

Projet Spark - Descente de gradient en batch et minibatch

Victor Journe, Ludovic Lelievre

Introduction

Ce rapport donne des éléments théoriques de descente de gradient distribué pour une régression logistique. Le notebook associé fournit les codes Spark.

Un marché financier (un des marchés de l'électricité européen dans notre cas) est qualifié de court lorsque il y a plus de demande que d'offre et long sinon. A partir de trois ans d'historique (100 000 observations), notre objectif dans ce travail est de classer le sens du marché grâce à une cinquantaine de variables explicatives.

La figure 1 donne la répartition des prix du marché; on retrouve deux régimes qui constituent la classification binaire que propose ce rapport.

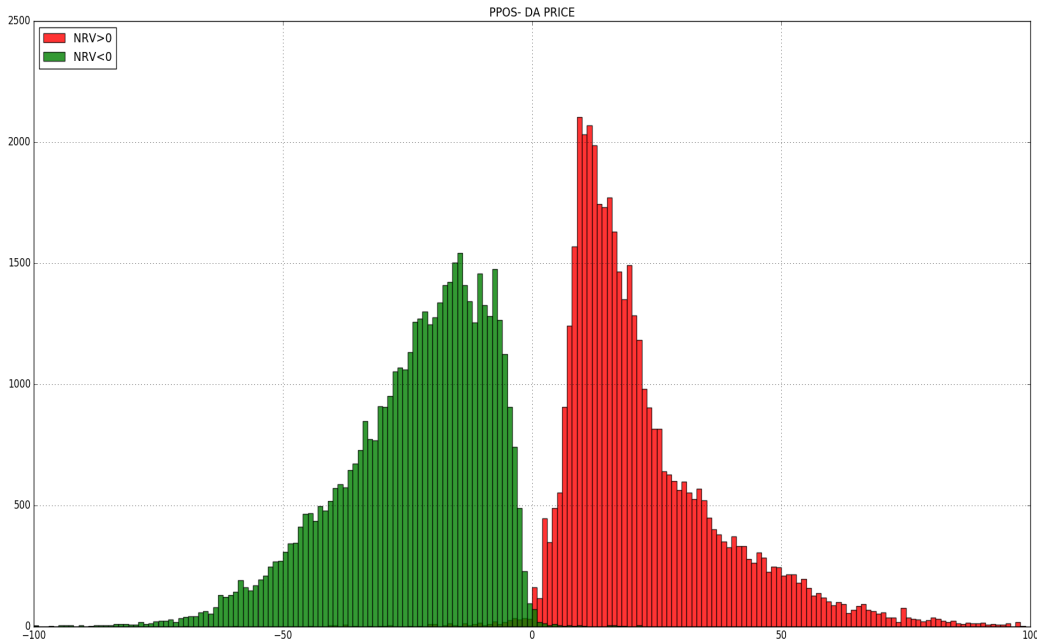


Figure 1: Les deux régimes de prix

Part I - Choix du modèle: Régression logistique régularisée

Nous souhaitons, à partir des données des années 2014 et 2015, prédire la situation du marché sur l'année 2016. Chaque quart d'heure, le marché est soit court ($y_i = 0$), soit long ($y_i = 1$). Nous sommes donc face à un problème de classification binaire.

Pour ce faire, nous allons utiliser une régression logistique régularisée afin de prédire la classe de chaque quart d'heure de l'année 2016. Nous choisissons la version régularisée de la régression logistique afin d'éviter que le modèle ne surapprenne.

La fonction logistique s'écrit:

$$h_w(x) = \frac{1}{1 + e^{-w^T x}}$$

où x représente les inputs et w les coefficients.

La fonction de coût de la régression logistique régularisée s'écrit:

$$J(w) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(h_w(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_w(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$

avec m est le nombre d'exemples et n le nombre de features.

Le problème de la régression logistique consiste à trouver les coefficients w tel que:

$$w = \operatorname{argmin} \{J(w)\}$$

Ce problème de minimisation peut se résoudre à l'aide d'un algorithme de descente de gradient. Il s'agit de calculer le gradient de $J(w)$ par rapport à chaque w_j puis de faire varier w_j dans le sens du gradient de manière itérative jusqu'à ce que le minimum de $J(w)$ soit atteint.

Le calcul du gradient s'écrit:

$$\begin{cases} \nabla w_0 = \frac{1}{m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \nabla w_j = \frac{1}{m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} w_j \end{cases}$$

pour $j = 1, \dots, n$.

Et les nouveaux coefficients sont obtenus de la manière suivante:

$$\begin{cases} w_{0,t+1} = w_{0,t} - \alpha \nabla w_{0,t} \\ w_{j,t+1} = w_{j,t} - \alpha \nabla w_{j,t} \end{cases}$$

L'algorithme de descente de gradient ne fait intervenir que des calculs de somme. Cet algorithme peut donc être implémenté en Spark afin de pouvoir calculer des descentes de gradient parallélisées.

Dans les parties II et III, nous expliquons la méthode que nous avons suivie pour réaliser la descente de gradient distribuée de la régression logistique ainsi que les résultats obtenus dans le cas du batch et dans le cas du minibatch respectivement.

Part II - Descente de gradient à pas constant en Batch

Dans cette partie, la méthode de la descente de gradient distribuée en batch est détaillée par la figure 2. Les 4 grandes étapes sont:

1. A partir d'un RDD texte chargé en mémoire, X_i et y_i sont formatés en objet *numpy array* pour faciliter le traitement.
2. Le gradient est calculé par ligne, constituant une réalisation $(X_i, y_i, grad_i)$. La valeur du vecteur w_t , nécessaire au calcul du gradient, est connue par tous les clusters par un broadcast.
3. Le gradient total s'obtient par somme de tous les gradients. Il faut lui ajouter le terme de pénalisation λw_t .
4. La dernière étape consiste à mettre à jour w_{t+1} par une descente de gradient à pas constant, contrôlé par α .

Les résultats après 30 itérations sont donnés dans le tableau 1. Le taux de bonne classification sur le set de test est de 74% et de 75% sur le set d'entraînement.

L'algorithme a convergé d'après le tracé de la fonction coût de la figure 3.

Part III - Descente de gradient à pas constant en minibatch

Dans cette partie, la méthode de la descente de gradient distribuée en minibatch est détaillée par la figure 4. Les 5 grandes étapes sont:

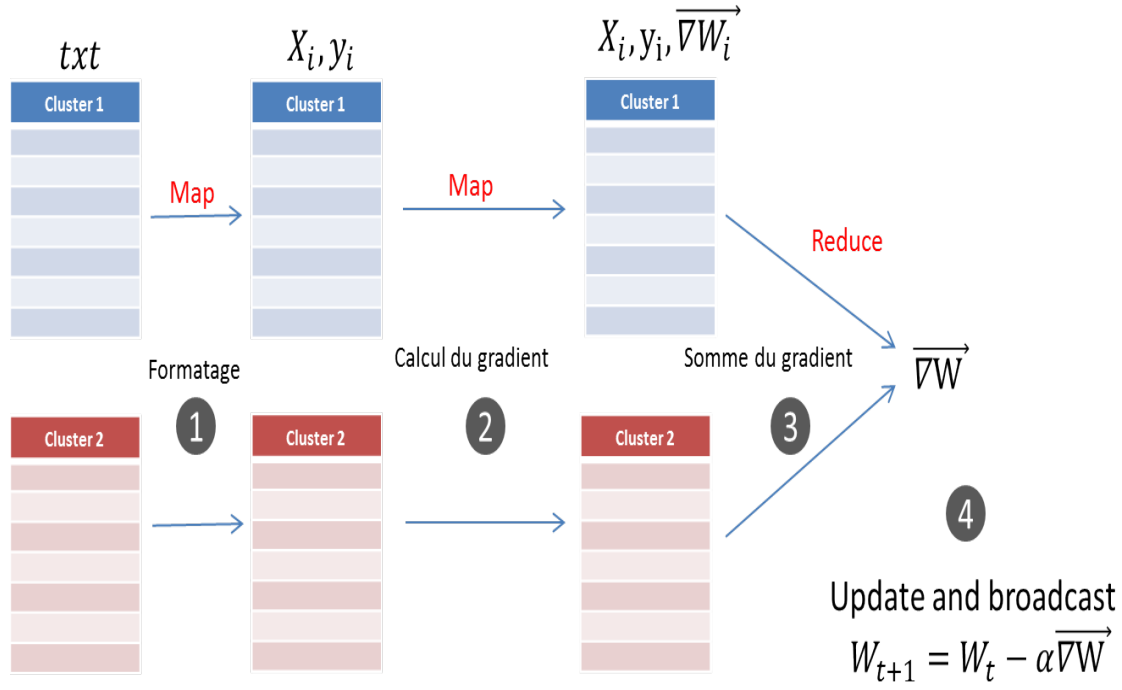


Figure 2: Descente de gradient distribuée en batch

	test			train	
Actual/predicted	0	1	Actual/predicted	0	1
0	10701	3223	0	26891	8921
1	3348	7977	1	8437	24668

Table 1: Matrices de confusion dans le cas du batch

1. A partir d'un RDD texte chargé en mémoire, nous ajoutons un index à chaque ligne de manière aléatoire afin de constituer les minibatches. Nous créons un vecteur de coefficients pour chaque minibatch que nous ajoutons à chaque ligne du RDD par index grâce à la transformation join.
2. Le gradient est calculé pour chaque ligne du RDD, constituant une réalisation $index_i, (X_i, y_i, grad_i)$.
3. Nous calculons le gradient pour chaque minibatch grâce à la transformation reduceByKey.
4. Les coefficients pour chaque minibatch sont modifiés par une descente de gradient à pas constant en prenant en compte le gradient correspondant au minibatch.
5. Le vecteur des coefficients finaux s'obtient par la moyenne des coefficients de chaque minibatch.

Les résultats après 30 itérations sont donnés dans le tableau 3. Le taux de bonne classification sur le set de test est de 74% et de 75% sur le set d'entraînement.

L'algorithme a convergé d'après le tracé de la fonction coût de la figure 5.

	test			train	
Actual/predicted	0	1	Actual/predicted	0	1
0	10706	3226	0	26885	8923
1	3343	7974	1	8443	24666

Table 2: Matrices de confusion dans le cas du minibatch

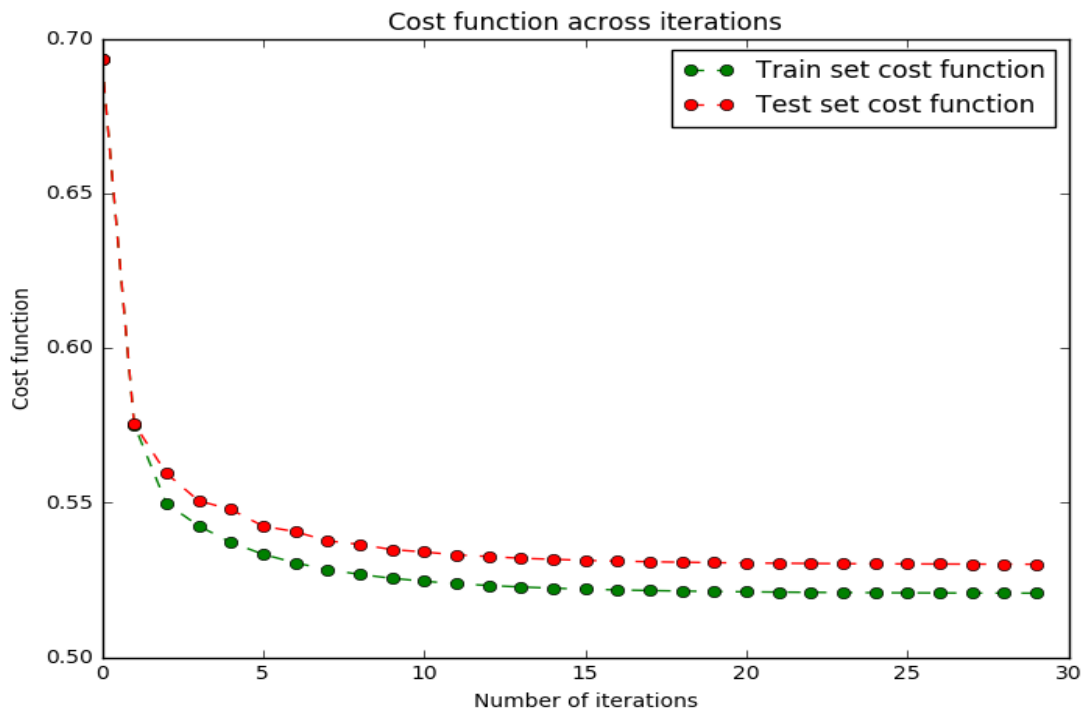


Figure 3: Convergence de l'algorithme dans le cas du batch

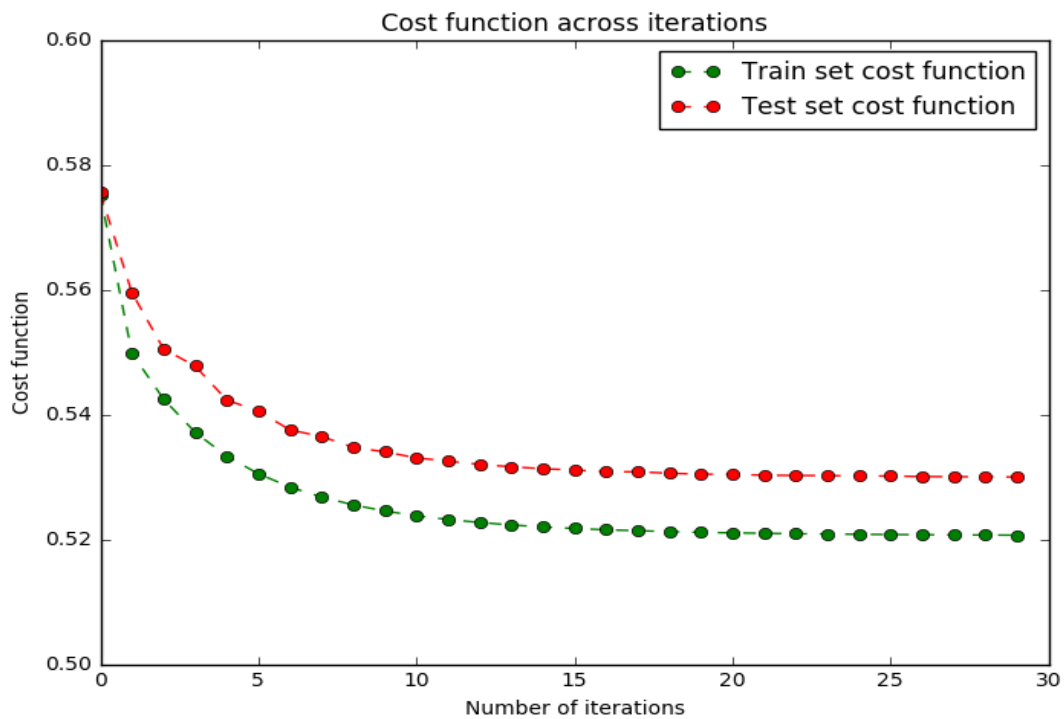


Figure 5: Convergence de l'algorithme dans le cas du minibatch

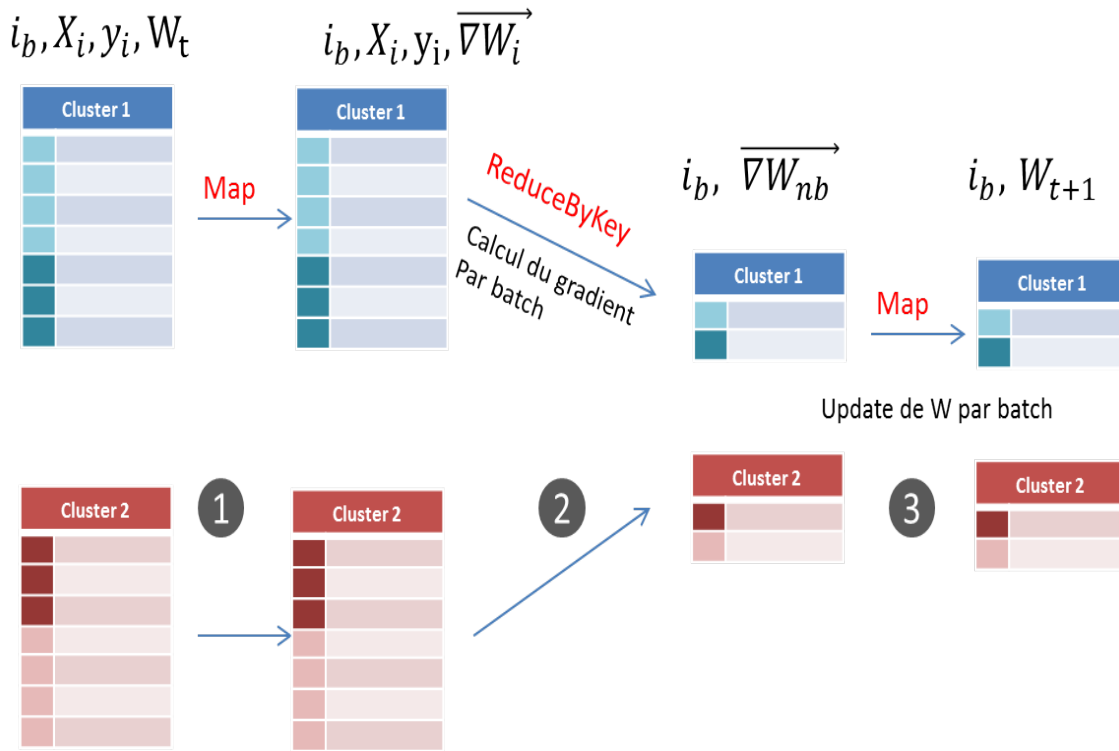


Figure 4: Descente de gradient distribuée en minibatch

Conclusion

L'algorithme de descente de gradient s'adapte très bien au calcul distribué, effectuée dans le cadre d'une régression logistique. De plus les résultats obtenus dans le cas du minibatch sont très proches de ceux obtenus dans le cas du batch, ce qui montre que la descente de gradient en minibatch converge vers la descente de gradient en batch.

Les résultats obtenus sont très proches de ceux donnés par *Sklearn*, la bibliothèque d'apprentissage en Python. Cependant, les temps de calcul sont bien inférieurs car les données sont d'une taille raisonnable, ce qui permet une résolution vectorielle. Pour atteindre la limite de *Sklearn*, il faudrait peut être 100 fois plus d'exemples. De plus, il serait intéressant de comparer les performances de Spark sur un plus grand nombre de clusters.

Enfin, un "Grid Search" pour le paramètre de régularisation λ serait envisageable avec Spark. Il faudrait alors dans l'étape du calcul du gradient, le faire pour différentes valeurs de λ , et conserver w pour chacun des λ .