Études des principaux algorithmes de *data mining*

Guillaume CALAS
guillaume.calas@gmail.com

Spécialisation *Sciences Cognitives et Informatique Avancée*

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

14-16 rue Voltaire,
94270 Le Kremlin-Bicêtre,
France

Mots clés: *data mining*, arbres de décision, *clustering*, règles associatives, *sequence mining*, classification.

Table des matières

1 Introduction

Le *Data mining* est un domaine pluridisciplinaire permettant, à partir d'une très importante quantité de données brutes, d'en extraire de façon automatique ou semi-automatique des informations cachées, pertinentes et inconnues auparavant en vue d'une utilisation industrielle ou opérationnelle de ce savoir. Il peut également mettre en avant les associations et les tendances et donc servir d'outil de prévisions au service de l'organe décisionnel.

On distingue le *data mining* supervisé qui sert essentiellement à la classification des données et le *data mining* non supervisé qui est utilisé dans la recherche d'associations ou de groupes d'individus.

Le champ d'action du *data mining* s'étend du chargement et du nettoyage des données (ETL[1]) dans les bases de données, à la mise en forme des résultats, en passant par le plus important : la classification et la mise en relation de différentes données.

1.1 Apprentissage supervisé

En sciences cognitives, l'**apprentissage supervisé** est une technique d'apprentissage automatique — plus connu sous le terme anglais de *machine-learning* — qui permet à une machine d'apprendre à réaliser des tâches à partir d'une base d'apprentissage contenant des exemples déjà traités. Chaque élément (*item*) de l'ensemble d'apprentissage (*training set*) étant un couple entrée-sortie.

De part sa nature, l'apprentissage supervisé concerne essentiellement les méthodes de *classification* de données (on connaît l'entrée et l'on veut déterminer la sortie) et de *régression* (on connaît la sortie et l'on veut retrouver l'entrée).

1.1.1 Les arbres de décision

Un **arbre de décision** est, comme son nom le suggère, un outil d'aide à la décision qui permet de répartir une population d'individus en groupes homogènes selon des attributs discriminants en fonction d'un objectif fixés et connu. Il permet d'émettre des prédictions à partir des données connues sur le problème par réduction, niveau par niveau, du domaine des solutions.

Chaque nœud interne d'un arbre de décision porte sur un attribut discriminant des éléments à classifier qui permet de répartir ces éléments de façon homogène entre les différents fils de ce nœud. Les branches liant un nœud à ses fils représentent les valeurs discriminantes de l'attribut du nœud. Et enfin, les feuilles d'un arbre de décision sont ses prédictions concernant les données à classifier.

C'est une méthode qui a l'avantage d'être *lisible* pour les analystes et *permet de déterminer les couples <attribut, valeur> discriminants* à partir d'un très grand nombre d'attributs et de valeurs.

Les algorithmes d'apprentissage automatique que nous étudierons à la section 2.1 permettent l'induction de ces arbres de décision.

1.1.2 Les réseaux de neurones

Un **réseau de neurones** est un modèle de calcul dont le fonctionnement schématique est inspiré du fonctionnement des neurones biologique. Chaque neurone fait une somme pondérée de ses entrées (ou synapses) et retourne une valeur en fonction de sa *fonction d'activation*[2]. Cette valeur peut être utilisée soit comme une des entrées d'une nouvelle couche de neurones, soit comme un résultat qu'il appartient à l'utilisateur d'interpréter (classe, résultat d'un calcul, etc.).

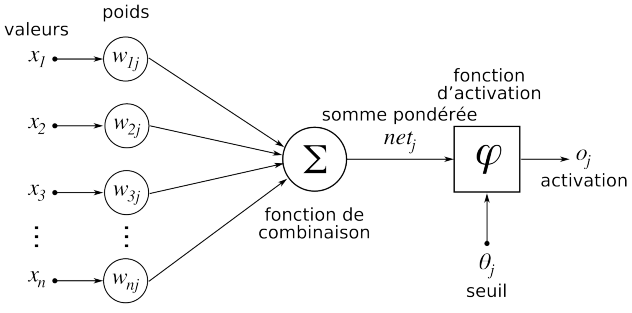

FIG. 1 – Structure d'un neurone artificiel.
Par Chrislb sous licence *Creative Common Attribution ShareAlike 2.5.*

La phase d'apprentissage d'un réseau de neurones

[1] *Extract, Transform and Load*

[2] Une fonction d'activation permet de déterminer l'état d'activation d'un neurone formel.

permet de régler le poids associé à chaque synapse d'entrée (on parle également de *coefficient synaptique*). C'est un processus long qui doit être réitéré à chaque modification structurelle de la base de données traitée.

1.2 Apprentissage non supervisé

On parle d'**apprentissage non supervisé** lorsque l'on cherche à extraire des informations nouvelles et originales d'un ensemble de données dont aucun attribut n'est plus important qu'un autre.

Le résultat des algorithmes de *data mining* non supervisé doit être analysé afin d'être retenu pour un usage ou tout simplement rejeté.

1.2.1 *Clustering*

Le ***clustering*** est une méthode statistique d'analyse de données qui a pour but de regrouper un ensemble de données en différents groupes homogènes. Chaque sous-ensemble regroupe des éléments ayant des caractéristiques communes qui correspondent à des critères de proximité.

Le but des algorithmes de *clustering* est donc de minimiser la distance intra-classe (grappes d'éléments homogènes) et de maximiser la distance inter-classe afin d'obtenir des sous-ensembles le plus distincts possible.

La mesure des distances est un élément prépondérant pour la qualité de l'algorithme de *clustering*.

Cette classe d'algorithmes ne sera pas traitée dans le présent document.

1.2.2 Les règles associatives

Les **règles associatives** sont des règles qui sont extraites d'une base de données transactionnelles (*itemset*) et qui décrivent des associations entre certains éléments. Cette technique permet de faire ressortir les associations entre les produits de base (les produits essentiels, ceux pour lesquels le client se déplace) et les produits complémentaires, ce qui permet de mettre en place des stratégies commerciales visant à accroître les profits en favorisant, par exemple, les ventes complémentaires. Ces algorithmes permettent de résoudre des problèmes dits de *Frequent Set Counting* (FSC).

1.2.3 *Sequence mining*

Le ***sequence mining*** concerne la détection de motifs dans les flux de données dont les valeurs sont délivrées par séquences.

Cette technique est particulièrement utilisé en biologie pour l'analyse de gènes et des protéines mais également afin de faire du *text mining*, les phrases étant considérées comme des séquences ordonnées de mots.

1.3 Apprentissage incrémental

L'**apprentissage incrémental** permet à une machine d'apprendre par ajout successif d'informations. Pour être considéré comme tel, un système d'apprentissage doit :

- être capable d'apprentre de nouvelles informations à partir de nouvelles données ;
- être capable de se passer des données d'origine pour entraîner le nouveau classifieur ;
- préserver le savoir précédemment acquis ;
- être capable de reconnaître de nouvelles classes indroduites dans les nouvelles données.

2 Présentation des algorithmes

2.1 Induction d'arbres de décision

2.1.1 CART

Référence : [4]
Date de publication : 1984
Auteur : Breiman
Incrémental : non
Description : CART (*Classification And Regression Tree*) construit un arbre de décision strictement binaire avec exactement deux branches pour chaque nœud de décision. L'algorithme partitionne l'ensemble d'entraînement de façon récursive selon la méthode *diviser pour mieux régner*[3].

[3] Du latin *Divide ut imperes*, méthode de conception consistant à diviser un problème de grande taille en plusieurs sous-problèmes analogues, l'étape de subdivision

Pour chaque nœud de décision, CART fait une recherche exhaustive sur tous les attributs et valeurs de séparation disponibles et sélectionne la séparation s qui maximise le critère suivant au nœud t :

$$\phi(s|t) = 2P_L P_R \sum_{j=1}^{Card(Classes)} |P(j|t_L) - P(j|t_R)| \quad (1)$$

où

$$\begin{aligned} S &= \text{ensemble d'entraînement} \\ t_L &= \text{fils gauche du nœud } t \\ t_R &= \text{fils droit du nœud } t \\ P_L &= \frac{Card(t_L)}{Card(S)} \\ P_R &= \frac{Card(t_R)}{Card(S)} \\ P(j|t_L) &= \frac{Card(\text{classe j} \in t_L)}{Card(t)} \\ P(j|t_R) &= \frac{Card(\text{classe j} \in t_R)}{Card(t)} \end{aligned}$$

Dans le cas où l'attribut testé porte sur des valeurs continues, CART peut identifier un ensemble fini de valeurs possibles. Cependant, il est préférable que l'utilisateur discrétise les variables continues afin de mieux contrôler la génération de l'arbre et de le rendre plus robuste.

L'algorithme s'arrête lorsque l'un des cas d'arrêt suivant est rencontré :

- le nœud est pur, ie. tous les éléments du nœud appartiennent à la même classe ;
- tous les attributs ont été utilisés précédemment ;
- la profondeur de l'arbre à atteint la valeur maximale définie par l'utilisateur ;
- la taille du nœud est inférieure à la taille minimale définie par l'utilisateur ;
- la taille d'un des fils qui résulterait de la séparation optimale est inférieure à la taille minimale définie par l'utilisateur.

CART n'est pas adapté pour travailler sur de très grandes quantités d'enregistrements (à cause de l'exhaustivité de ses recherches de valeurs de séparation optimales) et bien qu'ayant des performances honorables, il est sujet au sur-apprentissage. Le post-élagage de l'arbre de décision généré est donc fortement conseillée.

étant appliquée récursivement

2.1.2 ID3

Référence : [11]
Date de publication : 1986
Auteur : Quinlan
Incrémental : par extension (cf. ID4[13], ID5 et ID5R[16])
Algorithme : figure 1 page 10
Description : ID3 construit un arbre de décision de façon récursive en choisissant l'attribut qui maxime le gain (3) d'information selon l'entropie de Shannon (2). Cet algorithme fonctionne exclusivement avec des attributs catégoriques et un nœud est créé pour chaque valeur des attributs sélectionnés.

ID3 est un algorithme basique facile à implémenter dont la première fonction est de remplacer les experts dans la construction d'un arbre de décision. Cependant, les arbres de décisions ne sont ni robustes, ni compacts ce qui les rends inadaptés aux grosses bases de données.

Entropie de Shannon :

$$E(S) = -\sum_{j=1}^{|S|} p(j) \log_2 p(j) \quad (2)$$

où $p(j)$ est la probabilité d'avoir un élément de caractéristique j dans l'ensemble S.

$$Gain(S,A) = E(S) - \sum^{v} \left(\frac{|S_v|}{|S|} * E(S_v)\right) \quad (3)$$

où S est un ensemble d'entraînement, A est l'attribut cible, S_v le sous-ensemble des éléments dont la valeur de l'attribut A est v, $|S_v|$ = nombre d'éléments de S_v et $|S|$ = nombre d'éléments de S.

2.1.3 C4.5

Référence : [12]
Date de publication : 1993
Auteur : Quinlan
Incrémental : non
Description : C4.5 est une amélioration d'ID3 qui permet de travailler à la fois avec des données discrètes

et des données continues. Il permet également de travailler avec des valeurs d'attribut absentes.

Enfin, C4.5 élague l'arbre construit afin de supprimer les règles inutiles et de rendre l'arbre plus compact. L'algorithme C5, qui est une solution commerciale, est une amélioration supplémentaire de C4.5.

2.1.4 OC1

Référence : [8]
Date de publication : 1993
Auteurs : Murphy, Kasif, Salzberg, Beigel
Incrémental : non
Algorithme : figures 5 et 6 page 12
Description : OC1 (*Oblique Classifier 1*) est une extension de CART-LC (CART avec des Combinaisons Linéaires). Mais contrairement à CART, d'une part, OC1 utilise le hasard pour trouver le meilleur hyperplan de séparation de l'ensemble d'entraînement, et d'autre part, l'hyperplan de séparation peut ne pas être parallèle aux axes. De ce fait, l'algorithme peut générer une multitude d'arbres de décision ce qui peut permettre de réaliser un *k-decision-tree* classifieur dont le résultat de la classification résulte de la majorité des votes des k arbres de décision.

Afin de ne pas perdre de performance par rapport à un classifieur classique à arbre de décision, OC1 utilise une séparation oblique uniquement quand cela améliore les performances par rapport à la meilleure séparation parallèle à un axe.

Pour trouver le meilleur hyperplan l'algorithme travaille attribut par attribut. Il considère (d-1) attributs comme des constantes et va émettre n (le nombre d'exemples) contraintes sur l'attribut restant et déterminer la valeur qui en respecte le plus. Pour cela l'algorithme utilise les équations suivantes :

$$\frac{a_m x_{jm} - V_j}{x_{jm}} = U_j \tag{4}$$

où x_{jm} est la valeur de l'attribut m de l'exemple j, a_m est le coefficient réel de l'attribut m de l'hyperplan recherché et V_j est définit par :

$$V_j = \sum_{i=1}^{d} (a_i x_{ji}) + a_{d+1} \tag{5}$$

Pour mesurer l'impureté de la séparation, OC1 peut utiliser n'importe quelle mesure d'impureté mais le Gain (3), Gini Index (7), et Twoing Rule (6) obtiennent les meilleures performances.

$$T_{Value} = \left(\frac{|T_L|}{n}\right) * \left(\frac{|T_R|}{n}\right) * \left(\sum_{i=1}^{k} \left| \frac{L_i}{|T_L|} - \frac{R_i}{|T_R|} \right|\right)^2 \tag{6}$$

Enfin, en ce qui concerne l'élagage de l'arbre de décision obtenu, n'importe quel algorithme prévu à cet effet est utilisable et obtient de bons résultats.

2.1.5 SLIQ

Référence : [1]
Date de publication : 1996
Auteurs : Metha, Agrawal, Rissanen
Incrémental : oui, par extension
Algorithme : figures 2,3 et 4 page 10
Description : SLIQ (*Supervised Learning In Quest*) est un algorithme performant capable de traiter des attributs numériques et catégoriques. L'algorithme construit un arbre binaire de décision de façon récursive en utilisant le coefficient de Gini (7) comme critère pour la sélection du meilleur attribut et de la valeur associée.

Coefficient de Gini :

$$gini(S) = 1 - \sum_{j=1}^{m} p_c^2 \tag{7}$$

où p_c est la fréquence relative de la classe c dans l'ensemble S contenant m classes. Si S est pure, $gini(S) = 0$.

Le coefficient de Gini pour une partition de S en deux ensembles S_T et S_F selon le critère de séparation t est défini par la relation :

$$gini(S,t)_{split} = \frac{|S_T|}{|S|} gini(S_T) + \frac{|S_F|}{|S|} gini(S_F) \tag{8}$$

Afin de gagner en performances, l'algorithme tri les attributs sur leurs valeurs (quand c'est possible) et utilise une liste triée par attribut associée à un histogramme pour chaque nœud ce qui lui permet de calculer très rapidement le coefficient de Gini, cependant, ces listes doivent être stockés dans la mémoire vive et leur taille dépend directement du nombre d'éléments

dans la base de données ce qui peut constituer une limitation physique.

Pour les attributs catégoriques à large cardinalité on utilise un algorithme glouton pour trouver la meilleure séparation.

Enfin, SLIQ utilise un algorithme basé sur le principe de la Longueur Minimale de Description (*Minimum Description Length*, MDL) pour faire l'élagage final.

2.1.6 SPRINT

Référence : [14]
Date de publication : 1996
Auteurs : Shafer, Mehta, Agrawal
Incrémental : non
Description : SPRINT est un algorithme basé que SLIQ qui a été conçu afin de remédier au principal problème de SLIQ : son utilisation excessive de la mémoire. Il peut également être facilement sérialisé afin d'augmenter ses performances.

La différence entre les deux algorithmes réside dans le fait que SPRINT ne conserve en mémoire que la partie utilisée des listes triées alors que SLIQ conserve la liste entière en mémoire.

2.2 Les réseaux de neurones

2.2.1 AdaBoost

Référence : [6]
Date de publication : 1997
Auteurs : Freund, Schapire
Incrémental : oui
Algorithme : figure 7 page 13
Description : *AdaBoost* est un méta-algorithme qui permet de booster les performances d'un classifieur à base de réseau de neurones. L'algorithme regroupe des hypothèses faibles émises par un classiffieur faible entraînés sur un sous-ensemble d'entraînement dont la distribution des éléments est renforcée, itération après itération, afin que l'apprentissage puisse être concentré sur les exemples qui posent le plus de difficultés au classifieur ainsi entraîné.

La distribution est modifiée selon l'équation suivante :

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & si \quad h_t(x_i) = y_i \\ e^{\alpha_t} & si \quad h_t \neq y_i \end{cases} \tag{9}$$

À la fin des T itérations, les T hypothèses faibles — pondérées en fonction de leurs performances — sont combinées afin de donner naissance à une hypothèse finale

$$H(x) : X \rightarrow Y$$

AdaBoost est rapide et simple à implémenter, il permet d'avoir de bons résultats et d'améliorer les performances d'autres classifieurs.

2.2.2 Learn++

Référence : [10]
Date de publication : 2001
Auteurs : Polikar, Udpa, S. Udpa, Honovar
Incrémental : oui
Algorithme : figure 8 page 14
Description : *Learn++* s'inspire de l'algorithme *AdaBoost*. Il génère un certain nombre de classifieurs faibles à partir d'un ensemble de données dont on connaît le label. En fonction des erreurs du classifieur faible généré, l'algorithme modifie la distribution des éléments dans le sous-ensemble suivant afin de renforcer la présence des éléments les plus difficile à classifier. Cette procédure est ensuite répétée avec un ensemble différent de données du même *dataset* et des nouveaux classifieurs sont générés. En combinant leurs sorties selon le schéma de majorité de votes de Littlestone on obtient la règle finale de classification.

Les classifieurs faibles sont des classifieurs qui fournissent une estimation grossière — de l'ordre de 50% ou plus de bonne classification — d'une règle de décision car ils doivent être très rapide à générer. Un classifieur fort passant la majorité de son temps d'entraînement à affiner ses critères de décision. La recherche d'un classifieur faible n'est pas un problème trivial et la complexité de la tâche croît avec le nombre de classes différentes, cependant, l'utilisation d'algorithmes NN correctement réglés permet efficacement de contourner le problème.

L'erreur est calculée selon l'équation :

$$error_t = \sum_{i:h_t(x_i)\neq y_i} S_t(i) = \sum_{i=1}^{m} S_t(i)[|h_t(x_i) \neq y_i|] \quad (10)$$

avec $h_t : X \rightarrow Y$ une hypothèse et $S_t = TR_t \cup TE_t$ où TR_t est le sous-ensemble d'entraînement et TE_t le sous-ensemble de test.

Les coefficients synaptiques sont mis à jour selon l'équation suivante :

$$w_{t+1}(i) = w_t(i) \times \begin{cases} B_t & si \quad H_t(x_i) = y_i \\ 1 & sinon \end{cases} \quad (11)$$

où t est le numéro d'itération, B_t l'erreur composite normalisée et H_t l'hypothèse composite.

Contrairement à *AdaBoost* qui est orienté vers l'amélioration des performances tout en permettant — de part sa structure — l'apprentissage incrémental, *Learn++* est essentiellement orienté sur l'apprentissage incrémental.

2.3 Induction de règles associatives

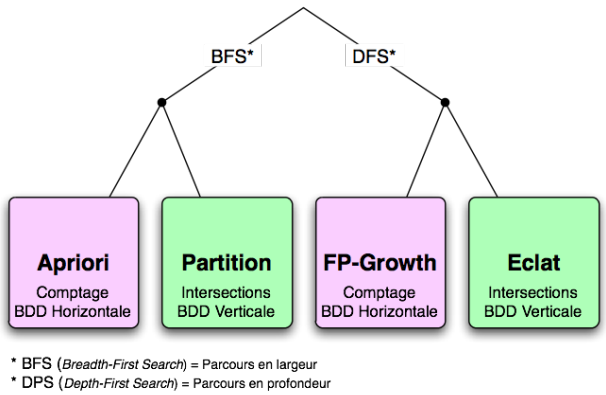

FIG. 2 – Algorithmes de *Frequent Set Counting* (FSC).

Les algorithmes suivants utilisent les notions de *support* et de *confidence* pour déterminer la pertinence des associations.

Le support d'une règle est défini par :

$$Support(A) = \frac{count(A)}{|T|} \quad (12)$$

où A est *item* (un produit) et T une base de données transactionnelle dont est issu A.

La confidence d'une règle est définie par :

$$Confidence(A \Rightarrow B) = \frac{Support(A \Rightarrow B)}{Support(A)} \quad (13)$$

Les règles associatives générées sont retenues si elles ont un support supérieur à *minSupp* et une confidence supérieure à $minConf$. Ces deux constantes — dépendantes de la base de données — sont définies par l'utilisateur du système de façon empirique.

2.3.1 Apriori

Référence : [2]
Date de publication : 1993
Auteurs : Agrawall, Imielinski, Swami
Algorithme : figure 9 page 15
Description : *Apriori* est un algorithme classique de recherche de règles d'association. Comme tous les algorithmes de découvertes d'associations, il travaille sur des bases de données transactionnelles (des enregistrements de transactions).

Pour révéler la pertinence d'une règle on utilise deux concepts qui sont le support (12) et la confidence (13). Afin d'être retenue, chaque règle devra avoir un support supérieur à *minSupp* et une confidence supérieure à $minConf$. Ces deux valeurs étant définies empiriquement par l'utilisateur du système.

L'algorithme démarre avec la liste des produits les plus fréquents dans la base de données respectant les seuils de support. Un ensemble de règles (des candidats) est généré à partir de cette liste. Les candidats sont testés sur la base de données (ie. on recherche les instances des règles générées et leurs occurrences) et les candidats ne respectant pas *minSupp* et $minConf$ sont retirées. L'algorithme réitère ce processus en augmentant à chaque fois la dimension des candidats d'une unité tant que des règles pertinentes sont découvertes. A la fin, les ensembles de règles découvertes sont fusionnés.

La génération des candidats se fait en deux étapes : 1) la jointure ; 2) l'élagage. La jointure consiste en une jointure d'un ensemble de règles à k-1 éléments sur lui même qui aboutit à la génération d'un ensemble de candidats à k éléments. Enfin, l'élagage supprime

les candidats dont au moins une des sous-chaîne à k-1 éléments n'est pas présente dans l'ensemble des règles à k-1 éléments.

Cet algorithme est très performant, mais souffre si les ensembles d'éléments fréquents sont trop grands. De plus, scanner la base données à la recherche d'un motif de façon répété devient rapidement un frein aux performances sur de grosses bases de données.

2.3.2 FP-Growth

Référence : [7]
Date de publication : 2000
Auteurs : Han, Pei, Yin, Mao
Description : *FP-Growth* (*Frequent-Pattern Growth*) est un algorithme complètement innovant par rapport aux autres algorithmes de recherche de règles associatives presque tous basés sur *Apriori*.

L'algorithme utilise une structure de données compacte appelé *Frequent-Pattern tree* et qui apporte une solution au problème de la fouille de motifs fréquents dans une grande base de données transactionnelle. En stockant l'ensemble des éléments fréquents de la base de transactions dans une structure compacte, on supprimer la nécessiter de devoir scanner de façon répétée la base de transactions. De plus, en triant les éléments dans la structure compacte, on accélère la recherche des motifs.

Un *FP-tree* est composé d'une racine nulle et d'un ensemble de nœuds préfixé par l'élément représenté. Un nœud est composé par : le nom de l'élément, le nombre d'occurrence de transaction où figure la portion de chemin jusqu'à ce nœud, un lien inter-nœud vers les autres occurrences du même élément figurant dans d'autres séquences de transactions. Une table d'en-tête pointe sur la première occurrence de chaque élément.

L'avantage de cette représentation des données est qu'il suffit de suivre les liens inter-nœuds pour connaître toutes les associations fréquentes où figure l'élément fréquent.

2.3.3 Eclat

Référence : [17]
Date de publication : 2000
Auteur : Zaki
Algorithme : figures 10 et 11 page 15
Description : *Eclat* est le fondateur de la seconde grande famille d'algorithmes de recherche de règles d'associations avec l'algorithme *Apriori*. L'algorithme démarre avec la liste des éléments fréquents puis l'algorithme est réitéré en rajoutant les ensembles fréquents à l'ensemble des candidats C jusqu'à ce que cet ensemble soit vide.

L'ensemble des transactions de D qui contiennent l'élément X est définit par :

$$D(X) = \{T \in D | X \subseteq T\} \tag{14}$$

Les relations sont filtrées selon l'équation suivante :

$$freq(C_0) = \{(x, D_x) | (x, D_x) \in C_0, |D_x| \geq minsup\} \tag{15}$$

2.3.4 SSDM

Référence : [5]
Date de publication : 2005
Auteurs : Escovar, Biajiz, Vieira
Algorithme : figures 9 et 12 pages 15 et 16
Description : SSDM (*Semantically Similar Data Miner*) est un algorithme de la famille de l'algorithme *Apriori* auquel est rajouté des notions d'ensembles flous afin de renforcer la recherche d'associations sur des bases plus robustes.

Comme le nom de l'algorithme l'indique, SSDM s'appuie sur les similarité sémantique des éléments pour effectuer ses recherches. L'algorithme utilise des matrices de similarité — remplie par l'utilisateur du système — pour chaque domaine de produits afin d'effectuer ses recherches. C'est le principal inconvénient de cette méthode basée sur des critères humains, des études sont aujourd'hui menées afin de trouver des méthodes pour remplir ces matrices de façon automatique. L'algorithme nécessite de fixer une valeur minimale de similarité *minSim* qui détermine si l'algorithme doit confondre deux éléments — d'un même domaine — en une seule entité.

L'algorithme recherche les cycles de similarité entre les éléments et ces cycles sont utilisés lors de la génération des candidats (identique à *Apriori*. On se retrouve alors avec un ensemble de candidats purs

contenant éventuellement des candidats flous. Le cas échéant, le support du candidat flou est évalué selon le poids de ses similarités par la formule suivante :

$$poids_C = \frac{[poids(a) + poids(b)][1 + sim(a,b)]}{2} \quad (16)$$

où $poids_C$ est le poids du candidat C contenant les éléments a et b et où $poids(a)$ correspond au nombre d'occurrence de a dans la base de données transactionnelles.

La formule générale pour n éléments est donnée par l'équation suivante :

$$poids = \left[\sum_{i=1}^{+\infty} poids\,(Item_i)\right]\left[\frac{1+f}{2}\right] \quad (17)$$

où f est le plus petit degré de similarité contenu dans les associations considérées.

La génération des règles est également identique à celle de l'*Apriori*.

2.3.5 kDCI

Référence : [9]
Date de publication : 2003
Auteurs : Orlando, Lucchese, Palmerini, Silvestri
Algorithme : figures 10 et 11 page 15
Description : kDCI (*k Direct Count & Intersect*) est un algorithme hybride multi-stratégies basé sur l'algorithme DCI. L'algorithme recueille des informations sur la base de données durant l'exécution afin de déterminer la meilleure stratégie à adopter selon que les données sont denses ou éparses.

kDCI est optimisé pour les grandes bases de données et utilise des structures de données compactes ainsi qu'une représentation des données minimale en mémoire. Lorsque la base de données élaguée peut tenir en mémoire, DCI construit une *tidlist* d'intersections à la volée. Ce vecteur binaire à deux dimensions comprend les identifiants des éléments en ordonnée et ceux des transactions en abscisse, la présence d'un élément dans une transaction étant marqué d'un 1 et l'absence d'un 0.

La technique utilisée pour les bases de données éparses se résume à une projection suivie d'un élagage (uniquement si le gain de ce dernier est supérieur à son coût).

Pour les bases de données denses où le nombre d'intersection est important, l'heuristique de DCI est utilisé. Il consiste en une réorganisation de la base de données verticale en réordonnant les colonnes en plaçant les segments identiques associés aux éléments les plus fréquents aux premières positions, ce qui permet de réduire le nombre d'intersections en évitant la répétition de segments identiques.

2.4 *Sequence mining*

Afin de facilité l'explication des algorithmes étudiés ci-dessus, nous utiliserons le formalisme suivant : soit $I = \{i_1, i_2, \ldots, i_m\}$ un ensemble de m éléments distincts. Un *événement* est une collection non vide non ordonnée de m éléments notée $(i_1, i_2, \ldots, i_m)$. Une *séquence* α est une liste ordonnée d'événements α_i notée $(\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_q)$. Une séquence avec k éléments ($k = \sum_j |\alpha_j|$) est appelée une $k-$séquence. Enfin, α (de cardinal n) est une sous-séquence de β ssi $\alpha \subseteq \beta$, ie. il existe des entiers $i_1 < i_2 < \ldots < i_n$ tels que $\alpha_1 \subseteq \beta_{i_1}, \alpha_2 \subseteq \beta_{i_2}, \ldots, \alpha_n \subseteq \beta_{i_n}$,. Une sous-séquence c de s est dite *contiguë* si elle résulte d'une suppression à la fois du premier et du dernier élément de s, ou de la suppression d'une des éléments de s contenu dans un événement ayant au moins 2 éléments, ou si c est une sous-séquence contiguë de c' laquelle est une sous-séquence contiguë de s.

2.4.1 GSP

Référence : [15]
Date de publication : 1996
Auteurs : Srikant, Agrawal
Algorithme : figure 14 page 17
Description : GSP (*Generalized Sequential Patterns*) est un algorithme multipasse. La première passe détermine le support (ie. le nombre de séquence contenant cet élément) de chaque élément et permet de ne travailler que sur les éléments ayant le support minimum et générer les séquences candidates. Le support de ces candidats étant déterminé lors des passages suivants.

La génération des candidats se fait en 2 phases : la jointure et l'élagage. La jointure se fait en joignant l'ensemble des (k-1)-séquences fréquentes L_{k-1} avec lui-même, ce qui consiste — en considérant les séquences $s_1, s_2 \in L_{k-1}$ — à rajouter le dernier élément de s_2 à s_1. Si l'élément constituait un événement à

lui tout seul dans s_2 alors il constituera un événement dans s_1. De la même manière s'il faisait parti d'un événement dans s_2 alors il sera rajouter au dernier événement de s_1. La jointure est valable uniquement si en enlevant le premier élément de s_1 on obtient la même sous-séquences qu'en retirant le dernier élément de s_2. Enfin, l'élagage consiste à retirer les candidats ayant des sous-séquences ou des sous-séquences contiguës dont le support serait inférieur au *minSupp*.

Une fois l'ensemble des $k-$séquences candidates généré, chaque candidat est dénombré dans la base de données. Afin d'optimiser le dénombrement, une table de hachage est utilisée ainsi que la mise en place d'une durée maximale entre éléments d'une séquence ce qui permet de limiter les recherches. Pour plus de détails sur l'implémentation de ces techniques, merci de vous référer à [15].

Enfin, l'algorithme GSP permet également d'utiliser les informations contenues dans une hiérarchisation des données de type *est-un*. Dans ce cas, on généralise la recherche aux descendants des éléments recherchés.

2.4.2 SPADE

Référence : [18]
Date de publication : 2001
Auteurs : Zaki
Algorithme : figures 15, 16 et 17 page 17
Description : SPADE (*Sequential PAttern Discovery using Equivalence classes*) est un algorithme qui utilise la théorie des Treillis afin de limiter son espace de recherche, ie. il utilise pour chaque ensemble une borne supérieure et une borne inférieure.

SPADE utilise des ID-listes verticales, *id est* une liste de couple (SID^4, EID^5) par atome. Concernant la recherche des séquences fréquentes, deux stratégies sont proposées : *Breadth-First Search* (BFS) et *Depth-First Search* (DFS). BFS correspond à la stratégie utilisée par GSP, elle procède niveau par niveau ce qui à l'avantage de permettre un élagage beaucoup plus intéressant que via DPS, cependant, DPS prend beaucoup moins de place en mémoire et cette technique — qui consiste à effectuer des recherches branche par branche — est la seule utilisable dans le cas où les ensembles de séquences fréquentes seraient amener à être beaucoup trop importants.

Lors de la génération des séquences candidates, les jointures se font de la façon suivante :

1. **Élément contre élément :** PB jointe avec PD en PBD.
2. **Élément contre séquence :** PB jointe avec $P \rightarrow A$ en $PB \rightarrow A$.
3. **Séquence contre séquence :** $P \rightarrow A$ jointe avec $P \rightarrow F$ en $P \rightarrow AF$, $P \rightarrow A \rightarrow F$ et $P \rightarrow F \rightarrow A$.

Le principal avantage de SPADE par rapport à GSP est l'utilisation de listes verticales temporaires pour les jointures. Ces listes, bornées, sont beaucoup plus petites et rapides à générer que celles utilisées par GSP ce qui permet d'améliorer grandement le temps de calcul nécessaire ainsi que de réduire les besoins de mémoire. Enfin, l'utilisation d'ensembles de treillis permet de restreindre le domaine de recherche autour des éléments fréquents ce qui occasion un important gain de temps.

[4] SID = Séquence IDentifiant .
[5] EID = Evénement IDentifiant.

Algorithmes

Algorithme 1 ID3

ENTRÉES: Exemples, attributCible, AttributsNonCible

```
si estVide(Exemples) alors
  retourner un nœud Erreur
sinon
  si estVide(AttributsNonCible) alors
    retourner un nœud ayant la valeur la plus représenté pour attributCible
  sinon
    si tous les éléments de Exemples ont la même valeur pour attributCible alors
      retourner un nœud ayant cette valeur
    sinon
      AttributSélectionné = attribut maximisant le gain d'information parmi AttributsNonCible
      AttributsNonCibleRestants = AttributsNonCible −{AttributSélectionné}
      nouveauNœud = nœud étiqueté avec AttributSélectionné
      pour chaque valeur de AttributSélectionné faire
        ExemplesFiltrés= exempleAyantValeurPourAttribut(Exemples, AttributSélectionné, valeur)
        nouveauNœud.fils(valeur)=ID3(ExemplesFiltrés, AttributSélectionné, AttributsNonCibleRestants)
      fin pour
      retourner nouveauNœud
    fin si
  fin si
fin si
```

Algorithme 2 SLIQ-Partition

ENTRÉES: Dataset S

```
si (tous les points de S appartiennent à la même classe) alors
  Sortir de l'algorithme
fin si
Évaluer les partitionnements pour chaque attribut A
Utiliser le meilleur partitionnement de S aboutissant sur la création de S1 et S2
Partition(S1) /* Appel récursif à gauche */
Partition(S2) /* Appel récursif à droite */
```

Algorithme 3 SLIQ-EvaluateSplits

pour chaque attribut A **faire**
  pour chaque valeur v de la liste d'attribut **faire**
    $l =$ feuille contenant l'élément courrant
    mettre à jour l'histogramme de la feuille l
    si A est un attribut continu **alors**
      tester la séparation pour le critère $A \leq v$ de la feuille l
    fin si
  fin pour
  si A est un attribut discret **alors**
    pour chaque feuille de l'arbre **faire**
      trouver le sous-ensemble de A qui réalise la meilleure séparation
    fin pour
  fin si
fin pour

Algorithme 4 SLIQ-UpdateLabels

pour chaque attribut A utiliser comme critère de séparation **faire**
  création des nouveaux nœuds et histogrammes associés
  pour chaque valeur v de la liste de l'attribut A **faire**
    soit e la classe correspondant à l'élément courrant
    trouver la nouvelle classe c de A correspondant à la nouvelle séparation
    modifier la référence de classe de l'élément de e vers c
  fin pour
fin pour

Algorithme 5 OC1

ENTRÉES: T /* L'ensemble d'entrainement */
  $H =$ bestAxis-Parallel(T) /* Meilleur hyperplan de séparation parallèle à un axe */
  $I = impurity(H)$ /* Impureté de cette séparation */
  pour $i = 1$ **jusqu'à** R **faire**
    si $i > 1$ **alors**
      $H =$ randomisedHyperplan() /* Hyperplan aléatoire */
    fin si
    label Etape_1
    répéter
      pour $j = 1$ **jusqu'à** d **faire**
        $Perturb(H, j)$ /* Perturber le coefficient j de H */
      fin pour
    jusqu'à I soit améliorée
    label Etape_2
    pour $i = 1$ **jusqu'à** J **faire**
      Choisir aléatoirement une direction et tenter de perturber H dans cette direction.
      si $impurity(H) < I$ **alors**
        aller à Etape_1
      fin si
    fin pour
    $I_l = impurity(H)$
    si $I_l < I$ **alors**
      $I = I_l$
    fin si
  fin pour
  retourner la séparation correspondant à I

Algorithme 6 OC1-Perturb

ENTRÉES: H, m
  pour $j = 1, .., n$ **faire**
    Calcule(U_j) /* Calcule U_j selon l'équation (4) */
    Trier $U_1, ..., U_n$ en ordre décroissant
    $a'_m =$ meilleure séparation de la liste triée des U_j
    $H_l =$ résultat de la substitution des a_m par a'_m dans H
    si impurity(H_l)<impurity(H) **alors**
      $a_m = a'_m$
      $P_{move} = P_{stag}$
    sinon
      /* Pour sortir d'un minimum local */
      si impurité(H_l)=impurité(H) **alors**
        $a_m = a'_m$ avec probabilité P_{move}
        $P_{move} = P_{move} - 0.1 * P_{stag}$
      fin si
    fin si
  fin pour

Algorithme 7 AdaBoost

ENTRÉES: $S = (x_1, y_1), ..., (x_m, y_m)$ avec $x_i \in X, y_i \in Y = -1, +1$

$D_1(i) = \frac{1}{m}, \forall i$ /* Initialisation */

pour $t = 1$ **jusqu'à** T **faire**

$S_t = ChoisirAleatoirement(S, D_t)$

$WeakLearner = EntrainerClassifieurFaible(S_t, D_t)$

$h_t = ObtenirHypotheseFaible(WeakLearner)$

$\varepsilon_t = ComputeError(h_t, S_t)$

$\alpha_t = \frac{1}{2} \ln(\frac{1-\varepsilon_t}{\varepsilon_t})$

$UpdateWeights(t)$ /* Mise à jour de la distribution selon l'équation (9) */

fin pour

retourner

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Algorithme 8 Learn++

ENTRÉES: Pour chaque distribution de la base de données $D_k \quad k = 1,2,...,K$:
- une séquence de m exemples d'entraînement $S = [(x_1,y_1),...,(x_m,y_m)]$,
- un algorithme d'apprentissage faible *WeakLearn*,
- un entier T_k pour spécifier le nombre d'itérations.

pour $k = 1$ **jusqu'à** K **faire**
  $w_1(i) = D(i) = \frac{1}{m}, \forall i$ /* A moins que l'on ai des connaissances pour faire autrement */
  pour $t = 1$ **jusqu'à** T_k **faire**
    label 1 : $D_t = w_t / \sum_{i=1}^{m} w_t(i)$ /* D_t est une distribution */
    label 2 : $TR_t = RandomSubset(S, D_t)$ /* Ensemble d'entraînement choisi aléatoirement dans S selon D_t */
    $TE_t = RandomSubset(S, D_t)$ /* Ensemble de test choisi aléatoirement dans S selon D_t */
    label 3 : $h_t = WeakLearn(TR_t)$ /* Hypothèse de $X \rightarrow Y$ */
    label 4 : $\varepsilon_t = ComputeError(h_t, S_t)$ /* Calcul de l'erreur de h_t selon l'équation (10) */
    si $\varepsilon_t > 1/2$ **alors**
      $t = t - 1$
      Supprimer(h_t)
      goto label 2
    sinon
      $\beta_t = normalise(\varepsilon_t)$ /* Erreur normalisée : $\frac{\varepsilon}{1-\varepsilon}$ */
    fin si
    label 5 : $H_t = \underset{y \in Y}{argmax} \sum_{t:h_t(x)=y} \log(1/\beta_t)$ /* Hypothèse composée */
    $E_t = ComputeError(H_t, S_t)$
    si $E_t > 1/2$ **alors**
      $t = t - 1$
      Supprimer(H_t)
      goto label 2
    fin si
    label 6 : $B_t = normalise(E_t)$ /* Erreur composite normalisée */
    UpdateWeight(t) /* Mise à jour des poids des instances selon l'équation (11) */
  fin pour
fin pour
retourner

$$H_{final} = \underset{y \in Y}{argmax} \sum_{k=1}^{K} \sum_{t:H_t(x)=y} \log \frac{1}{B_t}$$

Algorithme 9 Apriori

ENTRÉES: T, ε

$L_1 = \{$ensemble de 1-item qui apparaissent dans au moins ε transactions$\}$
$k = 2$
tant que $L_{k-1} \neq \varnothing$ **faire**
  $C_k = Generate(L_{k-1})$ /* Génère l'ensemble de candidats */
  pour $t \in T$ **faire**
    $C_t = SousEnsemble(C_k, t)$ /* Sélection des candidats de C_k présents dans t */
    pour $c \in C_t$ **faire**
      $count[c] = count[c] + 1$
    fin pour
  fin pour
  $L_k = \{c \in C_k | count[c] \geq \varepsilon\}$
  $k = k + 1$
fin tant que
retourner $\bigcup_k L_k$

Algorithme 10 Eclat

ENTRÉES: Liste des arguments :
- A /* Liste des éléments ordonnée en ordre croissant */
- $D \subseteq P(A)$ /* La base de données transactionnelles issue de A */
- $minSup \in \mathbb{N}$ /* Le support minimum définit par l'utilisateur */

$F = \{(\varnothing, |D|)\}$ /* Initialisation de l'ensemble des fréquent et de leur support */
$C_0 = \{(x, D(\{x\}) | x \in A\}$ /* Ensemble des transactions contenant un élément de A */
$C'_0 = freq(C_0)$ /* C_0 est filtrée selon (15) */
$F = \{\varnothing\}$
$F = addFrequentSupersets(F, C'_0)$ /* Appel recursif */
retourner F /* Ensemble d'éléments fréquents et leur support */

Algorithme 11 Eclat-addFrequentSupersets

ENTRÉES: Liste des arguments :
- $p \in P(A)$ /* Préfixe */
- C /* Matrice des motifs fréquents de p */
- F /* Ensemble des motifs fréquents et de leur support */

pour $(x, D_x) \in C$ **faire**
  $q = p \cup \{x\}$
  $C_q = \{(y, D_x \cap D_y) | (y, D_y) \in C, y > x\}$
  $C'_q = freq(C_q)$ /* C_q est filtrée selon (15) */
  si $C'_q \neq \varnothing$ **alors**
    $F = addFrequentSupersets(q, C'_q)$
  fin si
  $F = F \cup \{(q, |D_x|)\}$
fin pour
retourner F

Algorithme 12 SSDM-CyclesDeSimilarité

$A_2 =$ ensemble d'associations floues de taille 2
pour $(k = 3; k < size(D_n); k{+}{+})$ **faire**
  comparer chaque paire d'association floues dans A_{k-1}
  si $(\text{prefixe}(a_i) = \text{prefixe}(a_j), i \neq j)$ **alors**
    // Si l'union des suffixes est suffisament similaire
    si $((\text{suffix}(a_i) \cup \text{suffix}(a_j)) \in A_2)$ **alors**
      $a_k = a_i \cup \text{suffix}(a_j)$
    fin si
  fin si
  fincomparer
  $A_k =$ ensemble de tous les a_k
fin pour
$S_n =$ groupe de tous les A_k

Algorithme 13 kDCI

ENTRÉES: $D, minSupp$
$F_1 = premierScan(D, minSupp)$ /* Obtenir les figures d'optimisation durant le premier scan */
$F_2 = secondScan(D', minSupp)$ /* A partir du second scan on utilise une base de données D' temporaire */
$k = 2$
tant que $D'.taille\_verticale() > memoire\_disponible()$ **faire**
  $k{+}{+}$
  $F_k = DCP(D', minSupp, k)$
fin tant que
$k{+}{+}$
$F_k = DCP(D', VD, minSupp, k)$ /* Création d'une base de données verticale */
$dense = VD.est\_dense()$
tant que $F_k \neq \varnothing$ **faire**
  $k{+}{+}$
  si $use_clef_motif()$ **alors**
    si *dense* **alors**
      $F_k = DCI\_dense\_keyp(VD, minSupp, k)$
    sinon
      $F_k = DCI\_sparse\_keyp(VD, minSupp, k)$
    fin si
  sinon
    si *dense* **alors**
      $F_k = DCI\_dense(VD, minSupp, k)$
    sinon
      $F_k = DCI\_sparse(VD, minSupp, k)$
    fin si
  fin si
fin tant que

Algorithme 14 GSP

$F_1 = \{$1-séquence fréquentes$\}$
pour $k = 1$ **jusqu'à** $F_{k-1} \neq \varnothing$ **faire**
  $C_k =$ Ensemble de $k-$séquences candidates
  pour $\forall \varepsilon \in DB$ **faire**
    pour $\alpha \in C_k$ **faire**
      si $\varepsilon \subset \alpha$ **alors**
        Incrémenter le compteur de α
      fin si
    fin pour
  fin pour
  $F_k = \{\alpha \in C_k | \alpha.sup \geq minSupp\}$
fin pour
retourner $\bigcup_k F_k$ /* Ensemble des séquences fréquentes */

Algorithme 15 SPADE

ENTRÉES: $minSupp, D$ /* Support minimal et base de données */
$F_1 =$ { élements fréquents ou 1-séquences fréquentes }
$F_2 =$ {2-séquences fréquentes}
$\varepsilon =$ {classes équivalentes $[X]_{\theta_1}$}}
pour $\forall [X] \in \varepsilon$ **faire**
  $EnumerateFrequentSeq([X])$
fin pour

Algorithme 16 SPADE-EnumerateFrequentSeq

ENTRÉES: S /* Un sous-treillis et son ID-liste */
  pour $\forall A_i \in S$ **faire**
    $T_i = \varnothing$
    pour $\forall A_j \in S$ avec $j > i$ **faire**
      $R = A_i \vee A_j$
      si $Prune(R) = FAUX$ **alors**
        $L(R) = L(A_i) \cap L(A_j)$
        si $\sigma(R) \geq minSupp$ **alors**
          $T_i = T_i \cup \{R\}$
          $F_{|R|} = F_{|R|} \cup \{R\}$
        fin si
      fin si
    fin pour
    si DepthFirstSearch **alors**
      $EnumerateFrequenteSeq(T_i)$
    fin si
  fin pour
  si BreadthFirstSearch **alors**
    pour $\forall T_i \neq \varnothing$ **faire**
      $EnumerateFrequenteSeq(T_i)$
    fin pour
  fin si

Algorithme 17 SPADE-Prune

ENTRÉES: β
  pour $\forall$ (k-1)-sous-séquence, $\alpha \prec \beta$ **faire**
    si $[\alpha_1]$ a été calculé et $\alpha \notin F_{k-1}$ **alors**
      retourner Vrai
    fin si
  fin pour
  retourner Faux

Références

[1] Manish Mehta 0002, Rakesh Agrawal, and Jorma Rissanen. SLIQ : A fast scalable classifier for data mining. In Apers et al. [3], pages 18–32.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, SIGMOD Conference, pages 207–216. ACM Press, 1993.

[3] Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors. Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings, volume 1057 of Lecture Notes in Computer Science. Springer, 1996.

[4] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and Regression Trees. Wadsworth, 1984.

[5] Eduardo L. G. Escovar, Mauro Biajiz, and Marina Teresa Pires Vieira. SSDM : A semantically similar data mining algorithm. In Carlos A. Heuser, editor, SBBD, pages 265–279. UFU, 2005.

[6] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. Sci., 55(1) :119–139, 1997.

[7] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation : A frequent-pattern tree approach. Data Min. Knowl. Discov., 8(1) :53–87, 2004.

[8] Sreerama K. Murthy, Simon Kasif, Steven Salzberg, and Richard Beigel. Oc1 : A randomized induction of oblique decision trees. In AAAI, pages 322–327, 1993.

[9] Salvatore Orlando, Claudio Lucchese, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci : a multi-strategy algorithm for mining frequent sets. In FIMI, 2003.

[10] Robi Polikar, L. Upda, S. S. Upda, and Vasant Honavar. Learn++ : an incremental learning algorithm for supervised neural networks. IEEE Transactions on Systems, Man, and Cybernetics, Part C, 31(4) :497–508, 2001.

[11] J. Ross Quinlan. Induction of decision trees. Machine Learning, 1(1) :81–106, 1986.

[12] Ross R. Quinlan. C4.5 : programs for machine learning. Morgan Kaufmann Publishers Inc., 1993.

[13] Jeffrey C. Schlimmer and Douglas H. Fisher. A case study of incremental concept induction. In AAAI, pages 496–501, 1986.

[14] John C. Shafer, Rakesh Agrawal, and Manish Mehta 0002. SPRINT : A scalable parallel classifier for data mining. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, VLDB, pages 544–555. Morgan Kaufmann, 1996.

[15] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns : Generalizations and performance improvements. In Apers et al. [3], pages 3–17.

[16] Paul E. Utgoff. Incremental induction of decision trees. Machine Learning, 4 :161–186, 1989.

[17] Mohammed J. Zaki. Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering, 12 :372–390, 2000.

[18] Mohammed Javeed Zaki. SPADE : An efficient algorithm for mining frequent sequences. Machine Learning, 42(1/2) :31–60, 2001.