Création d'un package pour R

Régression logistique
Descente de gradient - Programmation parallèle

LE GALÈZE Pierre, DANET Chloé, DUDOIT Romain
M2 SISE - Université Lumière Lyon 2

29 novembre 2021

Table des matières

1 Introduction

Dans le cadre de notre projet de programmation R de deuxième année de Master SISE (Statistique et Informatique pour la Science des Données), nous avons implémenté un package pour R. L'enjeu de ce package est de proposer la régression logistique binaire avec la méthode de la descente de gradient, mais aussi la possibilité d'exploiter les capacités des processeurs multicœurs par le biais de la programmation parallèle.

La régression logistique constitue un cas particulier de modèle linéaire et est largement utilisée en apprentissage automatique. C'est un modèle de régression binomiale permettant d'associer à un vecteur de variables aléatoires, une variable aléatoire binomiale génériquement notée Y.
De manière générale, la probabilité de la classe 1 est celle qui est modélisée par défaut. Par exemple, dans un jeu de données avec deux catégories 0 et 1, la première classe peut être la catégorie 1, et le modèle de régression logistique peut alors s'écrire comme étant la probabilité qu'un individu du jeu de données appartienne à la catégorie 1.

Concernant la descente de gradient, son but est de trouver des valeurs de paramètres/coefficients d'une fonction qui minimise une fonction de coût : on parle d'algorithme d'optimisation.

Ce package a donc pour objectif de modéliser la probabilité qu'une entrée (X) appartienne à la classe par défaut (Y=1) et ce, de manière optimisée.

Dans ce rapport, nous allons donc présenter l'architecture du package et les fonctions le composant, mais aussi les différentes méthodes de descente de gradient : batch, mini-batch et online. Nous expliquerons également, la parallélisation que nous avons implémenté pour la méthode batch.

2 Présentation des fonctions du package

Notre package intègre un fichier d'aide comprenant son installation, les descriptions des fonctions, de leurs paramètres, des objets fournis en sortie et des exemples d'utilisation. Ce dernier est disponible sur GitHub à l'adresse : `https://github.com/Romain8816/M2_RPackage/tree/main/rlgd`.

Notre package rlgd comprend de multiple fonction, ayant chacune un rôle spécifique que nous allons les présenter ci-dessous.

2.1 sigmoid

La fonction sigmoïde, dite aussi courbe en "S", est définie par :

$$f(x) = \frac{1}{1 + e^{-x}}$$

Elle représente la fonction de répartition de la loi logistique et va nous permettre de calculer des prédictions.

2.2 x_dot_theta

Nous utilisons cette fonction afin d'effectuer une multiplication matricielle entre une X et theta :

$$F(x) = X.\theta$$

où X est une matrice de dimension m x n regroupant les variables explicatives du jeu de données ; et theta est la matrice de dimension n x 1 renseignant les paramètres du modèle. Elle fournit donc en sortie une matrice de dimension n x 1, c'est-à-dire le modèle de régression logistique.

2.3 probability

Notre fonction de probabilité est donnée par la formule suivante :

$$h_\theta(x) = \frac{1}{1 + e^{-X.\theta}}$$

Dans notre package, cette fonction est utilisée pour calculer la probabilité que les observations soient dans une catégorie spécifiée d'une variable binaire. Elle fait appelle aux fonctions sigmoid et x_dot_theta.

2.4 cost_function

Pour la régression linéaire, la fonction de coût donne une courbe convexe, qui a un seul minimum. En revanche, l'utilisation de cette fonction pour le modèle logistique ne donne pas de courbe convexe, en raison de la non-linéarité. L'algorithme de la descente de gradient se

figera au premier minimum rencontré, sans trouver le minimum global.

Il est donc nécessaire de développer une nouvelle fonction de coût spécifique pour la régression logistique. La fonction logarithmique est ensuite utilisée pour transformer la fonction de probabilité en une fonction convexe en séparant les cas où y = 1, des cas où y = 0.
En pratique, si notre modèle prévoit une appartenance de x à la classe 0, alors que y = 1, et inversement, il est nécessaire de pénaliser le calcul par une grosse erreur, c'est-à-dire par un gros coût. De plus, la fonction logarithmique permet de tracer cette courbe avec une propriété convexe, ce qui va pousser la descente de gradient à trouver les paramètres θ pour un coût minimal et donc un coût qui tend vers 0.
Finalement, la fonction de coût s'écrit de la manière suivante :

$$J(\theta) = -\frac{1}{m}\sum y * log(\sigma(X.\theta)) + (1-y) * log(1-\sigma(X.\theta))$$

Dans notre package, cette fonction nous sert uniquement pour afficher la courbe de décroissance de la fonction de coût.

2.5 cost_plot

Cette fonction permet simplement d'afficher le coût en fonction des itérations avec en entrée le résultat de la fonction rlgd.fit.

2.6 gradient

Le calcul du gradient de la fonction de coût de la régression logistique est illustré de la manière suivante :

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T.(\sigma(X.\theta) - y)$$

Il permet de calculer des dérivées partielles de chacun des paramètres de θ.

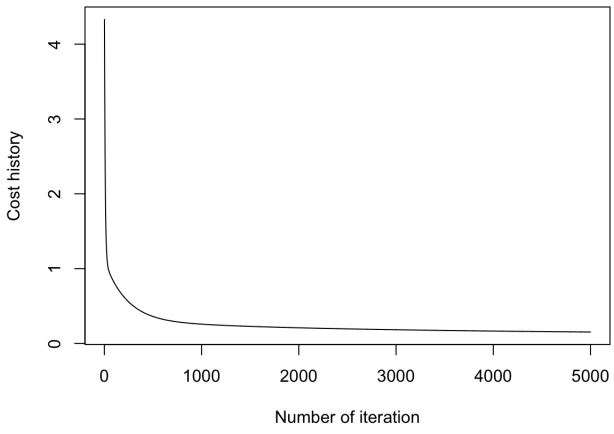
Dès lors, dans ce package, nous avons implémenté la descente de gradient que l'on peut représenter par :

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

La descente de gradient est un algorithme d'optimisation utilisé pour trouver les valeurs de coefficients d'une fonction qui minimise une fonction de coût. Elle est utilisée lorsque les paramètres doivent être recherchés par un algorithme d'optimisation.
Ici, nous avons implémenté 3 modes distincts de la descente de gradient : le mode batch, le mode online et le mode mini-batch. Ils sont décrits ci-dessous.

2.7 batch_gradient_descent

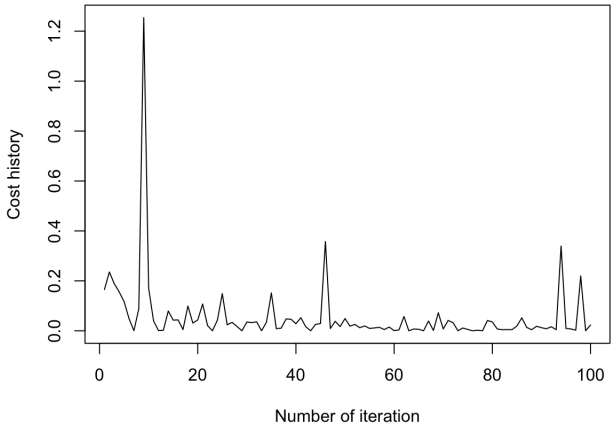
La fonction prend en entrée la matrice X du biais et des variables, la matrice y de la variable cible, les paramètres θ qui ont été initialisés aléatoirement, un nombre maximum d'itération, un paramètre de tolérance et un nombre de coeur. Si le nombre de coeur est égale à 1, on n'utilise pas la parallélisation et on effectue la descente de gradient en calculant pour chaque itération le gradient sur la totalité des observations et en mettant à jour ce dernier à l'itération suivante.
L'algorithme s'arrête lorsque le nombre maximum d'itération est atteint ou si la somme des différences en valeur absolue entre les paramètres de l'itération actuelle et les paramètres de l'itération précédente est inférieur à un seuil de tolérance.
À chaque itération, nous avons récupéré la valeur de la fonction de coût de manière à pouvoir afficher la courbe de décroissance de la fonction de coût comme ci-dessous et observer la convergence de l'algorithme après un certain nombre d'itérations.



Si le nombre de coeur est supérieur à 1, on effectue la parallélisation. Cela est abordé dans la 3ème partie de ce rapport.

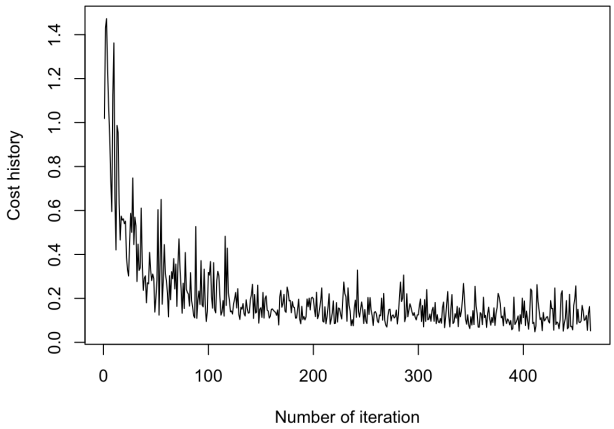
2.8 stochastic_gradient_descent

La fonction prend en entrée les mêmes paramètres que la fonction batch. Néanmoins, dans le cas online, on calcule pour chaque itération le gradient pour une ligne pris au hasard dans le jeu de donnée et on met à jour celui-ci à l'itération suivante.
Sinon, les conditions d'arrêts ainsi que la récupération du coût sont les mêmes que pour le batch. Ci-dessous, on peut observer un exemple du coût pour chaque itération de notre algorithme :



2.9 mini_batch_gradient_descent

Dans le cas du mini-batch, à chaque itération, le jeu de données est découpé en un certain nombre de lots. Ce dernier est déterminé par le nombre d'observations et la taille de batch_size, indiqué en paramètre de la fonction. Concernant les autres paramètres, ce sont les mêmes que ceux pour le mode batch et le mode online. De même que pour les modes précédents, on peut observer ci-dessous le coût au fur et à mesure des itérations :



2.10 rldg.fit

La fonction rlgd.fit effectue une régression logistique binaire en utilisant l'optimisation par descente de gradient (batch, online ou mini-batch). Elle prend en entrée une formule (objet de la classe formula du package stats de R), un dataframe, le mode, une taille de batch pour le mode mini-batch, un taux d'apprentissage, un nombre maximum d'itération, un paramètre

de tolérance, un nombre de coeur et un booléen pour effectuer ou non la standardisation sur le dataframe.

Le contrôle des données et les prétraitement (standardisation, one hot encoding) sont effectués avec la fonction get_x_y_.
Après l'initialisation aléatoire des paramètres dans θ, et selon le mode choisi, une des trois fonctions d'optimisation du gradient que nous avons évoqué en amont est appelée.

Elle retourne une instance de la classe S3 Reg.log avec des attributs correspondant aux entrées de la fonction rlgd.fit, aux sorties de la fonction get_x_y et aux sorties d'une fonction de descente de gradient.
L'instance retournée contient les paramètres d'entrée et comporte un certain nombre d'attributs :

- x, matrice (de la fonction get_x_y)
- y, matrice de la variable cible (de la fonction get_x_y)
- y_name : le nom de la variable cible (de la fonction get_x_y)
- x_names : liste des noms des variables explicatives (de la fonction get_x_y)
- preprocess : NULL ou sortie de la fonction preProcess du package "caret" appelée dans la fonction get_x_y. Elle sera utilisée dans la fonction rlgd.predict() pour appliquer ou non la normalisation sur les nouvelles données.
- xlevs : liste des niveaux pour chaque facteur dans le cadre de données de la fonction get_x_y. Elle sera utilisée dans la fonction rlgd.predict() pour traiter la même codification à chaud sur de nouvelles données.
- thêta : une matrice des paramètres finaux de la régression logistique binaire à partir d'une fonction de descente de gradient.
- cost_history : liste des résultats de la fonction de coût pour chaque itération d'une fonction de descente de gradient

2.11 get_x_y

Cette fonction est intégrée dans la fonction rlgd.predict et utilise 3 de ses paramètres : la formule, le dataframe et le booléen de standardisation. La fonction identifie les variables explicatives et la variable cible de la formule. Un modèle matriciel est créé (model.matrix) dans lequel un encodage one hot des variables de type facteur est fait automatiquement. Si le paramètre de standardisation est mis à "TRUE", une standardisation est effectuée à l'aide du package "caret".

2.12 rlgd.predict

La fonction rlgd.predict renvoie les prédictions pour la fonction rlgd.fit. Elle renvoie ainsi les classes prédites ou les probabilités de classe de la variable cible binaire définie dans la fonction rlgd.fit.

Pour fonctionner, la fonction a besoin d'un cadre de données comme celui utilisé dans la fonction rlgd.fit, c'est-à-dire le même ordre et le même nombre de variables explicatives. Les types de données doivent également être identiques. Un contrôle est effectué uniquement sur l'ordre et le nombre de variables explicatives dans le nouveau cadre de données. Les données sont ensuite transformées en un modèle matriciel comme dans la fonction rlgd.fit avec le même encodage des facteurs. La normalisation est également effectuée si elle a été faite dans la fonction rlgd.fit en regardant les attributs de l'objet Reg.log.

2.13 Surcharge du print et du summary

Nous avons aussi surchargé la fonction print afin d'afficher :

- la formule
- la variable cible
- les variables explicatives
- le mode du gradient utilisé
- le learning rate et le max_iter sélectionné
- la matrice des coefficients

Nous avons fait la même chose pour summary en affichant seulement la formule, le mode et la matrice des coefficients.

3 Parallélisation

Le but de la parallélisation est d'accélérer le temps de calcul d'un algorithme en rentabilisant au maximum les capacités accrues des processeurs multicœurs. Pour faire cela, il faut d'abord séparer le jeu de données en plusieurs blocs (décidé en fonction du nombre de coeurs que l'on souhaite), puis l'algorithme sera exécuté en même temps sur tous les couples coeurs / blocs : chaque coeur prend en charge un seul bloc. Puis, on va pouvoir rassembler les résultats de chaque cœur en un seul pour obtenir le résultat final.

Dans notre cas, la parallélisation se fait sur le calcul du gradient et on rassemble les résultats en faisant l'addition des gradients obtenus.

3.1 Division en blocs

Comme on l'a vu précédemment, le prétraitement de la parallélisation consiste à séparer le jeu de donnée en plusieurs blocs. Pour résumé la fonction que nous avons utilisée, regardons le pseudo code suivant :

Algorithm 1 Divise x et y en plusieurs blocs en fonction du nombre de coeurs utilisé

Require: x and y deux matrices et un nombre de coeurs $ncores$
  $m \leftarrow$ nrow(y)
  $bloc\_size \leftarrow$ arrondi($m/ncores$)
  $bloc \leftarrow$ liste vide
  $index \leftarrow$ contient les index des limites de chaque bloc en fonction de $bloc\_size$
  for i in 1 :(taille(index)-1) **do**
    $y\_i \leftarrow$ Récupère le bloc i dans y avec $index$
    $x\_i \leftarrow$ Récupère le bloc i dans x avec $index$
    $bloc[[i]] \leftarrow$ Ajout du bloc x_i / y_i dans une liste
  end for
  return une liste contenant plusieurs blocs x_i / y_i

On se retrouve donc avec une liste contenant plusieurs blocs x_i / y_i sur lesquels nous allons pouvoir appliquer la parallélisation.

3.2 parSapply

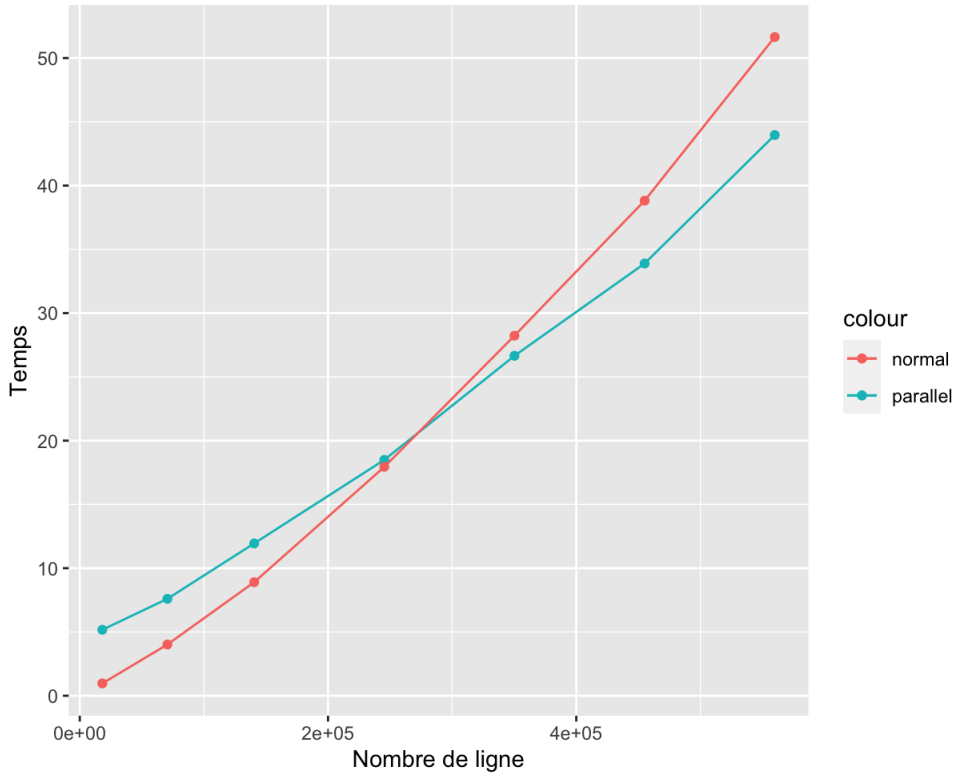
La fonction parSapply du package "parallel" est celle que nous avons utilisée, car elle facilite l'application de la parallélisation. En effet, il nous suffit de permettre l'utilisation de plus de cœurs qu'habituellement avec la fonction makeCluster(), puis d'indiquer dans les paramètres de parSapply, les blocs sur lesquels seront lancés le gradient.

3.3 Durée d'exécution de la parallélisation

À noter : les tests ont été faits avec 6 cœurs, 100 itérations et sur le gradient batch. Pour les reproduire, un fichier, nommé "Test_durée_execution", se trouve dans le github du projet.

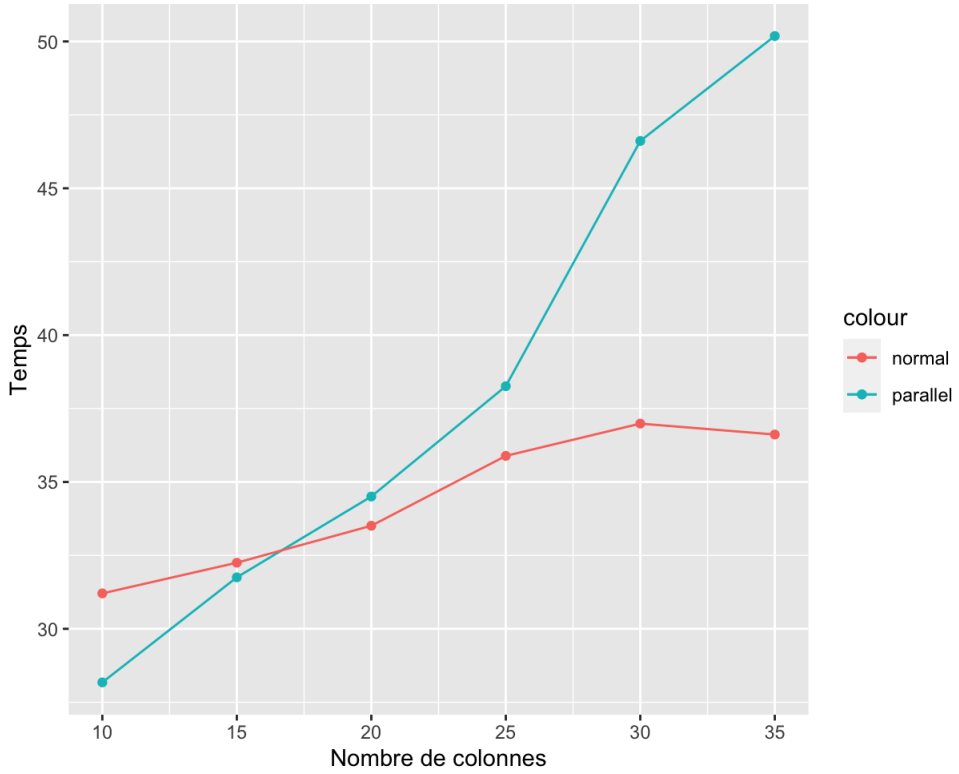
Afin de voir si notre l'implémentation était efficace, nous l'avons comparé avec la version « normale » du gradient batch, et ce, sur un jeu de donnée de plus en plus grand (que ce soit en ligne ou en colonne). Pour plus de précision quant au jeu de donnée, une fonction a été créée pour faire en sorte d'ajouter les lignes ou colonnes du dataset initial autant de fois qu'on le désire (dans notre cas, *breast_cancer*).

Dans un premier temps, nous avons juste étudié l'impact du nombre de lignes sur la durée d'exécution :



On remarque bien qu'au départ, le calcul parallèle dure plus longtemps que s'il n'était pas utilisé (si on se réfère au 1er point du graphique, l'exécution « normal » est 5 fois plus rapide). Cela pourrait s'expliquer par le temps de mise en place de la parallélisation qui dure tout de même quelque seconde ou encore à cause du transfert des données aux moteurs de calcul. Malgré tout, plus il y a de ligne et plus cette pénalité disparaît : comme on peut le voir sur le graphique, le retournement se fait à environ 300 000 lignes (soit environ 500x le dataset initial).

Ensuite, nous avons refait la même chose, mais cette fois-ci pour les colonnes. Pour le coup, nous savions qu'un grand nombre de variables à une influence négative sur un modèle en règle général (d'où les sélections de variables) mais nous aimerions savoir à quel point. Afin de bien voir son impact, nous avons augmenté le nombre de lignes à 350 000 :



On voit que pour 10, le nombre de colonnes initiale, le calcul parallèle à bien un impact, mais que plus il y a de colonne, moins cela marche, contrairement à la version « normal ». Ce changement se fait plutôt rapidement car, à peine le nombre de colonnes a été doublé que, déjà, on peut observer une grosse différence (contrairement aux lignes où il a fallu les multiplier par environ 500).

Pour aller plus loin, il pourrait être intéressant de comparer nos résultats avec ceux d'autres méthodes telles que le foreach, mclapply ou encore parLapply. On pourrait aussi faire tourner notre algorithme sur un nombre plus important de coeur et observer à quel point cela réduit le temps d'exécution.

4 Analyse des performances

4.1 Les données test

Le jeu de donnée que nous avons décidé d'utiliser pour tester notre modèle se nomme *breast_cancer*, avec un total de 666 lignes pour 10 colonnes. La variable à prédire est celle nommée "classe" et indique si une personne à une tumeur bénigne (69%) ou maligne (31%). Pour les tests ci-dessous, nous avons recodé la variable "classe" en 0 (bénigne) et 1(maligne).

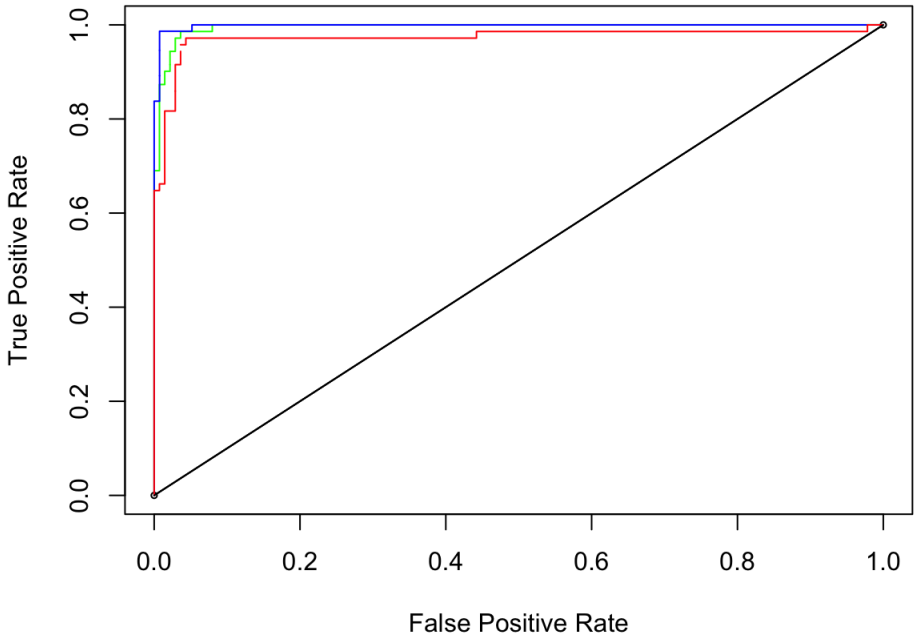
4.2 Précision

Afin d'observer le comportement de la précision pour les différentes méthodes (batch / mini-batch / online), nous avons sélectionné 70% des données de *breast_cancer* pour l'entraînement et 30% pour le test. Ensuite, chaque méthode a été lancée avec un nombre d'itérations croissant. À noter que, pour obtenir une précision la plus fiable possible, nous avons fait la moyenne des méthodes sur plusieurs essais pour chaque itération.

	iter	list_batch	list_mini_batch	list_online
1	50	0.5354067	0.8727273	0.9574163
2	100	0.5770335	0.9196172	0.9674641
3	200	0.6995215	0.9526316	0.9574163
4	500	0.8540670	0.9588517	0.9631579
5	800	0.9081340	0.9521531	0.9617225
6	1000	0.8851675	0.9665072	0.9612440
7	2000	0.9081340	0.9660287	0.9612440
8	5000	0.9631579	0.9712919	0.9641148

Comme on peut le voir ci-dessus, les résultats ne sont pas choquants avec la méthode batch qui met le plus d'itération à converger vers 100% de précision et online qui en met beaucoup moins.

On peut ensuite afficher les courbes ROC de ces méthodes en utilisant le tableau précédent. Nous sommes donc partis sur 5000 itérations pour batch et mini-batch et 100 pour online. La couleur bleue est celle de la méthode online, le vert, mini-batch et le rouge batch.



On retrouve ce que nous avions remarqué précédemment, c'est-à-dire que batch converge moins vite que les autres méthodes. Néanmoins, l'AUC reste tout de même élevée, ce qui est un bon signe pour notre modèle en règle général.

On peut aussi regarder la matrice de confusion qui nous permet d'avoir plus de précision quant aux erreurs de classification. Pour avoir une idée générale, nous avons fait la moyenne des résultats des 3 méthodes avec, pour rappel, un 0 si la tumeur est bénigne et un 1 si elle est maligne :

pred	0	1
0	64.114833	4.306220
1	1.913876	29.665072

En observant le tableau, on remarque que le plus d'erreurs est fait sur la classification de la classe 1 avec environ 4,5% contre 2% pour la classe 0.

5 Conclusion

En conclusion, avec la création de ce package, nous avons présenté une manière d'implémenter la fonction régression logistique pour les problèmes de classification binaire, et ce, avec la descente de gradient pour ses différentes méthodes : batch / mini-batch / online.

Une parallélisation a aussi été faites afin d'accélérer la durée d'exécution de notre algorithme, mais nous nous sommes vite rendu compte à quel point les gains étaient décevant sur un petit jeu de donnée. Sans doute, cela était-il dû à la mise en place de la parallélisation ou encore à cause du transfert des données aux moteurs de calcul. Ce n'est qu'à partir d'environ 350 000 lignes que celle-ci eu un vrai impact (en utilisant 6 coeurs).
On pourrait tout de même se demander si cette démultiplication des tâches n'aurait pas une faiblesse, notamment au niveau de la mémoire. Néanmoins, s'il y a la possibilité de l'utiliser sur des serveurs à distance, cela rendrait la parallélisation encore plus intéressante.