variables continues (telles que les notres) en utilisant cette formule : $P(C|X) = \frac{f(X|C) \cdot P(C)}{f(X)}$ où f correspond à la densité de probabilité conditionnelle de X sachant que la classe est C.
Nous avons analysé nos données pour déterminer la fonction de densité la plus appropriée à utiliser. Le graphique suivant illustre la distribution de RPDE en fonction de la target :



Voici les principales observations :

- **Classe 0 (bleue) :** La distribution est étalée et présente plusieurs petits pics, indiquant une forme de distribution complexe.
- **Classe 1 (rouge) :**La distribution a un pic principal autour de 0.4 à 0.5 mais n'est pas parfaitement unimodale.

Ces observations suggèrent que les distributions de "RPDE" ne suivent pas une loi continue spécifique, comme la loi normale. En conséquence, il est préférable d'utiliser des méthodes non paramétriques pour estimer les probabilités conditionnelles $P(X_i|Y)$.

5.2 Vérification de l'hypothèse naive

Nous avons en premier lieu décidé de vérifier si l'hypothèse naive est vraie. Pour cela, nous effectuons un test Khi2 d'indépendance sur les attributs gardés lors du prétraitement en mettant les valeurs entre 0 et 1 grâce au minmax Scaler :

```
from scipy.stats import chi2_contingency
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)
chi2, p, dof, ex = chi2_contingency(X.T)
alpha = 0.05  # Niveau de signification
if p < alpha:
    print("Les colonnes sont d pendantes (rejeter l'
        hypoth se nulle).")
```

```
else:
    print("Les colonnes sont ind pendantes (ne pas rejeter
        l'hypoth se nulle).")
```

Les colonnes sont indépendantes (ne pas rejeter l'hypothèse nulle).
Nous allons utiliser le modèle GaussianNB avec l'hyperparamètre var_smoothing réglé à 1 (Lissage de La Place) .Nous l'avons choisis car nos variables sont continues, et le modèle Gaussian Naive Bayes est bien adapté à ce type de données.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix
model = GaussianNB(var_smoothing=1)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

5.3 Évaluation

Ce modèle nous a donné les résultats suivants :
Accuracy : 0.82
Matrice de confusion :

$$\begin{bmatrix} 3 & 4 \\ 3 & 29 \end{bmatrix}$$

Vrai Positif (VP) : 29
Faux Positif (FP) : 4
Faux Négatif (FN) : 3
Vrai Négatif (VN) : 3

- **Vrai Positif (VP)** : Il y a 29 cas où le modèle a correctement prédit la classe positive.
- **Faux Positif (FP)** : Il y a 4 cas où le modèle a prédit positivement à tort (faux alarmes).
- **Faux Négatif (FN)** : Il y a 3 cas où le modèle a manqué de prédire positivement (faux négatifs).
- **Vrai Négatif (VN)** : Il y a 3 cas où le modèle a correctement prédit la classe négative.
- **Accuracy** :La précision globale du modèle est de 82 %. Cela signifie que le modèle est correct dans environ 82% des cas.

Nous avons rajouté un rapport de classification avec :

```
classification_report(y_test, y_pred)
```

	Precision	Recall	F1-Score	Support
0	0.50	0.43	0.46	7
1	0.88	0.91	0.89	32
Accuracy			0.82	39
Macro Avg	0.69	0.67	0.68	39
Weighted Avg	0.81	0.82	0.81	39

Table 1: Résultats des métriques de classification

- **Classe 0:**
  - **Précision :** 0.50
    Parmi toutes les instances prédites comme appartenant à la classe 0, 50 % sont réellement de la classe 0.
  - **Rappel :** 0.43
    Parmi toutes les instances réellement appartenant à la classe 0, 43 % ont été correctement identifiées par le modèle.
  - **F1-Score :** 0.46
    Le F1-score est la moyenne harmonique de la précision et du rappel. Un F1-score de 0.46 montre un équilibre relativement faible entre précision et rappel pour la classe 0.
  - **Support :** 7
    Il y a 7 instances de la classe 0 dans l'ensemble de test.
- **Classe 1:**
  - **Précision :** 0.88
    Parmi toutes les instances prédites comme appartenant à la classe 1, 88 % sont réellement de la classe 1.
  - **Rappel :** 0.91
    Parmi toutes les instances réellement appartenant à la classe 1, 91 % ont été correctement identifiées par le modèle.
  - **F1-Score :** 0.89
    Un F1-score de 0.89 montre un bon équilibre entre précision et rappel pour la classe 1.
  - **Support :** 32
    Il y a 32 instances de la classe 1 dans l'ensemble de test.
- **Moyenne Macro:** (Moyenne arithmétique pour toutes les classes.)
  - **Précision :** 0.69
  - **Rappel :** 0.67
  - **F1-Score :** 0.68
  - **Support :** 39 c'est le total des instances dans l'ensemble de test.

- **Moyenne Pondérée:** (Moyenne pondérée pour toutes les classes, en tenant compte du nombre d'instances de chaque classe.)
  - **Précision :** 0.81
  - **Rappel :** 0.82
  - **F1-Score :** 0.81
  - **Support :** 39 c'est le total des instances dans l'ensemble de test.

6 Réseaux de neurones

6.1 Définition

Les réseaux de neurones exploitent des opérations mathématiques simples mais puissantes pour traiter les données. Chaque neurone effectue une combinaison linéaire des entrées pondérées par des poids, auxquelles est ajouté un biais. Cette combinaison linéaire est ensuite soumise à une fonction d'activation non linéaire, telle que la sigmoïde, ReLU ou Tanh. Ces fonctions introduisent de la non-linéarité dans le modèle, permettant aux réseaux de neurones d'apprendre des relations complexes entre les données. La combinaison linéaire et l'activation se répètent à travers les différentes couches du réseau, chaque couche produisant des représentations de plus en plus abstraites des données.

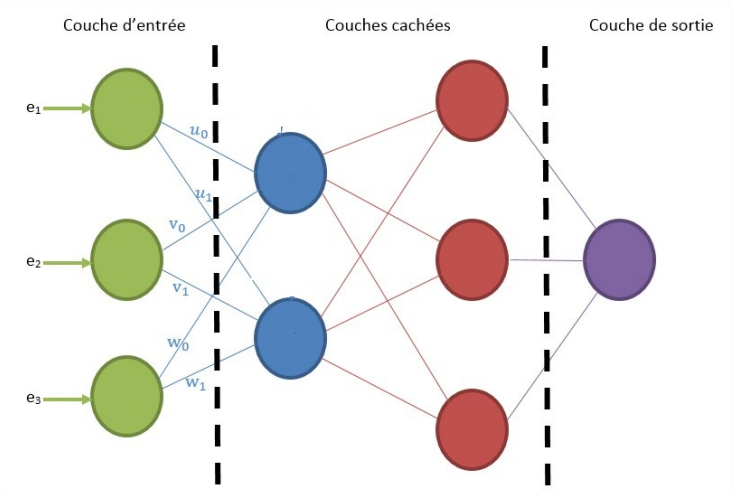


Figure 1: Réseau de neurones

6.2 Choix des hyperparamètres

Dans les réseaux de neurones, les hyperparamètres sont : la fonction d'activation, le nombre de couches et le nombre de neurones par couche. Partons du principe que nous voulons uniquement utiliser 2 couches cachées, nous allons donc, grâce au code qui suit, essayer toutes les possibilités et ainsi choisir celles qui maximisent la précision.

```
def create_and_evaluate_model(X_train, y_train, X_test,
    y_test, neurons1, neurons2,f1,f2, epochs=50, batch_size
    =10):
    model = Sequential()
    model.add(Input(shape=(X_train.shape[1],)))
    model.add(Dense(neurons1, activation=f1))
    model.add(Dense(neurons2, activation=f2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='sgd
        ', metrics=['accuracy'])

    model.fit(X_train, y_train, epochs=epochs, batch_size=
        batch_size, verbose=0)

    predictions = (model.predict(X_test) > 0.5).astype("
        int32")
    accuracy = accuracy_score(y_test, predictions)

    return accuracy
```

ensuite nous essayons les différentes possibilités

```
for neurons1 in tqdm(neurons1_options):
    for neurons2 in tqdm(neurons2_options):
        for f1 in tqdm(function_options):
            for f2 in tqdm(function_options):
                accuracy = create_and_evaluate_model(X_train
                    , y_train, X_test, y_test, neurons1,
                    neurons2,f1,f2)
                print(f"Accuracy with neurons1={neurons1},
                    neurons2={neurons2},f1={f1}, f2={f2}: {
                    accuracy}")
                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    best_params['neurons1'] = neurons1
                    best_params['neurons2'] = neurons2
                    best_params['f1'] = f1
                    best_params['f2'] = f2
```

Best accuracy: 0.8974358974358975 with parameters: 'neurons1': 2, 'neurons2': 3, 'f1': 'selu', 'f2': 'linear'

6.3 Modélisation

Pour avoir une meilleure précision nous devons donc avoir 2 neurones dans la première couche cachée avec une fonction d'activation selu définie comme :

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{si } x > 0 \\ \alpha e^x - \alpha & \text{sinon} \end{cases}$$

Où λ est une constante de mise à l'échelle et α est une constante de pente, généralement définie comme $\lambda = 1.0507$ et $\alpha = 1.6733$. Notre réseaux de neurones sera également constituée d'une seconde couche cachée constituée de 3 neurones avec la fonction d'activation linéaire définie comme :

$$\text{Linear}(x) = x$$

Enfin notre couche de sortie est constituée d'un seul neurone dont la fonction d'activation est sigmoide :

$$\sigma(x) = \begin{cases} 1 & \text{si } \frac{1}{1+e^{-x}} > 0.5 \\ 0 & \text{sinon} \end{cases}$$

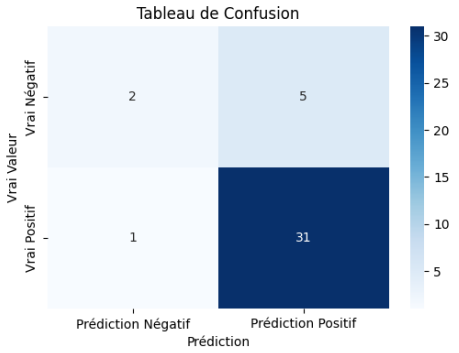
Nous définissons donc notre modèle :

```
model = Sequential()
model.add(Input(shape=(X_train.shape[1],)))
model.add(Dense(2, input_dim=20, activation='selu'))
model.add(Dense(3, activation='linear'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd',
    metrics=['accuracy'])
```

Nous remarquons que nous utilisons la fonction de gradient descendant stochastique pour optimiser les poids

6.4 Évaluation

Nous avons obtenu cette matrice de confusion :



Vrai Positif (VP) : 31

Faux Positif (FP) : 5

Faux Négatif (FN) : 1

Vrai Négatif (VN) : 2

- **Vrai Positif (VP)** : Il y a 31 cas où le modèle a correctement prédit la classe positive.
- **Faux Positif (FP)** : Il y a 5 cas où le modèle a prédit positivement à tort (faux alarmes).
- **Faux Négatif (FN)** : Il y a 1 cas où le modèle a manqué de prédire positivement (faux négatifs).
- **Vrai Négatif (VN)** : Il y a 2 cas où le modèle a correctement prédit la classe négative.

Nous remarquons que le modèle ce qui pourrait etre du au désequilibre de la base de données.

7 Conclusion

Dans ce projet, nous avons diagnostiqué la maladie de Parkinson en nous basant sur les métriques biomédicales des enregistrements. Nous avons commencé par nettoyer les données. Ensuite, nous avons procédé à la phase de sélection des variables pour identifier les caractéristiques les plus pertinentes pour la prédiction de la maladie. Une fois les variables sélectionnées, nous avons créé plusieurs modèles prédictifs, dont les résultats ont été évalués pour déterminer leur précision dans la détection de la maladie de Parkinson. Après une évaluation de nos quatre modèles, les résultats révèlent que le modèle K plus proches voisins (KNN) et le réseau de neurones surpassent les autres modèles, démontrant une précision d'environ 89.7%. La régression logistique suit de près avec une précision de 87.2%, indiquant également une performance solide. Cependant, le classifieur naïf bayésien présente une performance légèrement inférieure, avec une précision d'environ 82.0%.
Il est important de noter que ces résultats sont basés sur un ensemble de données relativement petit, ce qui peut limiter la capacité des modèles d'apprentissage automatique à capturer les schémas sous-jacents et à généraliser correctement, ce qui peut entraîner des performances moins précises. La taille restreinte de l'ensemble de données peut rendre plus difficile pour les modèles d'extraire des informations significatives ce qui peut affecter la précision des résultats obtenus.

8 Sources de données

Le jeu de données utilisé dans cette étude a été créé par Max Little de l'Université d'Oxford, en collaboration avec le National Centre for Voice and Speech, Denver, Colorado, qui a enregistré les signaux vocaux. L'étude originale a publié les méthodes d'extraction des caractéristiques pour les troubles vocaux généraux.
Article :
Little MA, McSharry PE, Roberts SJ, Costello DAE, Moroz IM. "Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection". BioMedical Engineering OnLine 2007, 6:23 (26 June 2007)