

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

Janvier 2024

Contents

1 – Introduction & Rappels théoriques	1
1.1 – Régression Logistique	1
1.2 – Naive Bayes	1
2 – Méthodologie	2
2.0 – Choix du dataset & outils utilisés	2
2.1 – Gradient Descent	3
2.2 – Régression Logistique	4
2.2.1 – Fonction de coût pour la régression logistique	4
2.2.2 – Apprentissage	5
2.2.3 – Prédictions	6
2.2.4 – Résultats	7
2.3 – Naive Bayes	8
2.3.1 – Extraction des distributions	8
2.3.2 – Prédictions	8
2.3.3 – Résultats	9
Références	10

Projet – Regression Logistique / Naive Bayes

1 – Introduction & Rappels théoriques

Dans ce document, nous approfondirons des techniques de regression 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 précises 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:

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

où $\mathbf{x}_i \in \mathbb{R}^K$ 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 simplement modifié la manière dont nous lui appliquons le seuillage, 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 plus que satisfaisants comme nous le verrons en [section 2.2](#).

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 classifié de “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 model est dit génératif contrairement à la regression logistique étant considéré comme “méthode discriminante” [1] et consiste à modéliser les probabilités conditionnelles $P(\mathbf{x}|\text{classe})$ pour chaque classe y et sample \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:



Iris

Donated on 6/30/1988

A small classic dataset from Fisher, 1936. One of the earliest known datasets used for evaluating classification methods.

Dataset Characteristics

Tabular

Subject Area

Biology

Associated Tasks

Classification

Variables Table

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` [6] 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:

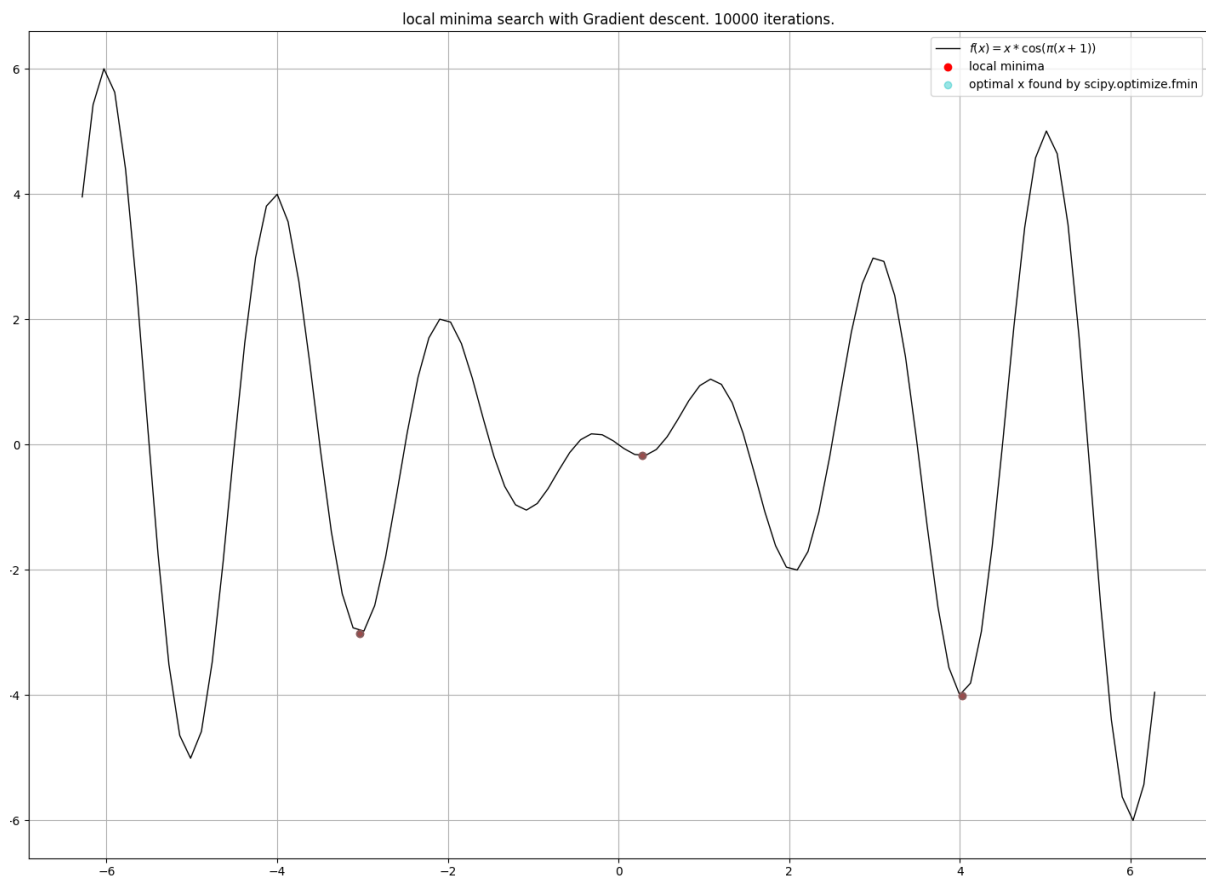


Figure 2: minimas locaux_gradient descent

Ce résultat illustre bien 2 choses: la première est que l’implémentation de la descente en gradient fonctionne correctement puisque pour 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

MSE – Une mauvaise idée :

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

Pour cela, on pourrait penser utiliser quelque chose comme la **Mean Squared Error (MSE)**, qui est une moyenne du carré de la différence entre le résultat obtenu par la régression (donné par z) et la valeur estimée y .

La MSE nous donne une estimation de l'erreur moyenne faite entre la fonction approximative f et la valeur attendue y .

L'objectif est donc de minimiser la MSE afin de minimiser l'erreur entre les valeurs estimées et les valeurs attendues.

Ce qui nous donnerait

$$MSE = \frac{1}{n} \sum_i^n (\sigma(z_i) - y_i)^2$$

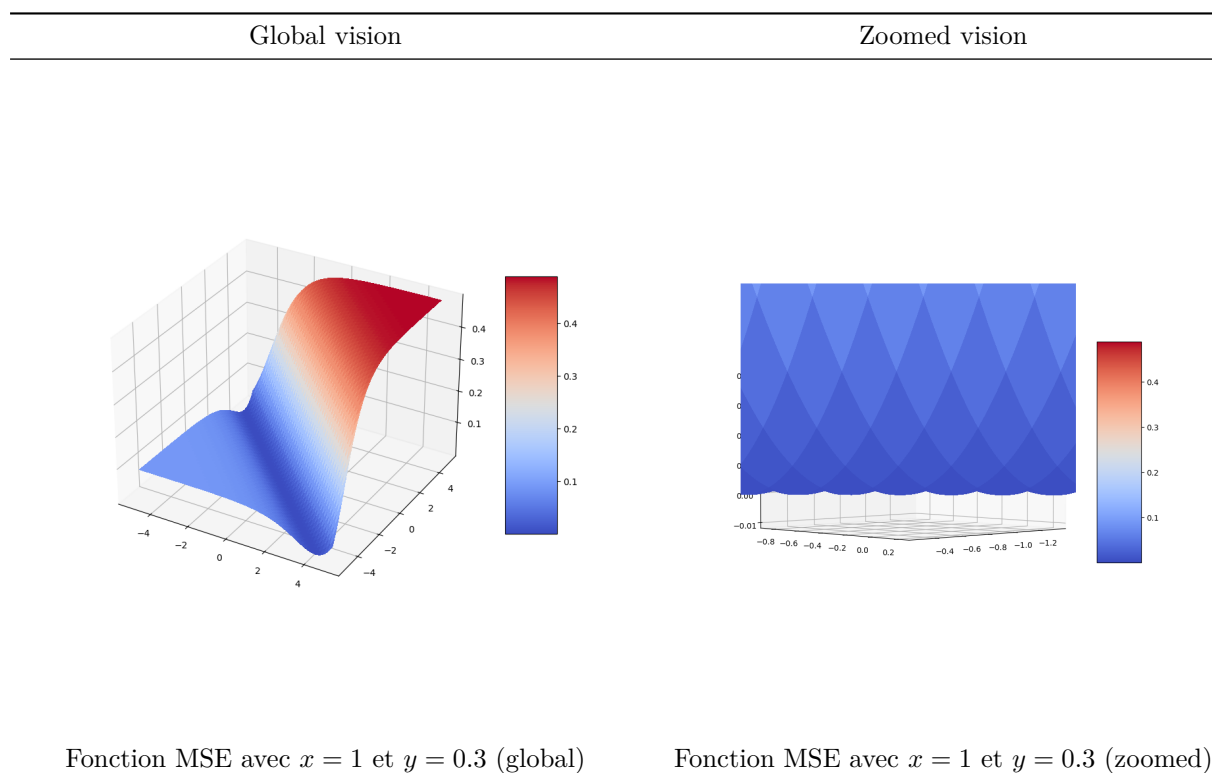
avec σ la fonction sigmoïde utilisée pour la régression logistique, définie comme en section 1.1 notre MSE nous donnerait:

$$MSE = \frac{1}{n} \sum_i^n \left(\frac{1}{1 + e^{-z_i}} - y_i \right)^2$$

Afin de visualiser la MSE obtenue, nous avons créé un graph de la fonction

$$MSE(w, b) = \frac{1}{n} \left(\frac{1}{1 + e^{w^T x + b}} - y_i \right)^2$$

pour $x = 1$ et $y = 0.3$. Nous obtenons alors les graphes suivants:



Nous pouvons remarquer sur la figure zoomé, que la fonction admet plusieurs minimum locaux. La fonction MSE n'est donc pas convexe. Ceci est problématique pour la descente en gradient, car celle-ci risquera de se retrouver coincé dans un minimum local et ne trouvera jamais le minimum global.

Si on pouvait faire des plots pour la fonction avec plus de paramètres, on verrait mieux et plus de minimum locaux.

Cela est dû au fait que la fonction $\sigma(z)$ n'est pas linéaire. (Découle trivialement du fait qu'il y ait une exponentielle dans la fonction.) Ce qui empêche la MSE d'être convexe.

La descente en gradient ne pourra donc pas fonctionner correctement, car on pourrait trouver des minimum locaux à la place du minimum global, ce qui nous empêcherait de trouver les paramètres optimaux pour la régression logistique.

Log Loss – Cross-Entropy :

Pour résoudre ce problème, on utilise plutôt la **log loss** fonction, appelée également **cross-entropy**.

La formule de la log loss function est donnée par:

$$C(\mathbf{w}, b) = \frac{1}{N} \sum_{i=0}^N \log p(y_i | \mathbf{x}_i; \mathbf{w}, b)$$

avec N le nombre de sample, y_i la “vrai” classe associé au sample \mathbf{x}_i et \mathbf{w}, b le vecteur de poids et biais. Où l'on peut ensuite développer $p(y_i | \mathbf{x}_i; \mathbf{w}, b)$ pour retomber sur la cross-entropy, avec $p(x) = y$ et $q(x) = z$

La fonction ci-dessus pénalise fortement (du moins plus que les autres cas) les fausses prédictions “confiantes” (i.e. annonce faux + haute probabilités) et son domaine d'arrivé est de 0 à ∞ , un modèle parfait aurait une log-loss de 0.

Un modèle complètement incorrect aurait, quant à lui, une log loss qui tend vers ∞

2.2.2 – 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 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 calcul le gradient en argument, que d'essayer de le calculer dynamiquement, c'est pourquoi la signature de notre implémentation prend un **df** en argument plutôt que la fonction de coût elle même.

Où le calcul des dérivées partielles a été défini comme ci-dessous.

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:

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

Or le \mathbf{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)

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

De même pour \mathbf{dw} :

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

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 implémentée.

```
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)`.

2.2.3 – Prédiction

Pour la prédiction, nous avons utilisé la fonction suivante:

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

qui prend simplement $\sigma(\mathbf{w}^T X + b)$ 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 – Résultats

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 efficient si besoin est 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 des sklearn alors que nous devions les implémenter nous mêmes, (telles que les metrics f1-score...). Ces fonctions ont bien été implémenté mais pour une raison de simplicité, elle n'ont pas été utilisée 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, plus que satisfaisants, 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:

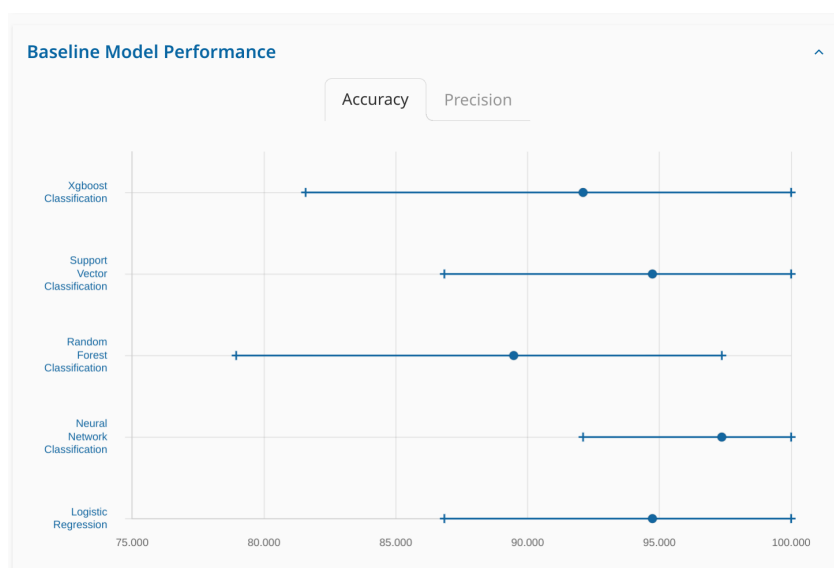


Figure 3: performances attendu d'après le UCI ML Repo [4]

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.

2.3 – 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.

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] *Scipy.Optimize.Fmin — SciPy v1.11.4 Manual*. URL: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html#scipy.optimize.fmin>.