# Projet Spark - Descente de gradient en batch et minibatch

Victor Journe, Ludovic Lelievre

# Introduction

Le marché de l'électricité européen se décompose en plusieurs marchés suivant l'échéance du produit. La veille de l'échéance, pour planifier la production, un marché au « fixing » a lieu, c'est le « day-ahead » (DA). Quelques heures avant l'échéance, un marché dit « intraday » permet aux acteurs de s'équilibrer pour pallier les pannes de centrales, ou un changement météorologique impactant la demande ou la production éolienne et solaire. Vient en dernier lieu le marché du « balancing », qui permet aux acteurs de proposer un volume de production ou de consommation directement utilisable pour le quart d'heure à venir. Au moment du transfert, le réseau doit être équilibré parfaitement entre la consommation et la production. Le responsable du réseau -ELIA en Belgique- engage ces ressources par le Net Regulation Volume (NRV) qui active à la hausse ou à la baisse le produit le moins cher encore disponible. Le marché est donc court ou long chaque quart d'heure, en fonction du signe du NRV. Le tableau 1 explique comment l'acteur en déséquilibre est rémunéré. MIP et MDP signifient marginal increasing ou decreasing price qui correspondent aux prix du dernier produit activé. Dans la tarification, on rajoute $\alpha$ pour pénaliser ou récompenser ceux qui aident le réseau à atteindre l'équilibre.

| marché | Long (NRV<0) | Court (NRV>0) |
|---|---|---|
| Prix par rapport au day-ahead | Au dessous | Au dessus |
| L'acteur est long, il est payé PPOS | MIP - $\alpha$ | MDP |
| L'acteur est court, il recoit | MIP | MDP + $\alpha$ |

Table 1: Principe de tarification

On peut trouver par exemple des prix négatifs pour une consommation (On est payé pour recevoir de l'électricité) quand on aide le réseau surchargé. Engie comme beaucoup d'acteurs du marché ont des contrats d'effacement avec des industriels par exemple ; il s'agit d'arrêter leurs productions pour une certaine durée dans l'idée d'aider le réseau lorsqu'il est court et toucher un prix noté PPOS. Il faut donc être en mesure de prédire le sens du marché dans le quart d'heure qui suit. La figure 1 rappelle la répartition des prix du marché; on retrouve deux régimes qui constituent la classifiction binaire que propose ce rapport.

La plupart des données sont accessibles sur le site d'Elia, puisque les marchés de l'éléctricité sont soumis à la transparence.

Les prédicteurs rassemblés pour prédire $NRV_{t+1}$ à ce jour sont :

- $NRV_t$,$PPOS_{t-1}$ : Net Regulation Volume et un retard de une demi-heure de la cible,
- $WR_{t-2}, SR_{t-2}, LR_{t-2}$ Production de Vent, Solaire et consommation réalisés ainsi que les derniers prévisions disponibles.
- $(Prix_i, Volume_i)$ De chaque produit i
- Eventuellement l'heure et le mois

Nous avons ajouté les 5 derniers prédécesseurs de chacune de ces variables, plus leurs valeurs moyenne sur la dernière heure, plus leurs séries différenciées. Ce qui fait un total de 55 variables. L'historique couvre 2014,2015 et 2016 au pas quart d'heure; soit 15x24x365x3 = 105120 observations. Le set d'entraînement comprend les années 2014 et 2015, le set de test est l'année 2016.

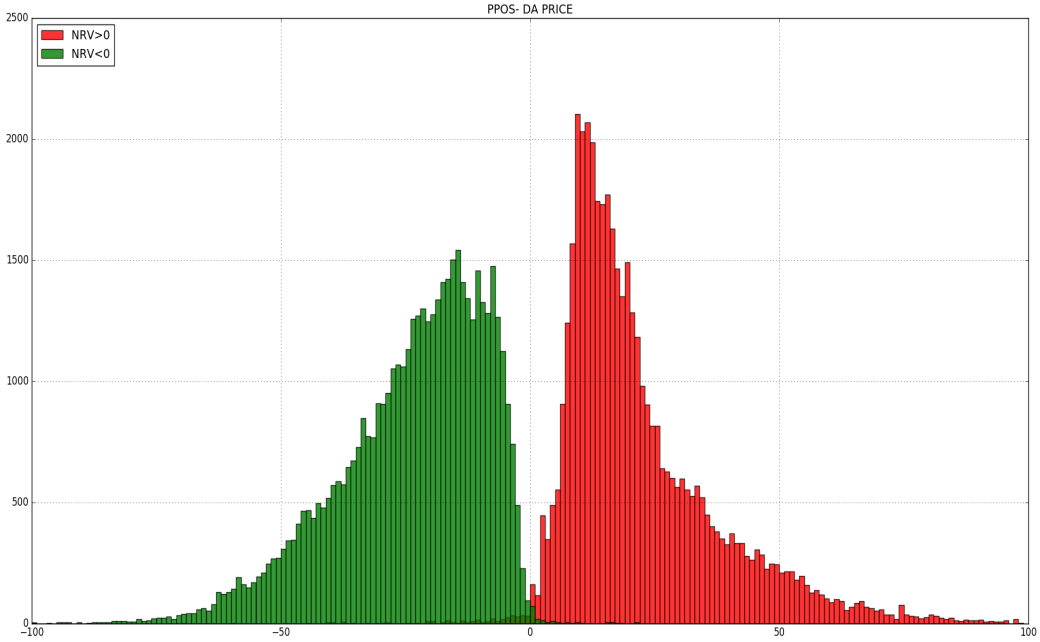


Figure 1: Les deux régimes de prix

# 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 coéfficients.

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

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

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

Le problème de la regression logistique consiste à trouver les coéffcieints $w$ tel que:

$$w = argmin\left\{J(w)\right\}$$

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:

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

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

Et les nouveaux coefficients sont obtenus de la manière suivante:

$$\left\{ \begin{array}{ll} w_{0,t+1} = & w_{0,t} - \alpha \nabla w_{0,t} \\ w_{j,t+1} = & w_{j,t} - \alpha \nabla w_{j,t} \end{array} \right\}$$

L'algorithme de descente de gradient ne fait intérvenir 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 suivi pour réaliser la descente de gradient distribuée de la régression logistique aisni 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 formater en object *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 $\lambda w_t$.
4. La dernière étape consite à mettre à jour $w_{t+1}$ par une descente de gradient à pas constant, controlé par $\alpha$.

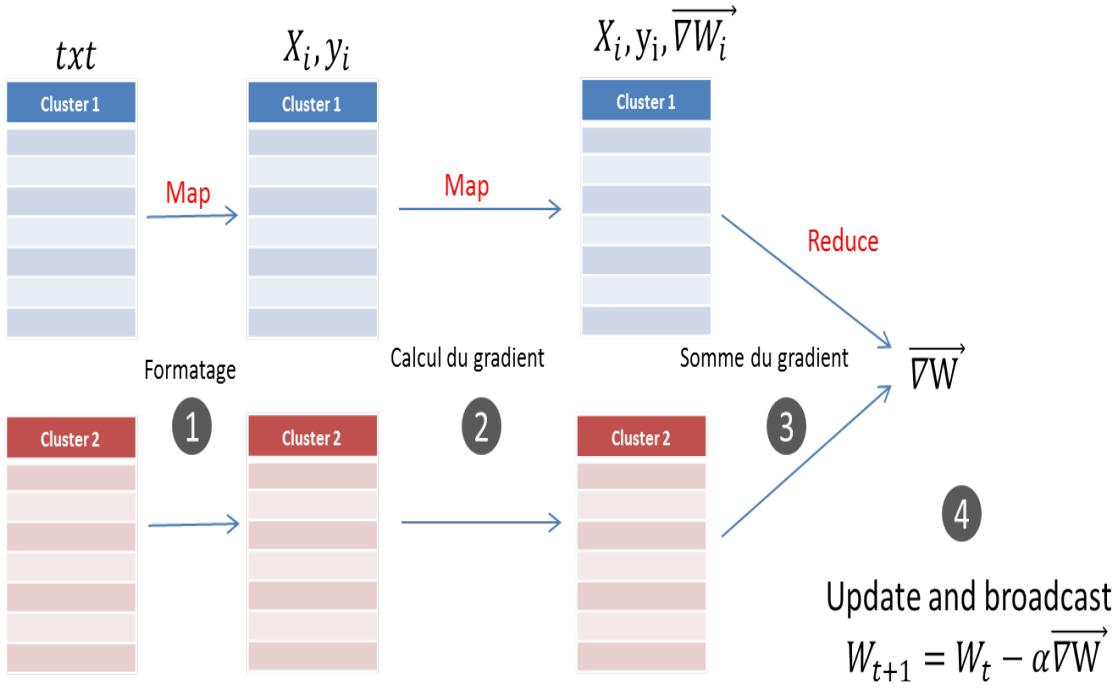


Figure 2: Descente de gradient distribuée en batch

Les résultats après 30 itérations sont donnés dans le tableau 2. 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.

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

Table 2: Matrices de confusion dans le cas du batch

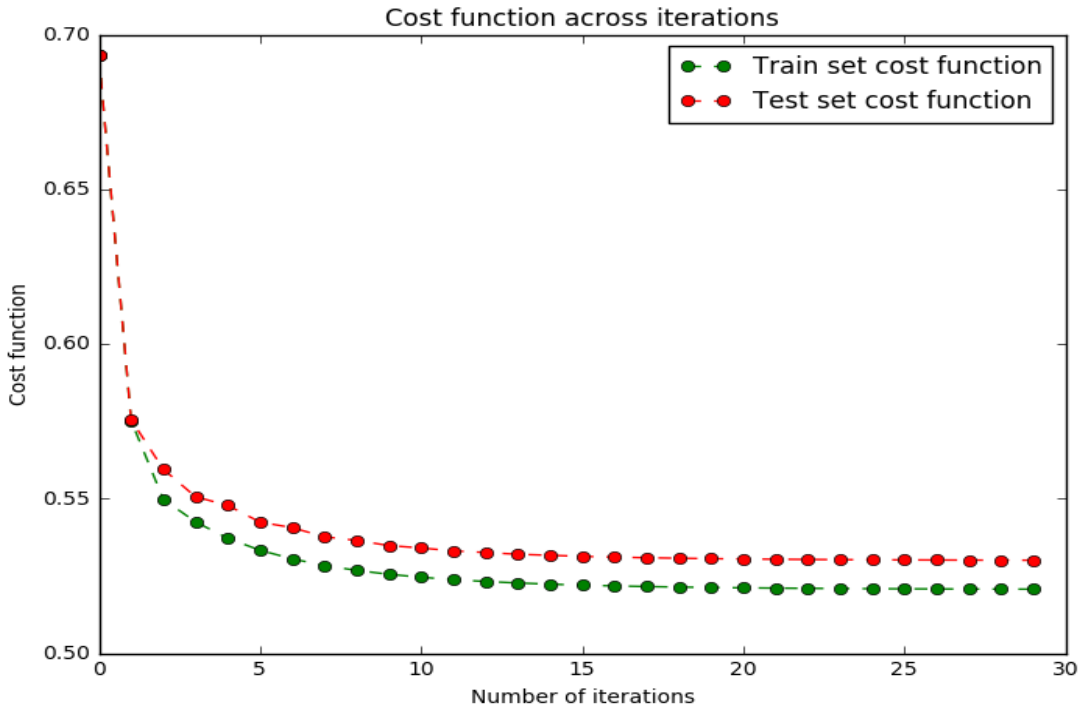


Figure 3: Convergence de l'algorithme dans le cas du batch

# 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:

1. A partir d'un RDD texte chargé en mémoire, nous ajoutons un index à chaque ligne de manière aléatoire afin de constituer les minibatchs. 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.

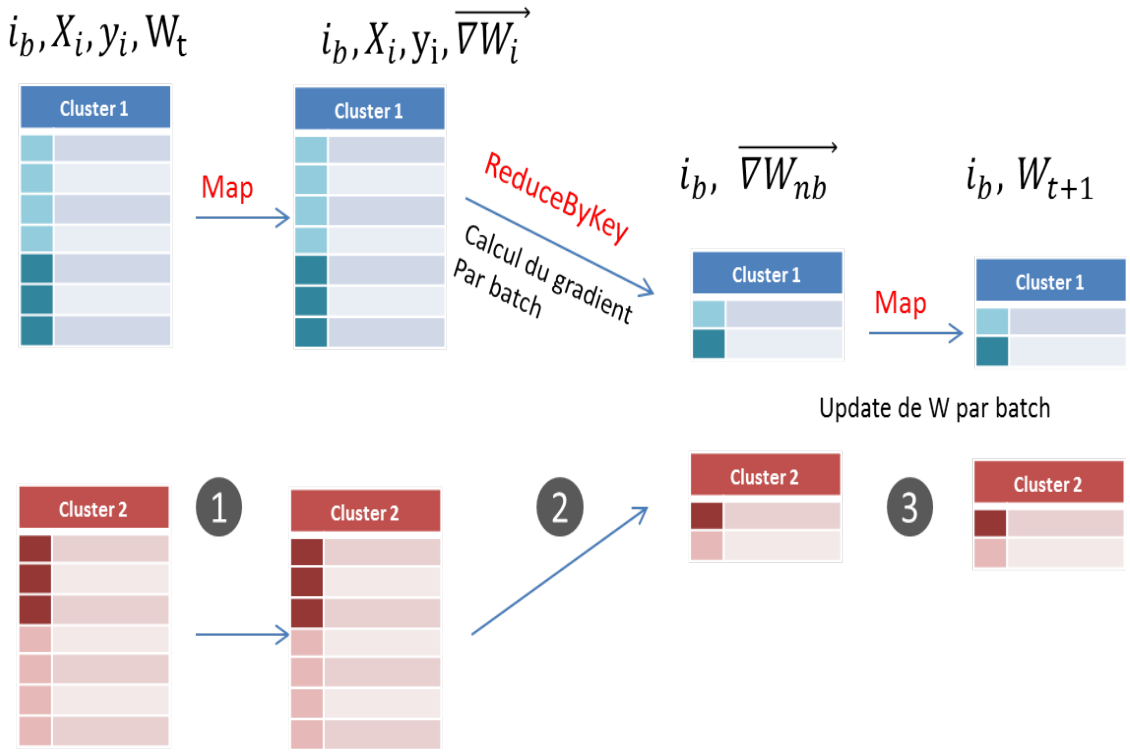


Figure 4: Descente de gradient distribuée en minibatch

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

Table 3: Matrices de confusion dans le cas du minibatch

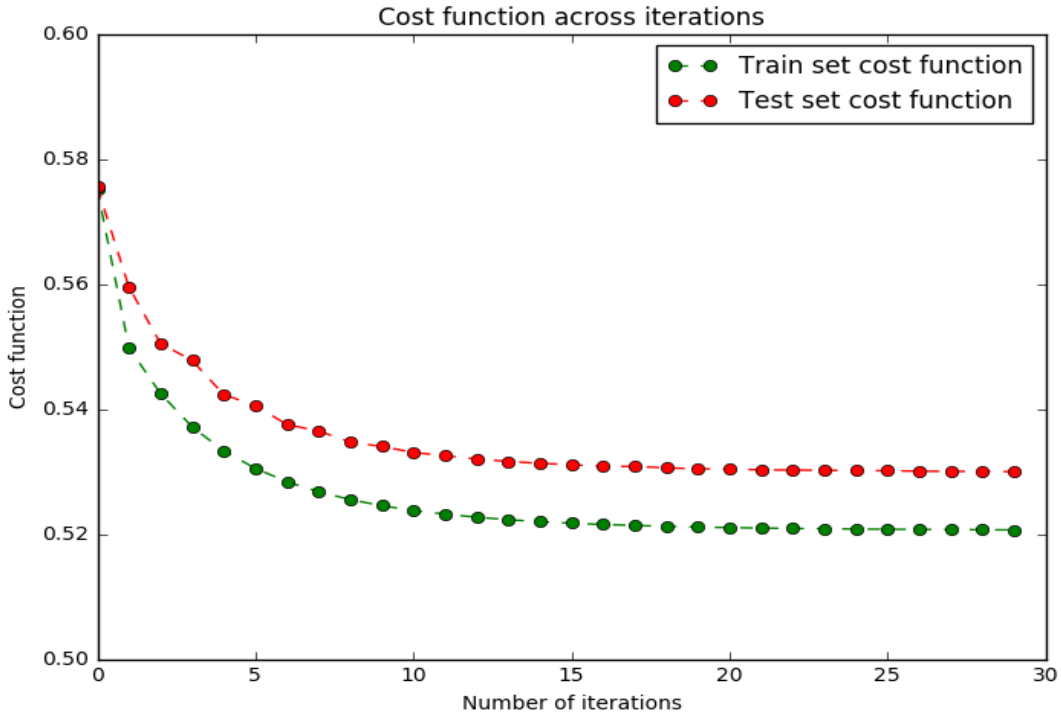


Figure 5: Convergence de l'algorithme dans le cas du minibatch

# Conlusion

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 $\lambda$ serait enviseagable avec Spark. Il faudrait alors dans l'étape du calcul du gradient, le faire pour différentes valeurs de $\lambda$, et conserver $w$ pour chacun des $\lambda$.