Affinity Propagation Clustering on PySpark

Elements logiciels de traitement des données massives

Maxime CALLOIX & Romane PERSCH

February 5, 2017

1 Introduction

L'Affinity Propagation est un algorithme de clustering très intéressant du point de vue de son interprétabilité. Sa première particularité est de ne pas nécessiter en entrée un nombre de classes a priori. Ainsi, contrairement à des algorithmes tels que k-means, l'Affinity Propagation renvoie un nombre de classes dépendant de la structure intrasèque des données. Son autre intérêt réside dans le fait que les centres des classes sont des points observés et non des points artificiels (dans le cas de k-means, il s'agit typiquement du barycentre des points de la classe). En termes d'interprétation, on a donc un "modèle réel" représentant les individus de la classe.

Cependant, il est en général utilisé sur des jeux de données de taille limitée car il est fondé sur la communication entre les individus observés de leur "intérêt" à se choisir en tant que centre. Il nécessite donc de ce fait un grand nombre d'itérations, et le paralléliser de façon "classique" engendrerait d'énormes coûts de communication entre les CPU. Nous nous inspirons donc de l'article "Map/Reduce Affinity Propagation Clustering" de Wei-Chih Hung, Chun-Yen Chu, et Yi-Leh Wu afin d'en proposer une version sur PySpark cherchant à limiter au maximum la communication entre les CPU.

2 Modèle

L'idée générale de l'Affinity Propagation est de considérer les individus observés comme les noeuds d'un graphe. Ces noeuds se transmettent des messages le long des arêtes de façon récursive jusqu'à ce qu'un ensemble de centres et que les classes correspondantes émergent. Ils cherchent en un sens au fur et à mesure de leurs communications à "élire" le meilleur centre pour eux.

L'algorithme est principalement fondé sur une mesure de similarité entre les individus. Nous utilisons ici la similarité euclidienne définie par :

$$s(i,k) = -||x_i - x_k||^2$$

D'autres mesures de similarité peuvent évidemment être utilisées, du moment qu'elles respectent la définition d'une mesure de similarité.

L'algorithme prend en entrée une valeur de préférence, qui représente la préférence d'un individu à se choisir lui-même comme centre plutôt que les autres. Comme évoqué plus haut, il n'y a donc pas besoin de définir un nombre de clusters a priori. Ici, nous considérons la valeur de préférence la plus communément utilisée : nous prenons la médiane des des similarités. Néanmoins, nous laissons dans la structure de notre code la possibilité d'ajouter d'autres stratégies de choix de la valeur de préférence (plusieurs possibilités sont notamment évoquées dans l'article).

2 types de messages passent alternativement entre les individus :

- la responsabilité $r(i,k)$ qui indique à quel point l'individu k est adapté pour servir de centre à l'individu i par rapport aux autres individus que pourrait choisir i comme centre
- la disponibilité (*availability* en anglais) $a(i,k)$ qui indique à quel point l'individu k est disponible pour servir de centre à l'individu i en tenant compte des préférences des autres points pour élire l'individu k comme centre

Les matrices de responsabilité et de disponibilité sont mises à jour comme suit :

$$r(i,k) = s(i,k) - max_{k' \neq k}\{a(i,k') + s(i,k')\}$$

$$a(i,k) = min\{0, r(k,k) + \sum_{i' \neq i, i' \neq k} max\{0, r(i',k)\}\}$$

$$a(k,k) = \sum_{i' \neq k} max\{0, r(i',k)\}\}$$

Les calculs utilisés dans le code sont détaillés dans le notebook. En pratique, pour éviter que les valeurs oscillent trop, un lissage est effectué entre l'ancienne valeur et la nouvelle à l'aide d'un coefficient de damping λ.

Ces mises à jour ont lieu successivement jusqu'à ce que l'un de ces critères d'arrêt soit atteint:

- Nombre maximum d'itérations (valeur par défaut = 200)
- Les centres restent les mêmes pendant un certain nombre d'itérations (valeur par défaut = 3)

D'autres critères d'arrêt peuvent évidemment être ajoutés mais nous sommes allés au plus simple.

3 Implementation sur PySpark

Comme on le voit, l'algorithme nécessite un grand nombre d'itérations et les coûts de communication entre les CPU seraient trop élevés si à chaque itération les CPU devaient s'échanger des données. L'article "Map/Reduce Affinity Propagation Clustering" de Wei-Chih Hung, Chun-Yen Chu, et Yi-Leh Wu propose une Impletementation avec Map/Reduce cherchant justement à limiter la communication entre les CPU. L'idée générale est de réaliser un Affinity Propagation clustering sur chaque CPU séparément et d'agréger ensuite les classes obtenues dont les centres sont proches (distance inférieure à un certain seuil).

Nous reprenons donc cette idée pour l'implementer sur Spark. Notre implementation est constituée de deux grandes étapes :

1. Affinity Propagation Clustering sur chaque CPU indépendamment: à l'aide de la commande mapPartitions essentiellement
2. Agrégation des centres "proches": en utilisant un algorithme de recherche des "connected components" au sein d'un graphe dont les noeuds sont les centres et dont les arêtes relient les centres considérés comme "proches"

Le détail de notre code et de nos choix de structuration des données est donné dans le notebook "Affinity Propagation Module Documentation". Le jeu de données Iris y est pris à titre d'exemple.

4 Test : Clustering d'activités sportives

4.1 Présentation des données

Nous avons appliqué notre algorithme de calcul distribué sur une base de données un peu plus conséquente qu'Iris : Daily and Sports Activities Data Set [1]. Huit personnes ont effectué chacune dix-neuf activités différentes pendant 5 minutes (découpées en 60 segments). Durant ces activités, des capteurs placés à différents endroits de leur corps ont pris 45 types de mesures différentes (25 mesures par seconde pour chaque type). La base de données correspond donc aux résultats de ces mesures (donc 45 séries temporelles de 25 * 5 éléments) pour chacune des 9120 observations (60 segments * 8 personnes * 19 activités). Le but est ici de regrouper les activités similaires du point de vue de ces mesures.

L'algorithme d'Affinity Propagation est particulièrement adapté à cette problématique. Comme vu en introduction, nous n'avons pas besoin de faire une hypothèse sur le nombre de classes a priori. Nous aurions en effet bien du mal à supposer un nombre de classes ici. De plus, nous sommes dans le cas d'un objet géométrique complexe : chaque observation pouvant être vue comme une série temporelle multidimensionelle avec intrinséquement des corrélations et des formes qui ont surement un pouvoir explicatif plus fort que leurs valeurs prisent indépendemment. L'algorithme k-mean avec distance euclidienne ne serait ni en mesure

[1] à télécharger sur le site de l'UCI : Daily and Sports Activities Data Set

d'utiliser cette information sur la géométrie des données ni capable de représenter l'individu central de la classe. A l'inverse l'Affinity Propagation renvoie un individu centre réel et permet d'utiliser une mesure de similarité différente de la mesure euclidienne.

Nous avons utilisé plusieurs approches afin de calculer nos différents clusters. Nous avons fait varier la sélection des préférences (valeur moyenne, valeur minimale) et le paramètre d'agrégation. Le paramètre d'agrégation est lié à la préférence : plus la préférence est élevé, plus le paramètre d'agrégation doit être élevé pour pourvoir regrouper les clusters. Enfin, nous avons utilisé plusieurs distances différentes:

- La distance euclidienne, ce n'est pas la plus adaptée mais elle a l'avantage d'être simple et rapide lors des calculs.
- La distance *Complexity Invariant Distance* introduite par Batista, Wang, Kheog[2] : conçue pour étudier la distance entre deux séries temporelles. Dans cette mesure, la distance euclidienne est multipliée par les *Complexity Estimate* de chaque série temporelle. Comme nous sommes en présence de séries temporelles multidimensionnelles, la distance entre chaque observation est calculée comme la somme des *Complexity Invariant Distance* de chacune des séries temporelles.

$$CE = \sqrt{\sum_{i=0}^{n-1}(x_i - x_{i+1})^2}$$

$$CID(a,b) = ED(a,b)\frac{\max(CE(a), CE(b))}{\min(CE(a), CE(b))}$$

- La distance euclidienne entre les features de chaque série temporelle (minimum, maximum, moyenne, médiane, écart-type et *Complexity Estimate*)

Il existe, bien sûr, de nombreuses autres distances. Certaines sont plus appropriées car elles permettent d'analyser directement les séries temporelles multidimensionnelle en tenant compte des corrélations (et qui nécessitent de décomposer la matrice de covariance en vecteur propre). Cependant, en utilisant l'algorithme en local, nous n'avions pas la puissance de calcul nécessaire pour utiliser ces distances dans un temps raisonnable.

Nos résultats ont été obtenus avec une valeur de préférence minimum et une valeur d'agrégation de 0.04.

4.2 Résultat pour la distance euclidienne

Avec la distance euclidienne, on obtient des résultats cohérents (cf. Table 1 et Figure 1). A l'exception de la classe 2924 dont l'effectif est très faible, les classes sont cohérentes avec les activités qui les composent. Chaque activité appartient (à l'exception parfois de quelques observations) à une classe unique. De plus, les activités des classes au sein de ces classes sont proches : la classe 368 regroupe des activités à posture debout (marche, monter des escaliers, sauter...) tandis que la classe 1005 regroupe des activités à postures assises (rester assis, aviron, vélo...).

Notons que la représentation graphique ne permet pas de représenter fidèlement la complexité géométique de nos observations.

4.3 Résultat pour la distance *Complexty Invariant Distance*

La figure 2 montre l'agrégation des classes avec la distance *Complexty Invariant Distance*. Les résultats que l'on obtient sont moins convaincant que les résultats avec la distance euclidienne bien qu'assez similaires (les deux principales classes sont rassemblées). La distance ne semble pas représenter assez bien la géométrie de l'observation : une distance qui prendrait en compte les corrélations entre séries temporelles d'un même individu serait peut-être plus appropriée.

4.4 Résultats en utilisant les features

Après avoir calculé différents features (minimum, maximum, moyenne, médiane, écart-type et *Complexity Estimate*), on utilise entre-eux la distance euclidienne. Les résultats peuvent être visualisés sur la Table 2 et la Figure 3. Avec les features, les résultats sont semblables aux précédants. On retrouve plus ou moins les mêmes classes qu'avec la distance euclidienne.

[2] http://www.cs.ucr.edu/~eamonn/Complexity-Invariant%20Distance%20Measure.pdf

Table 1 – Résultats en utilisant la distance euclidienne

	Classe 368	Classe 1005	Classe 2924	classe 2927
sitting	5	475	-	-
standing	480	-	-	-
lying on back	-	476	-	4
lying on right side	-	25	15	440
ascending stairs	480	-	-	-
descending stairs	480	-	-	-
standing in an elevator still	480	-	-	-
moving around in an elevator	478	2	-	-
walking in a parking lot	480	-	-	-
walking on a treadmill (flat)	480	-	-	-
walking on a treadmill (inclined)	480	-	-	-
walking on a treadmill (with speed)	480	-	-	-
exercising on a stepper	480	-	-	-
exercising on a cross trainer	480	-	-	-
cycling on an exercise bike (horizontal)	-	480	-	-
cycling on an exercise bike (vertical)	-	480	-	-
rowing	-	480	-	-
jumping	480	-	-	-
playing basketball	479	1	-	-

Table 2 – Résultats en utilisant les features

	Classe 49	Classe 186	Classe 548	classe 816	classe 7588	classe 7789
sitting	479	1	-	-	-	-
standing	-	480	-	-	-	-
lying on back	477	-	3	-	-	-
lying on right side	10	-	422	35	13	-
ascending stairs	-	480	-	-	-	-
descending stairs	-	480	-	-	-	-
standing in an elevator still	-	480	-	-	-	-
moving around in an elevator	1	478	1	-	-	-
walking in a parking lot	-	480	-	-	-	-
walking on a treadmill (flat)	-	480	-	-	-	-
walking on a treadmill (inclined)	-	480	-	-	-	-
walking on a treadmill (with speed)	-	475	-	-	-	5
exercising on a stepper	-	480	-	-	-	-
exercising on a cross trainer	-	480	-	-	-	-
cycling on an exercise bike (horizontal)	480	-	-	-	-	-
cycling on an exercise bike (vertical)	763	17	-	-	-	-
rowing	480	-	-	-	-	-
jumping	-	480	-	-	-	-
playing basketball	1	479	-	-	-	-

Figure 1 – Répartition des classes avec la distance euclidienne selon les valeurs du premier et second axe

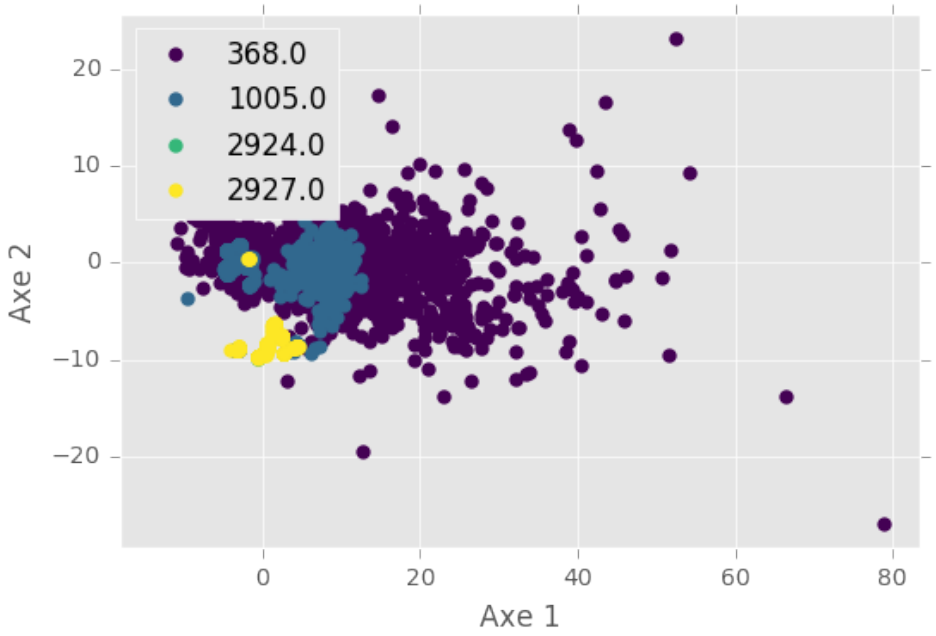
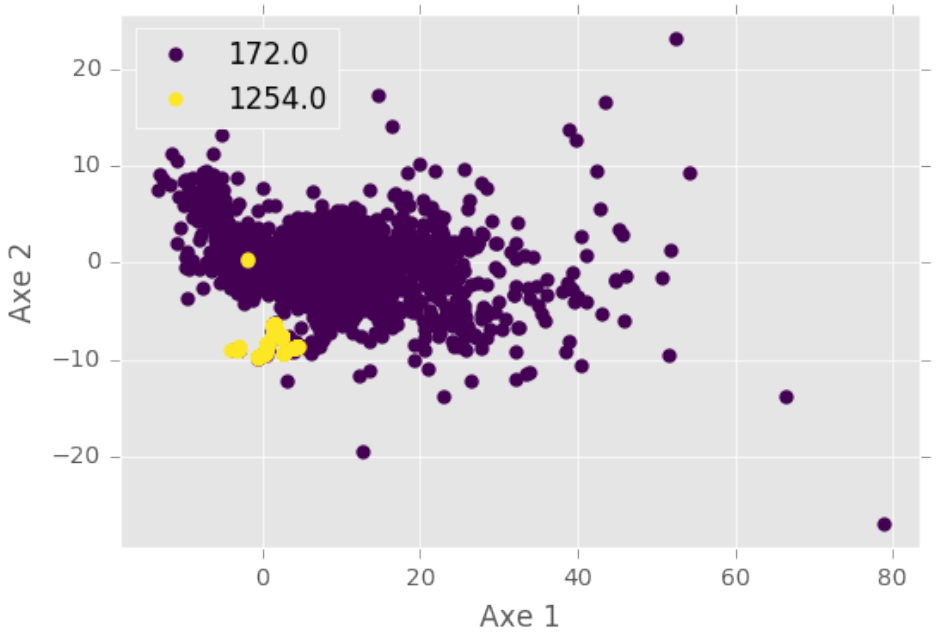


Figure 2 – Répartition des classes avec la *Complexty Invariant Distance* selon les valeurs du premier et second axe



5 Conclusion

Au cours de ce projet, nous nous sommes donc familiarisés avec l'outil Spark. Nous avons proposé la version distribuée sur Spark d'un algorithme de Clustering appelé Affinity Propagation dont la parallélisation était loin d'être évidente du fait d'un grand nombre d'itérations. Enfin, nous avons vérifié notre approche sur le petit jeu de données Iris et nous l'avons appliquée à des données issues de capteurs afin de clusteriser des activités sportives. Le choix d'une base de données avec des observations sous forme de séries temporelles multidimensionnelles nous a de plus permis d'utiliser sur PySpark des techniques de préparation des données un peu plus poussées (réduction des données / création de features, utilisation de distances non-euclidiennes).

Figure 3 – Répartition des classes avec les features selon les valeurs du premier et second axe