

Université de Genève
—
Sciences Informatiques



13X005 Intelligence Artificielle
Projet – Regression Logistique / Naive Bayes

Repository Github: [13X005-AI-Project](#)

Gregory Sedykh, Leandre Catogni, Noah Peterschmitt, Noah Munz, Michel Donnet

Janvier 2024

Contents

1 – Introduction & Rappels théoriques	1
1.1 – Régression Logistique	1
?	1
1.2 – Naive Bayes	2
2 – Méthodologie	3
2.0 – Choix du dataset & outils utilisés	3
2.1 – Gradient Descent	4
2.2 – Régression Logistique	5
2.2.1 – Fonction de coût pour la régression logistique binaire	5
2.2.2 – Fonction de coût pour la régression logistique multinomiale	7
2.2.3 – Apprentissage	12
2.2.4 – Prédictions	13
2.2.5 – Résultats	14
2.3 – Naive Bayes	16
2.3.1 – Extraction des distributions	16
2.3.2 – Prédictions	16
2.3.3 – Résultats	17
3. – Analyse	18
3.1 – Phénomène de sur-apprentissage	20
3.1.1 – Fonctions et signatures	25
4 – Comparaisons	26
4.1 - Vraisemblance et classification des échantillons	26
4.2 - Comparaison avec SKLearn	28
4.2.1 - Naïve Bayes	28
4.2.2 - Régression Logistique	28
4.3 - Conclusion sur les comparaisons	29
Références	30

Projet – Regression Logistique / Naive Bayes

1 – Introduction & Rappels théoriques

Dans ce document, nous approfondirons les techniques de “Régression logistique” et “Naive Bayes” comme outils d’apprentissage supervisés.

Dans le cadre de l’intelligence artificielle et de l’apprentissage supervisé, la compréhension et la classification précise des données revêtent une importance capitale. Parmi les diverses méthodologies existantes, la “Régression Logistique” et “Naive Bayes” se distinguent par leur efficacité et leur applicabilité dans de nombreux contextes. Ce document se propose d’étudier ces deux techniques, en mettant l’accent sur leur mise en œuvre pratique et leur efficacité comparative dans divers scénarios.

1.1 – Régression Logistique

?

En statistiques, la régression logistique, s’inscrit dans le cadre des modèles de régression pour les variables binaires. Bien qu’elle soit quasiment exclusivement utilisée en tant que méthode de classification.

En effet, c’est l’ajout d’un seuil, à la probabilité continue donnée par le model de regression qui nous permet de l’utiliser pour la classification.

Ce type de modèle vise à expliquer de manière optimale une variable binaire, qui représente la présence ou l’absence d’une caractéristique spécifique, à l’aide d’un ensemble conséquent de données réelles et d’un modèle mathématique.

Autrement dit, il s’agit de relier une variable aléatoire de Bernoulli, généralement notée y , aussi appelé “label” à un vecteur constitué de plusieurs variables aléatoires, (x_1, \dots, x_K) , aussi appelés “features”. [5].

La régression logistique s’appuie sur un classifieur linéaire [1] i.e. un classifieur dont la sortie (pour un vecteur de feature $x \in \mathbb{R}^n$) est donnée par:

$$g(x) = f(\langle w, x \rangle + b)$$

où $w \in \mathbb{R}^n$ est le vecteur de poids, $b \in \mathbb{R}$ le biais et $\langle \cdot, \cdot \rangle$ le produit scalaire usuel. f est une fonction dite de seuillage qui va séparer nos résultats. Un choix commun pour f est la sigmoïde ou la fonction signe [1].

Par exemple, dans le cas de la regression logistique binaire, on suppose le modèle suivant:

$$y_i \sim \text{Bernoulli}(p_i), \quad p_i = \sigma(\langle w, x_i \rangle + b), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

où $x_i \in \mathbb{R}^K$ représente un vecteur (ligne) de K valeurs pour les K features (aussi appelé un *sample*), et y_i la variable aléatoire qui représente le label qui leur est associé.

Cependant, dans notre dataset (voir [section 2.0](#)) nous avons 3 classes (3 espèces d’iris), y ne suit donc, évidemment, plus une loi de Bernoulli.

La sigmoïde étant continue, nous avons testé 2 méthodes de prédiction:

- La première consistait simplement à modifier la manière dont nous appliquons le seuillage sur la fonction sigmoïde, pour distinguer 3 cas au lieu de 2. i.e. Au lieu de séparer le domaine en 2 ($\sigma(z) \leq 0.5$, $\sigma(z) > 0.5$), nous l’avons séparé en N (ici $N = 3$). On a donc que $y_i = k \Leftrightarrow \frac{k}{N} \leq \sigma(z) < \frac{k+1}{N}$, ce qui a donné des résultats assez satisfaisants comme nous le verrons en [section 2.2](#).
- La deuxième consistait évidemment en l’application non pas de la régression logistique binaire, mais de la régression logistique multinomiale, fonctionnant avec plusieurs labels. Le principe de la régression logistique multinomiale est simplement de faire plusieurs régressions logistiques binaires. On possède donc un vecteur de poids et un biais par label, et on calcule à chaque fois la probabilité que l’élément appartienne à une certaine classe. La prédiction retournera la classe pour laquelle la probabilité que l’élément appartienne à la classe est la plus élevée.

1.2 – Naive Bayes

“Naive Bayes” se présente comme une méthode de classification probabiliste basée sur le [théorème de Bayes](#), caractérisée par l’adoption d’une hypothèse d’indépendance forte entre les features (attributs), qualifiée de “naïve”.

Plus simplement, le classifieur est considéré comme “naïf” car il part du principe que chaque feature (attribut) est indépendante des autres et a un poids égal quant à la probabilité qu’un point appartienne à une classe.

Ce modèle est dit génératif contrairement à la régression logistique, étant considéré comme “méthode discriminante” [1], et consiste à modéliser les probabilités conditionnelles $P(\mathbf{x}|\text{classe})$ pour chaque classe y et smaple \mathbf{x} afin de trouver celle qui maximise cette probabilité.

En d’autres termes, le problème revient à trouver, pour des attributs x_1, \dots, x_k , la classe \tilde{y} telle que:

$$\tilde{y} = \arg \max_{y \in \mathcal{Y}} \left[P(y) \prod_{k=1}^K P(x_k|Y) \right]$$

2 – Méthodologie

2.0 – Choix du dataset & outils utilisés

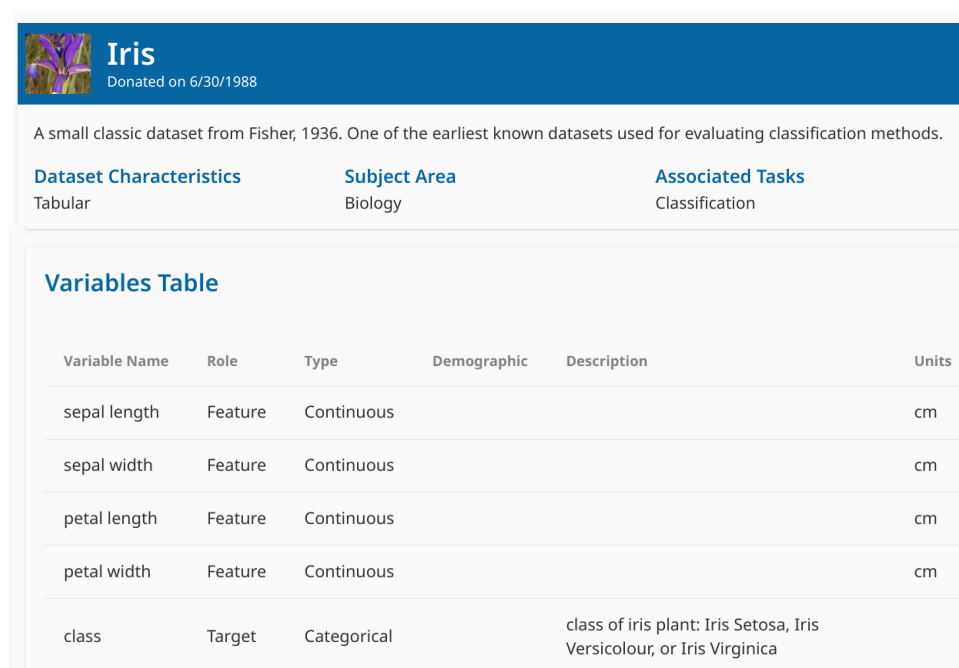
Pour la suite de ce projet les outils suivants ont été utilisés dans chaque parties:

- [python](#)
- [poetry](#)
- [Make](#)
- [numpy](#)
- [pandas](#)
- [sklearn](#)
- [matplotlib](#)
- [ucmilrepo](#)
- [pytest](#)

Le package `ucmilrepo` a été utilisé pour charger les données de notre dataset depuis la base de donnée du [UC Irvine Machine Learning Repository](#).

Le dataset que nous avons choisi est le fameux dataset “Iris” [3], un des plus anciens et connus dataset de classification. Il contient 150 observations de 3 espèces différentes d’iris (Iris setosa, Iris virginica et Iris versicolor) avec $K = 4$ features (longueur et largeur des sépales et pétales).

Voici un aperçu des points-clés du dataset:



Variable Name	Role	Type	Demographic	Description	Units
sepal length	Feature	Continuous			cm
sepal width	Feature	Continuous			cm
petal length	Feature	Continuous			cm
petal width	Feature	Continuous			cm
class	Target	Categorical		class of iris plant: Iris Setosa, Iris Versicolour, or Iris Virginica	

Figure 1: Iris descriptive table

Le label que nous allons prédire sera donc *class*, i.e. l’espèce de l’iris.

2.1 – Gradient Descent

Dans cette section, une implémentation de la “descente en gradient” a été réalisée. La fonction a la signature suivante

```
1 def gradient_descent(df, params: NDArray, alpha: float, num_iters: int) -> NDArray:
```

Elle calcule de manière itérative le(s) paramètre(s) `params` qui minimisent la fonction dont `df` est le gradient avec un “taux de convergence” `alpha`.

La fonction a été testée avec la fonction `scipy.optimize.fmin` [7] de la librairie `scipy` sur la fonction suivante:

$$f(x) = x \cdot \cos(\pi(x + 1))$$

avec différents $x_0 \in \{-\pi, 0, \pi\}$ (valeur initiale de `params`, i.e. `NDArray` avec `D=0`).

Les minimas locaux trouvés par les deux fonctions sont les suivants:

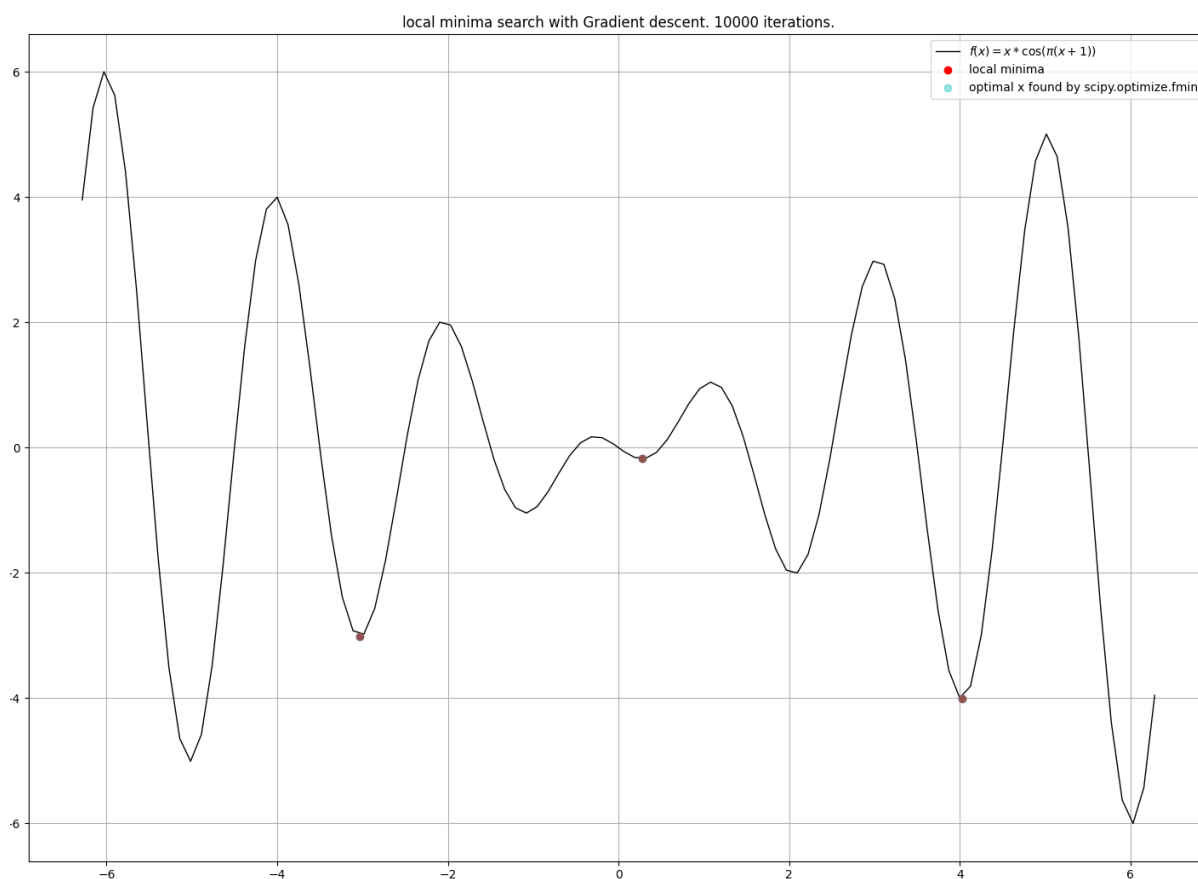


Figure 2: minimas locaux_gradient descent

Ce résultat illustre bien 2 choses: la première est que l’implémentation de la descente en gradient fonctionne correctement puisque chaque points trouvé par notre fonction est confondu avec celui trouvé par la fonction de `scipy` (c’est ce qui donne cette teinte “grise”). La deuxième est que la “qualité” du minima local (i.e. la distance avec le minima globale) dépend fortement de la valeur initiale et ce pour les deux fonctions.

2.2 – Régression Logistique

2.2.1 – Fonction de coût pour la régression logistique binaire

2.2.1.1 – Fonction de coût

Reprenons le modèle décrit dans la section 1.1. Nous avons donc:

$$y_i \sim \text{Bernoulli}(p_i), p_i = \sigma(w^T x_i + b), \sigma(z) = \frac{1}{1 + e^{-z}}$$

Notre but est de calculer $p(y_i|x_i)$, puis de seuiller le résultat obtenu afin de prédire si l'élément possédant les caractéristiques x_i appartient ou pas à la classe y_i . On cherche donc à trouver les paramètres de poids w et de biais b optimaux permettant la meilleure prédiction possible. On peut trouver la notation $p(y_i|x_i; w, b)$ indiquant que nous ne connaissons pas encore le vecteur poids w et le biais b .

La densité de probabilité de cette fonction peut donc s'exprimer comme

$$p(y_i|x_i; w, b) = p_i^{y_i} (1 - p_i)^{1-y_i}$$

Notre but est donc de maximiser cette fonction. Cependant, nous préférons une fonction à minimiser plutôt qu'à maximiser, car la descente en gradient permet de trouver un minimum et non pas un maximum. . .

Une solution habituelle est donc d'inverser la fonction, transformant ainsi le problème de maximisation en problème de minimisation, et de prendre le logarithme de l'inverse de cette fonction afin d'éviter des valeurs extrêmes lors de notre minimisation. L'application de la fonction logarithme sur l'inverse de la fonction est correcte car la fonction logarithme est strictement croissante, donc elle n'aura pas d'impact sur la convexité de la fonction. Cette solution est communément appelée **Negative Logarithm Likelihood**.

Donc on cherchera à minimiser la fonction:

$$\log\left(\frac{1}{p(y_i|x_i; w, b)}\right) = \log(1) - \log(p(y_i|x_i; w, b)) = -\log(p(y_i|x_i; w, b))$$

Pour n données, cette fonction peut s'écrire:

$$-\sum_i^n \log(p(y_i|x_i; w, b))$$

2.2.1.2 – Dérivée de la fonction de coût

Comme nous voulons utiliser la descente en gradient, nous devons trouver la dérivée de la fonction à minimiser, donc de notre fonction de coût.

Tout d'abord, remarquons que nous pouvons écrire:

$$\begin{aligned} & \log(p(y_i|x_i; w, b)) \\ &= \log(p_i^{y_i} (1 - p_i)^{1-y_i}) \\ &= \log(p_i^{y_i}) + \log((1 - p_i)^{1-y_i}) \\ &= y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \end{aligned}$$

De plus, voici ce que nous donne la dérivée de la fonction sigmoïde:

$$\begin{aligned} & \frac{d\sigma(z)}{dz} \\ &= ((1 + e^{-z})^{-1})' \\ &= -1 \times -e^{-z} \times (1 + e^{-z})^{-2} \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \end{aligned}$$

$$\begin{aligned}
 &= \sigma(z) \frac{e^{-z}}{1 + e^{-z}} \\
 &= \sigma(z) \frac{1 + e^{-z} - 1}{1 + e^{-z}} \\
 &= \sigma(z) \left(\frac{1 + e^{-z}}{1 + e^{-z}} - \frac{1}{1 + e^{-z}} \right) \\
 &= \sigma(z) \left(1 - \frac{1}{1 + e^{-z}} \right) \\
 &= \sigma(z)(1 - \sigma(z))
 \end{aligned}$$

Donc nous pouvons facilement calculer la dérivée par rapport au poids w et par rapport au biais de notre fonction de coût.

Voici ce que nous donne la dérivée partielle par rapport au poids $\frac{\partial}{\partial w_j}$:

$$\begin{aligned}
 &\frac{\partial}{\partial w_j} (-\log(p(y_i|x_i; w, b))) \\
 &= -\frac{\partial}{\partial w_j} (y_i \log(p_i) - (1 - y_i) \log(1 - p_i)) \\
 &= -y_i \frac{\partial}{\partial w_j} \log(p_i) - (1 - y_i) \frac{\partial}{\partial w_j} \log(1 - p_i) \\
 &= -y_i \frac{\partial}{\partial w_j} \log(\sigma(z)) - (1 - y_i) \frac{\partial}{\partial w_j} \log(1 - \sigma(z)), \quad z = w^T x_i + b \\
 &= -y_i \frac{1}{\sigma(z)} \frac{\partial}{\partial w_j} \sigma(z) - (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{\partial}{\partial w_j} (1 - \sigma(z))
 \end{aligned}$$

Or on a:

$$\frac{\partial}{\partial w_j} z = \frac{\partial}{\partial w_j} (w^T x_i + b) \Leftrightarrow \frac{dz}{\partial w_j} = x_{ij} \Leftrightarrow \frac{\partial}{\partial w_j} = \frac{d}{dz} x_{ij}$$

Donc:

$$\begin{aligned}
 &-y_i \frac{1}{\sigma(z)} \frac{\partial}{\partial w_j} \sigma(z) - (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{\partial}{\partial w_j} (1 - \sigma(z)) \\
 &= -y_i \frac{1}{\sigma(z)} \frac{d}{dz} \sigma(z) x_{ij} - (1 - y_i) \frac{1}{1 - \sigma(z)} \left(-\frac{d}{dz} \sigma(z) x_{ij} \right) \\
 &= -y_i \frac{1}{\sigma(z)} \sigma(z)(1 - \sigma(z)) x_{ij} - (1 - y_i) \frac{1}{1 - \sigma(z)} (-\sigma(z))(1 - \sigma(z)) x_{ij} \\
 &= -y_i (1 - \sigma(z)) x_{ij} + (1 - y_i) \sigma(z) x_{ij} \\
 &= -y_i x_{ij} + y_i \sigma(z) x_{ij} + (1 - y_i) \sigma(z) x_{ij} \\
 &= -y_i x_{ij} + (y_i + 1 - y_i) \sigma(z) x_{ij} \\
 &= (\sigma(z) - y_i) x_{ij}
 \end{aligned}$$

Voici ce que nous donne la dérivée partielle par rapport au biais $\frac{\partial}{\partial b}$:

$$\begin{aligned}
 &\frac{\partial}{\partial b} (-\log(p(y_i|x_i; w, b))) \\
 &= -\frac{\partial}{\partial b} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) \\
 &= -y_i \frac{\partial}{\partial b} \log(p_i) - (1 - y_i) \frac{\partial}{\partial b} \log(1 - p_i)
 \end{aligned}$$

$$= -y_i \frac{1}{\sigma(z)} \frac{\partial}{\partial b} \sigma(z) - (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{\partial}{\partial b} (1 - \sigma(z)), \quad z = w^T x_i + b$$

On a:

$$\frac{\partial}{\partial b} z = \frac{\partial}{\partial b} (w^T x_i + b) \Leftrightarrow \frac{dz}{db} = 1 \Leftrightarrow \frac{\partial}{\partial b} = \frac{d}{dz}$$

Donc:

$$\begin{aligned} & -y_i \frac{1}{\sigma(z)} \frac{\partial}{\partial b} \sigma(z) - (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{\partial}{\partial b} (1 - \sigma(z)) \\ &= -y_i \frac{1}{\sigma(z)} \frac{d}{dz} \sigma(z) - (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{d}{dz} (1 - \sigma(z)) \\ &= -y_i \frac{1}{\sigma(z)} \frac{d}{dz} \sigma(z) + (1 - y_i) \frac{1}{1 - \sigma(z)} \frac{d}{dz} \sigma(z) \\ &= -y_i \frac{1}{\sigma(z)} \sigma(z)(1 - \sigma(z)) + (1 - y_i) \frac{1}{1 - \sigma(z)} \sigma(z)(1 - \sigma(z)) \\ &= -y_i(1 - \sigma(z)) + (1 - y_i)\sigma(z) \\ &= -y_i + y_i\sigma(z) + (1 - y_i)\sigma(z) \\ &= -y_i + (y_i + 1 - y_i)\sigma(z) \\ &= \sigma(z) - y_i \end{aligned}$$

Donc pour n données, la dérivée de la fonction de coût par rapport au poids et au biais nous donne:

$$\frac{\partial}{\partial w_j} \left(- \sum_i^n \log(p(y_i|x_i; w, b)) \right) = \sum_i^n (\sigma(z) - y_i) x_{ij}$$

et:

$$\frac{\partial}{\partial b} \left(- \sum_i^n \log(p(y_i|x_i; w, b)) \right) = \sum_i^n (\sigma(z) - y_i)$$

2.2.2 – Fonction de coût pour la régression logistique multinomiale

Afin d'entraîner les paramètres de la régression logistique, il faut pouvoir comparer les résultats obtenus par la régression avec les résultats attendus.

On souhaite définir une fonction à minimiser permettant de trouver les paramètres optimaux de la régression logistique.

Notre classification se base sur la fonction sigmoïde $\sigma(z) = \frac{1}{1+e^{-z}}$.

Comme la fonction exponentielle est toujours positive, on a bien que $\sigma(z) \in [0, 1]$.

La fonction sigmoïde nous donne la probabilité que l'élément donné appartienne à un label.

Autrement dit, la fonction sigmoïde est la fonction de répartition de la régression logistique.

Soit $Y \in \{0, 1\}$ les différents labels que peut prendre l'élément que l'on considère et soit X l'ensemble des caractéristiques connues de l'élément, dont on cherche à déterminer dans quelle classe le mettre, donc quel label on doit lui attribuer. Soit θ le vecteur des poids des covariables, indiquant à quel point les covariables influencent sur la décision du label. On a donc:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(X_1 \cdot w_1 + X_2 \cdot w_2 + \dots + b)}}$$

et

$$P(Y = 0|X) = 1 - \frac{1}{1 + e^{-(X_1 \cdot w_1 + \dots + b)}}$$



Pour plus de simplicité, on va considérer que le biais est compris dans les poids: au lieu d'écrire $z = wX + b$, on écrit $z = \hat{X}\theta$ avec $\hat{X} = [X \ 1]$ modifié ou on a ajouté une colonne avec que des 1 à la fin de la matrice X et $\theta = [w \ b]$ afin d'avoir une bonne cohérence avec le rapport et le code. (On a trouvé cela plus facile d'avoir pour chaque labels les poids et bias sur une ligne, donc d'avoir θ_1 pour le label 1 etc...) Ainsi, on a: $\hat{X}\theta^T = X_1 \cdot w_1 + X_2 \cdot w_2 + \dots + b$

Pour la suite, on va noter $X = \hat{X}$

Notre régression logistique binaire peut donc s'écrire comme:

$$P(Y = 1|X) = \frac{1}{1 + e^{X\theta^T}} = \sigma(X\theta^T)$$

et

$$P(Y = 0|X) = 1 - \sigma(X\theta^T)$$

2.2.2.1 – Généralisation de la régression logistique binaire On désire donc trouver une nouvelle distribution $\phi(z)$ tel que:

$$\phi(z) \in [0, 1] \ \forall z$$

est une généralisation de la fonction $\sigma(z)$

On veut donc que pour une régression logistique binaire, on ait $\sigma(z) = \phi(z)$.

On peut remarquer que:

$$\begin{aligned} P(Y = 1|X) &= \frac{1}{1 + e^{-X\theta^T}} \\ &= \frac{1}{1 + e^{-X\theta^T}} \cdot \frac{e^{X\theta^T}}{e^{X\theta^T}} \\ &= \frac{e^{X\theta^T}}{e^{X\theta^T} + e^{X\theta^T - X\theta^T}} \\ &= \frac{e^{X\theta^T}}{e^{X\theta^T} + e^0} \\ &= \frac{e^{X\theta^T}}{e^{X\theta^T} + 1} \end{aligned}$$

On peut considérer que nous avons un vecteur de poids pour chaque label.

Ainsi, on a $\theta_0 = [w_0 \ b_0]$ pour le label 0 et $\theta_1 = [w_1 \ b_1]$ pour le label 1.

Comme on a besoin seulement d'un vecteur de poids pour déterminer le label de nouveaux éléments avec leurs caractéristiques, on peut considérer que $\theta_0 = [0 \ \dots \ 0]$.

Ainsi, la formule précédente nous donne:

$$\begin{aligned} P(Y = 1|X) &= \frac{e^{X\theta_1^T}}{e^{X\theta_1^T} + 1} \\ &= \frac{e^{X\theta_1^T}}{e^{X\theta_1^T} + e^0} \\ &= \frac{e^{X\theta_1^T}}{e^{X\theta_1^T} + e^{0 \cdot X}} \\ &= \frac{e^{X\theta_1^T}}{e^{X\theta_1^T} + e^{X\theta_0^T}} \end{aligned}$$

$$= \frac{e^{X\theta_1^T}}{\sum_{i=0}^1 e^{X\theta_i^T}}$$

On peut donc généraliser cette formule pour K labels.

Cela nous donne:

$$P(Y = k|X) = \frac{e^{X\theta_k^T}}{\sum_{i=0}^K e^{X\theta_i^T}}$$

Comme la fonction exponentielle est toujours positive, on a bien que:

$$\begin{aligned} 0 &\leq e^{X\theta_k^T} \leq e^{X\theta_k^T} + \sum_{i \neq k}^K e^{X\theta_i^T} \\ &\Leftrightarrow 0 \leq e^{X\theta_k^T} \leq \sum_i^K e^{X\theta_i^T} \\ &\Leftrightarrow 0 \leq \frac{e^{X\theta_k^T}}{\sum_i^K e^{X\theta_i^T}} \leq 1 \\ &\Leftrightarrow 0 \leq \phi(z) \leq 1 \end{aligned}$$

De plus, on a que:

$$\begin{aligned} &\sum_k^K P(Y = k|X) \\ &= \sum_k^K \frac{e^{X\theta_k^T}}{\sum_i^K e^{X\theta_i^T}} \\ &= \frac{\sum_k^K e^{X\theta_k^T}}{\sum_i^K e^{X\theta_i^T}} \\ &= \frac{\sum_i^K e^{X\theta_i^T}}{\sum_i^K e^{X\theta_i^T}} \\ &= 1 \end{aligned}$$

Donc la fonction $\phi(z)$ est bien une fonction de distribution de probabilité qui généralise la fonction sigmoïde pour des problèmes à plusieurs labels.

Cette fonction est couramment appelée fonction **softmax**.

2.2.2.2 – Fonction de coût Notre objectif est donc de trouver une fonction de coût pour pouvoir entraîner les paramètres de la régression multinomiale. On cherche à maximiser la vraisemblance des données. Donc pour un label Y donné, on veut maximiser:

$$\sum_k^K f(Y, k) P(Y = k|X)$$

avec $f(Y, k)$ la fonction qui vaut 1 si $Y = k$ et 0 sinon.

Comme on a plusieurs couples de données (X_i, Y_i) , on peut écrire la fonction précédente comme:

$$\sum_i^n \sum_k^K f(Y_i, k) P(Y_i = k|X_i)$$

En maximisant cette fonction, on fait en sorte que le paramètre θ_k permette d'obtenir la prédiction que le label soit égal à k avec la somme des probabilités où $Y_i = k$ est la plus grande possible.

Afin de pouvoir utiliser un algorithme comme la descente en gradient, il faut non pas maximiser une fonction, mais minimiser une fonction.

Tout d'abord, comme on travaille avec des exponentielles, on a intérêt à prendre un logarithme pour éviter d'avoir à travailler avec de trop grandes valeurs. Cette modification n'aura pas d'impact sur la convexité car la fonction logarithme est une fonction strictement croissante.

Enfin, comme on cherche une fonction à minimiser et non pas à maximiser pour pouvoir utiliser la descente en gradient, on va prendre l'inverse de la fonction.

Cela s'appelle couramment le **negative logarithm likelihood**.

Cela nous donne une fonction de coût comme suit:

$$\begin{aligned} & \sum_i^n \sum_k^K f(Y_i, k) \log\left(\frac{1}{P(Y_i = k|X_i)}\right) \\ & \sum_i^n \sum_k^K f(Y_i, k) (\log(1) - \log(P(Y_i = k|X_i))) \\ & - \sum_i^n \sum_k^K f(Y, k) \log(P(Y_i = k|X_i)) \end{aligned}$$

On peut minimiser cette fonction de coût grâce à une descente en gradient.

2.2.2.3 – Dérivée de la fonction de coût On va calculer la dérivée de la fonction de coût.

On a:

$$\begin{aligned} & \log(P(Y = k|X)) \\ & = \log\left(\frac{e^{X\theta_k^T}}{\sum_i^K e^{X\theta_i^T}}\right) \\ & = X\theta_k^T - \log\left(\sum_i^K e^{X\theta_i^T}\right) \end{aligned}$$

Donc:

$$\frac{\partial}{\partial \theta_j} \sum_i^K f(Y, i) \log(P(Y = i|X))$$

(NB: On considère que $Y = k$, tous les autres termes étant annulés car $f = 0$)

$$\begin{aligned} & = \frac{\partial}{\partial \theta_j} f(Y, k) \log(P(Y = k|X)) \\ & = \frac{\partial}{\partial \theta_j} \left(X\theta_k^T - \log\left(\sum_i^K e^{X\theta_i^T}\right) \right) \end{aligned}$$

Supposons que $j = k$.

$$\begin{aligned} & = X - \frac{\partial}{\partial \theta_j} \log\left(\sum_i^K e^{X\theta_i^T}\right) \\ & = X - \frac{1}{\sum_i^K e^{X\theta_i^T}} \frac{\partial}{\partial \theta_j} \sum_i^K e^{X\theta_i^T} \\ & = X - \frac{1}{\sum_i^K e^{X\theta_i^T}} \frac{\partial}{\partial \theta_j} e^{X\theta_j^T} \end{aligned}$$



$$\begin{aligned}
 &= X - \frac{X e^{X\theta_j^T}}{\sum_i^K e^{X\theta_i^T}} \\
 &= X - X P(Y = j|X) \\
 &= X(1 - P(Y = j|X))
 \end{aligned}$$

Supposons que $j \neq k$.

$$\begin{aligned}
 &\frac{\partial}{\partial \theta_j} \left(X\theta_k^T - \log \left(\sum_i^K e^{X\theta_i^T} \right) \right) \\
 &= -\frac{\partial}{\partial \theta_j} \log \left(\sum_i^K e^{X\theta_i^T} \right) \\
 &= -\frac{1}{\sum_i^K e^{X\theta_i^T}} \frac{\partial}{\partial \theta_j} \sum_i^K e^{X\theta_i^T} \\
 &= -\frac{1}{\sum_i^K e^{X\theta_i^T}} \frac{\partial}{\partial \theta_j} e^{X\theta_j^T} \\
 &= -\frac{X e^{X\theta_j^T}}{\sum_i^K e^{X\theta_i^T}} \\
 &= -X P(Y = j|X)
 \end{aligned}$$

On a donc:

$$\frac{\partial}{\partial \theta_j} \sum_i^K f(Y, i) \log(P(Y = i|X)) = X(f(Y, j) - P(Y = j|X))$$

car $f(Y, k)$ est égal à 1 si $Y = k$ et 0 sinon.

Donc pour n données, la dérivée de notre fonction de coût nous donne:

$$\frac{\partial}{\partial \theta_j} \left(-\sum_m^n \sum_i^K f(Y_m, i) \log(P(Y_m = i|X_m)) \right) = -\sum_m^n X_m (f(Y_m, j) - P(Y_m = j|X_m))$$

Maintenant, on est prêt pour entraîner notre régression logistique multinomiale !

2.2.3 – Apprentissage

Maintenant que nous avons une fonction de coût permettant de quantifier (en moyenne) à quel point un set de N prédictions est correct/incorrect à un point de l'apprentissage donné, il ne reste plus qu'à chercher les paramètres optimaux qui minimisent cette fonction de coût. Ce que l'on va réaliser à l'aide de la descente en gradient. C'est le processus d'apprentissage.

En effet, lors de l'apprentissage, on va chercher de manière itérative les \mathbf{w} et b (ou le θ) qui respectent les critères mentionnés ci-dessus en calculant le gradient de la fonction de coût à chaque itérations et en allant dans la direction opposé.

Concrètement cela revient à appliquer l'algorithme suivant:

Algorithm 1 gradient descent

```

function GRADIENTDESCENT( $f, \mathbf{w}_{init}, b_0, \alpha, \text{num\_iters}$ )
     $\mathbf{w} \leftarrow \mathbf{w}_{init}$ 
     $b \leftarrow b_0$ 
    for 1 to num_iters do
         $d\mathbf{w}, db \leftarrow \nabla f(w, b)$ 
         $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot d\mathbf{w}$ 
         $b \leftarrow b - \alpha \cdot db$ 
    end for
    return  $w, b$ 
end function

```

En pratique, il est plus simple de passer directement la fonction qui calcule le gradient en argument, que d'essayer de le calculer dynamiquement, c'est pourquoi la signature de notre implémentation prend un \mathbf{df} en argument plutôt que la fonction de coût elle même.

Où le calcul des dérivées partielles a été défini comme ci-dessous (plus de détails de calculs sont présents dans le point 2.2.1).

Soit $\nabla C(\mathbf{w}, b) = (\frac{\partial C(\mathbf{w}, b)}{\partial \mathbf{w}}, \frac{\partial C(\mathbf{w}, b)}{\partial b})$, pour un sample \mathbf{x}_i et sa classe y_i , on obtient:

$$\begin{aligned}
 -\frac{\partial \log(y_i | \mathbf{x}_i; \mathbf{w}, b)}{\partial b} &= \sigma(z_i) - y_i = \sigma(\mathbf{w}^T X_i + b) - y_i \\
 -\frac{\partial \log(y_i | \mathbf{x}_i; \mathbf{w}, b)}{\partial w_j} &= x_{ij} \cdot (\sigma(z_i) - y_i) = (\sigma(\mathbf{w}^T X_i + b) - y_i) \cdot x_{ij}
 \end{aligned}$$

Or le db dans l'algorithme ci-dessus se réfère à la moyenne (pour tout i) de ces valeurs (i.e. distance moyenne *classes prédites* – *“vrai” classes*).

On l'obtient donc comme suit: (la somme des dérivées est la dérivée de la somme, linéarité de la dérivée)

$$\nabla_b C = -\frac{1}{N} \sum_{i=1}^N \frac{\partial \log(y_i | \mathbf{x}_i; \mathbf{w}, b)}{\partial b} = \frac{1}{N} \sum_{i=1}^N \sigma(\mathbf{w}^T X_i + b) - y_i$$

De même pour $d\mathbf{w}$:

$$\begin{aligned}
 \nabla_{\mathbf{w}} C &= -\frac{1}{N} \sum_{i=1}^N (x_{ij}(y_i - p_i))_{1 \leq j \leq k} = \frac{1}{N} \sum_{i=1}^N (\sigma(z_i) - y_i) \cdot (x_{ij})_{1 \leq j \leq k} \\
 &= \frac{1}{N} \sum_{i=1}^N (\sigma(\mathbf{w}^T \mathbf{x}_i + b) - y_i) \mathbf{x}_i
 \end{aligned}$$

On retrouve ainsi, le calcul effectué dans la fonction `grad` de `log_reg.py` de signature suivante:

```

1  def grad(X: NDArray, y: NDArray, w: NDArray, b: float) -> tuple:

```

Etant donné que pour le calcul du gradient il est nécessaire d'avoir une matrice de feature X et vecteur de label y , une version “modifiée” de la descente en gradient a été implémenté.

```
1 def grad_desc_ml(features: NDArray, labels: NDArray, df, w: NDArray, b: float, alpha: float,
    num_iters: int) -> tuple[NDArray, float]:
```

Cette fonction se comporte exactement de la même manière que celle décrite en [section 2.1](#). La seule différence est qu'elle passe `features` et `labels` comme `X` et `y` à la fonction `df` (dans notre cas `df` est toujours la fonction `grad`), i.e. on a `df(features, labels, w, b)` au lieu de `df(params)`.

La régression logistique multinomiale est entraînée de manière similaire à la régression logistique binaire. Cependant, comme expliqué précédemment, on utilise la matrice de poids et biais θ et la matrice \hat{X} qui est la matrice X avec une colonne de 1 ajoutée.

Le calcul du gradient (plus de détails dans la section 2.2.2) est effectué par la fonction `gradient_cost_function` de `softmax.py` de signature suivante:

```
1 def gradient_cost_function(X: np.ndarray, theta: np.ndarray, Y: np.ndarray) -> np.ndarray
```

La descente en gradient aura de nouveau dû être modifiée pour être fonctionnelle: on a besoin de pouvoir passer les paramètres \hat{X} , θ et Y à la fonction `gradient_cost_function`. Ainsi, la signature de la fonction `gradient_descent_softmax` définie dans `softmax.py` est donnée par:

```
1 def gradient_descent_softmax(df, X: np.ndarray, Y: np.ndarray, theta: np.ndarray, alpha:
    float, num_iters: int) -> np.ndarray:
```

2.2.4 – Prédictions

2.2.4.1 – Régression logistique binaire Pour la prédiction, nous avons utilisé la fonction suivante:

```
1 def predict_log_reg(X: NDArray, w: NDArray, b):
```

qui prend simplement $\sigma(w^T X + b)$ et seuile la sortie du sigmoïde de manière à retourner un nombre entre 0 et 2 (avec les poids et biais entraînés).

2.2.4.2 – Régression logistique multinomiale Pour la prédiction de la régression logistique multinomiale, nous avons utilisé la fonction suivante:

```
1 def predict_log_reg_2(X: NDArray, theta: NDArray):
```

qui est définie également dans `softmax.py`.

Cette fonction applique softmax sur les données d'entrée, puis retourne le label qui contient la valeur maximale obtenue par softmax.

2.2.5 – Résultats

2.2.5.1 – Résultats obtenus par la régression logistique avec la fonction sigmoïde Suite à l'apprentissage, nous avons obtenu les résultats suivants:

$$w = [0.53452349, 0.36463584, 1.16132476, 1.08204578]$$

$$b = 0.45146791$$

N.B.:

L'apprentissage peut être ré-effectué de manière efficiente si besoin et à l'aide du jupyter notebook [training_test.ipynb](#) disponible sur la branche [gpu-training](#) du repository github. Le code de l'entraînement (uniquement sur cette branche) a été "porté" sur cuda / gpgpu à l'aide de la librairie [cupy](#) [2].

A noter qu'il utilise des fonctions de sklearn alors que nous devions les implémenter nous mêmes, (telles que les metrics f1-score...). Ces fonctions ont bien été implémentées mais pour une raison de simplicité, elles n'ont pas été utilisées pour l'entraînement. Le code de cette branche ne fera donc pas partie du rendu mais reste publiquement accessible sur github.

Comme dit en section 1.1, ces paramètres sont, en effet, très satisfaisant, comme on peut le voir sur l'output de `pytest` suivant:

```
1 src/log_reg.py::test_log_reg_f1score
2 weights & biases: [0.53452349, 0.36463584, 1.16132476, 1.08204578], 0.45146791
3 { 'accuracy': 1.0, 'f1_score': 1.0, 'precision': 1.0, 'recall': 1.0 }
4 PASSED
5
6 src/naive_bayes.py::test_predict_bayes_f1score_all
7 { 'accuracy': 0.97, 'f1_score': 0.975, 'precision': 0.976, 'recall': 0.974 }
8 PASSED
```

NB: pour reproduire cette output, lancer `make test_model`.

Ce résultat a été obtenu avec une séparation 70/30 de training/test data. Lorsque l'on essaye de changer la portion qui est prise aléatoirement dans chaque catégorie, on obtient un F1-score qui varie entre 0.93 et 1.0 (avec, dans de rares exceptions 0.91 ou 0.89).

De plus, l'on voit que les performances que nous avons obtenus rentrent tout à fait dans le cadre de celles annoncées par le UCI ML Repository:

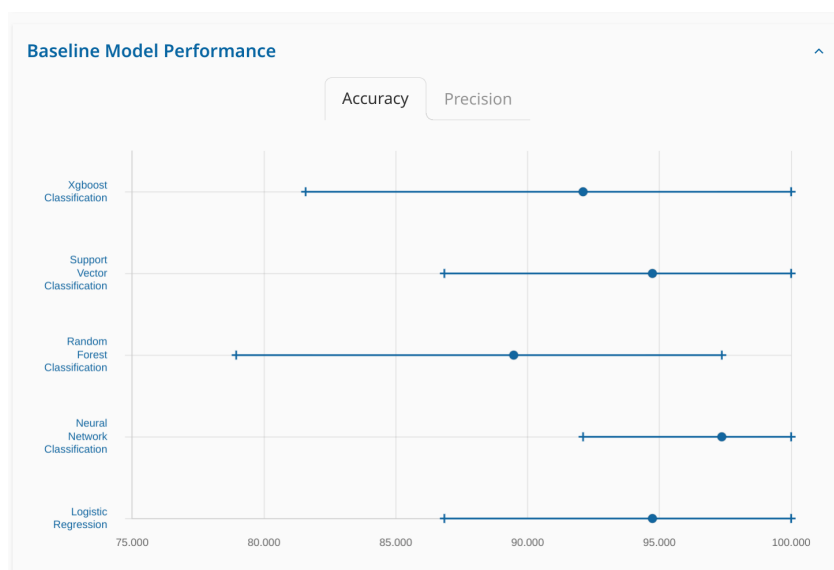


Figure 3: performances attendu d'après le UCI ML Repo [4]

Ce résultat illustre bien que notre démarche est correcte et que nos 2 modèles sont efficaces, avec un penchant pour la régression logistique qui semble être plus efficace que Naive Bayes.

2.2.5.2 Résultats obtenus pour la régression logistique multinomiale Suite à l'apprentissage du modèle multinomiale, on obtien les performances suivantes:

```
python3 main.py
```

Results:

```
Found theta (bias are last column and weights are the rest):  
[[ 0.32626066  0.83466238 -1.21626121 -0.55348121  0.17005888]  
 [ 0.20948959 -0.28684921  0.15659967 -0.18787991  0.11458452]  
 [-0.53575025 -0.54781317  1.05966154  0.74136112 -0.2846434 ]]
```

Metrics obtained:

```
{'precision': 1.0, 'recall': 1.0, 'accuracy': 1.0, 'f1_score': 1.0}
```

On peut être très satisfait de ces résultats. Pour les obtenir, nous avons fait le choix de ne pas initialiser aléatoirement la matrice θ , mais de l'initialiser à zéros. En effet, comme on n'utilise que 1000 itérations et un `learning_rate` de 10^{-4} , initialiser le vecteur avec des valeurs comprises par exemple entre 1 et 10 fera que la descente en gradient optimisera moins bien les paramètres nécessaire (à cause du faible nombre d'itérations), ce qui n'est pas l'objectif de l'initialisation aléatoire de la matrice θ .

Afin de reproduire ces résultats, il suffit de décommenter la ligne `#softmax.main()` dans `main.py`

Ici, sur la matrice theta, nous avons chaque ligne qui représente les poids et biais pour chaque label, avec à chaque fois le dernier élément de la ligne qui est le biais et le reste qui est le poids.

2.3 – Naive Bayes

Dans cette section, une implémentation d'un classifieur linéaire bayésien (naive bayes) a été réalisée.

2.3.1 – Extraction des distributions

Dans cette implémentation, étant données que toutes nos features sont continues, nous avons considéré que *sepal length*, *sepal width*, *petal length* et *petal width* seront représenté comme 4 variables aléatoires X_0, \dots, X_3 suivant 4 lois normales normales de paramètre (μ_k, σ_k) .

C'est à dire:

$$X_k \sim \mathcal{N}(\mu_k, \sigma_k) \quad k \in \llbracket 0, 3 \rrbracket$$

Elles peuvent être récupérées à l'aide de la fonction suivante:

```
1 def get_distrib_parameters(features: DataFrame, labels) -> dict[Any, list[tuple[f1, f1]]]:
```

qui va retourner un dict mappant chaque classe à une liste contenant les paramètres des distributions conditionnelles (normales) des features pour cette classe.

2.3.2 – Prédiction

Deux fonctions de prédictions ont été implémenté,

1. Prennant un sample et prédisant sa classe
2. Une deuxième qui prend tous les samples et applique, en parallèle, la première fonction à chacun d'eux.

Elles ont les signatures suivantes:

```
1 def predict_bayes(x, params_by_class: dict[Any, list[tuple[f1, f1]]]) -> Any:
2 def predict_bayes_all(X: DataFrame, params_by_class: dict[Any, list[tuple[f1, f1]]] | None
   = None) -> list[Any]:
```

Comme dit précédemment, pour pouvoir prédire la classe d'un sample, il faut calculer les probabilité conditionnelle $P(\mathbf{x}|classe)$ pour chaque classe y et sample \mathbf{x} et prendre la classe qui maximise cette dernière.

Cela revient à chercher le \tilde{y} défini en [section 1.2](#), développons le calcul qui nous amené à cette formule:

$$\tilde{y} = \arg \max_{y \in \mathcal{Y}} P(y|\mathbf{x}) = \arg \max_{y \in \mathcal{Y}} \frac{P(\mathbf{x}|y)P(y)}{P(\mathbf{x})} = \arg \max_{y \in \mathcal{Y}} P(\mathbf{x}|y)P(y)$$

Or

$$P(\mathbf{x}|y) = P(x_1|y) \prod_{i=2}^n P(x_i|x_{i-1}, \dots, x_1, y)$$

Avec l'hypothèse que les $\{X_i\}_{i \leq n}$ sont indépendants, on obtient que:

$$P(x_i|x_{i-1}, \dots, x_1, y) = P(x_i|y)$$

Donc

$$P(\mathbf{x}|y) = P(x_1|y) \prod_{k=2}^K P(x_k|y) = \prod_{k=1}^K P(x_k|y)$$

En conclusion:

$$\tilde{y} = \arg \max_{y \in \mathcal{Y}} \left[P(y) \prod_{k=1}^K P(x_k|y) \right]$$

(où K reste le nombre de features.)



Où au début on cherche à maximiser $P(y|x)$ car idéalement on voudrait savoir la probabilité que y soit le bon label pour n'importe quel sample \mathbf{x} . Cependant, on aimerait pouvoir effectuer cette prédictions pour des \mathbf{x} qui n'appartiennent pas à notre dataset d'apprentissage, i.e. on ne doit pas avoir besoin d'avoir déjà vu exactement ce sample. On a donc besoin d'une généralisation, c'est ainsi que l'on fini par retomber sur

$$\tilde{y} = \arg \max_{y \in \mathcal{Y}} \left[P(y) \prod_{k=1}^K P(x_k|y) \right]$$

qui est ce que calculent les fonctions dont on a donné la signature ci-dessus.

2.3.3 – Résultats

Dans cette section, nous allons simplement reprendre ce qui a été fait dit dans la [section 2.2.4](#) et remonter les mêmes tests.

Voici l'output du test `pytest` pour les rapports de performances du model bayesien:

```

1 src/log_reg.py::test_log_reg_f1score
2 weights & biases: [0.53452349, 0.36463584, 1.16132476, 1.08204578], 0.45146791
3 { 'accuracy': 1.0, 'f1_score': 1.0, 'precision': 1.0, 'recall': 1.0 }
4 PASSED
5
6 src/naive_bayes.py::test_predict_bayes_f1score_all
7 { 'accuracy': 0.97, 'f1_score': 0.975, 'precision': 0.976, 'recall': 0.974 }
8 PASSED

```

Ce résultat a été obtenu avec une séparation 70/30 de training/test data.

Ces résultats illustrent bien que notre démarche est correcte et que nos 2 modèles sont efficaces, avec un penchant pour la régression logistique qui semble être plus efficace que Naive Bayes.

Cependant, un f1-score de > 0.95 reste excellent.

3. – Analyse

Pour chaque classe y , on peut tracer les fonctions de distribution de probabilité pour chaque donnée X_k sachant la classe y afin d'analyser la structure des données.

Pour la classe $Y=0$, on obtient le graphe suivant :

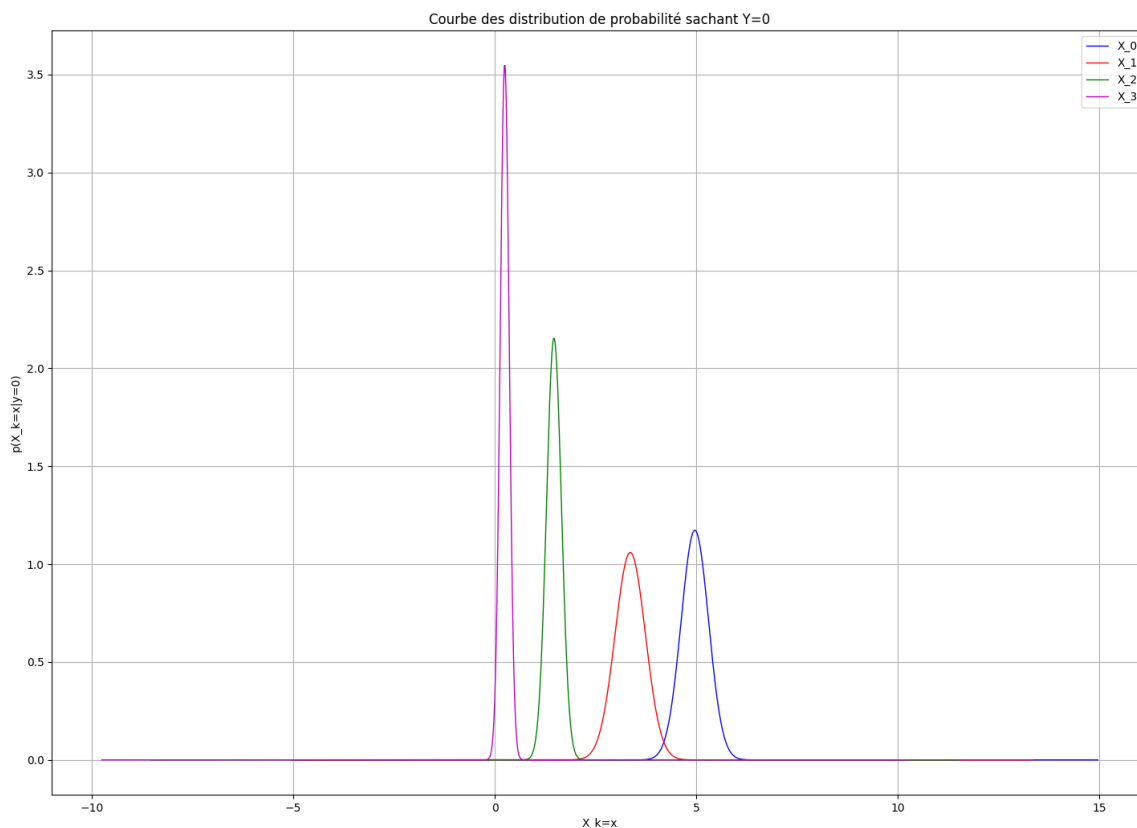


Figure 4: graphe des fonctions de distribution sachant $Y=0$

On peut voir tout d'abord que pour cette classe, les pics des courbes bleue et rouge sont bien inférieurs aux pics des courbes vert et magenta. Ainsi, on en conclut que les variables X_0 et X_1 ont moins d'influence dans la prédiction de cette classe. Alors que le pic de la courbe magenta est bien supérieur aux autres, indiquant que la variable X_3 a une forte influence sur la prédiction de cette classe. De plus, on observe que seules les courbes bleue et rouge ont un chevauchement perceptible mais quand même assez petit, on en conclut que les variables sont pour cette classe très peu indépendantes les unes des autres.

Pour la classe $Y=1$, on obtient le graphe suivant :

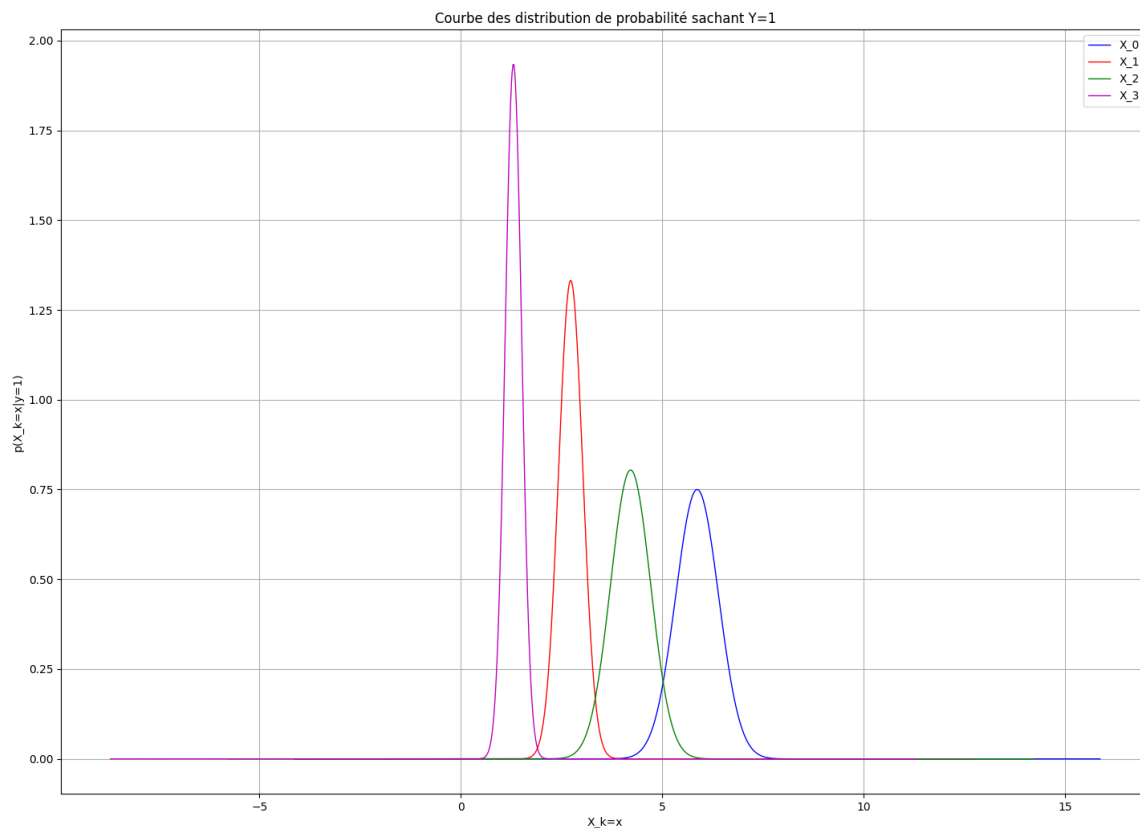


Figure 5: graphe des fonctions de distribution sachant $Y=1$

On peut voir tout d'abord que pour cette classe, les pics des courbes bleus et verte sont bien inférieurs aux pics des courbes rouge et magenta. Ainsi, on en conclut que les variables X_0 et X_2 ont moins d'influence dans la prédiction de cette classe. Alors que le pic de la courbe magenta est bien supérieur aux autres, indiquant que la variable X_3 a une forte influence sur la prédiction de cette classe. De plus, on observe que les courbes rouge et magenta ont un faible chevauchement indiquant une faible interdépendance entre X_3 et X_1 alors que les courbes rouge et verte ainsi que verte et bleue ont un chevauchement assez élevé montrant une certaine interdépendance entre les variables X_1 et X_2 ainsi qu'entre les variables X_0 et X_2 .

Enfin pour la classe $Y=2$, on obtient le graphe suivant :

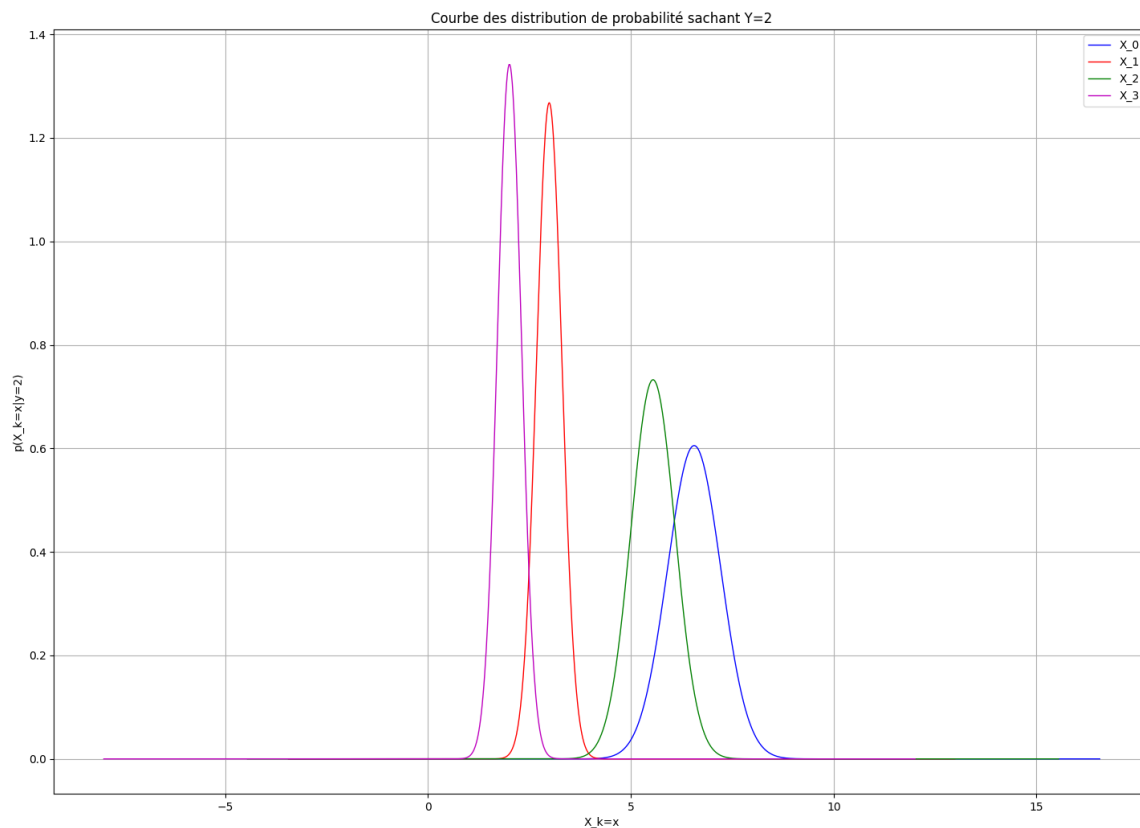


Figure 6: graphe des fonctions de distribution sachant $Y=2$

On observe que les pics des courbes rouge et magenta sont presque deux fois plus grand que ceux des courbes bleue et verte, de plus les courbes rouge se chevauchent fortement. Ainsi les variables X_3 et X_1 ont une très forte influence sur cette classe et sont assez interdépendant alors que les variables X_0 et X_2 ont très peu d'impacte sur la prédiction de cette classe. Les courbes bleue et verte se chevauchent aussi énormément montrant aussi une forte interdépendance entre les variables X_0 et X_2 .

Ainsi on peut remarquer que globalement la variable X_3 a une forte influence sur la classification alors que la variable X_0 a une plus faible. De plus, les variables indépendantes ne sont pas séparables les unes des autres.

3.1 – Phénomène de sur-apprentissage

Dans cette partie, nous allons voir les problèmes de sur-apprentissage du modèle de régression logistique multinomial.

Le phénomène de sur-apprentissage est lorsque le modèle entraîné s'adapte trop bien aux données d'apprentissage, si bien qu'il s'adapte au bruit des données d'apprentissage. Cela a pour conséquences de produire de moins bonnes prédictions pour des données de test. En effet, le modèle est "trompé" par le bruit des données d'apprentissage et prédira ainsi de mauvais résultats.

Dans l'énoncé, on nous propose d'utiliser un volume de données réduit afin de montrer le phénomène de sur-apprentissage. Nous étions alors perplexe et nous nous posions cette simple question: pourquoi utiliser moins de données pour montrer le phénomène engendré lorsqu'on entraîne trop un modèle ? À cette question, nous n'avons hélas pas encore trouvé de réponse qui tranche. En effet, peut-être qu'il faut montrer qu'avec moins de données, le modèle entraîné peut avoir de bonnes performances sur les données d'apprentissage, mais pas sur les données de test. Mais cela ne serait-il pas plutôt un phénomène

de sous-apprentissage ? Ou alors faut-il essayer d'entraîner un maximum notre paramètre θ sur les données d'apprentissage en espérant que notre modèle s'adaptera trop bien aux données d'apprentissage en donnant de mauvaises prédictions pour des données de test ? Ou encore doit-on ajouter beaucoup de bruit dans les données d'apprentissage pour que le modèle s'entraîne sur le bruit des données d'apprentissage et donne de moins bonnes prédictions pour des données de test ? Bref, ne sachant pas trop quelle voie explorer, nous avons décidé d'en explorer plusieurs.

Tout d'abord, nous avons testé et entraîné les modèles pour un volume de données réduit.

Voici 2 graphes montrant les résultats obtenus avec la régression logistique multinomiale et l'approche naive bayes. Notez que nous avons tenté de faire une courbe approximant les données obtenues afin de mieux visualiser ce qu'il se passe. Cette courbe est obtenue, en faisant un peu de bricolage, par le code suivant:

```
a, b = np.polyfit(x, log(y), 1)
plt.plot(x, a * log(y) + b)
```

Ce qui est une sorte de régression linéaire "adaptée" pour une courbe logarithmique.

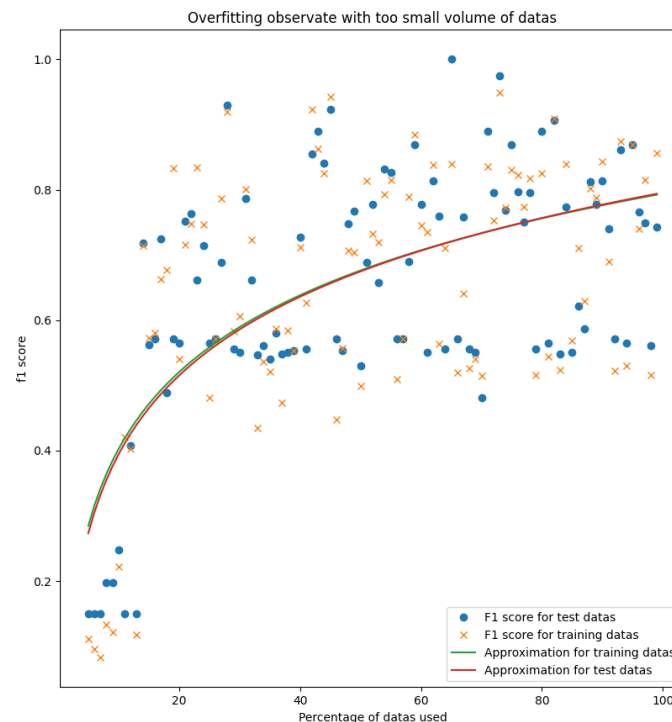


Figure 7: Naive bayes

Sur le graphique donné par Naive Bayes, nous pouvons constater que les f1 scores varient beaucoup, aussi bien pour les données d'entraînement que pour les données de test. Cependant, on peut remarquer que pour un volume faible de données, la courbe verte, correspondant au f1 score obtenu avec les données d'entraînement, est légèrement au dessus de la courbe rouge, représentant le f1 score des données de test. Cela signifie que lorsqu'on n'a pas assez de données d'entraînement, le modèle entraîné ne fera pas de bonnes prédictions sur les données de test, car celui-ci n'a pas été suffisamment entraîné pour bien ajuster ses paramètres: ses paramètres sont uniquement ajustés pour les données d'entraînement.

Cependant, nous pouvons constater que les f1 scores obtenus par Naive Bayes ne convergent pas... C'est pourquoi, afin de montrer les phénomènes de sur-apprentissage, nous allons utiliser la régression logistique qui possède des f1 scores convergeant, comme nous pouvons l'observer sur le graphique.

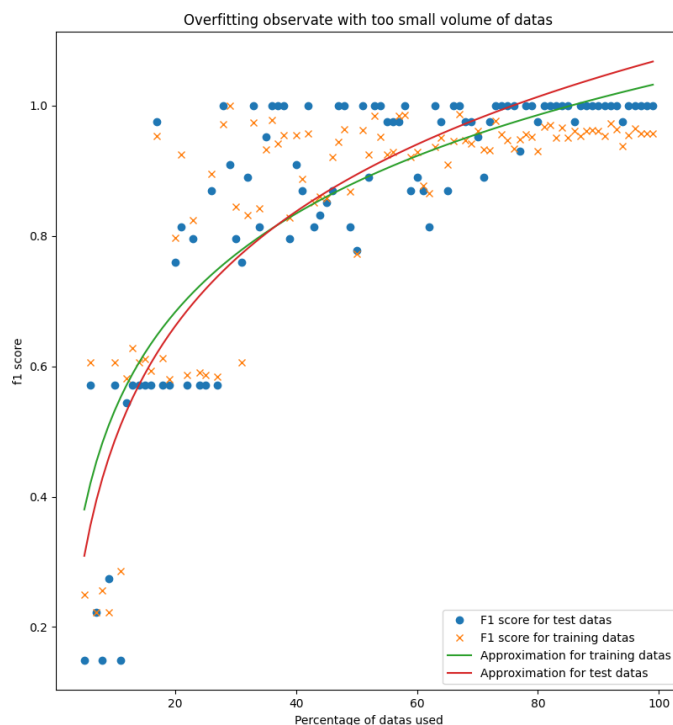


Figure 8: Logistic regression

Sur le graphique donné par la régression logistique, nous pouvons observer que les f1 scores sont très variés pour un volume faible de données, mais que ceux-ci convergent lorsque le volume de données augmente (On suppose que la même chose se produit avec Naive Bayes, mais qu'on n'a pas assez de données pour le remarquer...). C'est pour cette raison que nous allons préférer utiliser la régression logistique plutôt que Naive Bayes pour montrer les phénomènes de sur-apprentissage.

Nous pouvons également constater que les performances sur les données d'entraînement sont meilleures que les performances sur les données de test. Cela montre un "sur-ajustement" du modèle sur les données d'entraînement. En effet, comme le volume de données est faible, les paramètres de la régression logistique sont bien entraînés pour les données d'entraînement, mais pas pour les données de test.

Cependant, lorsque le volume de données augmente, on voit que les paramètres de la régression logistique s'ajustent correctement, et on a des performances sur les données de test qui deviennent meilleures que les performances sur les données d'entraînement.

Donc plus on a des données, mieux on pourra ajuster notre modèle. Dans notre cas, on ne peut pas tester ce que trop de données peuvent faire, car on n'a hélas pas un stock de données illimité.

Mais nous pouvons nous demander à quel point le bruit peut influencer la performance de notre modèle.

Afin d'éclaircir ce point, nous allons vous épargner la visualisation du graphique obtenu par Naive Bayes car celui-ci est très éparse: aucune donnée ne converge. Mais nous allons vous montrer le graphique obtenu par la régression logistique.

Tout d'abord, pour ajouter du bruit aux données, nous indiquons un pourcentage des données qu'on veut bruite. Ensuite, nous prenons d'une manière aléatoire le pourcentage donné de données, et pour ces données sélectionnées, nous attribuons un label aléatoire parmi la liste de labels initiaux des données. Cela entraîne forcément qu'une donnée qu'on voulait bruite a repris le même label qu'elle avait, et n'est donc pas bruité. Donc le bruitage est approximatif.

Voici le graphique obtenu pour la régression logistique:

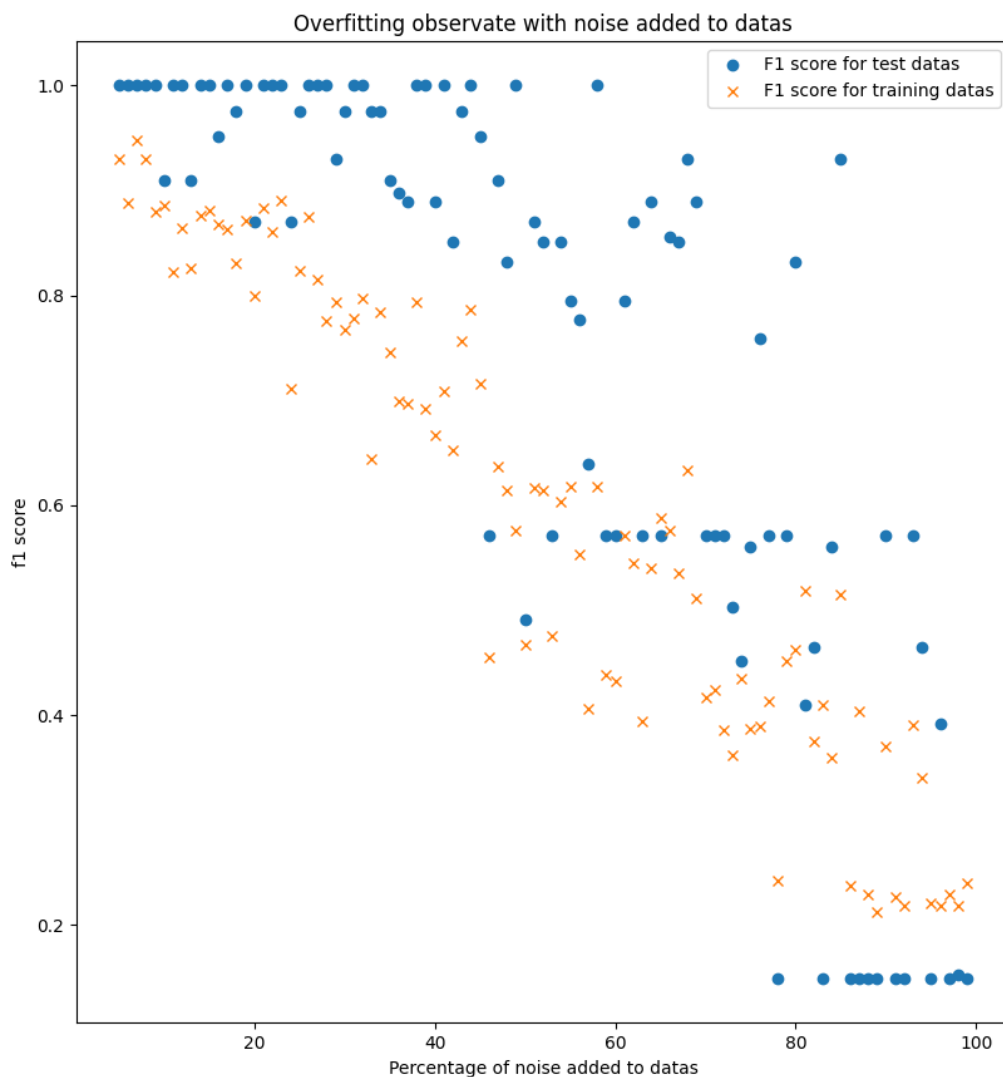


Figure 9: Logistic regression: noise added

Nous pouvons constater que pour des données pas très bruitées, le modèle possède de bonnes performances. Mais plus le bruit introduit dans les données augmente, plus on obtient des résultats éparpillés et décroissant dans l'ensemble. Enfin, on peut observer pour 80 à 100% des données bruitées que le modèle possède de

très mauvaises performances sur les données de test, qui sont moins bonnes que les performances obtenues sur les données d'entraînement. Cela est causé par le bruit introduit dans les données: il y a suffisamment de bruit dans les données pour que le modèle soit entraîné au bruit des données d'apprentissage, ce qui cause de mauvaises performances sur des données de test.

Donc trop de bruit dans les données d'apprentissage d'un modèle peut rendre un modèle avec de mauvaises performances, car trop entraîné au bruit des données d'apprentissages.

Enfin, nous avons tenté de faire varier le nombre d'itérations lors de la descente en gradient de la régression logistique et ajouté du bruit aux données pour que les paramètres de la régression logistique soient perturbés par le bruit des données. Nous avons mis 50% de bruit dans les données d'apprentissage, ce qui nous a donné le graphe suivant:

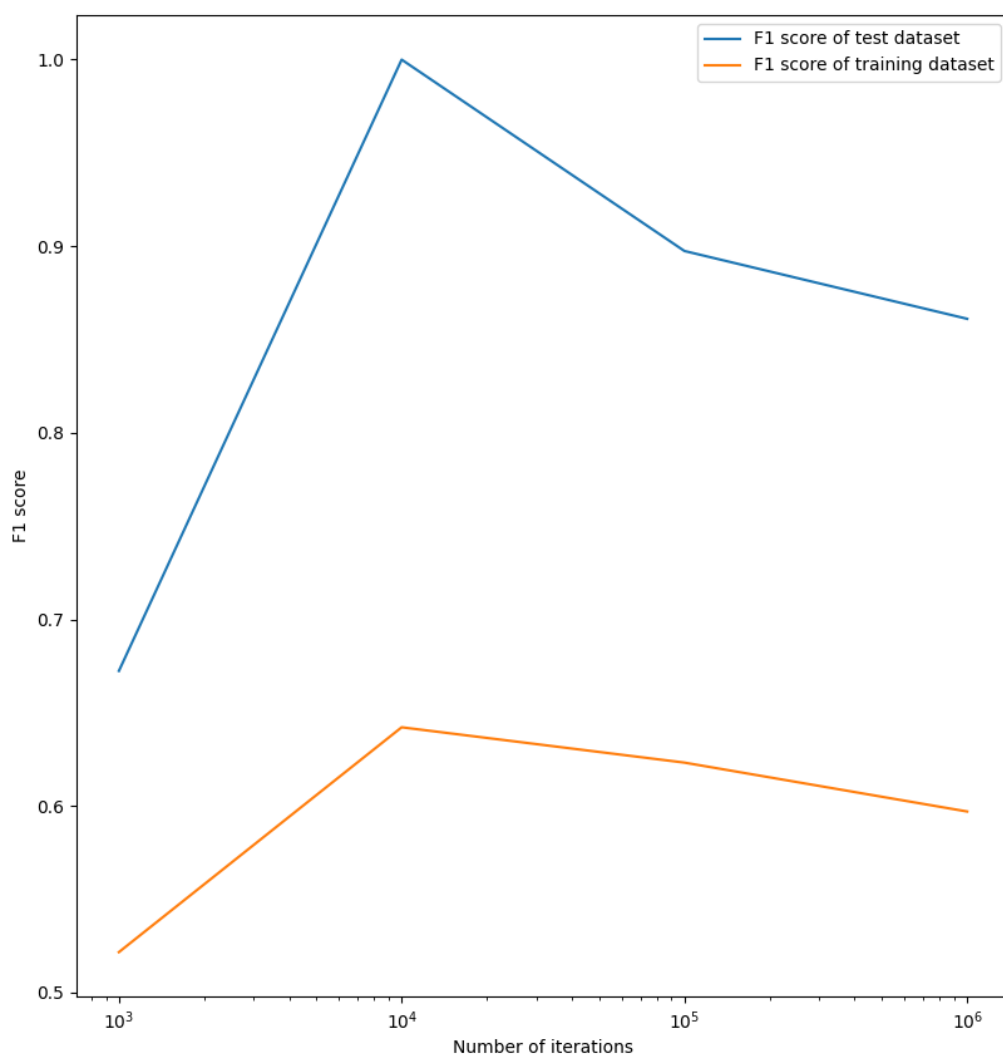


Figure 10: Logistic regression: varying number of iterations

Les résultats nous plaisent car ils montrent que pour un faible nombre d'itérations, les performances du modèle sont bonnes, mais moins bonnes que pour un nombre moyen d'itérations, et pour un nombre trop

grand d'itérations, les performances du modèle sont moins bonnes.

Cela signifie que si la descente en gradient est trop précise, on aura un modèle bien entraîné pour les données d'apprentissage, mais pas pour les données de test, car la performance du modèle sur les données de test diminue lorsqu'on a un nombre d'itérations élevé, tandis que la performance du modèle reste à peu près constante pour les données d'entraînement.

Ainsi, il faut également ajuster la précision de la descente en gradient afin d'obtenir non pas les paramètres optimaux pour les données d'apprentissage, mais pour obtenir des paramètres à la fois généraux et précis, permettant de donner des performances optimales sur des données de test.

Nous pouvons conclure que le sur-apprentissage ou sur-ajustement peut causer des problèmes dans les performances d'un modèle de classification, comme la régression logistique, car le modèle peut soit s'entraîner trop au bruit des données d'apprentissages, entraînant de mauvais résultats de test, soit ne pas avoir suffisamment de données d'apprentissage pour bien ajuster ses paramètres, soit trop s'entraîner aux données d'apprentissage, ce qui entraîne une baisse de performance sur les données de tests, car le modèle est également entraîné au bruit des données d'apprentissage.

Enfin, il faut toujours s'assurer que le modèle donne les meilleures performances possibles sur les données de test, car les performances sur les données de test comptent plus que les performances sur les données d'apprentissage...

3.1.1 – Fonctions et signatures

Pour bruite les données, une fonction a été créée dans `overfitting.py`. Cette fonction possède la signature suivante:

```
1 def add_noise_to_data(labels: np.ndarray, percentage: int) -> np.ndarray:
```

Elle prend des données aléatoirement et les bruite en réattribuant le label correspondant aux caractéristiques. On peut avoir le label qui est défini d'une manière aléatoire à sa valeur initiale.

Pour prendre seulement un pourcentage des données, une autre fonction a été créée dans `overfitting.py`. Cette fonction possède la signature suivante:

```
1 def get_percentage_of_data(feats: np.ndarray, labels: np.ndarray, percentage: int) -> np.ndarray:
```

Cette fonction prend un pourcentage des données qui ont été, au préalable, "mélangées" aléatoirement.

Les graphiques ont été obtenus grâce aux autres fonctions définies dans `overfitting.py`. On peut les exécuter en décommentant les lignes suivantes de la toute fin de `overfitting.py` et en décommentant la ligne `overfitting.main()` de main:

```
#overfitting_naive_bayes(FEAT, LABELS, FEAT_test, LABELS_test)
#overfitting_log_reg(FEAT, LABELS, FEAT_test, LABELS_test)
#overfitting(FEAT, LABELS, FEAT_test, LABELS_test)
```

Le code peut cependant prendre du temps à l'exécution: il prenait environ 2 minutes à tourner sur la machine d'un des étudiants.

4 – Comparaisons

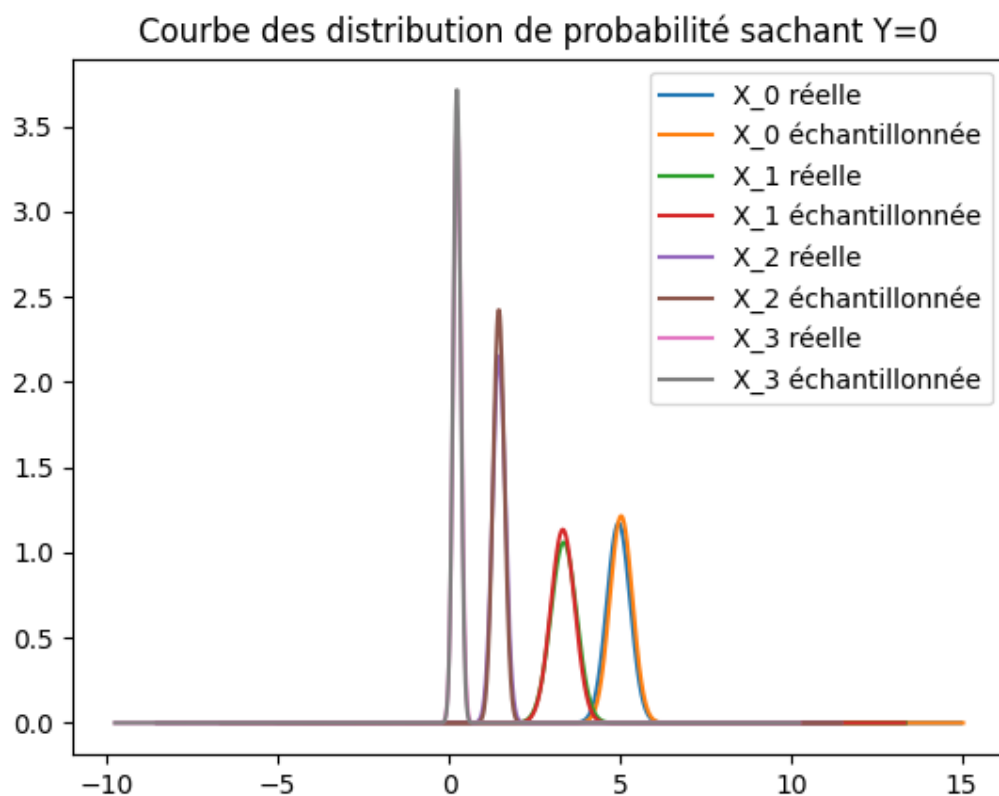
4.1 - Vraisemblance et classification des échantillons

Une fois que les paramètres des classes sont obtenus en supposant l'indépendance des variables, on échantillonne de nouvelles données afin de comparer les résultats obtenus avec les données d'origine.

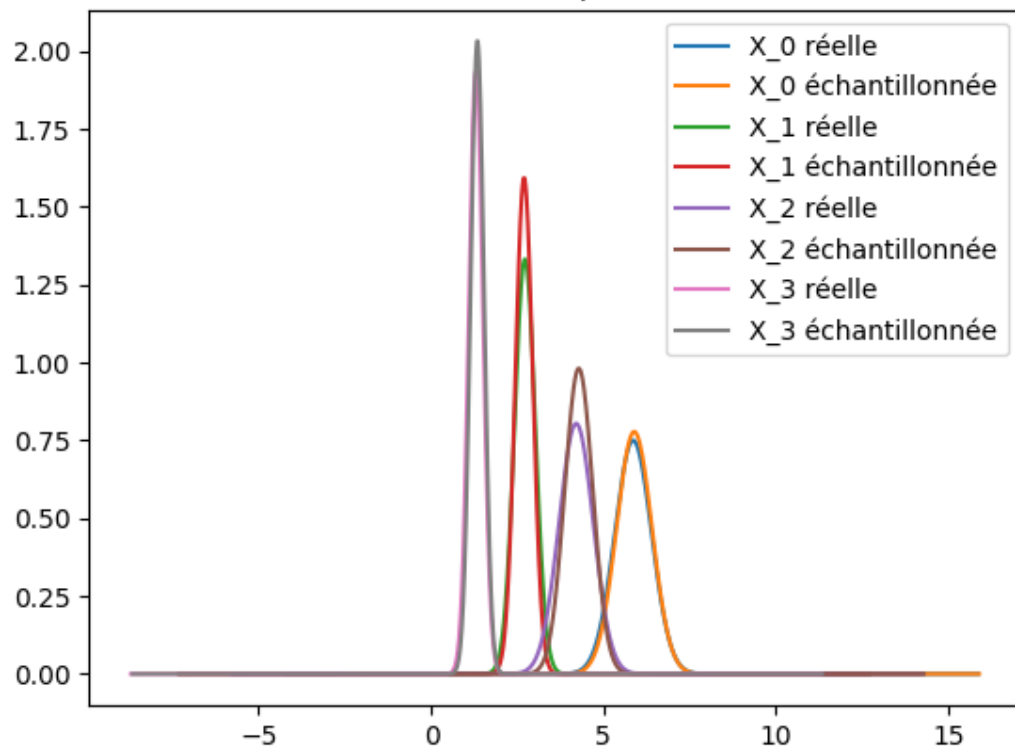
L'échantillonnage est fait dans le fichier `sampling.py`.

On fait 50 échantillons pour chaque classe, à partir des paramètres des distributions obtenus dans la section précédente.

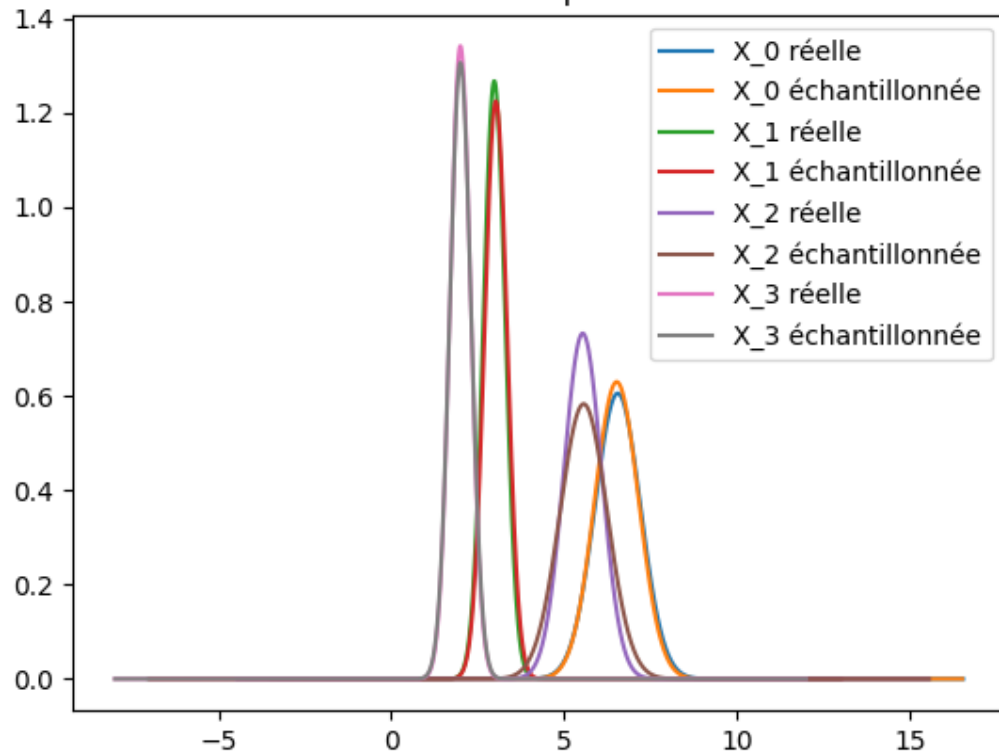
On obtient les résultats suivants (la moyenne et l'écart-type sont donnés pour chaque classe et chaque variable):



Courbe des distribution de probabilité sachant $Y=1$



Courbe des distribution de probabilité sachant $Y=2$



- Pour la classe 0:

- Mean:
 - * réelle: 4.964516, 3.3612902, 1.467742, 0.2451613
 - * échantillon: 5.04191904, 3.33580458, 1.46340112, 0.23831319
- Ecart-type:
 - * réel: 0.34014544, 0.37654343, 0.18508933, 0.112067565
 - * échantillon: 0.32847194, 0.35161457, 0.16439592, 0.10696828
- Pour la classe 1:
 - Mean:
 - * réelle: 5.862162, 2.7243242, 4.2108107, 1.3027027
 - * échantillon: 5.89406553, 2.70037139, 4.28611674, 1.3473438
 - Ecart-type:
 - * réel: 0.531952, 0.29944894, 0.49597478, 0.20613708
 - * échantillon: 0.51264886, 0.25024787, 0.40643571, 0.19599569
- Pour la classe 2:
 - Mean:
 - * réelle: 6.5594597, 2.9864864, 5.545946, 2.0054054
 - * échantillon: 6.52999239, 3.0324595, 5.57314614, 2.00670609
 - Ecart-type:
 - * réel: 0.65889615, 0.31460926, 0.54446435, 0.29715872
 - * échantillon: 0.63336966, 0.32560175, 0.68418903, 0.30524206

Les nombres et les graphiques montrent que les échantillons sont très proches des données réelles, donc on a bien la vraisemblance.

4.2 - Comparaison avec SKLearn

4.2.1 - Naïve Bayes

Notre implémentation de Naïve Bayes a été comparée avec celle de SKLearn dans le fichier `sampling.py`, avec un split des données échantillonnées en 70% training et 30% test.

On obtient les résultats suivants:

Notre Naive Bayes

- Precision: 0.9761904761904763
- Recall: 0.9743589743589745
- Accuracy: 0.9777777777777777
- F1_score: 0.9752738654147106

Sklearn Naive Bayes

- precision: 0.9761904761904763
- recall: 0.9743589743589745
- accuracy: 0.9777777777777777
- f1_score: 0.9752738654147106

4.2.2 - Régression Logistique

Notre implémentation de Régression Logistique a été comparée avec celle de SKLearn dans le fichier `sampling.py`, avec un split des données échantillonnées en 70% training et 30% test.

Pour SKLearn, le modèle utilisé est `lr = LogisticRegression(multi_class="multinomial")`, car on a 3 classes et donc il nous faut un modèle multinomial. [6]

Notre Logistic Regression

- Precision: 0.8505050505050505
- Recall: 0.8461538461538461



- Accuracy: 0.8666666666666667
- F1_score: 0.848323868840447

Sklearn Logistic Regression

- Precision: 0.9761904761904763
- Recall: 0.9743589743589745
- Accuracy: 0.9777777777777777
- F1_score: 0.9752738654147106

4.3 - Conclusion sur les comparaisons

On a vu qu'on avait la vraisemblance, puisque les échantillons sont très proches des données réelles, comme le montrent les graphiques.

En ce qui concerne SKLearn, on peut voir que les métriques pour Naïve Bayes sont identiques car les 2 implémentations sont simplement une application du théorème de Bayes, donc on espère avoir les mêmes résultats.

Pour le logistic regression, SKLearn fournit de meilleurs métriques car il est certainement plus optimisé que notre implémentation.

Malgré cela, notre implémentation donne quand même des très bons résultats, avec chaque métrique tournant autour de 85%.

On a donc aussi la classification.

On peut donc conclure que les deux implémentations arrivent à bien classer les données IRIS.

Références

- [1] *Classifieur linéaire*. In: *Wikipédia*. URL: https://fr.wikipedia.org/w/index.php?title=Classifieur_lin%C3%A9aire&oldid=189518969 (visited on 01/05/2024).
- [2] *NumPy & SciPy for GPU*. CuPy. URL: <https://cupy.dev/> (visited on 01/11/2024).
- [3] R. A. Fisher. *Iris*. 1936. DOI: [10.24432/C56C76](https://doi.org/10.24432/C56C76). URL: <https://archive.ics.uci.edu/dataset/53> (visited on 01/07/2024).
- [4] R. A. Fisher. *Iris*. 1936, *UCI ML repository website*. URL: <https://archive.ics.uci.edu/dataset/53/iris>.
- [5] *Régression logistique*. In: *Wikipédia*. URL: https://fr.wikipedia.org/w/index.php?title=R%C3%A9gression_logistique&oldid=210479759 (visited on 01/05/2024).
- [6] scikit-learn developers. *LogisticRegression - scikit-learn 1.0.2 documentation*. Accessed: 2024-01-25. 2023. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.
- [7] *Scipy.Optimize.Fmin — SciPy v1.11.4 Manual*. URL: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html#scipy.optimize.fmin>.