Master 2 SISE Statistiques et Informatiques pour la Science des Données

Université Lumière Lyon 2
Année universitaire 2023–2024

Naive Bayes Classifier

Création d'un package pour R

Natacha Perez
Abdourahmane Ndiaye
Annabelle Narsama

Contents

1 Introduction

L'objectif de ce projet a été de créer un package pour R proposant une méthode "Bayésien Naïf" pour la classification supervisée, codé en classe R6. Ce package intégre une méthode de parallélisation des calculs pour réduire le temps d'exécution des calculs et pour s'adapter à de gros volumes de données.Il peut être directement installer à partir de Github et il est documenté par un tutoriel en anglais qui montre l'utilisation de ces fonctionnalités. La programmation du coeur de l'algorithme Naive Bayes n'a pas nécessité de package existants. Ce package a été conçu de façon à être le plus facile d'utilisation possible.

2 Naïve Bayes Classifier

2.1 Choix du model Naïve Bayes Gaussien

Le Naïve Bayes est une famille de modèle probabilistes qui utilisent le théorème de Bayes en supposant une indépendance entre les variables explicatives pour prédire l'étiquette de la variable à prédire. Il existe différents types de Naives Bayes classifier qui dépendent de la distribution des variables explicatives: Bernoulli, multinomial, poisson, non paramétrique et Gaussien.

Nous avons choisi d'implémenter le classifieur bayésien naïf Gaussien. La variable cible qualitative présente $K \geq 2$ modalités tandis que les variables explicatives peuvent être quantitatives ou qualitatives. Les variables données en entrée à un classificateur doivent être adaptés à ce dernier, c'est pourquoi pour implémenter un classifieur bayésien naïf gaussien, il a fallu harmoniser les types de données avant de procéder aux calculs.

2.2 Méthodes privées: Préparation des données

2.2.1 Fonction `Binarize`

Pour harmoniser les données, notre classe R6 comporte des méthodes privées (utilisées uniquement en interne). La fonction privé `binarize` *utilise 'model.matrix', qui lorsqu'elle rencontre des variables catégorielles, les encode en '0' ou '1. Chaque niveau d'une variable catégorielle devient une colonne de la matrice. Ainsi, la fonction "binarize" permet d'obtenir des variables quantitatives:*

Algorithm 1 Fonction `Binarize`

```
Function Binarize(column):
    if is.numericcolumn then
        return model.matrix~column - 1
    end
    else if is.factorcolumn or is.charactercolumn or is.logicalcolumn then
        return model.matrix~column - 1
    end
    else
        message"Column with type ", classcolumn, " encountered."  stop"Only character, factor, logical, or numerical
        columns must be entered."
    end
```

2.2.2 Fonction `check_numeric`

Après l'encodage des variables catégorielles, une fonction privée au sein de notre classe R6, appelée **`check_numeric`**, *permet de s'assurer que les données d'entrée X sont sous forme de matrice et contiennent des valeurs numériques. Elle effectue des coercitions si nécessaires. En sortie, la fonction renvoie une matrice où tous les éléments sont de type numérique.*

Algorithm 2 Fonction `check_numeric`

Function `check_numeric(`X`)`**:**
- **if** $\neg is.matrix X$ **then**
  - *warning"X was coerced to a matrix.", call.* $=$ ***FALSE*** $X \leftarrow as.matrix X$
- **end**
- **if** $\neg is.numericunlist X$ **then**
  - *warning"Matrix elements were coerced to numeric"* $X \leftarrow as.matrixas.numericunlist X$
- **end**
- **return** X

2.3 Théorème de Bayes

Soient $\mathcal{X} = (X_1, \ldots, X_J)$ l'ensemble des variables explicatives et Y la variable à prédire (l'attribut classe comportant $K \geq 2$ modalités). Ce problème de classification multiclasse est abordé en appliquant d'abord le théorème de Bayes aux probabilités conditionnelles spécifiques à chaque classe $P(Y = C_k|\mathcal{X} = \mathbf{x})$, en le décomposant ainsi en produit de la vraisemblance (likelihood) et de la probabilité a priori (prior) normalisé par la vraisemblance des données (facteur de normalisation):

$$P(Y = C_k|X = \mathbf{x}) = \frac{P(Y = C_k)P(X = \mathbf{x}|Y = C_k)}{P(X = \mathbf{x})}$$

Étant donné que les variables aléatoires $X = (X_1, X_2, \ldots, X_d)$ sont (naïvement) supposées être conditionnellement indépendantes, la vraissemblance $P(X = \mathbf{x}|Y = C_k)$ peut être réécrite comme suit :

$$P(Y = C_k|X = \mathbf{x}) = \frac{P(Y = C_k)\prod_{i=1}^{d} P(X_i = x_i|Y = C_k)}{P(X_1 = x_1, \ldots, X_d = x_d)}$$

Étant donné que $P(X_1 = x_1, \ldots, X_d = x_d)$ est constant, la probabilité conditionnelle $P(Y = C_k|X = \mathbf{x})$ peut se réécrire ainsi :

$$P(Y = C_k|X = \mathbf{x}) \propto P(Y = C_k)\prod_{i=1}^{d} P(X_i = x_i|Y = C_k)$$

On peut transformer ces calculs en logarithme pour transformés les multiplications en additions:

$$\log P(Y = C_k|X = x) \propto \log P(Y = C_k) + \sum_{i=1}^{d} \log P(X_i = x_i|Y = C_k).$$

La classe avec la probabilité a posteriori logarithmique la plus élevée est choisie comme prédiction:

$$\hat{C} = \arg\max \left(\log P(Y = C_k) + \sum_{i=1}^{d} \log P(X_i = x_i|Y = C_k) \right) \tag{1}$$

2.4 Méthodes publiques

2.4.1 Fonction `fit`

Avant d'utiliser le package **`GaussianNaiveBayes`**, *l'utilisateur doit préalablement subdiviser ses données en un échantillon d'apprentissage et un échantillon de test. Puis l'utilisateur appelel la méthode* **`fit`** *de notre classe en fournissant en paramètres* **`Xtrain`** *et* **`yTrain`**.

La méthode **`fit`** *prend en entrée* **`Xtrain`** *et* **`ytrain`**. *Dans un premier temps, elle s'assure que la variable réponse Y soit un vecteur de type factor, character ou logical. Elle binarise les variables explicatives en appelant la fonction* **`binarize`** *et s'assure que les données soient de types* **`numeric`** *en et les transformant en matrice avec la fonction* **`check_numeric`**. *Dans un second temps, la fonction* **`fit`** *permet de calculer les probabilités a priori de chaque classe : Étant donné que la variable de réponse Y peut prendre K valeurs distinctes notées* $C_1, \ldots, C_K$, *chaque probabilité a priori* $P(Y = C_k)$ *dans l'équation peut être interprétée comme la probabilité d'observer l'étiquette* C_k. *Les probabilités a priori correspondent aux proportions de classes dans l'échantillon* $\left(\frac{\text{nombre d'échantillons dans la classe}}{\text{nombre total d'échantillons}}\right)$. *Les probabilités a priori peuvent également être renvoyé en utilisant le paramètre* **`prior`** *de notre classe* **`Gaussian_Naive_Bayes`**. *Enfin, les paramètres* μ *et* σ *sont estimés pour chaque classe et chaque prédicteur sur la base de l'échantillon d'entrainement. En sorti, la fonction* **`fit`** *stocke les résultats dans les membres privés de la classe.*

Algorithm 3 Fonction `fit`

Function `fit`(*X, y*)**:**

- **if** *is.nullX* ***or*** *is.nully* **then**
  - stop"X and y are required."
- **end**
- **if** $\neg is.factory$ ***and*** $\neg is.charactery$ ***and*** $\neg is.logicaly$ **then**
  - *stop"y must be a factor, character, or logical vector."*
- **end**
- **if** $\neg is.data.frameX$ ***and*** $\neg is.matrixX$ **then**
  - *stop"X must be a data frame or matrix."*
- **end**
- **if** *anyis.naX* ***or*** *anyis.nay* **then**
  - *stop"X and y cannot contain NA values."*
- **end**
- **`self$X`** $\leftarrow$ *X* **`self$y`** $\leftarrow$ *y*
- **if** $\neg is.factory$ **then**
  - **`self$y`** $\leftarrow$ *factory*
- **end**
- **`levels_y`** $\leftarrow$ *levelsself$y*
- **if** *nlevelsself$y ¡ 2* **then**
  - *stop"y must contain at least two classes."*
- **end**
- **`prior`** $\leftarrow$ *prop.tabletableself$y* **`lev`** $\leftarrow$ *levelsself$y*
- **`self$X`** $\leftarrow$ *lapplyX, private$binarize* **`self$X`** $\leftarrow$ *cbindas.data.frameself$X* **`self$X`** $\leftarrow$ *private$check_numericself$X*
- **`vars`** $\leftarrow$ *colnamesself$X*
- **`params`** $\leftarrow$ *lapplylev,* ***function****(lev)* { **`lev_subset`** $\leftarrow$ **`self$X[self$y == lev, , drop = FALSE]`** **`mu`** $\leftarrow$ *colMeanslev_subset, na.rm = TRUE* **`sd`** $\leftarrow$ *applylev_subset, 2,* ***function****(x)* { **`sqrt`**$(meanx^2, na.rm = TRUE - meanx, na.rm = TRUE^2)$ } **`list(mu = mu, sd = sd)`** }
- **`self$vars`** $\leftarrow$ *vars* **`private$data`** $\leftarrow$ **`list(x = self$X, y = self$y)`** **`self$levels_y`** $\leftarrow$ *levels_y* **`self$params`** $\leftarrow$ *params* **`self$prior`** $\leftarrow$ *prior* **`private$call`** $\leftarrow$ *match.call*

2.4.2 Fonction `predict`

La fonction ***`predict`*** *prend en entrée un ensemble de données* ***`Xtest`*** *et retourne en sorti les prédictions sous forme de facteur, où chaque observation est attribuée à la classe avec la probabilité postérieure maximale. Cette fonction récupère les paramètres du modèle entraîné tels que les niveaux de classe (*`lev`*), les probabilités a priori (*`prior`*), les moyennes (μ et les écarts-typesσ). Les prédicteurs étant des variables numériques, la distribution gaussienne est assumée et le calcul du likelihood s'est basé sur cette formule:*

$$\mathbb{P}\left(X_i = v \mid Y = C_k\right) = \frac{1}{\sqrt{2\pi\sigma_{ik}^2}} \exp\left(-\frac{(v-\mu_{ik})^2}{2\sigma_{ik}^2}\right)$$

Enfin, la classe ayant la probabilité log-postérieure la plus élevée est choisie comme prédiction, selon la formule(1).

Algorithm 4 Fonction `predict`

Function `predict`(*X_test, threshold = 0.001, eps = 0*)**:**

`lev` ← `self$levels_y prior` ← `self$prior mu` ← `self$params$mu sd` ← `self$params$sd`

... Validation des données d'entrée

`num_cores` ← detectCores `cl` ← makeCluster`num_cores X_test` ← parLapply`cl, X_test, private$binarize` stopCluster`cl`

... Prétraitement des données d'entrée

`features` ← `col_names[col_names %in% colnames(mu)]`

... Validation des caractéristiques

`sd[sd <= eps]` ← `threshold eps` ← `ifelse(eps == 0, log(.Machine$double.xmin), log(eps))` `threshold` ← `log(threshold)`

... Calcul des probabilités a posteriori

`post` ← matrix`nrow = nrow(`$X_{\texttt{test}}$`), ncol = lengthlev` colnames`post` ← `lev`

for *`ith_class in seq_along(lev)`* **do**

`ith_class_sd` ← `sd[ith_class, ]` **end**

`ith_post` ← `-0.5 * log(2 * pi * ith_class_sd`^{2}`)`$-0.5*$`((`$\texttt{X}_{\texttt{test}}$ `- mu[ith_class, ]) / ith_class_sd)`^{2}

`ith_post[ith_post <= eps]` ← `threshold`

`post[, ith_class]` ← `(rowSums(ith_post) + log(prior[ith_class]))`

`self$post` ← `post cat"Exponentiel "`

return *`factor(lev[max.col(post, "first")], lev)`*

2.4.3 Fonction `predict_proba`

L'utilisateur peut appeler la fonction ***`predict_proba`*** *afin d'obtenir, pour chaque observation, les probabilités d'appartenance à chaque classe $P(Y = C_k|X = \mathbf{x})$. Cette fonction prend en entrée l'échantillon X_{test} et renvoie en sortie les probabilités a posteriori pour chaque classe sous forme de matrice. Le calcul des probabilités a posteriori utilise la fonction exponentielle pour transformer les log-probabilités stockées dans* ***`self$post`*** *en probabilités. Ensuite, elle normalise les probabilités en divisant par la somme de toutes les probabilités.*

Algorithm 5 Fonction `predict_proba`

Function `predict_proba`(X_{test})**:**

```
... Validation des données d'entrée
n_obs ← nrow(X_test) n_lev ← length(self$levels_y) post ← matrix(0, nrow = n_obs, ncol = n_lev)
if n_obs == 1 then
    post ← t(apply(post, 2, function(x) 1 / sum(exp(post - x)))) colnames(post) ←
    self$levels_y return post
end
else
    colnames(post) ← self$levels_y result ← matrix(0, nrow = n_obs, ncol = n_lev)
    for i in seq_len(n_obs) do
        probabilities ← exp(self$post[i,]) sum_per_row ← sum(probabilities)
        for j in seq_along(self$levels_y) do
            result[i, j] ← probabilities[j] / sum_per_row
        end
    end
    colnames(result) ← self$levels_y return result
end
```

2.4.4 Fonction `print`

L'utilisateur peut inspecter les paramètres du modèle en appelant la fonction ***print*** *qui a pour objectif d'afficher les probabilités a priori pour chaque classe et les tables de probabilités conditionnelles associées à chaque variable. Cette fonction obtient les tables de probabilités conditionnelles en appelant la méthode* ***get_gaussian_tables****.*

Algorithm 6 Fonction `print`

Function `print()`**:**

```
cat("probabilities:  ") cat("-----------------------------") for lev in names(self$prior) do
    cat(paste(" ", lev, ":  ", self$prior[[lev]], ""))
end
cat("Probabilities:")              cat("-----------------------------")              tables ←
private$get_gaussian_tables() for var in names(tables) do
    cat(paste("Variable:", var, "")) print(tables[[var]])
end
```

2.4.5 Fonction `summary`

La fonction ***summary*** *fournit à l'utilisateur un résumé des principales informations du modèle, incluant :*

- *Le nombre total d'observations dans l'ensemble de données.*
- *Le nombre d'observations de chaque classe.*
- *Les probabilités a priori de chaque classe.*
- *Le nombre et les noms des descripteurs.*
- *Les écarts types et les moyennes de chaque descripteur pour chaque classe.*

Algorithm 7 Function `summary`

Function `summary()`:

```
cat("Number of observations:  ", length(self$y), "\n")
cat("Number of training observations in each class:\n")
print(table(self$y))
cat("Prior probabilities in y:\n")
print(prop.table(table(self$y)))
cat("Number of Features:", length(self$vars), "\n")
cat("Features:", self$vars, "\n")
cat("Standard deviation of each feature:\n")
print(self$params$sd)
cat("Mean of each feature per class:\n")
print(self$params$mu) }
```

2.4.6 Fonction `confusion_matrix`

L'utilisateur peut à présent évaluer les performances du modèle en appelant la fonction **`confusion_matrix`**. *Celle-ci prend en entrée deux vecteurs,* **`y_test`** *et* **`y_pred`**, *qui représentent respectivement les valeurs réelles des classes et les valeurs prédites par le modèle. Elle génère en sortie une matrice de confusion et montre le nombre de prédictions correctes et incorrectes pour chaque classe. L'utilisateur peut ainsi calculer les métriques d'évaluation qu'il souhaite.*

Algorithm 8 Fonction `confusion_matrix`

Function `confusion_matrix(`*y_test, y_pred*`)`:

if *is.nully_test* ***or*** *is.nully_pred* **then**
  stop"predict.gaussian_naive_bayes(): y_test and y_pred are required."
end
if $\neg$*is.factory_test* ***and*** $\neg$*is.charactery_test* ***and*** $\neg$*is.logicaly_test* **then**
  stop"y_test must be either a factor, character, or logical vector"
end
if $\neg$*is.factory_pred* ***and*** $\neg$*is.charactery_pred* ***and*** $\neg$*is.logicaly_pred* **then**
  stop"y_pred must be either a factor, character, or logical vector"
end
if *anyis.nay_test* ***or*** *anyis.nay_pred* **then**
  stop"y_test or y_pred cannot contain NA values."
end
`cols` $\leftarrow$ *paste0"pred-", NB$levels_y* **`conf_mat`** $\leftarrow$ *arraydim = c(length(self$levels_y),length(self$levels_y)), dimnames = list(self$levels_y,cols)*
for *clas in self$levels_y* **do**
  for *clas_pred in cols* **do**
    compt $\leftarrow$ *tabley_pred*[*y_test* == *clas*] *conf_mat[clas,]* $\leftarrow$ *compt*
  end
end
return *conf_mat*

3 Stratégie de parallélisation des calculs

Dans notre classe R6 Naive Bayes Gaussian, nous avons utilisé une méthode de parallélisation des calculs pour réduire le temps de calcul sur des données volumineuses. La parallélisation des calculs a été réalisée en utilisant le package `parallel` *et la fonction* `parLapply` *en exemple dans la fonction* `predict` *:*

- `detectCores` *détermine le nombre de cœurs disponibles sur la machine.*
- `makeCluster` *crée un cluster de calcul parallèle avec le nombre de cœurs déterminé précédemment. Les tâches peuvent être réparties entre ces cœurs pour accélérer le traitement.*
- `parLapply` *applique la fonction* `private$binarize` *de manière parallèle à chaque élément de la liste* `X_test`*. Chaque élément de la liste est traité indépendamment par un cœur différent.*
- `stopCluster(cl)` *: Une fois que le traitement parallèle est terminé, la fonction* `stopCluster` *arrête le cluster parallèle.*

Function `predict`(*X_test, threshold = 0.001, eps = 0*)**:**

`num_cores ← detectCores() cl ← makeCluster(num_cores) X_test ← parLapply(cl, X_test, private$binarize) stopCluster(cl)`

---

4 Exemple d'utilisation

Voici un exemple d'utilisation de notre package `GaussianNaiveBayes` *pour l'instancier, l'entraîner et effectuer des prédictions:*

```
# Installer le package GaussianNaiveBayes
install.packages("devtools")
library(devtools)
devtools::install_github('Abdouragit/GaussianNaiveBayes')
library(GaussianNaiveBayes)

# Créer une instance de la classe Gaussian_Naive_Bayes
NB = Gaussian_Naive_Bayes$new()

# Entraîner le modèle
NB$fit(Xtrain, ytrain)

# Effectuer des prédictions
predictions <- NB$predict(Xtest)

# Afficher les prédictions
print(predictions)

# Obtenir les probabilités de prédiction
NB$predict_proba(Xtest)

# Afficher un résumé du modèle
NB$print()
```

```
# Afficher un résumé des informations du modèle
NB$summary()

# Générer une matrice de confusion
NB$confusion_matrix(y_test = ytest, y_pred = NB$y)
```

5 Références

References

[1] *Package Naive Bayes sur CRAN, The Comprehensive R Archive Network,* https://cran.r-project.org/web/packages/naivebayes/index.html .

[2] *Documentation scikit-learn - Naive Bayes gaussien, scikit-learn,* https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html .