Forage de données
Mini-Projet
8INF854

Papa Alioune FALL, Quentin LETORT

5 mai 2020

UQAC
Université du Québec
à Chicoutimi

Résumé

Présentation d'une technique de fouille de données au niveau algorithmique et mise en place d'une expérimentation avec des données d'apprentissage dans un contexte réel.

Table des matières

1 Contexte de l'étude

Pour débuter ce rapport et mener au contexte de l'étude, il nous semblait pertinent de présenter une synthèse du papier sur le framework **DistributedWekaSpark** utilisé dans ce projet.

1.1 Synthèse de l'article

L'article se nomme "A Parallel Distributed Weka Framework for Big Data Mining using Spark" et porte sur l'utilisation d'un framework pour faire de la fouille de données de manière distribuée.
Ce papier parle des apports des nouvelles méthodes de traitement des données par rapport aux méthodes traditionnelles. Les solutions distribuées permettent aujourd'hui une extraction et une fouille efficace de grandes quantités de données, Hadoop une de ces solution permet un traitement des données sur disque mais est inefficace pour les applications qui nécessitent des itérations. Une solution plus récente pour palier à ce problème est Spark qui lui fait un traitement rapide en mémoire et prend en charge les calculs itératifs.

Ce papier s'intéresse ainsi à un environnement facile à utiliser pour l'extraction et la fouille de données pour les analystes de données. Weka est une des plateformes qui intègre ces différents algorithmes de fouille de données, et ce papier traite en particulier de l'utilisation de **DistributedWekaSpark**. Cette extension de Weka permet d'utiliser les méthodes d'apprentissage automatique offertes par Spark via l'interface proposée par Weka. Spark propose trois classes de méthodes pour l'exploration des données : la classification, la régression et la recherche de règles d'association.

Dans ce papier, une expérience est menée afin de comparer les performance des algorithmes fournies par **DistributedWekaSpark**. L'expérience a été faite avec différentes instances d'Amazon EC2 avec trois configurations distinctes : petite échelle (8 cœurs / 28,2 Go), moyenne échelle (32 cœurs / 112,8 Go) et grande échelle (128 cœurs / 451,2 Go). Chacune des classes d'algorithmes de **DistributedWekaSpark** a été évaluée à l'aide d'un algorithme représentatif : la classification à l'aide des SVM, la régression à l'aide de la régression linéaire et la recherche de règles d'association à l'aide de l'algorithme FP-Growth en utilisant à chaque fois les trois échelles.
En parallèle, les tests ont également été effectués sur Hadoop pour comparer les performances. Les résultats en termes d'efficacité, de mise à l'échelle et de temps d'exécution par rapport à Hadoop ont été rapportées dans le papier et montrent que **DistributedWekaSpark** réalise une mise à l'échelle presque linéaire sur différentes charges de travail et montre des accélérations jusqu'à 4x plus rapides en moyenne que Hadoop sur des charges de travail identiques.

1.2 Contexte

Le traitement des données occupe aujourd'hui une place importante dans le big data. Avec l'explosion des données, il est nécessaire de trouver une technologie capable de traiter ces ensembles de données. Les systèmes distribués offre des solutions pour l'extraction et une fouille efficace des données. Hadoop est la plateforme distribuée la plus utilisée mais traite les données sur le disque, ce qui rend les traitements inefficaces pour les applications d'exploration de données nécessitant souvent des itérations. Pour traiter les données plus rapidement, il existe le framework Spark, qui lui assure les traitements des données en mémoire facilitant les traitements itératifs.
Dans ce projet, nous nous intéresserons à ce framework et son utilisation à travers le logiciel Weka et son extension **DistributedWekaSpark**.

2 Problématique et objectifs

La problématique consiste à savoir comment analyser et traiter plus efficacement les grandes quantités de données (dites Big Data), en assurant à la fois rapidité et facilité d'utilisation. En ce sens, nous verrons en détail le fonctionnement d'un algorithme de classification supervisé par arbre de décision , l'algorithme C4.5, utilisant le paradigme de programmation distribuée. Nous appliquerons ensuite ce type d'algorithme distribué sur des données réelles grâce à Spark et Weka et comparerons les performances avec des algorithmes traditionnels d'apprentissage automatique présents dans Weka.

3 Description de l'algorithme

Spark, plateforme de calcul distribuée pour les grandes quantités de donnée, se base principalement sur le modèle de programmation distribué appelé MapReduce. Le principe est de faire des calculs parallèles et distribués et ceci en mémoire à l'aide de deux types d'opérations : Map et Reduce.
L'opération Map consiste à appliquer une fonction pour transformer chaque instance des données en une paire clé-valeur et l'opération Reduce applique une fonction pour chaque clé et fournit un résultat à partir des valeurs associées à une clé. Le fonctionnement est le suivant :

- diviser l'ensemble des données en plusieurs sous ensemble
- puis appliquer l'opération map à chaque sous ensemble de données. Ce qui permet de transformer la représentation des éléments des sous ensembles en paires clé-valeur.
- les paires trouvés précédemment sont regroupées et triées par clé.
- enfin l'opération reduce est appliquée pour chaque clé. Elle consiste à effectuer une opération pour combiner l'ensemble des valeurs associées à une clé.

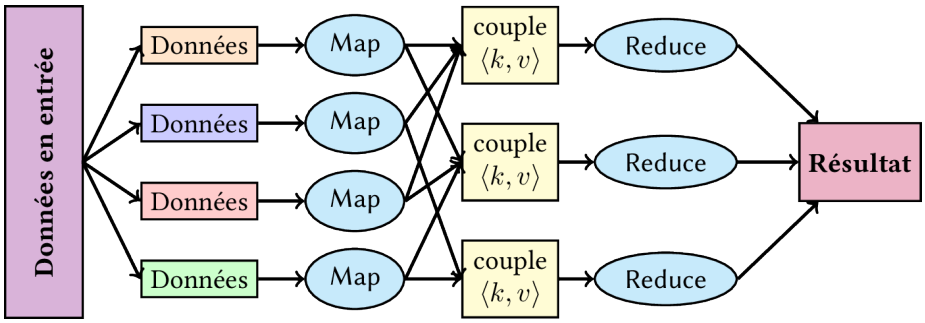

FIGURE 1 – Schéma de fonctionnement de MapReduce

Intéressons nous plus particulièrement au fonctionnement de l'algorithme C4.5 qui est un algorithme de classification supervisé produisant un modèle de type arbre de décision. Traditionnellement, l'algorithme fonctionne de la sorte :

Soit un ensemble S de données labellisées avec chaque instance constituée de p valeurs d'attributs.
A chaque noeud, l'algorithme choisit l'attribut qui sépare le plus efficacement l'ensemble en sous-ensembles en essayant d'isoler les classes. En ce sens, il utilise le critère du gain d'information (basé sur l'entropie) et choisit l'attribut avec le gain d'information le plus élevé.
L'algorithme agit ensuite récursivement sur les sous-ensembles générés. Pour que l'algorithme puisse arrêter la récursion, trois conditions d'arrêts sont déclarées :

- Toutes les instances de l'ensemble appartiennent à la même classe. Dans ce cas, l'algorithme crée une feuille dans l'arbre de décision associée à cette classe.
- Aucun des attributs ne fournit de gain d'information. Dans ce cas, l'algorithme crée un noeud de décision plus haut dans l'arbre en utilisant la valeur attendue de la classe.
- Une instance d'une nouvelle classe est rencontrée. Dans ce cas, l'algorithme agit comme dans le cas précédent en créant un noeud de décision plus haut dans l'arbre avec la valeur attendue.

Une fois l'arbre construit, il suffit simplement de le parcourir en fonction des valeurs d'attributs de nouvelles instances afin d'en prédire la classe.

Voyons désormais sa version distribuée (issue du papier de Mu Y, Liu X, Yang Z, Liu X) :
L'algorithme se divise cette fois-ci en trois parties distinctes :

- un algorithme central permettant la construction de l'arbre
- un algorithme chargé de sélectionner l'attribut séparant le plus efficacement les données d'un noeud (composé des opérations Map et Reduce)
- un algorithme chargé de séparer les données du noeud entre les différents noeuds enfants (composé également des opérations Map et Reduce)

L'algorithme central prend en entrée le jeu de données d'entraînement (chaque instance est composée d'un ensemble d'attributs et d'une classe) ainsi que des paramètres tels que la profondeur maximum de l'arbre.
L'algorithme central appelle les deux autres algorithmes (Sélection d'attribut et séparation des données) sur chaque noeud grâce à la récursivité (en partant du noeud racine).

Algorithm 1: Algorithme de construction de l'arbre

MapReduceC45 ($trainingData, maxDepth, currentDepth$)
- **Entrée:**
  - Ω_0 : ensemble de données d'entrainement
  - $maxDepth$: profondeur maximale de l'arbre
  - $currentDepth$: profondeur actuelle de l'arbre
- **Sortie:** L'arbre C4.5 construit

if $currentDepth > maxDepth$ **then**
  return
$\Omega = \{\}$
`// Itération sur l'ensemble des noeuds de` Ω_0
for $X : \Omega_0$ **do**
  $resultBestAttribute$ = GetBestAttribute($trainingData$)
  $bestAttribute = resultBestAttribute.bestAttr$
  $threshold = resultBestAttribute.threshold$
  `// Division des données en` n `noeuds enfants`
  $\{X_i\}_{i=1}^{n}$ = SplitData($trainingData, bestAttribute, threshold$)
  Add $\{X_i\}_{i=1}^{n}$ to Ω
if Ω *is not empty* **then**
  $\Omega_0 = \Omega$
  MapReduceC45($\Omega_0, maxDepth, currentDepth + 1$)

L'algorithme de sélection du meilleur attribut s'appuie sur le modèle Map/Reduce afin de calculer le gain ratio et le seuil de chaque attribut (pour une variable qualitative, il s'agit d'un nombre permettant de séparer les données en deux ensembles tandis que pour une variable qualitative, il s'agit des différentes valeurs de cette variable) pour ensuite déterminer l'attribut ayant le meilleur gain ratio (ce dernier étant directement corrélé au gain d'information de l'attribut).

L'algorithme de division des données s'appuie également sur le modèle Map/-Reduce afin d'attribuer à chaque instance un noeud en fonction de l'attribut sélectionné et le seuil de séparation.

Algorithm 2: Algorithme de sélection du meilleur attribut

GetBestAttribute (*trainingData*)

Entrée:

— *trainingData* : données d'entrainement. Chaque instance possède une classe et des valeurs d'attributs

Sortie: structure composée de $\{bestAttr, threshold\}$

$resultMapReduce$ = runMapReduceJob(*trainingData*)

```
/* resultMapReduce est constitué de paires <attribute,
   (totalRatio, finalThreshold)> suite à l'application
   des opérations map et reduce sur les données
   d'entrainement                                          */
```

$result$ = findBestAttribute($resultMapReduce$)

```
/* sélectionne l'attribut avec le gain ratio le plus
   élevé                                                   */
```

return *result*

map (*data*)

Entrée:

— *data* : partition de données

Sortie: Ensemble de paires $< clé, valeur >$ de la forme $< attribute, (gainRatio, threshold) >$

foreach *attribute in data.attributes* **do**

- $threshold$ = findThreshold($data, attribute$)
- $infoGain$ = computeInfoGain($data, attribute$)
- $splitInfo$ = computeSplitInfo($data, attribute$)
- $gainRatio = \frac{infoGain}{splitInfo}$
- emit($attribute, (gainRatio, threshold)$)

reduce $(attribute, List < (gainRatio, threshold) >)$

Entrée:

— Paire $< clé, valeur > = < attribute, List < (gainRatio, threshold) >>$. Chaque attribut est associé à un ensemble de ratio et seuil

Sortie: $< clé, valeur > = < attribute, (totalRatio, finalThreshold) >$

if *attribute is numerical* **then**

- $finalThreshold = \sum \frac{threshold}{m}$

```
/* m est la taille de la liste                            */
```

else

- $finalThreshold = \bigcup threshold$

$totalRatio = \sum gainRatio$

emit($attribute, (totalRatio, finalThreshold)$)

Algorithm 3: Algorithme de division des données

SplitData $(trainingData, attribute, threshold)$

Entrée:

— $trainingData$: données à diviser

— $attribute$: attribut à utiliser pour la division des données

— $threshold$: seuil de séparation (nombre en cas d'attribut numérique et catégorie en cas d'attribut catégoriel)

Sortie: Ensemble des données des noeuds enfants

$resultMapReduce$ = runMapReduceJob$(data, attribute, threshold)$

return $getDataSets(resultMapReduce)$

map $(data, attribute, threshold)$

Entrée:

— $data$: partition de données

— $attribute$: attribut à utiliser pour la division des données

— $threshold$: seuil de séparation (nombre en cas d'attribut numérique et catégorie en cas d'attribut catégoriel)

Sortie: Ensemble de paires $< clé, valeur >$ de la forme $< id, instance >$ avec id correspondant à la valeur d'attribut de l'instance

foreach *instance in data* **do**

  if *attribute is numerical* **then**

    if $instance[attribute] > threshold$ **then**

      $id = 1$

    else

      $id = 0$

  else

    $id = instance[attribute]$

  emit$(id, instance)$

reduce $(id, List < instance >)$

Entrée:

— Paire $< clé, valeur > = < id, List < instance >>$. Toutes les instances associées à une valeur de l'attribut sont regroupées dans une liste.

Sortie: $< clé, valeur > = < id, List < instance >>$

$nbClasses$ = distinctClassValues$(List < instance >)$

if $nbClasses \leq 1$ **then**

  Build a no leaf node

else

  Build a leaf node

emit$(id, List < instance >)$

4 Structure de données utilisée

La structure de donnée utilisé est un format arff de Weka. Il s'agit d'un ensemble de données de phishing de sites Web collectés au niveau des archives de données du site Phishtank : PhishingData.arff.

5 Échantillonnage de données d'apprentissage et de test

Le dataset est composé de 1353 de sites web (instances) et comprend 10 attributs. Les sites web collectés sont répartis comme suit : 548 sites légitimes, 702 URL de phishing et 103 URL suspectes.

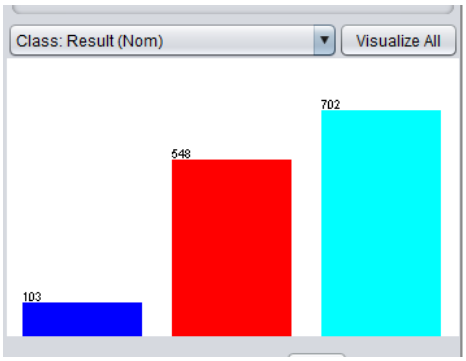

FIGURE 2 – Classification des instances

Le dataset se trouve sur le lien suivant : Lien dataset de données d'apprentissage et de test. Le tableau suivant donne les sept premiers exemples de données suivant les attributs ainsi que les résultats trouvés Légitime (1), Suspect (0). et Phishy (-1) à l'issue de l'analyse.
En ce qui concerne l'échantillonnage des données pour l'expérimentation qui va suivre, nous utiliserons deux méthodes distinctes : la méthode split (66% des données pour l'apprentissage et 34% pour les tests) et la méthode cross validation (10 folds). Cette dernière technique divise notre échantillon de données en 10 sous-échantillons puis sélectionne un de ces sous-échantillons pour les tests tandis que les autres sont utilisés pour l'entrainement du modèle. Elle procède de même pour chaque sous-échantillon de notre échantillon initial.

SFH	popUpWidnow	SSLfinal_State	Request_URL	URL_of_Anchor	web_traffic	URL_Length	age_of_domain	having_IP_Address	Result
1	-1	1	-1	-1	1	1	1	0	**0**
-1	-1	-1	-1	-1	0	1	1	1	**1**
1	-1	0	0	-1	0	-1	1	0	**1**
1	0	1	-1	-1	0	1	1	0	**0**
-1	-1	1	-1	0	0	-1	1	0	**1**
-1	-1	1	-1	-1	1	0	-1	0	**1**
1	-1	0	1	-1	0	0	1	0	**-1**

FIGURE 3 – Extrait des sept premières instances du dataset

6 Application

Pour la réalisation des tests nous avons utilisé le package **DistributedWekaSparkDev** qui fournit des algorithmes de classification de MLlib la bibliothèque d'apprentissage automatique de Spark. Ce qui nous a permis de faire les tests en utilisant **Weka Explorer**.

Avec la méthode **split**, nous avons obtenus les résultats suivants :

- **taux de succès** c'est à dire instance correctement classée : 84.3478 %
- **taux d'erreur** : 15.6522 %
- **indice Kappa** (mesure de qualité) : 0.7197 qui peut être qualifié un bon kappa car compris dans l'intervalle 0.8-0.61
- **Courbe ROC** : aire couverte par la courbe égale à 0.8698 (proche de 1)

Ces résultats prouvent l'efficacité de l'algorithme utilisé sur notre dataset. Il arrive à prédire correctement presque tous les données de test et l'indice kappa obtenu témoigne de la bonne qualité de notre classifieur.
La courbe ROC montre également que notre modèle est bon car l'aire sous la courbe est proche de 1.

Avec la méthode **cross validation** nous avons les résultats suivants :

- **taux de succès** c'est à dire correctement classé : 85.2919 %
- **taux d'erreur** : 14.7081
- **indice Kappa** (mesure de qualité) : 0.7347 qui peut être qualifié un bon kappa car compris dans l'intervalle 0.8-0.61
- **Courbe ROC** : aire couverte par la courbe égale à 0.8728 (proche de 1).

Les résultats issus de la méthode cross validation montrent également des performances remarquables et rejoignent les résultats en termes d'efficacité.

7 Analyse comparative

Maintenant nous allons faire une analyse comparative des résultats avec d'autres algorithmes de classification dans Weka.
Nous allons appliquer les deux méthodes et relever les taux de succès des différents algorithmes.

	MLibDecisionTree	id3	J48
Méthode split	84.3478 %	86.7391 %	89.7826 %
Méthode cross validation	85.2919 %	87.5831 %	90.7613 %

FIGURE 4 – Tableau comparatif des algorithmes en fonction du taux de succès

Nous constatons que les différents algorithmes offrent tous de bons résultats en termes de taux de succès. Les résultats sont aussi comparables mais l'algorithme J48 (implémentation de C4.5 dans Weka) donne tout de même les meilleurs performances.

8 Résultats obtenus

Les résultats des tests sont illustrés sur les figures suivantes :

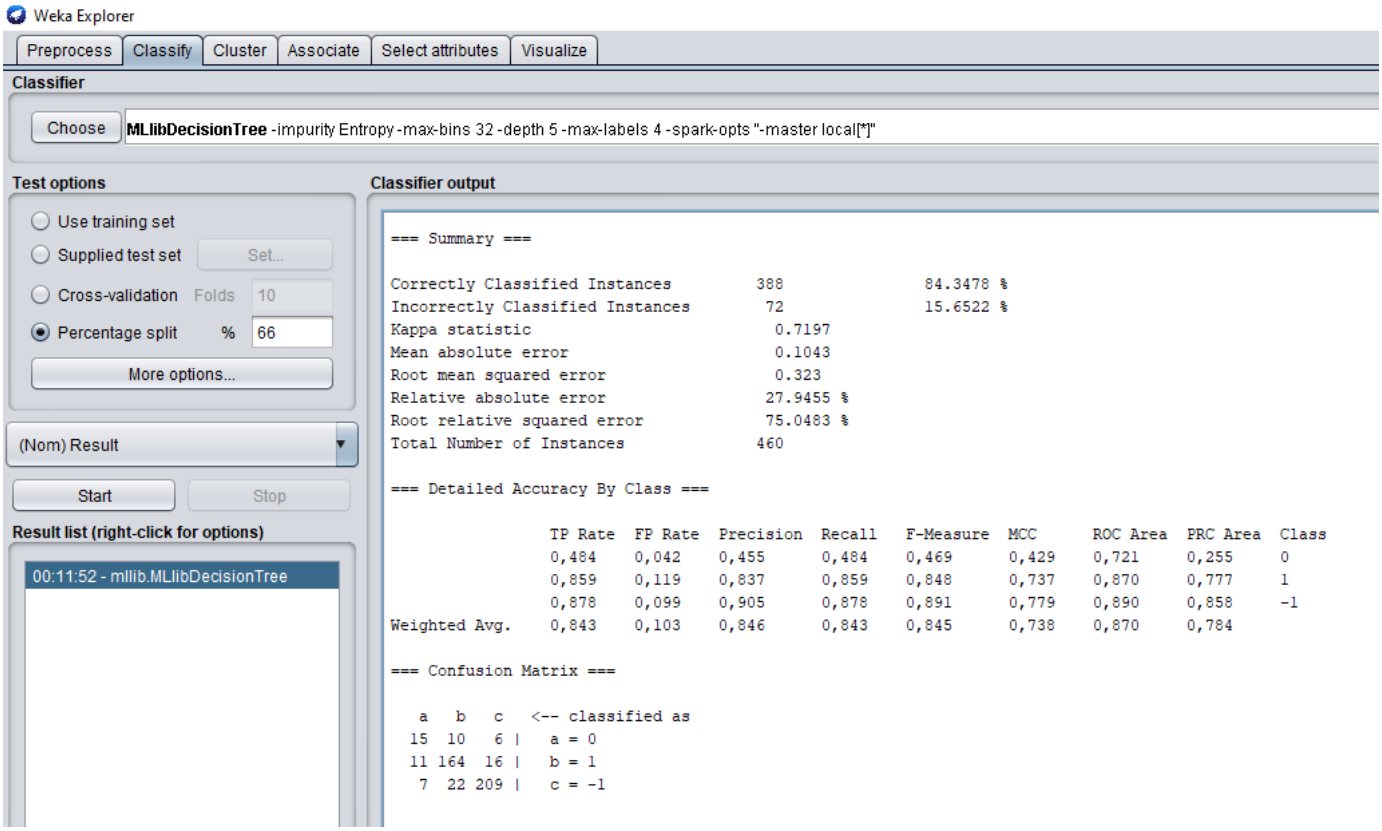

FIGURE 5 – MLlibDecisionTree - Méthode split

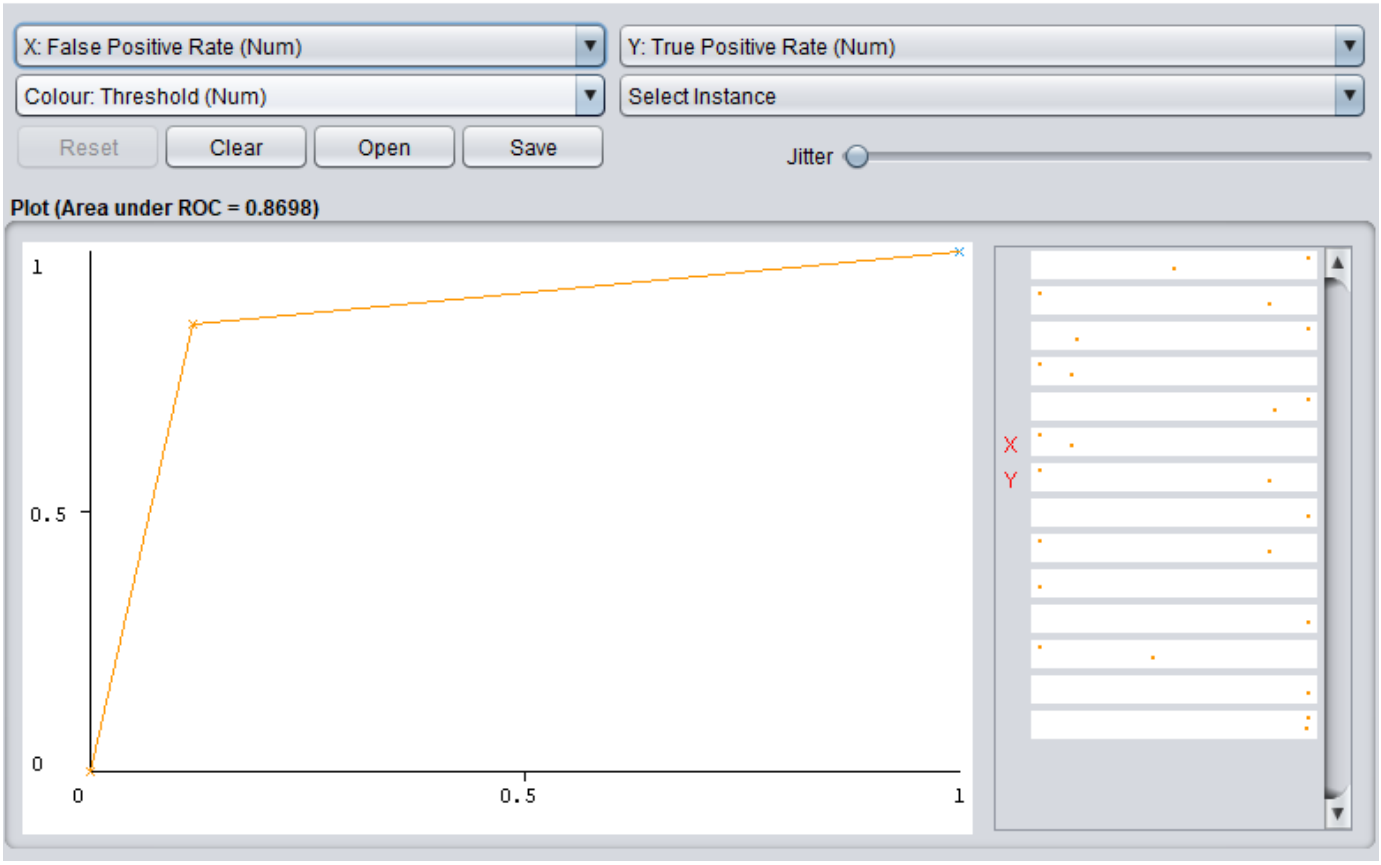

FIGURE 6 – ROC Curve - MLlibDecisionTree(split)

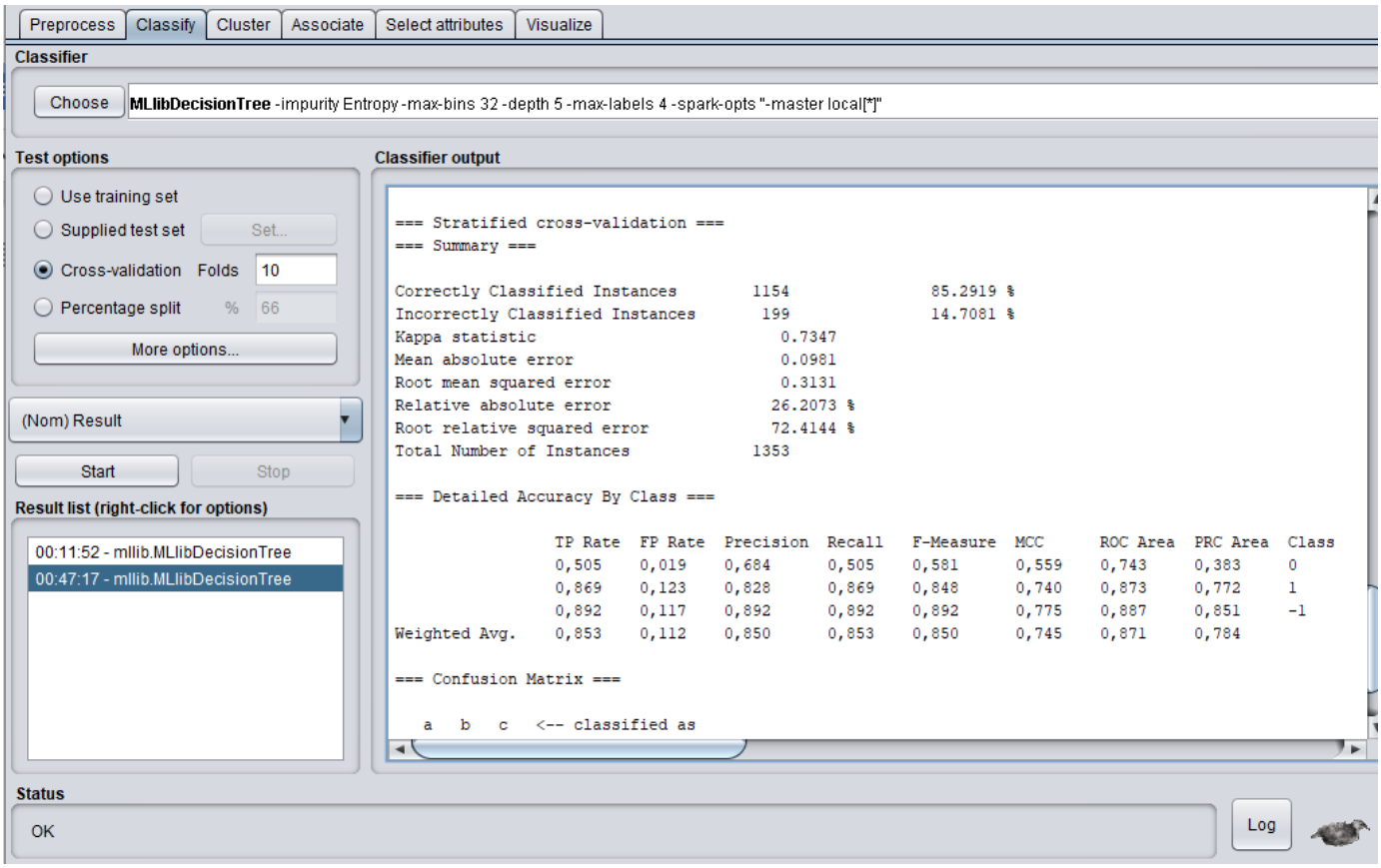

FIGURE 7 – MLlibDecisionTree - Méthode cross-validation

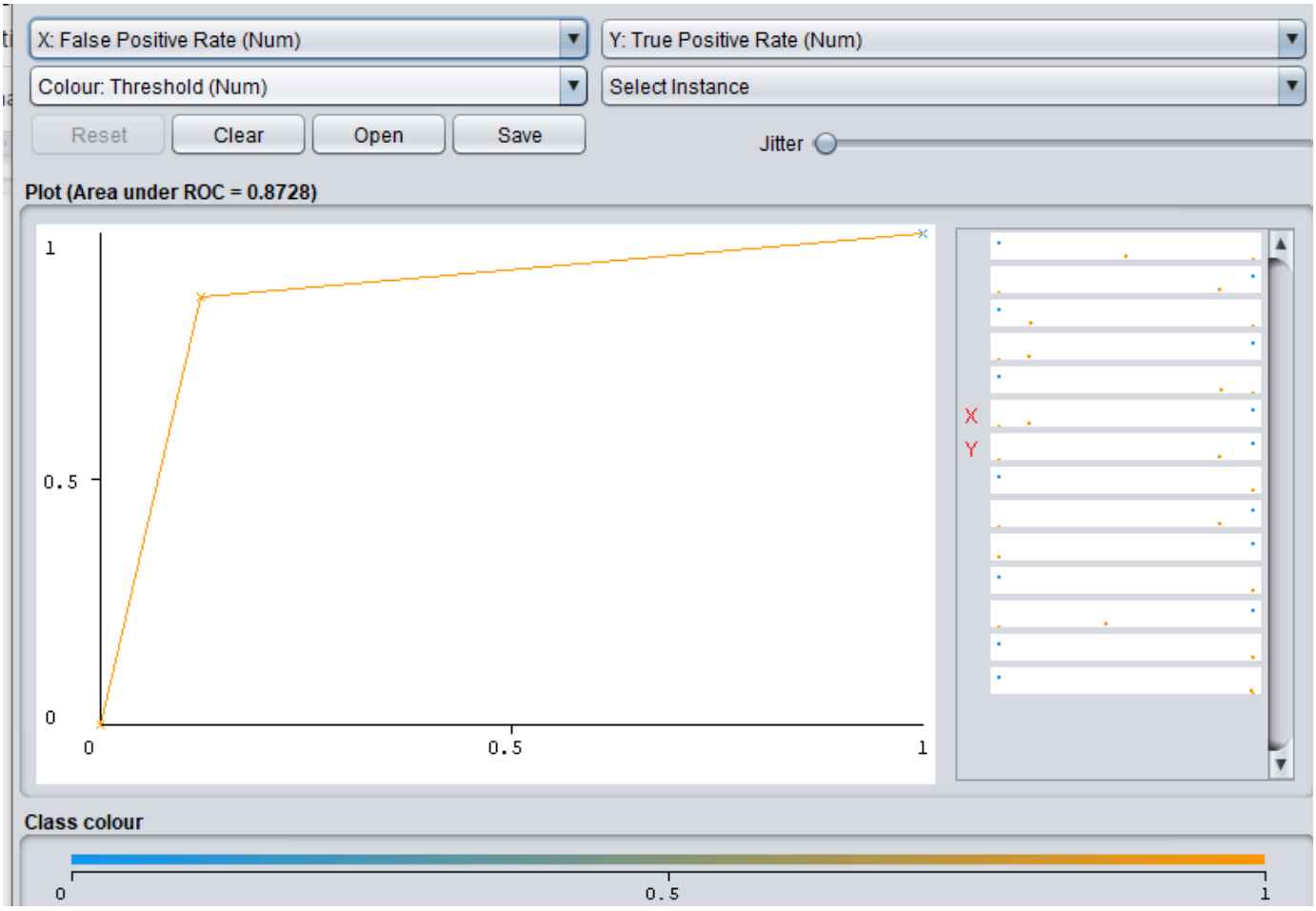

FIGURE 8 – ROC Curve - MLlibDecisionTree(cross-validation)

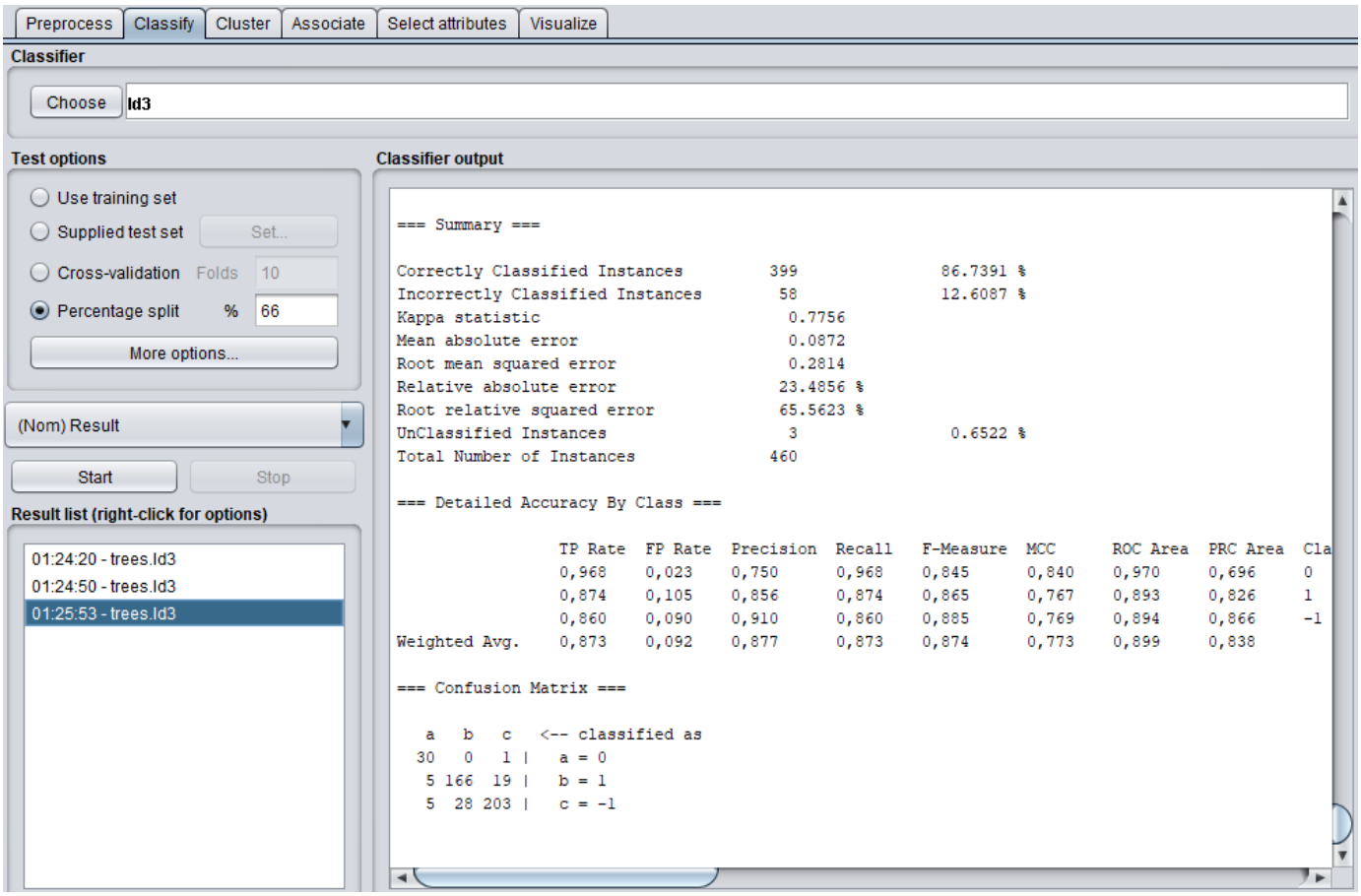

FIGURE 9 – ID3 - Méthode split

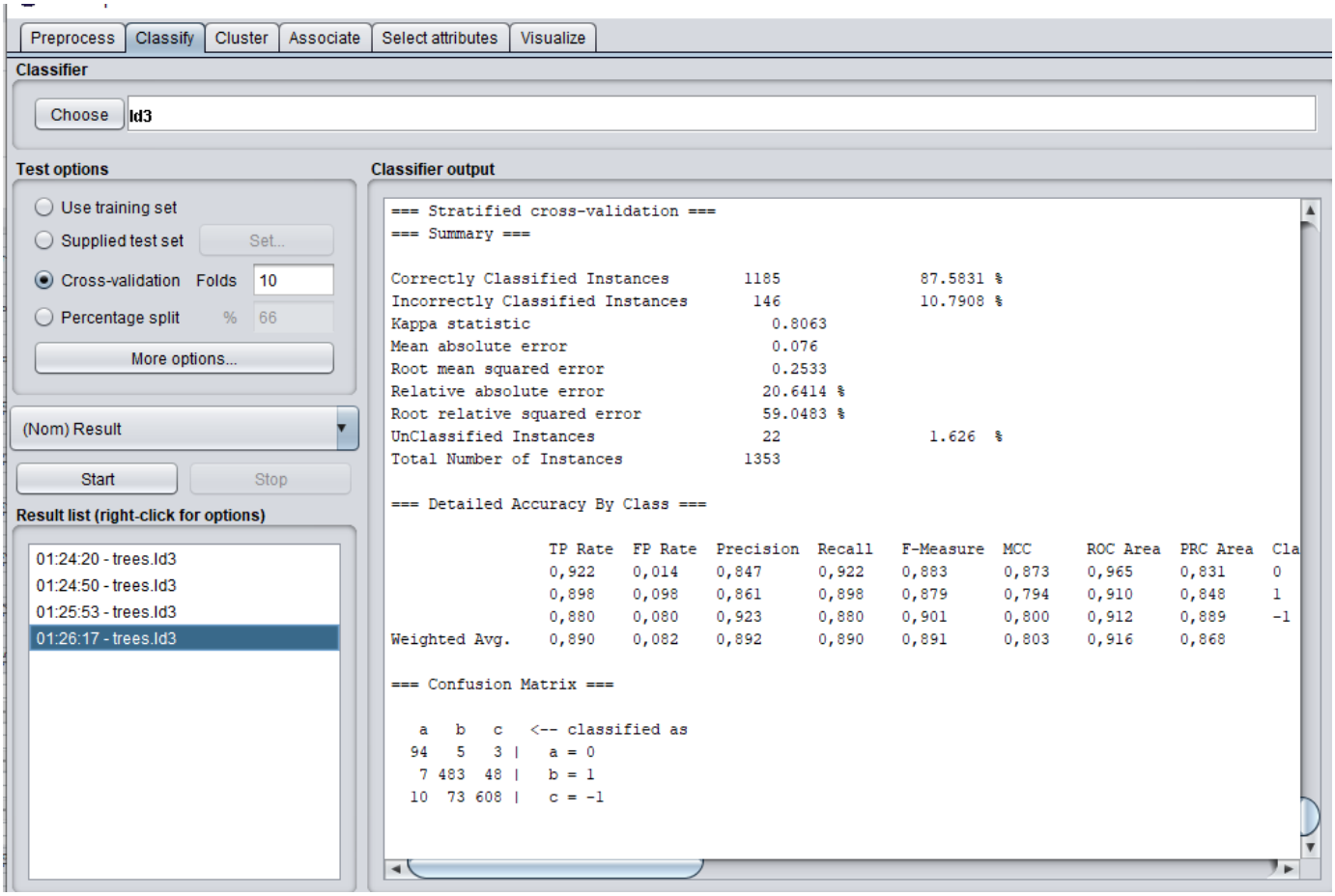

FIGURE 10 – ID3 - Méthode cross-validation

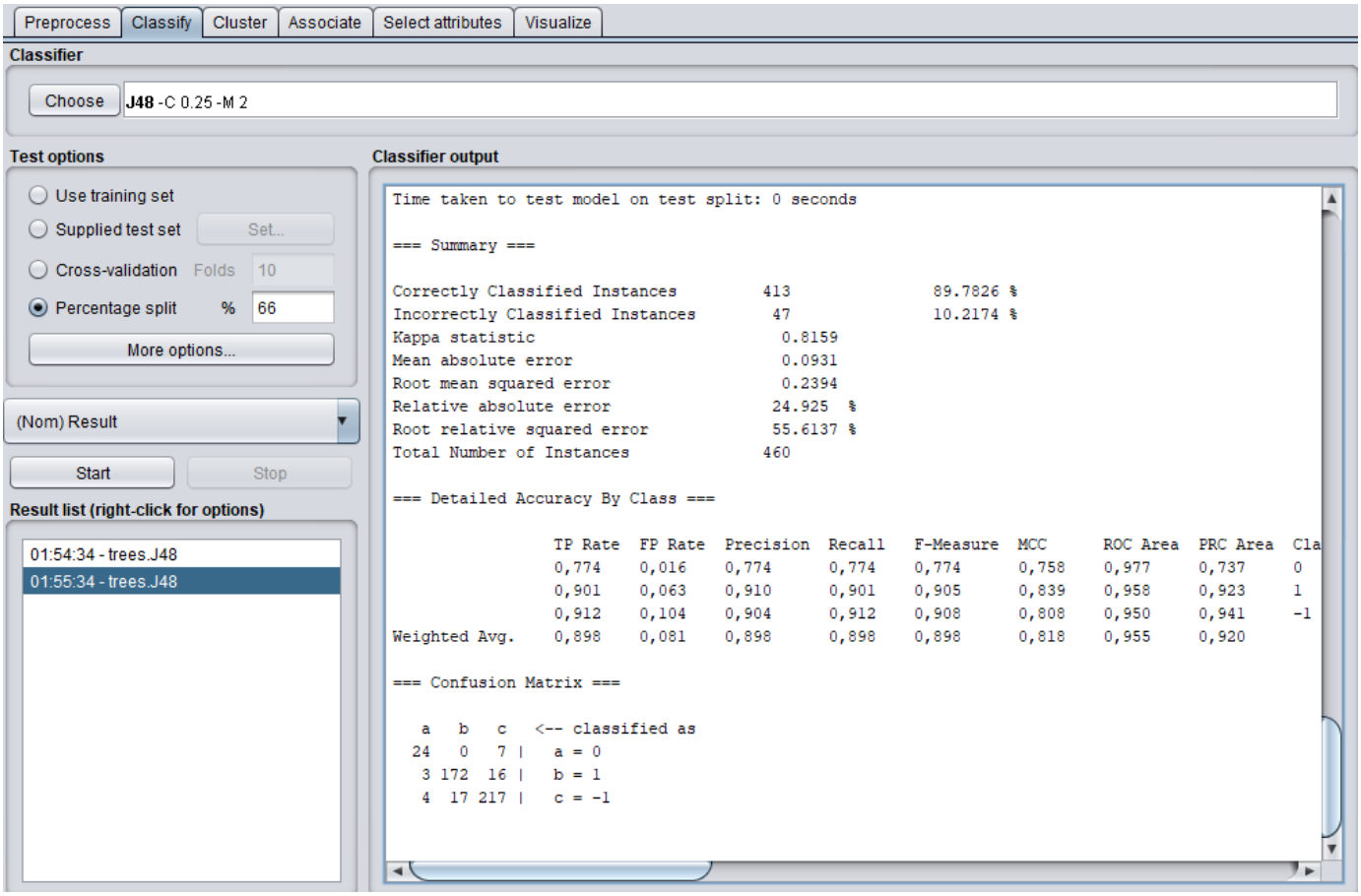

FIGURE 11 – J48 - Méthode split

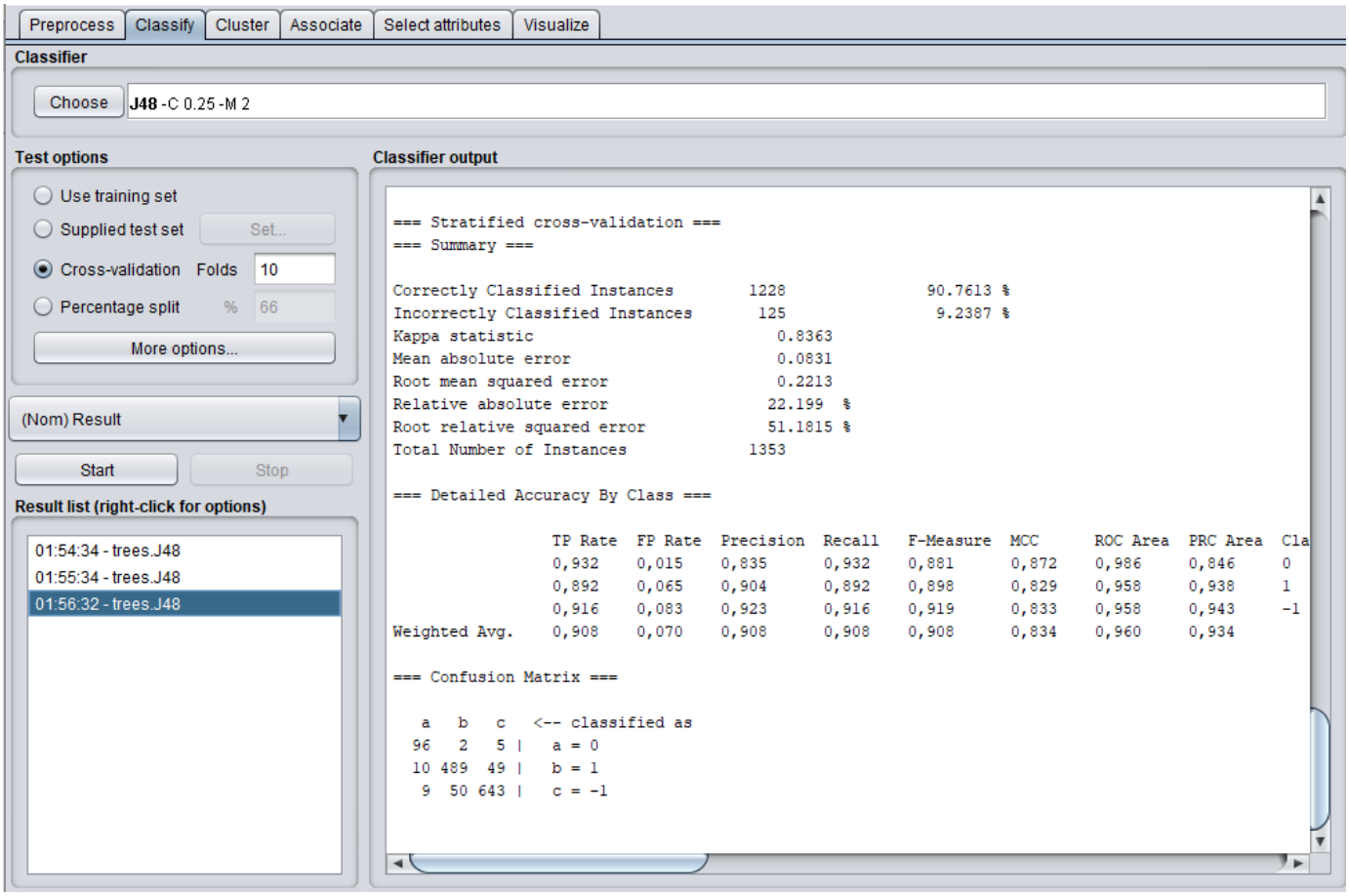

FIGURE 12 – J48 - Méthode cross-validation

9 Conclusion

A travers ce mini-projet, nous souhaitions montrer comment un algorithme pensé à l'origine de manière séquentielle pouvait être adapté pour utiliser le paradigme de programmation Map/Reduce et ainsi pouvoir s'effectuer de manière distribuée.
Cette approche permet en effet de traiter de plus grandes quantités de données en profitant de la scalabilité horizontale. Spark propose de nombreux algorithmes adaptés à ce paradigme grâce à sa librairie MLlib et nous souhaitions comparer un algorithme en particulier : MLlibDecisionTree avec d'autres algorithmes présents dans Weka : ID3 et J48.
L'intérêt de l'étude n'était pas de montrer la faculté de Spark à s'adapter à une montée en charge (de grandes quantités de données à analyser) car les deux autres algorithmes n'auraient pu être testés mais de comparer les performances entre ces algorithmes pour une quantité de données moyenne.
Nous avons pu voir que Spark présente de très bons résultats pour son algorithme d'arbre de décision même si ces derniers restent un peu en deça des résultats des deux autres algorithmes (notamment J48).

10 Référence

Mu Y, Liu X, Yang Z, Liu X. A parallel C4.5 decision tree algorithm based on MapReduce. Concurrency Computat : Pract Exper. 2017 ;29 :e4015. https ://doi.org/10.1002/cpe.4015

Koliopoulos, Aris-Kyriakos & Yiapanis, Paraskevas & Tekiner, Firat & Nenadic, Goran & Keane, John. (2015). A Parallel Distributed Weka Framework for Big Data Mining Using Spark. 10.1109/BigDataCongress.2015.12.

Schéma Map/Reduce : Par Clém IAGL — Travail personnel, CC BY-SA 3.0, https ://commons.wikimedia.org/w/index.php ?curid=22688163