



Résumé de Datamining et Datawarehousing

Résumé réalisé par: Anthony Rouneau

Section: 1^{er} Bloc Master en Sciences Informatiques

Contents

I	Classification	3
1	Approche générale	3
2	Arbre de décision (DTI)	4
2.1	Algorithme de Hunt	4
2.2	Choisir la condition de test	5
2.3	Qualité d'un partitionnement	6
2.4	Algorithme	9
2.5	Caractéristiques	10
3	Overfitting	11
3.1	Bruit	11
3.2	Manque de représentativité	12
3.3	Procédure de comparaison multiple	12
3.4	Estimation des erreurs de généralisation	12
3.4.1	Estimation par resubstitution	12
3.4.2	Incorporation de la complexité du modèle	13
3.4.3	Estimer les bornes statistiques	15
3.4.4	Utiliser des données de validation	15
3.5	Overfitting des arbres de décision	15
3.6	Évaluer les performances d'un classificateur	16
3.6.1	Les métriques	16
3.6.2	Méthode <i>Holdout</i>	18
3.6.3	Random subsampling	18
3.6.4	Cross-validation	18
3.6.5	Bootstrap	18
3.7	Méthode de comparaison de classificateurs	19
4	Types d'algorithmes	22
4.1	Nearest Neighbours (IBk, Lazy)	22
4.2	Règles de classification	22
4.3	Arbres IBk	24
4.4	Naive Bayes	24
II	Analyse par association	25
5	Découverte de règles	26

6	Génération de <i>frequent itemsets</i>	27
6.1	Le principe <i>Apriori</i>	27
6.2	Génération d'itemsets fréquents avec <i>Apriori</i>	28
6.3	Génération de candidats et élagage	29
6.3.1	Force brute	29
6.3.2	$F_{k-1} \times F_1$	29
6.3.3	$F_{k-1} \times F_{k-1}$	30
6.4	Compter le <i>support count</i>	30
7	Génération de règles d'associations	33
7.1	Élagage basé sur la <i>confidence</i>	33
7.2	Génération de règle dans l'algorithme <i>Apriori</i>	33
8	Représentations compactes des itemsets fréquents	35
8.1	Maximal Frequent Itemset	35
8.2	Closed Frequent Itemsets	36
9	Optimiser <i>Apriori</i>	38
9.1	Traverser un treillis	38
9.1.1	General-To-Specific	38
9.1.2	Specific-To-General	38
9.1.3	Bidirectionnel	39
9.1.4	Classes d'équivalence	39
9.1.5	Profondeur / Largeur	40
9.2	Représentation des transactions	41
10	Algorithme FP-Growth	43
10.1	FP-Tree	43
10.2	Génération d'itemsets fréquents	43
III	Analyse par clustering	45

Part I

Classification

Le rôle d'une classification est d'assigner des objets (instances) à une classe selon une suite d'attributs. Chaque objet peut donc être représenté par un tuple (x, y) , où x est une séquence d'attributs et y le nom de sa classe.

Un **attribut** peut être discret ou continu, contrairement au **label de classe**, qui lui doit être discret. Ce qui veut dire que le nombre de classes dans lesquelles les instances seront classé est fini, et que toutes les classes sont à priori connues. **C'est ce qui différencie une classification d'une régression**

Classification – Tâche de construire/d'apprendre une fonction f qui fera correspondre un ensemble d'attributs, un objet, x à une des classes prédéfinies, y .

• Buts d'une classification

Modélisation descriptive Utilisation d'une classification dans le but de lister les attributs qui définissent une certaine classe. (e.g. attributs d'animaux → famille d'animaux.

Modélisation prédictive Utilisation d'une classification pour prédire la classe d'objets inconnus. Notons qu'une classification n'est pas performante pour des classes ordonnées (avec une hiérarchie ou un ordre définie).

1 Approche générale

Une technique de classification consiste à construire un modèle de classification depuis un **ensemble de données d'entraînement**. Ce modèle peut ensuite être évalué en y faisant passer un **ensemble de données de test**, idéalement **différent de l'ensemble d'entraînement pour éviter l'overfitting** (voir plus loin).

Un modèle est évalué selon plusieurs métriques, dont par exemple la *Précision* et le *Taux d'erreur*.

Confusion matrix for a 2-class problem.

		Predicted Class	
		<i>Class = 1</i>	<i>Class = 0</i>
Actual Class	<i>Class = 1</i>	f_{11}	f_{10}
	<i>Class = 0</i>	f_{01}	f_{00}

$$\begin{aligned} \text{Précision} &= \frac{\text{Nombre de prédictions correctes}}{\text{Nombre total de prédiction}} = \frac{f_{11} + f_{00}}{f_{11} + f_{10} + f_{01} + f_{00}} \\ \text{Taux d'erreur} &= \frac{\text{Nombre de prédictions fausses}}{\text{Nombre total de prédiction}} = \frac{f_{10} + f_{01}}{f_{11} + f_{10} + f_{01} + f_{00}} \end{aligned}$$

2 Arbres de décision (DTI)

DTI = Decision Tree Induction.

Le nombre d'arbres de décision pouvant découler d'un ensemble d'attributs est exponentiel. Il est donc impossible de trouver l'arbre optimal pour un problème de classification donné. C'est pourquoi **la plupart des techniques de construction utilisent un algorithme glouton (greedy).**

2.1 Algorithme de Hunt

Les méthodes de construction ID3, C4.5, CART découlent toutes du même algorithme : l'algorithme de Hunt. L'algorithme consiste à partitionner récursivement les données d'entraînement dans de sous-ensembles plus "purs". Celui-ci est décrit ci-dessous.

Soit D_t le sous-ensemble des données d'entraînement qui est associés au noeud t de l'arbre, et $y = \{y_1, y_2, \dots, y_c\}$ les labels de classe.

- Étape 1** Si tous les objets de D_t appartiennent à la même classe y_t , alors, le noeud t est une feuille avec le label y_t .
- Étape 2** Si D_t contient des instances qui appartiennent à plus qu'à une classe, une **condition de test d'attribut** est sélectionnée pour partitionner les objets dans de plus petits sous-ensembles. Un noeud fils est créé pour chaque sortie de la condition de test et les objets de D_t sont répartis dans les noeuds fils selon la condition sélectionnée. Pour bien partitionner, il faut tenir compte du type d'attribut (cf. 2.2) et savoir évaluer la qualité d'une partition (cf. 2.3).
- Étape 3** **Recommencer jusqu'à ce qu'on ne doive plus diviser aucun noeud.** Il y a plusieurs méthodes pour déterminer cela. Soit on peut attendre qu'il n'y ait plus que des feuilles parfaites, ne contenant qu'une seule classe chacune, soit on peut arrêter le processus plus tôt.

Pour que cet algorithme fonctionne, il faudrait que chaque combinaison des valeurs d'attributs soit présente dans les données d'entraînement et que chaque combinaison ait une classe unique (classification parfaite car on aurait géré tous les cas possibles). Ce n'est évidemment pas possible, c'est pourquoi **les deux conditions ci-dessous sont nécessaires en pratique.**

1. Il est possible qu'un des noeuds fils créés à l'étape 2 soit vide (aucune des instances ne correspond à la condition). Dans ce cas, **le noeud fils vide en question devient une feuille et aura comme label la classe majoritaire présent dans son parent.**

2. Si dans l'étape 2, tous les objets sont les même, mais qu'ils n'appartiennent pas à la même classe (contradictions, bruits), il n'est dès lors plus possible de diviser le noeud. Il deviendra donc une feuille et aura comme label la classe majoritairement représentée par ses objets.

2.2 Choisir la condition de test

Il faut pouvoir diviser les différents attributs en fonction de leur type.

Attributs binaire La condition de test génère deux noeuds fils, un par valeur possible.

Attributs nominaux Les deux manières de division sont illustrées par la figure 1. À savoir que des algo. de construction d'arbres, comme CART ne produise que des sorties binaires, en choisissant la meilleure découpe binaire parmi les $2^k - 1$ possibilités.

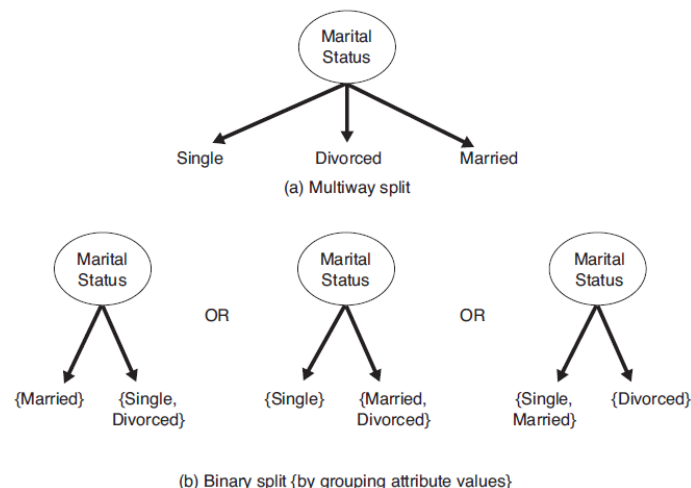


Figure 1: Division d'attributs binaires

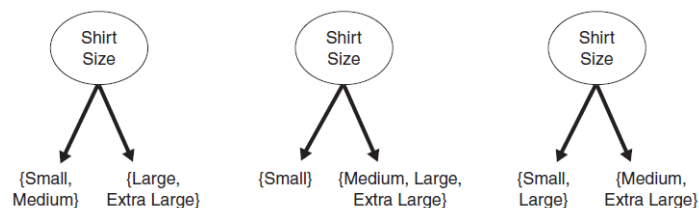


Figure 2: Division d'attributs ordonnés

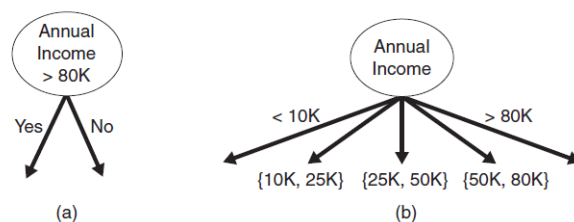


Figure 3: Division d'attributs continus

Attributs ordonnés De la même manière que les nominaux, tant que l'ordre des attributs n'est pas cassé par les partitions, voir la figure 2.

Attributs continus Même chose que les deux derniers, même si cette fois, on a une condition sur la valeur continue, comme représenté sur la figure 3. L'algorithme va évaluer les différents splits possibles et choisir le meilleur. Pour une division *multiway*, une discrétisation est généralement opérée sur les valeurs des attributs.

2.3 Qualité d'un partitionnement

Pour évaluer la qualité d'un partitionnement, définissons $p(i|t)$ comme étant le **pourcentage** (entre 0 et 1) d'objets appartenant à la classe i , pour un noeud d'arbre t . Quand le noeud n'est pas utile, la notation peut se réduire à p_i . Pour une classe binaire, (p_0, p_1) peut-être utilisé avec $p_1 = 1 - p_0$.

• Impureté d'un noeud

Plus une classe majoritairement représentée par les objets du noeud, au plus ce noeud, cette partition, sera considérée comme *pure*. En pratique, c'est le degré d'impureté qui sera mesuré. un noeud $(0, 1)$ n'a aucune impureté, tandis qu'un noeud $(0.5, 0.5)$ est le plus impur possible (aucune classe n'est majoritairement représentée).

Quelques **exemples de mesures d'impuretés** (c = le nombre de classes différentes, et $0 \log_2 0 = 0$ pour le calcul de l'Entropy) :

$$\text{Entropy}(t) = - \sum_{i=0}^{c-1} p(i|t) \cdot \log_2 \cdot (p(i|t))$$

$$\text{Gini}(t) = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2$$

$$\text{Classification error}(t) = 1 - \max_i [p(i|t)]$$

Les trois mesures, bien que différentes, respecte toutes la mesure "d'impureté" et ont donc le même pire cas et le même meilleur cas. Cependant, la mesure choisie peut influencer le résultat du partitionnement.

• Impureté d'un split

Pour évaluer l'impureté d'un partitionnement, on va utiliser une moyenne pondérée par rapport au nombre d'objet par partition.

$$I(\text{split}) = \sum_{j=1}^k \frac{N(v_j)}{N} I(v_j)$$

Avec I la mesure d'impureté, k le nombre de partitions, N le nombre total d'objets, et $N(v_j)$ le nombre d'objets présents dans la partition (noeud fils) v_j .

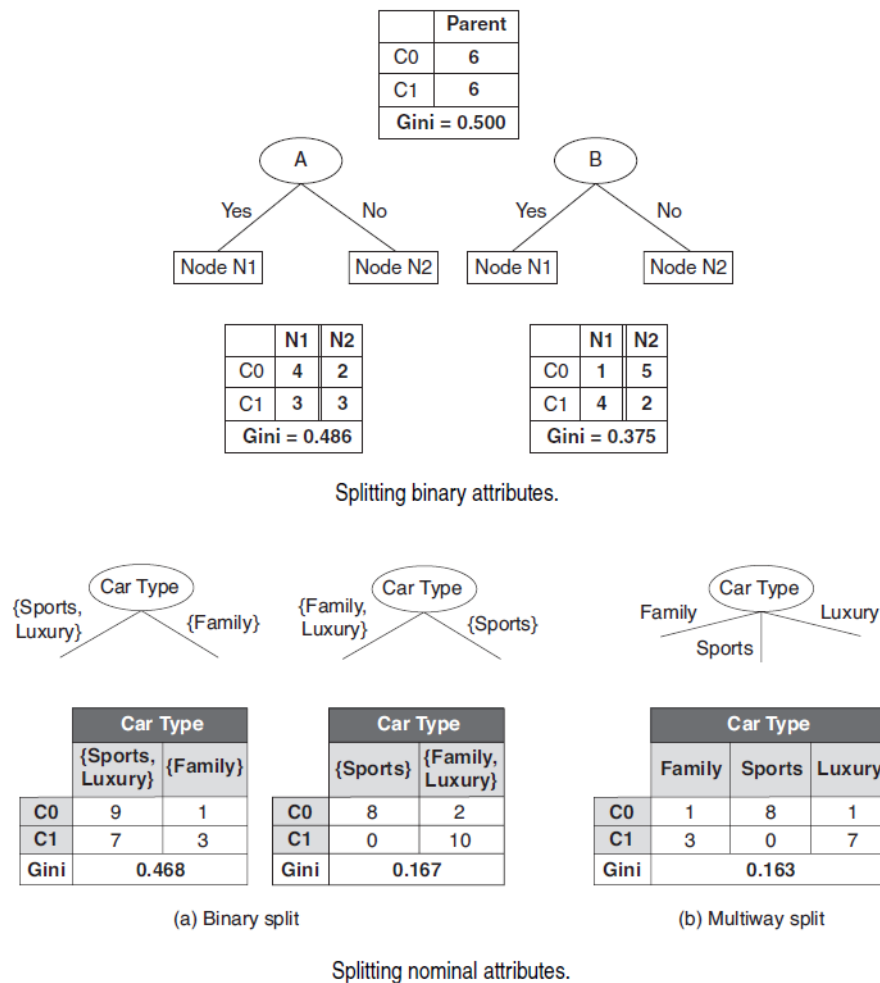


Figure 4: Exemple de Gini(split) de partitionnements d'attributs binaires et nominaux

Par exemple, prenons la figure 4, le tableau Car Type ((b), split *multiway*). Le calcul se fait comme suit ($\{value\}$ représentant un noeud fils, une partition) :

$$\begin{aligned}
 \text{Gini}(\text{split}) &= \frac{4}{20} \cdot \text{Gini}(\{\text{Family}\}) + \frac{8}{20} \cdot \text{Gini}(\{\text{Sports}\}) + \frac{8}{20} \cdot \text{Gini}(\{\text{Luxury}\}) \\
 \text{Gini}(\text{split}) &= \frac{4}{20} \cdot [1 - (\sum_{i=0}^1 [p(C_i|\{\text{Family}\})]^2)] + \frac{8}{20} \cdot [1 - (\sum_{i=0}^1 [p(C_i|\{\text{Sports}\})]^2)] \\
 &\quad + \frac{8}{20} \cdot [1 - (\sum_{i=0}^1 [p(C_i|\{\text{Luxury}\})]^2)] \\
 \text{Gini}(\text{split}) &= \frac{4}{20} \cdot 0.375 + \frac{8}{20} \cdot 0 + \frac{8}{20} \cdot 0.163 = 0.163
 \end{aligned}$$

C'est normal que le *multiway* ait de meilleurs résultats que les binaires, car faire du binaire, c'est en quelques sorte fusionner des résultats de *multiway*

• Ratio de gain

Il peut arriver qu'un attribut soit inapproprié à la classification. Typiquement, on retrouve souvent un attribut "ID" dans les tables. Chaque objet ayant un ID unique, l'attribut ne peut en aucun cas aider à la classification. De manière générale, on voudrait éviter qu'un attribut ne divise l'arbre en trop de branches.

Pour éviter ceci, il y a deux manières de faire :

1. Se restreindre à des splits binaires
2. Prendre en compte dans le score d'un split le nombre de partitions, comme le fait par exemple le **Gain ratio**.

$$\text{Gain ratio} = \frac{\Delta_{\text{info}}}{\text{Entropy}(\text{split})}$$

Ou $\text{Entropy}(\text{split})$ est l'impureté du split, calculé avec l'Entropy. L'idée est que si le split a beaucoup de branches, alors $\text{Entropy}(\text{split})$ sera grand, réduisant ainsi son *Gain ratio*.

2.4 Algorithme

Algorithm A skeleton decision tree induction algorithm.

```

TreeGrowth (E, F)
1: if stopping_cond(E,F) = true then
2:   leaf = createNode().
3:   leaf.label = Classify(E).
4:   return leaf.
5: else
6:   root = createNode().
7:   root.test_cond = find_best_split(E, F).
8:   let V = {v|v is a possible outcome of root.test_cond }.
9:   for each v ∈ V do
10:    E_v = {e | root.test_cond(e) = v and e ∈ E}.
11:    child = TreeGrowth(E_v, F).
12:    add child as descendent of root and label the edge (root → child) as v.
13:   end for
14: end if
15: return root.
```

Figure 6

`Classify()` consiste à chercher un label à la feuille. Dans la plupart des cas, cela consiste à faire:

$$\text{leaf.label} = \underset{i}{\operatorname{argmax}} p(i|t)$$

$p(i|t)$ peut dès lors être utilisé comme la probabilité qu'à un objet du noeud t d'appartenir à la classe i .

Après être passé dans l'algorithme, la taille de l'arbre peut-être réduite grâce à une étape d'élagage (pruning), car les arbres trop larges font souvent l'objet d'*overfitting*.

2.5 Caractéristiques

Les caractéristiques principales d'une classification par arbre de décision sont les suivantes:

1. Technique non-paramétrique. Aucune hypothèse sur le jeu de donnée n'est nécessaire.
2. La recherche de l'arbre optimale $\in NP - Complet$, les méthodes utilisées sont des heuristiques.
3. Les techniques développées sont très rapides d'exécution.
4. Un arbre est facile à interpréter et à comparer à d'autres modèles.
5. Donne une représentation expressive pour l'apprentissage de fonctions à valeurs discrètes. Même s'il y a quelques exceptions avec des problèmes Booléens, comme la fonction de parité, dont la valeur est 0 (*resp.* 1) quand il y a un nombre impair (*resp.* pair) d'attribut Booléen. Il faut un arbre complet avec 2^d noeud (d = nbre d'attribut Booléen) pour avoir un arbre permettant de répondre.
6. Les arbres sont robustes au bruit.
7. Les **attributs redondants** ne posent pas de problèmes. Un attribut est redondant lorsqu'il y a une forte corrélation entre cet attribut et un autre. En pratique, quand l'un des deux attributs redondants sera utilisé pour un split, l'autre ne sera pas utilisé. Si l'arbre est trop grand, il se peut qu'il y ait tout de même des splits redondant, qui seront coupé à l'élagage.
8. Si une feuille contient trop peu d'objets, on parle de **fragmentation des données**. On peut l'éviter en refusant de split lorsqu'on a plus assez d'objets.
9. Des sous-arbres peuvent être dupliqué dans l'arbre de décision. Ceux-ci seront fusionnés dans l'élagage.
10. Les frontières, formées par les attributs (cf. figure 7), entre les classes sont rectilignes pour les arbres car l'étape de division se fait pour un attribut à la fois, ce qui empêche de trouver les relations complexes qu'il peut y avoir entre les attributs. Des **arbres de décision obliques** peuvent être utilisés pour résoudre ce problème, ils prennent plus qu'un attribut pour diviser leurs attributs, ce qui multiplie les calculs... Il existe aussi la **construction inductive**, qui va créer de nouveaux attributs à partir des attributs de base qui sont liés entre eux. Ceci demande moins de calcul car les relations entre attributs se calculent une seule fois.
11. Le choix de l'impureté n'a pas beaucoup d'influence sur la qualité de l'arbre final. La stratégie d'élagage, par contre, peut l'influencer.

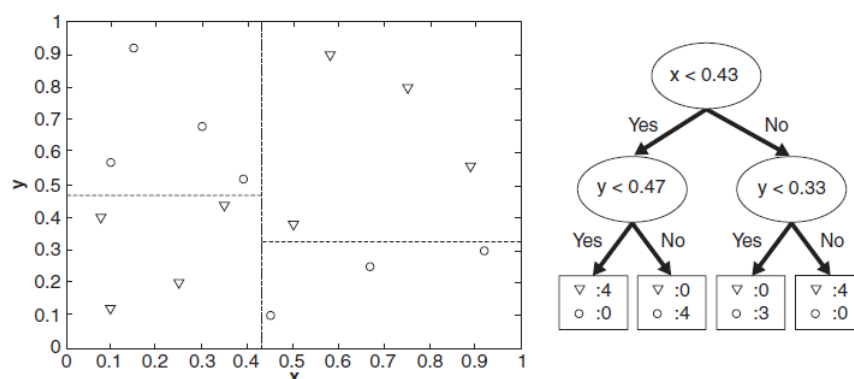


Figure 7: Exemple de frontières séparant les classes

3 Overfitting

On divise les erreurs de classification en deux types d'erreur :

1. **Les erreurs d'entraînement, ou erreurs de resubstitution** – Erreurs commises sur les données d'entraînement.
2. **Les erreurs de généralisation** – Erreurs commises sur des données extérieures, ne venant pas du jeu d'entraînement.

Dans les deux cas, l'erreur représente le nombre d'objets mal classifiés sur le nombre total d'objets du jeu de données.

Un modèle présentant plus d'erreur de généralisation que d'erreur d'entraînement est ce qu'on appelle un modèle qui fait de l'**overfitting**, c'est-à-dire qu'il colle trop à ses données d'entraînement et qu'il n'est pas très bon pour classer d'autres objets, même s'il est peut-être bon pour les données d'entraînement.

Lorsqu'un modèle est trop restreint, il peut y avoir le phénomène d'**underfitting** qui apparaît. Ceci veut dire que le modèle n'a pas saisi les caractéristiques qui permettent de classer correctement et est donc mauvais à la fois pour classer les données d'entraînement, mais aussi pour classer les données extérieures.

3.1 Bruit

Le bruit est une des causes de l'**overfitting**. Si par exemple des objets sont mal classifiés dans le jeu de données d'entraînement, cela va induire en erreur le modèle et peut donc commettre cette erreur en essayant de classer des données extérieures.

On peut parfois identifier la partie du modèle qui pose problème, et la supprimer.

3.2 Manque de représentativité

Si il y a un petit nombre d'objets, ou du moins, qu'une classe compte un petit nombre d'objets dans le jeu de données d'entraînement, le modèle va avoir du mal à identifier ce qui caractérise cette classe mal représentée (e.g. 5 *no*, 1 *yes*).

3.3 Procédure de comparaison multiple

Pour illustrer ce qu'est la comparaison multiple, imaginons qu'on doive prédire s'il va pleuvoir demain. Si on tire au hasard, la probabilité qu'on ait raison est de 0.5. Par contre, si on doit prédire pour les dix prochains jour, la probabilité qu'on ait raison au moins 8 fois sur 10 est de :

$$\frac{\binom{10}{8} + \binom{10}{9} + \binom{10}{10}}{2^{10}} = 0.0547$$

Ce n'est pas très élevé, mais si on demande à 50 personnes de faire le même travail pour les 10 jours à venir, *Nobody* étant la probabilité que personne n'ait raison au moins pour 8 jours, et *AtLeastOne* la probabilité qu'au moins une des 50 personnes ait raison 8 fois sur 10 :

$$Nobody = (1 - 0.0547)^{50}$$

$$AtLeastOne = 1 - Nobody = 0.9399$$

Mais même si l'on trouve la personne qui a trouvé, rien ne garanti que cette même personne est douée dans ce qu'elle fait. Peut-être qu'elle a juste deviné au hasard, huit jours de suite.

On peut faire l'analogie avec la construction d'un arbre de décision. Si l'on dit qu'on a un arbre T_0 et qu'on cherche, parmi k splits différents du noeud courant, $T_{x_{max}}$, l'arbre résultant de l'ajout du split, de gain maximum. Et si l'on dit que l'on choisit $T_{x_{max}}$ pour continuer plutôt que T_0 si ce gain dépasse un seuil α ($\Delta(T_0, T_{x_{max}}) > \alpha$). Et bien plus k augmente, plus on a de chance de trouver le split qui va donner un gain assez élevé. Mais si le nombre de donnée est réduit, la variance de ce gain est plus grand, et il y a de grandes chances qu'un gain "hasardeux" soit choisi car il dépassait α (à l'image du météorologiste chanceux, on a aucune garantie que cet ajout au modèle restera performant pour les données extérieures).

3.4 Estimation des erreurs de généralisation

La complexité d'un modèle influence l'*overfitting*. La question est donc quelle est la complexité idéale pour un modèle ? Il faudrait minimiser le taux d'erreur de généralisation, mais lors de la création du modèle, l'algorithme n'a accès qu'aux données d'entraînement.

3.4.1 Estimation par resubstitution

On suppose ici que le jeu de données d'entraînement représente bien les données globales du problème. Il suffit donc de minimiser l'erreur d'entraînement pour minimiser, par hypothèse, l'erreur de généralisation. **Pas bon en pratique.**

3.4.2 Incorporation de la complexité du modèle

Comme l'*overfitting* peut venir d'un modèle trop complexe, préférer des modèles plus simples permettrait de limiter l'*overfitting*, c'est le principe du **Rasoir d'Occam**.

Rasoir d'Occam – Parmi deux modèles avec la même erreur de généralisation, le modèle le plus simple des deux est préféré au plus complexe.

Comme disait Einstein, "Tout devrait être fait le plus simple possible, mais pas plus simple". Il ne faudrait donc pas prendre un modèle trop simple, au risque d'*underfitting*.

Ci-dessous sont présentés deux méthodes d'intégration de la complexité dans l'évaluation des modèles.

• Estimation pessimiste de l'erreur

On additionne l'erreur d'entraînement et la complexité du modèle. Appelons $n(t)$ le nombre d'objets d'entraînement classifiés par le noeud t et $e(t)$ le nombre d'objet mal classifiés. L'estimation pessimiste de l'erreur d'un arbre T , $e_g(T)$ est la suivante :

$$e_g(T) = \frac{\sum_{i=1}^k [e(t_i) + \Omega(t_i)]}{\sum_{i=1}^k n(t_i)} = \frac{e(T) + \Omega(T)}{N_t}$$

Avec k le nombre de feuilles, $e(T)$ l'erreur globale d'entraînement, N_t le nombre d'objets d'entraînement, et $\Omega(T)$ la pénalité associée à chaque noeud t_i . La valeur de cette pénalité décidera de l'importance que l'on accorde à la complexité dans l'estimation.

Pour un arbre binaire, une pénalité de 0.5 indique qu'un noeud doit toujours se diviser en deux si la division permet de mieux classer au moins un objet d'entraînement, en effet, ajouter une pénalité de 0.5 à l'erreur en créant une nouvelle feuille (on en crée deux, mais on ajoute qu'une seule feuille, étant donné que le parent était précédemment une feuille, et qu'il n'en est plus une) est moins coûteux que de commettre une erreur d'entraînement. Et une pénalité valant 1 indique que pour diviser un noeud, il faut que la division permette d'améliorer la classification d'au moins deux objets d'entraînement.

• Description de longueur minimal (MDL)

Approche venant de la théorie de l'information. En prenant la figure 8 comme exemple, où A et B partagent un jeu de donnée X , mais où il n'y a que A qui connaît y . A voudrait donc transmettre les labels (y) à B , ce qui représente $O(n)$ bits d'information. Si A arrive à créer un modèle permettant de déduire y_i de X_i , alors si la taille de l'encodage du modèle est inférieure à la taille de l'encodage des y , il est plus intéressant d'envoyer le modèle plutôt que y en entier.

Il se peut cependant que le modèle ne permettent pas une classification sans faute. Il faudra alors envoyer à B ces objets, mal classifiés, individuellement.

$$\text{Cost}(\text{model}, \text{data}) = \text{enc}(\text{model}) + \text{enc}(\text{misclassified})$$

$\text{enc}(\text{model})$ étant le coût de l'envoi de l'encodage du modèle et $\text{enc}(\text{misclassified})$ le coût de l'envoi de l'encodage des objets mal classifiés par le modèle. On devrait donc chercher un modèle qui minimise se coût.

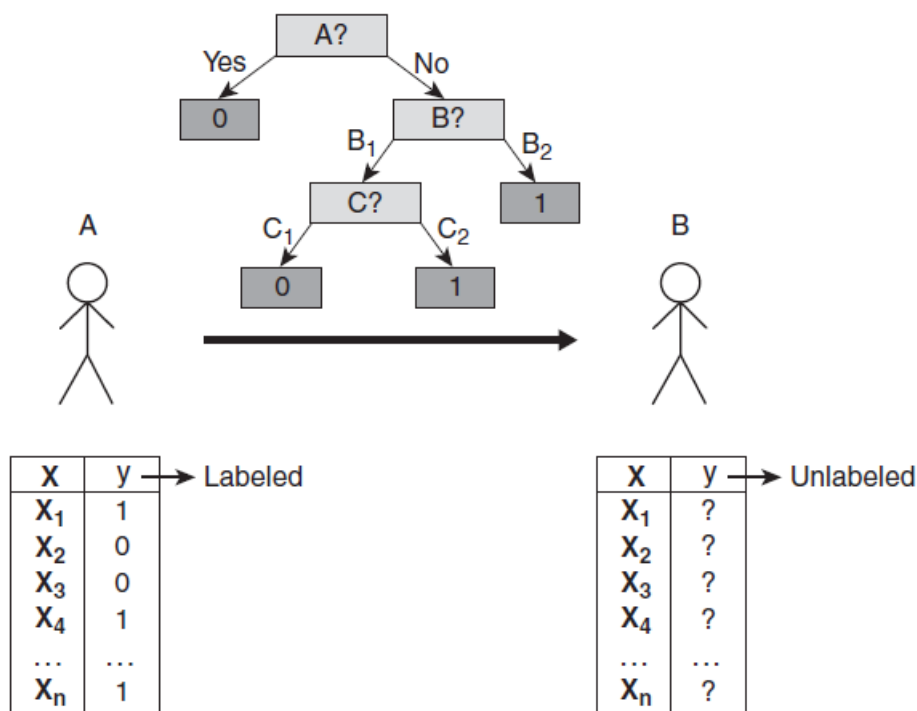


Figure 8: Minimum Description Length (MDL) Principle

3.4.3 Estimer les bornes statistiques

L'erreur de généralisation peut aussi être estimée comme une correction statistique de l'erreur d'entraînement. On calculera la correction statistique comme une borne supérieure de l'erreur d'entraînement, en prenant en compte le nombre d'objet qui atteignent une feuille en particulier. Par exemple, l'algorithme C4.5 suppose que le nombre d'erreurs commises par chaque feuille suit une distribution binomiale. Il faut donc déterminer la borne limite supérieure à l'erreur d'entraînement observée.

3.4.4 Utiliser des données de validation

Dans cette méthode, on coupe le jeu de donnée d'entraînement en deux. Une partie (généralement $\frac{2}{3}$) du jeu d'entraînement original est gardé pour l'entraînement du modèle et le reste des données est utilisé pour estimer l'erreur de généralisation (comme si cette partie des données était composée de données extérieures). Le point faible de cette méthode est que l'on retire des données du jeu d'entraînement, ce qui peut causer des lacunes dans le modèle (classe trop peu représentée, ...).

3.5 Overfitting des arbres de décision

Avoir une estimation fiable de l'erreur de généralisation permet à l'algorithme de classification de chercher un modèle précis sans *overfit* les données d'entraînement. Ci-dessous sont présentées deux stratégies qui permettent d'éviter l'*overfitting* dans les arbres de décision.

• Prepruning (Early Stopping Rule)

Le pré-élagage consiste à arrêter la croissance de l'arbre avant d'atteindre un arbre qui *overfit* les données d'entraînement. Il faut pour cela choisir une condition d'arrêt. Celle-ci peut se baser sur l'impureté ou le gain. La **principale difficulté** est alors de choisir le bon seuil qui arrêtera la croissance de l'arbre pour la mesure choisie. Un seuil trop restrictif implique de l'*underfitting* et un seuil trop large implique l'*overfitting*.

• Post-pruning

On applique le post-élagage sur un arbre qui n'a pas subi de *prepruning*. On élague cette fois-ci des feuilles vers la racine. Les deux manières d'élaguer sont les suivantes :

- (1) Remplacer un sous-arbre par une feuille avec le label présent majoritairement dans ce sous-arbre.
- (2) Remplacer un sous-arbre par sa branche la plus utilisée.

L'élagage s'arrête quand il n'y a plus d'améliorations possibles.

On obtient souvent de meilleurs résultats avec le *post-pruning* car on part d'un arbre complet, et on évite donc l'*underfitting*. Le point faible pouvant être que les calculs fait pendant la création de l'arbre peuvent être "gaspillés" quand on fini par couper une partie de l'arbre.

3.6 Évaluer les performances d'un classificateur

Maintenant qu'on peut trouver un modèle qui évite l'*overfitting*, on va vouloir le tester sur des données extérieures, des données de test. Ceci va permettre d'obtenir une estimation non-biaisée de l'erreur de généralisation (par la définition de ce type d'erreur).

3.6.1 Les métriques

Différentes métriques existent pour évaluer les performances d'un modèle sur un jeu de test. Soit TP le nombre de vrai positifs, TN le nombre de vrais négatifs, FP le nombre de faux positifs et FN le nombre de faux négatifs.

$$P = TP + FN$$

$$N = FP + TN$$

$$\text{True} - \text{Positive} - \text{Rate} = \frac{TP}{P} (= TPR)$$

$$\text{False} - \text{Positive} - \text{Rate} = \frac{FP}{N} (= FPR)$$

$$\text{Recall} = \frac{TP}{P}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Accuracy} = \frac{TP + TN}{P + N}$$

$$\begin{aligned} F - \text{Measure} &= 2 \cdot \frac{\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}} \\ &= \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \end{aligned}$$

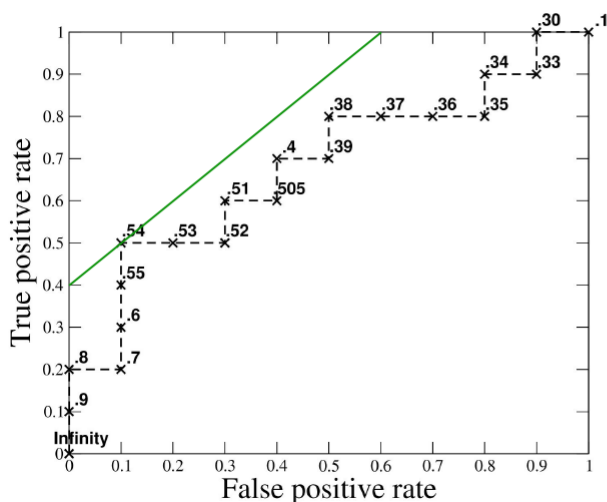
Il existe une autre mesure, plus complexe, appelée *ROC*. Pour l'expliquer, prenons les figures 9 à 11. Le but est d'évaluer la capacité d'un attribut (pour l'exemple : *Score*) à classer les instances. En effet, certains attributs permettent de mieux distinguer les classes grâce à une conditions sur ses valeurs.

En prenant l'exemple de la figure 9, la valeur la plus haute (au niveau des diagonales), est celle de l'instance 6. Au plus cette diagonale tend vers le haut-gauche du graphe, au mieux l'attribut coupe bien les instances en classes.

On sait que l'instance 6 est p (le point le plus haut est d'office dans la classe testée, ici p), et que l'instance d'après est n (car sinon, le point le plus haut aurait été celui-ci), donc, la condition sera :

$$\text{Classe } p \text{ si } \text{Score} \geq \frac{0.54 + 0.53}{2}; \text{ Classe } n \text{ sinon}$$

Se faisant, on sait que 0.54 sera de la classe p , et que 0.53 sera dans la class n .



Inst#	Class	Score	Inst#	Class	Score
1	p	.9	11	p	.4
2	p	.8	12	n	.39
3	n	.7	13	p	.38
4	p	.6	14	n	.37
5	p	.55	15	n	.36
6	p	.54	16	n	.35
7	n	.53	17	p	.34
8	n	.52	18	n	.33
9	p	.51	19	p	.30
10	n	.505	20	n	.1

Figure 9: Pour une instance de p , on monte, et pour une instance de n , on va à droite

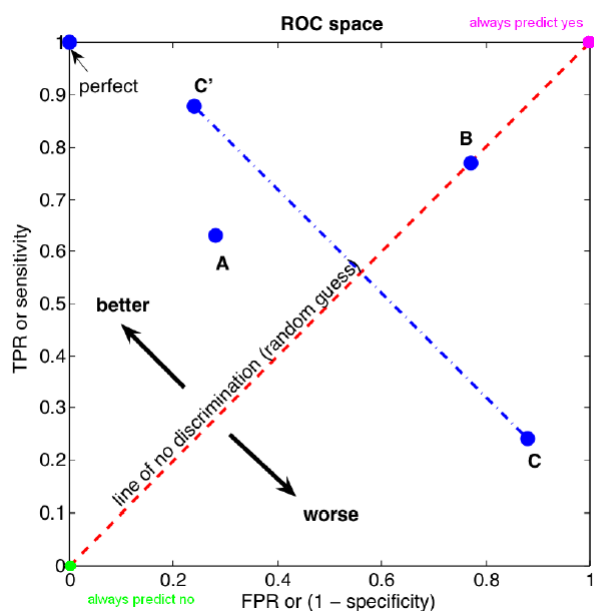


Figure 10: On prend la plus haute diagonale, parallèle à la ligne de discrimination, qui passe par un des points

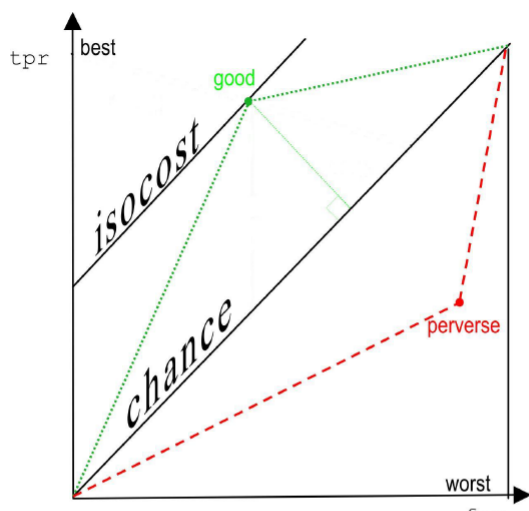


Illustration of ROC Analysis. The main diagonal represents chance with parallel isocost lines representing equal cost-performance. Points above the diagonal represent performance better than chance, those below worse than chance. For a single good (dotted=green) system, AUC is area under curve (trapezoid between green line and $x=[0,1]$). The perverse (dashed=red) system shown is the same (good) system with class labels reversed.

Figure 11: Au plus la diagonale est haute, au plus la classification est bonne

3.6.2 Méthode *Holdout*

Très semblable à l'usage de données de validation. On va diviser les données initiales en deux ensembles de même taille. Un des ensemble qui sert de données de test et l'autre de données d'entraînement.

Les deux ensembles sont donc liés, il peut y avoir un déséquilibre dans les labels. On pourrait donc avoir une classe moins représentée dans un ensemble, ce qui implique (si les classes sont bien réparties dans les données de base) que cette classe est représentée trop fort dans l'autre ensemble. Ceci peut impliquer un modèle biaisé de par le déséquilibre dans les données d'entraînement.

3.6.3 Random subsampling

Dans cette méthode, on va répéter la méthode *holdout* k fois, la découpe en deux ensemble étant aléatoire, chaque découpe sera différente.

On initialise un compteur i . Tant que $i < k$, on divise en deux les données, on améliore le modèle et on teste le modèle.

La précision du modèle obtenu est la suivante, avec acc_i la précision du modèle obtenu à l'itération i .

$$\sum_{i=1}^k \frac{acc_i}{k}$$

3.6.4 Cross-validation

Cette fois-ci, chaque objet est utilisé le même nombre de fois pour entraîner le modèle, et une seul et unique fois pour le tester. Cette méthode se divise en deux techniques principales.

k-fold On divise les données initiales en k sous-ensemble de même taille. On itère k fois, et pour l'itération i , on choisit l'ensemble i comme ensemble de test et les $k - 1$ autres comme ensemble d'entraînement.

leave-one-out Cas particulier du k-fold où $k = N$, N étant le nombre d'objets présents dans les données. On a donc un seul objet par sous-ensemble. Cette approche utilise donc $N - 1$ données d'entraînement à chaque itération (peu de chance d'*underfitting*) et un seul objet de test (si l'objet ne passe pas le test, le modèle sera considéré comme très mauvais). On a donc énormément de calculs à faire (N modélisations) mais en moyenne le modèle résultant est bon (même si la variance de la qualité du modèle est haute dû aux tests à un seul objet).

3.6.5 Bootstrap

Contrairement aux méthodes précédentes, les objets de cette méthode peuvent être réutilisés plusieurs fois pour entraîner le modèle. En effet, la construction d'un extrait de *bootstrap*, qui sera utilisé comme ensemble d'entraînement se fait comme suit :

1. On tire au hasard un objet et on l'ajoute à l'extrait de *bootstrap*.
2. L'objet tiré est remplacé afin qu'il puisse peut-être être de nouveau tiré dans une prochaine itération.
3. Recommencer jusqu'à obtenir un ensemble de taille voulue (paramètre).

Une fois l'extrait construit, on l'utilise comme ensemble d'entraînement, les objets non présents dans l'extrait sont utilisés comme données de test. On itère b fois, ce qui implique que b extraits de *bootstrap* sont créés au cours de l'opération.

On estime que dans un extrait de taille N , N étant le nombre d'objet dans le jeu de données de base, 63,2% des données de base y sont présent (et donc que 36,8% des données sont des doublons).

La précision d'un modèle construit peut se calculer de plusieurs manières, la plus connue est le **.632 bootstrap**, définie comme suit avec ϵ_i la précision du modèle créé avec l'extrait de *bootstrap* de l'itération i , et acc_s la précision d'un modèle créé avec tous les objets des données de base en entraînement.

$$acc_{boot} = \frac{1}{b} \sum_{i=1}^b (0.632 \cdot \epsilon_i + 0.368 \cdot acc_s)$$

3.7 Méthode de comparaison de classificateurs

Pour comparer des résultats de classificateurs, surtout sur des jeux de test différents, il faut passer par des tests statistiques. Par exemple, prenons deux modèles : M_a et M_b . M_a a une meilleure précision que M_b , mais la précision de M_a a été calculé sur un jeu de test moins grand que celui de M_b . Il faut donc pouvoir mesurer la confiance que l'on peut accorder à la précision de M_a par rapport à la grandeur de son ensemble de test.

Pour connaître l'intervalle de confiance, il faut déterminer la distribution de probabilité qui domine la mesure de précision. Pour ce faire, on peut modéliser le problème de classification en une expérience binomiale:

1. Expérience de N essais indépendants donnant soit un succès, soit un échec.
2. La probabilité p d'obtenir un succès à chaque essai est constante.

Soit X le nombre de succès sur N essais, la probabilité d'en avoir v est la suivante:

$$P(X = v) = \binom{N}{v} p^v (1 - p)^{N-v}$$

Pour la classification, p est la vraie précision du modèle, X est le nombre d'instances bien classifiées, N le nombre total d'objets. Si on prends la précision empirique, normalisée par rapport à la taille du jeu de données de test, $\frac{X}{N}$ (moyenne = p , variance = $\frac{p(1-p)}{N}$), c'est aussi une distribution normale. Son intervalle de confiance est souvent approximé par une distribution normale pour N suffisamment grand :

$$P \left(-Z_{\alpha/2} \leq \frac{acc - p}{\sqrt{p(1-p)/N}} \leq Z_{1-\alpha/2} \right) = 1 - \alpha$$

Avec $-Z_{\alpha/2}$ et $Z_{1-\alpha/2}$ les bords supérieurs et inférieurs obtenus par distribution normale avec une confiance de $1 - \alpha$.

Les formules suivantes permettent alors de conclure sur la différence entre les précisions comparées/

$$I(= d_t) = d \pm z_{\alpha/2} \cdot \hat{\sigma} = [d - z_{\alpha/2} \cdot \hat{\sigma}, d + z_{\alpha/2} \cdot \hat{\sigma}]$$

$$d = e - f$$

$$1 - \alpha = 90\% \text{ (Certitude)} \Leftrightarrow \alpha = 10\%$$

$$z_{\alpha/2} = 1.65 \text{ (Confiance)}$$

$$\hat{\sigma} = \sqrt{\frac{e(1-e)}{n} + \frac{f(1-f)}{m}}$$

Avec e (resp. f) représente le pourcentage d'instances mal classifiées dans le modèle 1 (resp. 2) et n (resp. m) représente le nombre total d'instances classées par le modèle 1 (resp. 2).

Les précisions des deux modèles sont alors considérées comme significativement différentes si 0 n'est pas couvert par I .

Remarque – On peut aussi utiliser l'intervalle de confiance pour évaluer un élagage d'arbre. Comme le montre la figure 12. On y voit qu'on regarde à élaguer le noeud qui a été divisé selon l'attribut "health plan contribution". On calcule alors l'intervalle de confiance pour chacun des fils de ce noeud, en prenant en compte la précision du fils concernant la classe "bad". On a donc n le nombre d'objet total, X le nombre d'objet libellé "bad", et p_1 et p_2 les bornes, inférieure et supérieures, de l'intervalle de confiance.

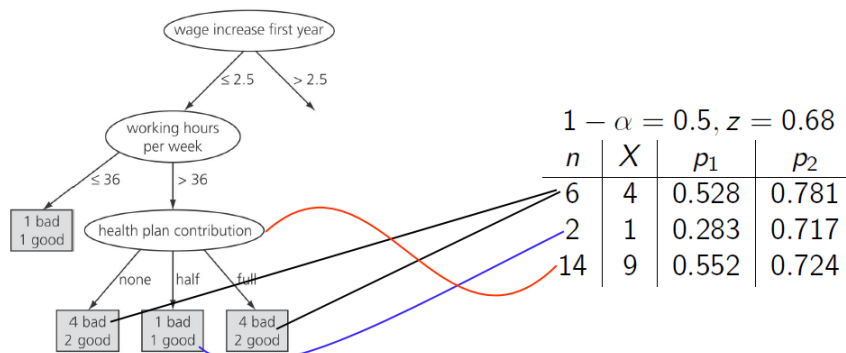


Figure 6.2 Pruning the labor negotiations decision tree.

Pessimistic precision estimate of “health plan contribution” is:

$$\frac{6}{14} \times 0.528 + \frac{2}{14} \times 0.283 + \frac{6}{14} \times 0.528 = 0.493$$

Since $0.493 < 0.552$, we prune away “health plan contribution”.

Source: Ian H. Witten and Eibe Frank: *Data Mining. Practical Machine Learning Tools and Techniques* (2nd Edition). Morgan Kaufmann, 2005

Figure 12: L'élitage est dans ce cas utile devrait être fait, car le split n'apporte pas d'informations utiles.

On compare donc ensuite la somme pondérée de la borne inférieure des fils (estimation pessimiste) avec la borne inférieure du parent. Si cette dernière est plus élevée (comme dans l'exemple), cela veut dire que la division n'était pas utile et fait perdre en confiance, on peut donc élaguer.

4 Types d'algorithmes

Il existe plusieurs type d'algorithmes construisant des modèles de classification. En voici quelques-uns.

4.1 Nearest Neighbours (IBk, Lazy)

Cette technique s'appuie sur la notion de distance entre les instances. Plusieurs distances peuvent être considérées (Euclidienne, Manhattan, ...). Voici quelques définitions de **distances** en fonction des types d'attributs.

- Un attribut numérique uniquement : différence.
- Plusieurs attributs numériques : normalisation des valeurs + distance Euclidienne, ou Manhattan.
- Attributs nominaux : 0 si valeur identique, 1 sinon.

Il reste ensuite à regrouper les objets en groupe en fonction de la distance qu'il y a entre chaque objets et leur k plus proches voisins.

4.2 Règles de classification

Classificateur qui consiste en une série de k règles, liées entre-elles par des "ET".

Exemple :

Si $a < 5$ et $b < 4$: classe = oui

Si $a > 2$ et $c > 3$: classe = non

Si $b > 3$ et $d > 2$: classe = oui

...

Pour le classificateur **OneR**, le but est de tester pleins de règles concernant chacune un seul attribut, chaque règle testées prenant les meilleurs valeurs "limites" de l'attribut pour classifier, et seule la meilleure règle est gardée.

Pour **Prism** par contre, on se concentre sur les classes. Prenons comme exemple la figure 13 comme exemple. Dans cet exemple, il y a deux classes : "oui", et "non", avec 5 objets dans chaque classe. On va ici s'intéresser à classer "non" au mieux. On va donc énumérer des règles simples (cf. figure 13), et regarder la précision de classification des instances correspondant à ces conditions.

Condition	Accuracy	Condition	Accuracy
outlook = sunny	3/5	humidity = high	4/7
outlook = overcast	0/4	humidity = normal	1/7
outlook = rainy	2/5	windy = false	2/8
temperature = hot	2/4	windy = true	3/6
temperature = mild	2/6		
temperature = cool	1/4		

Figure 13: Règle et précision de classification pour la classe "non"

La précision la plus élevée est 3/5. Nous choisissons donc la condition `outlook = sunny`. On va ensuite regarder s'il faut continuer à chercher des règles ou pas. La figure 14 montre les 5 instances correspondant à la condition `outlook = sunny` que l'on vient de choisir.

outlook	temperature	humidity	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
sunny	mild	high	false	no
sunny	cool	normal	false	yes
sunny	mild	normal	true	yes

Figure 14: On peut encore raffiner la condition en rajouter des conditions

Si on recommence à regarder pour quelle règle on classe le mieux les "non" pour les cinq instances restantes, on obtient le tableau de la figure 15.

Condition	Accuracy
<code>temperature = hot</code>	2/2
<code>temperature = mild</code>	1/2
<code>temperature = cool</code>	0/1
<code>humidity = high</code>	3/3
<code>humidity = normal</code>	0/2
<code>windy = false</code>	2/3
<code>windy = true</code>	1/2

Figure 15: L'égalité entre deux règles se brise par le le nombre d'instances couvertes

On remarque une égalité entre `temperature = hot` et `humidity = high`. Nous prendrons la deuxième condition car c'est elle des deux qui couvre le plus d'objets. Seulement, la règle entière si `outlook = sunny` et `humidity = high` ne couvre que 3 ces 5 instances "non". On recommence depuis le début, mais cette fois-ci en retirant les objets répondant à la règle déjà trouvée, etc... jusqu'à obtenir le maximum d'instances "non" bien classées. Puis, on peut recommencer le travail avec les classes "yes".

Remarque – Les règles trouvées par Prism sont *sound* mais pas *complete*. En effet, on est sûr que les règles sont vraies pour le jeu de données d'entraînement (*sound*) mais elles ne sont pas vraies pour n'importe quel jeu de données (non *complete*)

4.3 Arbres IBk

Arbres de décisions avec des règles sur les attributs. On peut noter que ces arbres sont parfois trop complexes pour rien (problème du sous arbres dupliqué), et que les règles s'y trouvant ne sont pas garantie *sound* ni *complete*.

On peut noter qu'il y a deux types de règles de manière générale (pas que pour les arbres). Ceux-ci sont expliqués sur la figure 16.

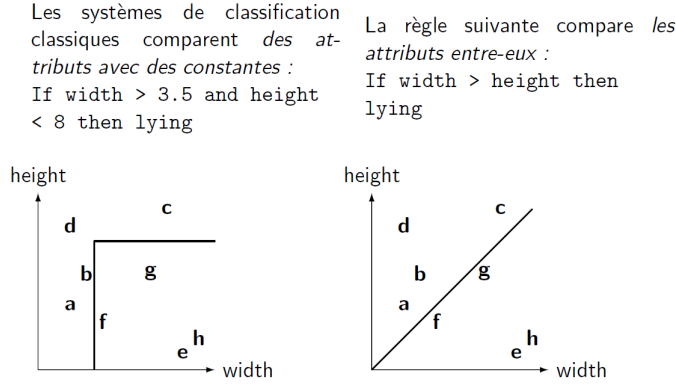


Figure 16: Il y a deux types de règles

4.4 Naive Bayes

Ce classificateur utilise la règle de Bayes. C'est une règle de probabilité qui dit :

$$Pr(H|E) = \frac{Pr(E|H) \cdot Pr(H)}{Pr(E)}$$

Avec $Pr(x|y)$ voulant dire la probabilité de x , sachant que y .

Pour classifier, on utilisera cette règle afin de trouver la probabilité pour l'objet k d'appartenir à une classe i , en sachant les attributs de cet objet ($=Pr(Class_i|Attributes_k)$). Normalement, un tel calcul est très coûteux et parfois infaisable quand on a pas toutes les informations, car :

$$Pr(Class_i|Attributes_k) = \frac{Pr(Att_{1_k} \& Att_{2_k} \& \dots \& Att_{j_k}|Class_i) \cdot Pr(Class_i)}{Pr(Attributes_k)}$$

Mais on admet une certaine naïveté, en pensant que tous les attributs sont indépendants l'un de l'autre, et donc que :

$$Pr(Class_i|Attributes_k) = \frac{Pr(Att_{1_k}|Class_i) \cdot Pr(Att_{2_k}|Class_i) \cdot \dots \cdot Pr(Att_{j_k}|Class_i) \cdot Pr(Class_i)}{Pr(Attributes_k)}$$

Avec Att_{l_k} l'attribut l de l'objet k .

Part II

Analyse par association

Le but de ce chapitre est de développer des techniques permettant de déduire des liens entre des entrées de jeu de données. Un exemple répandu est celui du panier d'achat d'un client de magasin, on pourrait trouver des liens entre l'achat de certaines marchandises avec l'achat d'autres. On appelle ces liens des **règles d'association**. Exemple :

$$\{Langes\} \longrightarrow \{Bieres\}$$

Ce qui impliquerait qu'il y a une forte corrélation entre la vente de langes et la vente de bières, ce qui veut dire que les gens qui achètent des langes ont de grandes chances d'acheter de la bière en même temps.

• Domaines d'application

En plus de ces paniers de marchandises, on trouve l'utilité dans la bio-informatique, les diagnostics médicaux, le Web mining et l'analyse de données scientifiques. Par exemple, en géologie, des modèles d'associations peuvent révéler d'intéressantes relations entre les océans, les continents, et les processus atmosphériques.

• Définition du problème

Nous allons définir ci-dessous les différents termes et hypothèses que nous utiliserons.

Représentation binaire Les jeux de données traités seront supposés représentables de manière binaire. Dans l'exemple du panier d'achat, on aura une variable par type de produit, cette variable vaudra 1 si l'objet a été acheté, et 0 sinon. Nous pouvons noter que dans ce cas, la variable est asymétrique de par le fait que si la variable est à 0, elle ne sera pas utilisée, en effet, il n'est pas très utile de savoir que nous n'avons pas acheté un certain objet ...

Itemsets Soit $I = \{i_1, i_2, \dots, i_d\}$ être l'ensemble de tous les objets (*itemset*) considérés et $T = \{t_1, t_2, \dots, t_N\}$ être l'ensemble des transactions effectuées. Chaque transaction de t_i est un sous-ensemble de I . Si un *itemset* comporte k objets, il est appelé *k-itemset*. L'*itemset* vide est accepté.

Support Count Soit la largeur de la transaction (*transaction width*) étant le nombre d'objets présents dans la transaction. Le *support count* ($\sigma(X)$) est défini comme suit :

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$$

On a donc le nombre de transactions dans T dans lesquels apparaît l'ensemble X d'objets.

Règles d'association Implication écrite sous la forme $X \longrightarrow Y$ dans laquelle X et Y sont disjoints ($X \cap Y = \emptyset$). La solidité d'une règle est mesurée en terme de son *Support* et de sa *Confidence*. Soit N le nombre total de transaction ($= |T|$), on a

$$\text{Support, } s(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{N}$$

$$\text{Confidence, } c(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

On peut noter qu'une règle ayant un faible *Support* peut être dû à la chance, et que la *Confidence* définit la confiance que l'on peut avoir dans la règle. La *Confidence* donne aussi une estimation de $\Pr(Y|X)$. De plus, le *support* d'une règle $X \longrightarrow Y$ ne dépend que de l'ensemble $X \cup Y$. Donc, les six règles suivantes ont le même *support* :

$$\begin{aligned} \{\text{Beer, Diapers}\} &\longrightarrow \{\text{Milk}\}, & \{\text{Beer, Milk}\} &\longrightarrow \{\text{Diapers}\}, \\ \{\text{Diapers, Milk}\} &\longrightarrow \{\text{Beer}\}, & \{\text{Beer}\} &\longrightarrow \{\text{Diapers, Milk}\}, \\ \{\text{Milk}\} &\longrightarrow \{\text{Beer, Diapers}\}, & \{\text{Diapers}\} &\longrightarrow \{\text{Beer, Milk}\}. \end{aligned}$$

Il faut noter que l'inférence faite par une règle d'association ne donne pas de causalité. L'achat des langes n'est pas la cause de l'achat de bières.

5 Découverte de règles

Découverte de règles d'association Étant donné un ensemble de transaction T , trouver toutes les règles ayant un *support* $\geq \text{minsup}$ et une *confidence* $\geq \text{minconf}$, où *minsup* et *minconf* sont deux seuils à définir.

Un ensemble de transaction qui contient d objets peut générer R règles au total.

$$R = 3^d - 2^{d+1} + 1$$

On peut le prouver de la manière suivante. Soit le tableau $[L, R, L, X, X, \dots, X]$ représentant une règle. Les indices du tableau sont les différents objets, L veut dire "l'objet se trouve dans la partie gauche de la règle", R veut dire "l'objet se trouve dans la partie droite de la règle" et X veut dire "ne fait pas partie de la règle". Le tableau donné donnerait donc la règle $\{obj_0, obj_2\} \longrightarrow \{obj_1\}$.

On a donc 3 possibilités par case, donc 3^d . Mais on ne peut pas avoir que des X et des R , ni que des X et des L , on doit donc retirer des 3^d possibilité 2 fois 2^d (X et L ou X et R = 2 possibilités par case), et $2 \cdot 2^d = 2^{d+1}$. Et enfin, on se rend compte que dans ce qu'on a retiré, il y avait deux fois l'ensemble de contenant que des X , on doit donc rajouter 1 pour compenser. On a donc bien $3^d - 2^{d+1} + 1$.

On peut conclure qu'il est impossible de toutes les générer par force brute, il faut donc trouver des astuces pour générer de bonnes règles en un temps raisonnable. On a donc deux tâches principales à remplir :

- (1) **Génération de Frequent Itemsets** Trouver tous les itemsets dont le *support* satisfait *minsup*.
- (2) **Génération de règle** Extraire les règles à haute *confidence*, qui seront appelées les règles "fortes". Généralement moins coûteux que (1).

6 Génération de frequent itemsets

Un ensemble de k objets peut potentiellement générer $2^k - 1$ itemsets fréquents, étant donné que dans un itemset candidat, un objet peut soit être présent, soit ne pas l'être. Un algorithme de force brute devrait donc effectuer $O((2^k - 1)Nw)$ opérations pour calculer les *support* de tous les itemsets possibles, car il faut comparer les itemsets possibles avec les N transaction qui ont une largeur de maximum w . Il y a deux manières de réduire le nombre de calculs.

1. Réduire le nombre d'itemset à comparer (cf. la section 6.1).
2. Réduire le nombre de comparaisons en utilisant des structures de données avancées.

6.1 Le principe *Apriori*

Ce principe permet de réduire le nombre d'itemsets à comparer aux transactions. Ce principe dit qu'un itemset fréquent ne contient que des sous-ensembles fréquents. Et donc, ce qui nous intéresse, que si un itemset est non-fréquent, tous ses super-ensembles sont non-fréquents.

Ceci est rendu possible grâce à l'anti-monotonie de la mesure *support*.

Propriété de Monotonie Soit I un ensemble d'objets, et $J = 2^I$ l'ensemble des parties de I . Une mesure f est monotone (ou bornée supérieurement) si:

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(X) \leq f(Y)$$

Par analogie, une propriété f est non-monotone si:

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(X) \geq f(Y)$$

Le principe *Apriori* peut être appliqué à n'importe quelle mesure non-monotone incorporée à l'algorithme.

6.2 Génération d'itemsets fréquents avec *Apriori*

La figure 17 illustre la génération d'éléments.

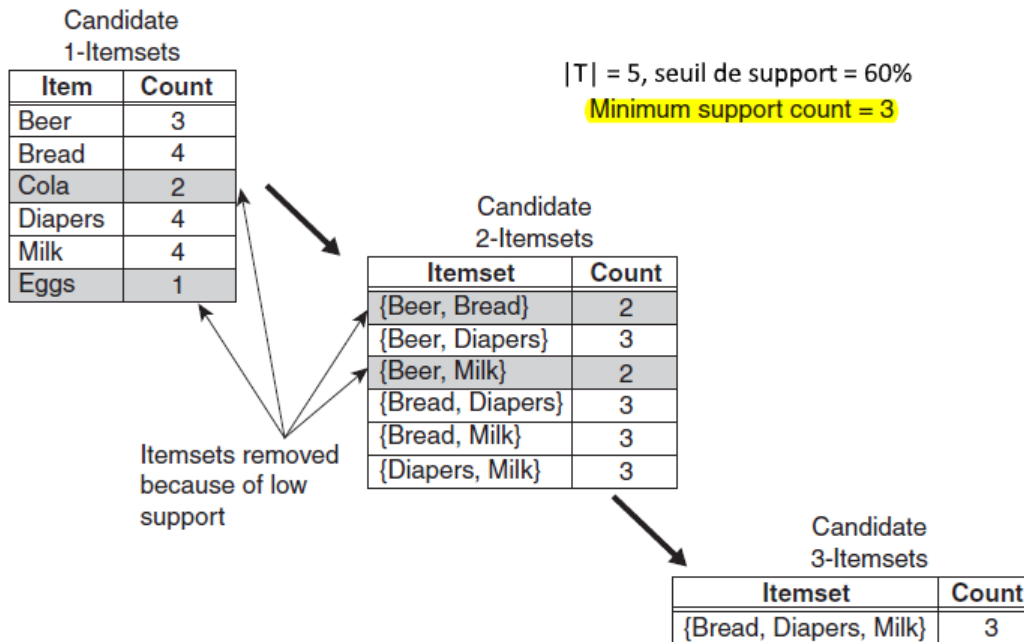


Figure 17

Algorithm	Frequent itemset generation of the <i>Apriori</i> algorithm.
1:	$k = 1$.
2:	$F_k = \{ i \mid i \in I \wedge \sigma(\{i\}) \geq N \times \text{minsup} \}$. {Find all frequent 1-itemsets}
3:	repeat
4:	$k = k + 1$.
5:	$C_k = \text{apriori-gen}(F_{k-1})$. {Generate candidate itemsets}
6:	for each transaction $t \in T$ do
7:	$C_t = \text{subset}(C_k, t)$. {Identify all candidates that belong to t }
8:	for each candidate itemset $c \in C_t$ do
9:	$\sigma(c) = \sigma(c) + 1$. {Increment support count}
10:	end for
11:	end for
12:	$F_k = \{ c \mid c \in C_k \wedge \sigma(c) \geq N \times \text{minsup} \}$. {Extract the frequent k -itemsets}
13:	until $F_k = \emptyset$
14:	Result = $\bigcup F_k$.

Figure 18: L'algorithme permettant d'extraire les itemsets fréquents

L'implémentation de la fonction *apriori-gen* est couverte dans la partie 6.3 et la fonction *subset* est expliquée dans la section 6.4. L'algorithme prend $k_{max} + 1$ itération, où k_{max} est la taille du plus grand itemset fréquent.

6.3 Génération de candidats et élagage

La fonction *apriori-gen* de l'algorithme Apriori génère des candidats suivants deux opérations:

- (1) **Génération de candidat** Génère des candidats de taille k à partir des itemsets de taille $k-1$ déjà générés.
- (2) **Élagage des candidats** Supprime les candidats n'ayant pas assez de *support*.

L'étape (2) doit en fait regarder tous les itemsets de taille k générés n (1), et supprimer ceux qui contiennent des sous-itemsets (pas besoin de regarder ceux de taille 1) non-fréquents, par le principe d'*Apriori*. Ceci représente $O(k)$ opérations pour chaque itemset de taille k , cependant, ce nombre sera réduit plus tard.

Une bonne méthode pour générer des itemsets candidats doit suivre les directives suivantes:

- (a) Elle devrait éviter de générer trop de candidats inutiles (avec sous-ensemble non-fréquent).
- (b) Elle doit être certaine de ne pas retirer de candidats valides
- (c) Elle ne devrait pas générer plus d'une fois le même candidat. En effet, il est possible de créer $\{a, b, c\}$ en fusionnant $\{a, b\}$ et $\{c\}$ mais aussi en fusionnant $\{a, c\}$ et $\{b\}$, il faut donc éviter les doublons.

6.3.1 Force brute

Le principe est de considérer tous les itemsets de taille k comme candidat potentiel. On a donc $\binom{d}{k}$ itemsets générés à chaque étape, et vu que $O(k)$ opérations doivent être effectuées par itemset, il y a $O(d \cdot 2^{d-1})$ opérations.

6.3.2 $F_{k-1} \times F_1$

une autre méthode de génération est de créer des itemsets de taille k en prenant les itemsets de taille $k-1$ et d'y rajouter un par un les objets fréquents (itemset de taille 1).

La méthode respecte (b), mais pas (c). Une manière de respecter (c) serait de s'assurer de respecter l'ordre lexicographique lors de la création de nouveaux itemsets.

Exemple : $\{a, b\}$ peut être augmenté avec $\{c\}$ pour former $\{a, b, c\}$, mais pas $\{a, c\}$ et $\{b\}$ car b vient se rajouter après c , qui est plus grand d'un point de vue lexicographique.

La méthode génère cependant pas mal d'itemsets inutiles. En effet, admettons que l'on sache que $\{a, c\}$ est non-fréquent, la méthode va tout de même créer $\{a, b, c\}$ en fusionnant $\{a, b\}$ et $\{c\}$.

6.3.3 $F_{k-1} \times F_{k-1}$

C'est la méthode utilisée en pratique, elle consiste à fusionner deux itemsets de taille $k - 1$ qui ont chacun $k - 2$ éléments en commun, tout en gardant l'ordre lexicographique discuté au point précédent.

Exemple : on fusionnerait $\{a, b, c\}$ avec $\{a, b, d\}$ pour obtenir $\{a, b, c, d\}$ car ils ont a et b en commun. Mais pour ce faire, il faut bien vérifier que le sous-ensemble commun de taille $k - 2$ est fréquent.

6.4 Compter le *support count*

Le principe de cette étape est de compter dans combien de transaction se retrouve un itemset candidat. Les étapes d'avant vont générer un ensemble itemsets de taille fixe k . On va donc préférer énumérer tous les sous-ensemble de taille k contenu dans chaque transaction et incrémenter les itemsets s'y retrouvant. Ce faisant, on ne va parcourir qu'une seule fois toutes les transactions. La figure 19 illustre la technique utilisée.

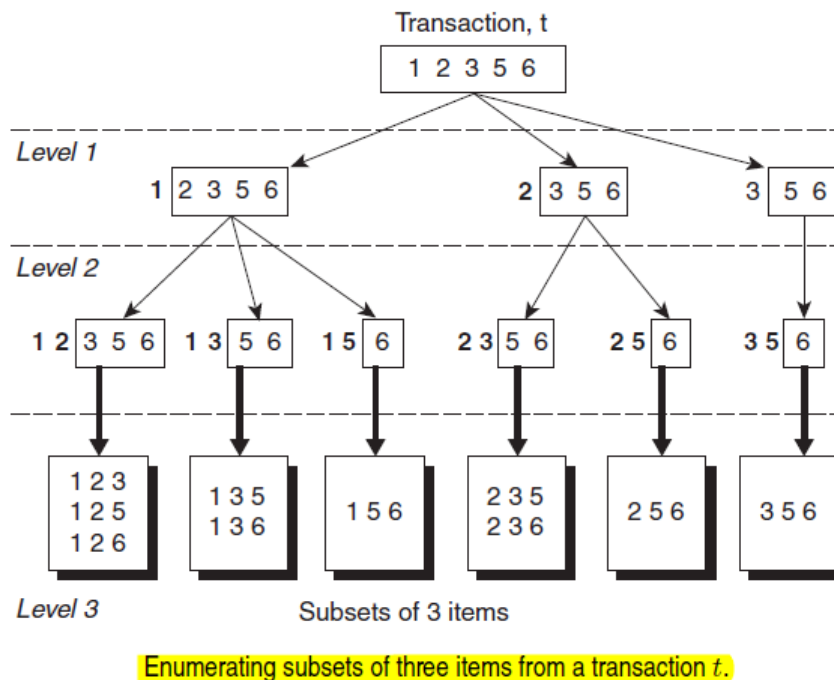


Figure 19: Chaque noeud du niveau j va se fixer un préfixe qui lui est propre de taille j

Il reste alors à retrouver les itemsets candidats à incrémenter. Une méthode pour faire ça est de "pousser" les sous-ensemble générés par l'énumération (e.g. ensembles de taille 3 de la figure 19) dans un arbre de hachage Dans lequel les candidats sont divisés selon une table de hachage.. Ceci est illustré sur la figure 20.

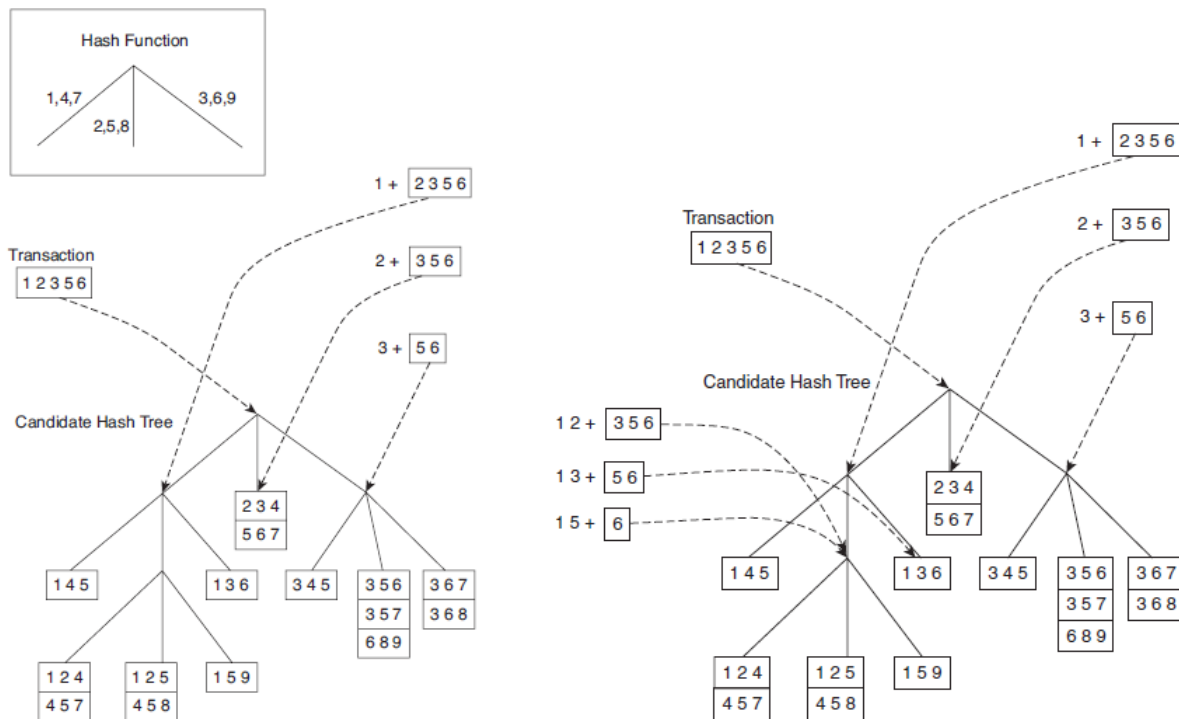


Figure 20: Dés lors, pas besoin de comparer tous les candidats, mais juste ceux présents dans la partie de l'arbre où le sous-ensemble a été "poussé"

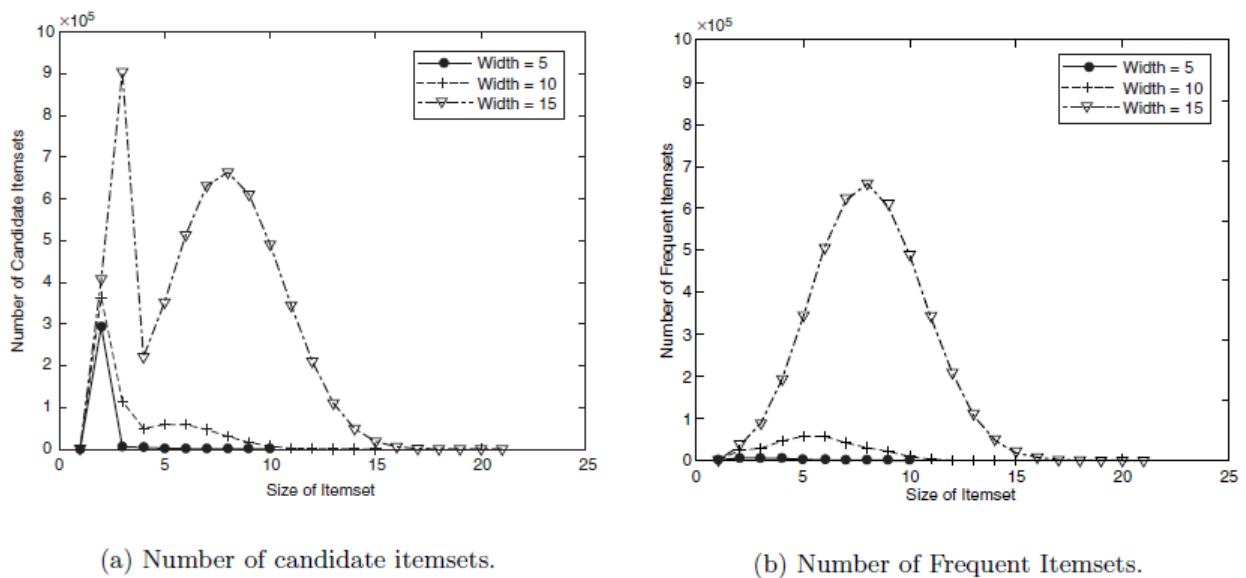
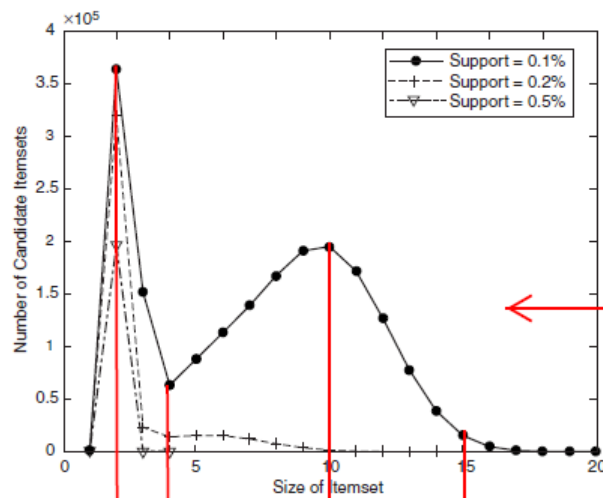
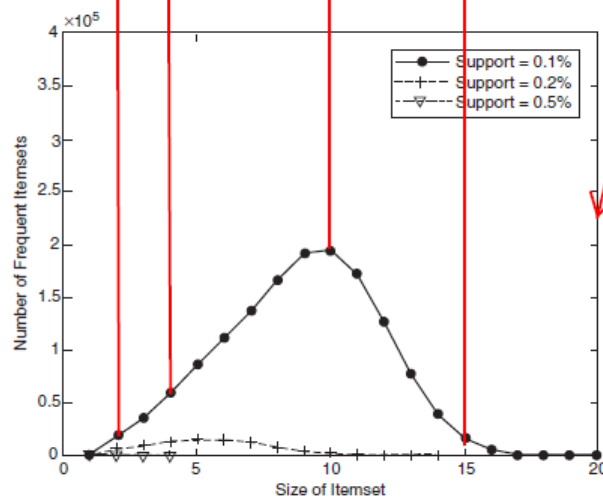


Figure 21: Si la largeur moyenne d'une transaction est de 10, les itemsets fréquents de taille bien plus grande que 10 sont peu probables



(a) Number of candidate itemsets.

La courbe représente l'évolution de la sélection des candidats par l'algorithme à sélection de préfixe commun $F_{k-1} \times F_{k-1}$. Donc, l'étape de taille 2 ne peut pas sélectionner selon les préfixes de l'étape $k-2$ car ils n'existent pas encore. On ne peut donc pas encore filtrer par itemsets de taille $k-2$ fréquents. A partir de la taille 3, le filtre par préfixe fréquent s'applique.



(b) Number of frequent itemsets.

En réalité, beaucoup d'ensembles sont fréquents. Donc la courbe représente la croissance du nombre de sous ensemble selon la taille de celui-ci.

Effect of support threshold on the number of candidate and frequent itemsets.

Figure 22: Important

7 Génération de règles d'associations

Cette partie est moins importante à optimiser car **on a plus besoin d'accéder à la base de donnée**, ce qui prenait du temps. En effet, on connaît, les itemsets fréquents, grâce à l'étape précédente, et pour calculer la *confidence*, on a juste besoin du support count des règles, ce qui est déjà calculé.

Chaque itemset candidat Y peut générer $2^k - 2$ règle. **On peut le prouver**. En effet, Chaque objets d' Y doit se retrouver dans la règle, chaque objet peut donc être soit à gauche, soit à droite de la règle. on a donc 2^k possibilités de combinaisons, mais on ne doit pas considérer les règle $\{\} \rightarrow Y$ et $Y \rightarrow \{\}$, on doit donc en retirer 2.

On peut donc générer des règles avec Y en découpant Y en deux sous-ensembles : X et $Y \setminus X$ afin de créer la règle $X \rightarrow Y \setminus X$.

7.1 Élagage basé sur la *confidence*

On ne peut plus se baser sur le principe d'*Apriori*, car la mesure de *confidence* n'est pas anti-monotone. Néanmoins, en comparant les règles générées par le même itemset fréquent Y , **le théorème suivant s'applique**:

Si une règle $X \rightarrow Y \setminus X$ ne satisfait pas le seuil *minconf*,
 Alors, $\forall X' \subseteq X, X' \rightarrow Y \setminus X'$ ne satisfait pas non plus le seuil *minconf*

7.2 Génération de règle dans l'algorithme Apriori

L'idée est de fusionner les parties droites de règles avec une *confidence* élevée pour en former des nouvelles, qui seront elles-mêmes de confiance de par le théorème précédent.

Algorithm 6.2 Rule generation of the *Apriori* algorithm.

```

1: for each frequent  $k$ -itemset  $f_k, k \geq 2$  do
2:    $H_1 = \{i \mid i \in f_k\}$       {1-item consequents of the rule.}
3:   call ap-genrules( $f_k, H_1$ .)
4: end for
```

Figure 23: Génération de règles à partir de tous les itemsets fréquents générés à l'étape précédente.

Algorithm 6.3 Procedure **ap-genrules**(f_k, H_m).

```

1:  $k = |f_k|$    {size of frequent itemset.}
2:  $m = |H_m|$    {size of rule consequent.}
3: if  $k > m + 1$  then
4:    $H_{m+1} = \text{apriori-gen}(H_m)$ .
5:   for each  $h_{m+1} \in H_{m+1}$  do
6:      $\text{conf} = \sigma(f_k) / \sigma(f_k - h_{m+1})$ .
7:     if  $\text{conf} \geq \text{minconf}$  then
8:       output the rule  $(f_k - h_{m+1}) \rightarrow h_{m+1}$ .
9:     else
10:      delete  $h_{m+1}$  from  $H_{m+1}$ .
11:    end if
12:  end for
13:  call ap-genrules( $f_k, H_{m+1}$ .)
14: end if

```

H_m est un ensemble de règles de confiance avec m éléments dans la partie droite des règles

f_k est un itemset fréquent

Pas de faute, $m =$ taille de la partie droite de la règle

On génère par exemple,
 $A \rightarrow BCD$
 grâce aux règles de H_m :
 $AD \rightarrow BC$
 $AC \rightarrow BD$
 $AB \rightarrow CD$

La règle générée est de confiance par le théorème

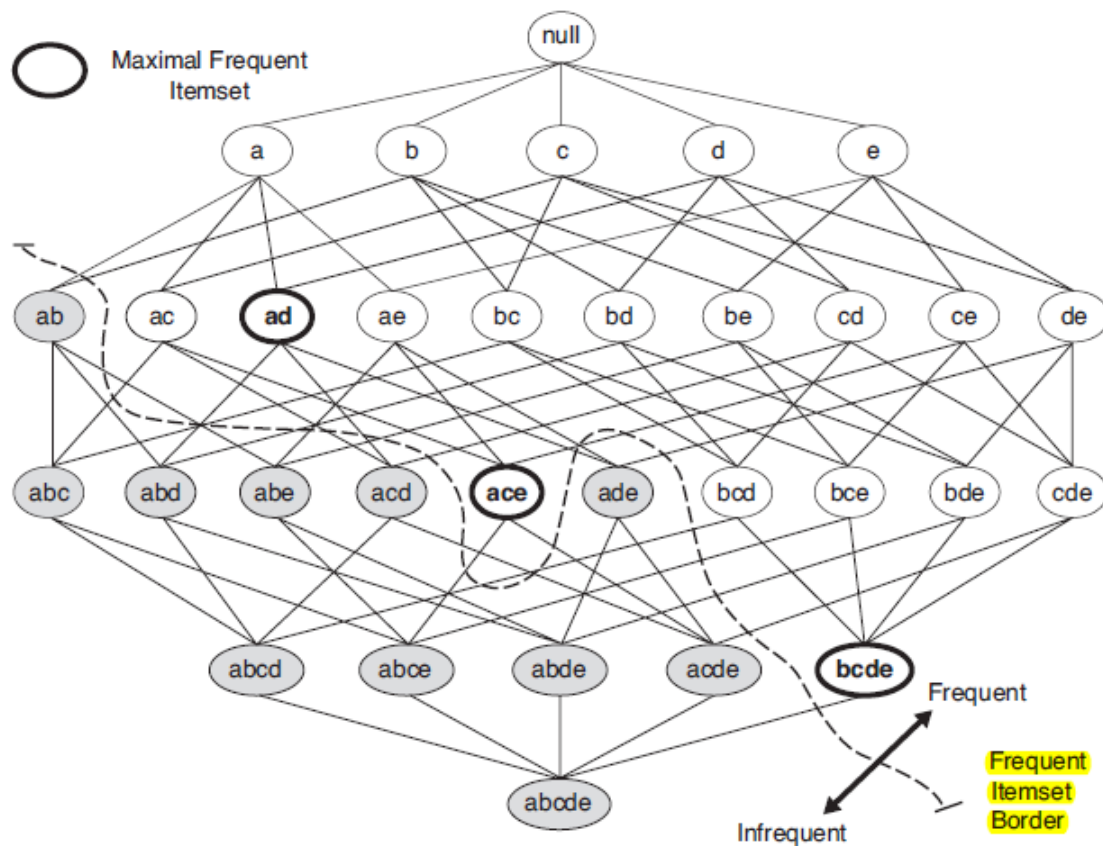
Figure 24: Génération de règles de confiance à partir de règles de confiance plus petites.

8 Représentations compactes des itemsets fréquents

Le nombre de transaction en pratique peut être très élevé. C'est pourquoi il est utile de rendre compact l'ensemble des itemsets fréquents.

8.1 Maximal Frequent Itemset

Maximal Frequent Itemset Itemset fréquent pour lequel aucun de ses super-ensembles immédiat ne sont fréquents.



Maximal frequent itemset.

Figure 25

On voit ici qu'on peut se limiter à donner les itemset en gras sur la figure 25, car en prenant tous leurs sous-ensemble, on obtient tous les itemsets fréquents. La **limitation** de cette représentation est qu'on abandonne le support count des sous-ensembles, ce qui est embêtant pour calculer la *confidence* des règles générées par ces premiers.

8.2 Closed Frequent Itemsets

Cette représentation propose un **ensemble compact sans abandonner le support count des itemset.**

Closed Itemset Un itemset X est fermé si aucun des super-ensembles immédiats de X n'a exactement le même support count que X .

On peut donc noter que si $\{A\}$ est non-fermé, alors, il existe un objet C t.q. la règle $A \rightarrow C$ a une confiance de 100%. En effet, si $\{A\}$ n'est pas fermé, ça veut dire qu'il y a un autre objet (C) t.q. A n'apparaît jamais sans C .

En effet, le fait d'avoir le même support count pour un super-ensemble veut dire que ce super-ensemble revient autant de fois dans les transactions que l'ensemble.

Exemple : Si on a $\sigma(\{X, Y\}) = 9$ et qu'on a $\sigma(\{X, Y, Z\}) = 9$, ça veut dire qu'à chaque fois qu'on a $\{X, Y\}$ dans une transaction, Z est également présent. On peut donc déduire que la confiance (*confidence*) de la règle $\{X, Y\} \rightarrow Z$ est de 100%.

Closed Frequent Itemset Un itemset est fermé et fréquent s'il est fermé et que son support dépasse *minsup*.

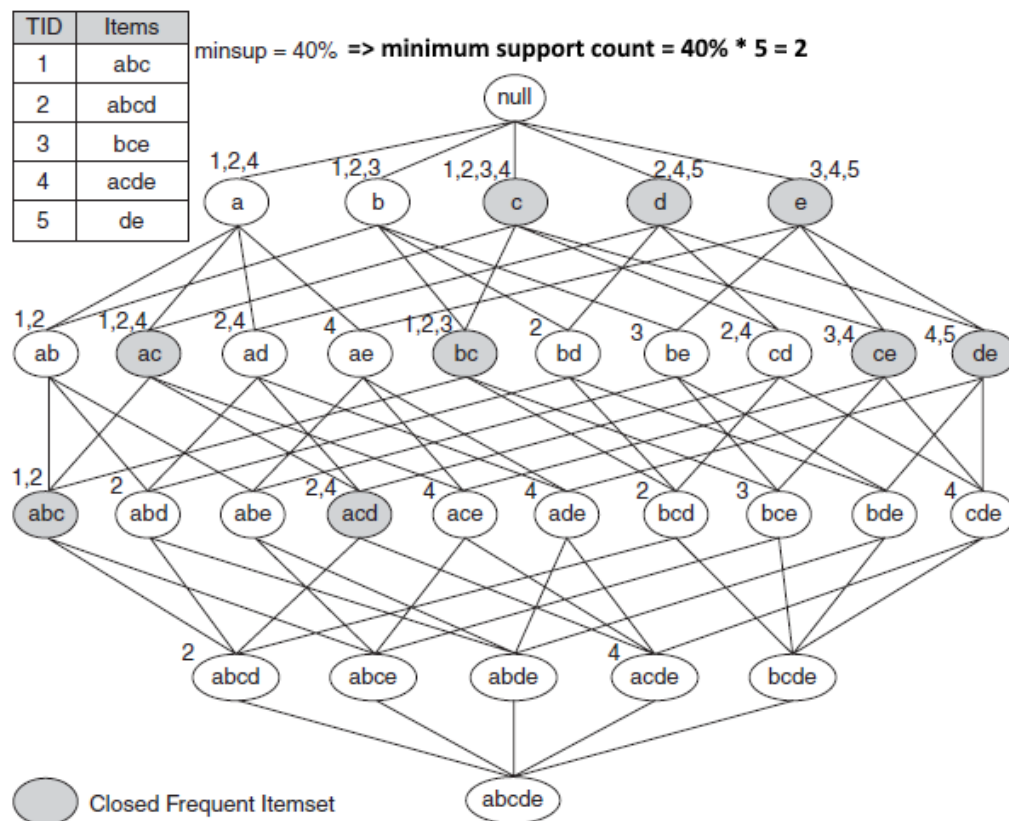


Figure 26

Le support count des itemsets fréquents fermés est sauvegardé dans la représentation.

Le support count de n'importe quel itemset fréquent non-fermé peut être obtenu en prenant le max des support count de ses super-ensembles directs. L'algorithme retrouvant le support count de tous les itemsets fréquents démarre donc de la plus grande taille de d'itemsets fréquents. En effet, l'algorithme a besoin de connaître le support count des super-ensembles avant de passer aux sous-ensembles.

Preuve : si $\{a, b\}$ est ouvert, on sait qu'il y a un de ces super-ensembles immédiats qui a exactement le même support count que lui (par la définition de non-fermé). La question serait alors de savoir duquel il s'agit. Mais on sait que le support count d'un super ensemble ne peut pas être plus grand que le support count de l'ensemble lui-même (anti-monotonie). Donc, le max des support count de ses super-ensembles immédiats sera son propre support count.

On peut noter que des règles peuvent être détectées comme redondantes grâce à cette notion d'itemsets fréquents fermés. En effet, si $\{b\}$ n'est pas fermé, mais que $\{b, c\}$ l'est, la règle $\{b\} \rightarrow X$ est redondante car elle a le même support et la même confiance que la règle $\{b, c\} \rightarrow X$. Ces règles redondantes ne sont donc pas générées.

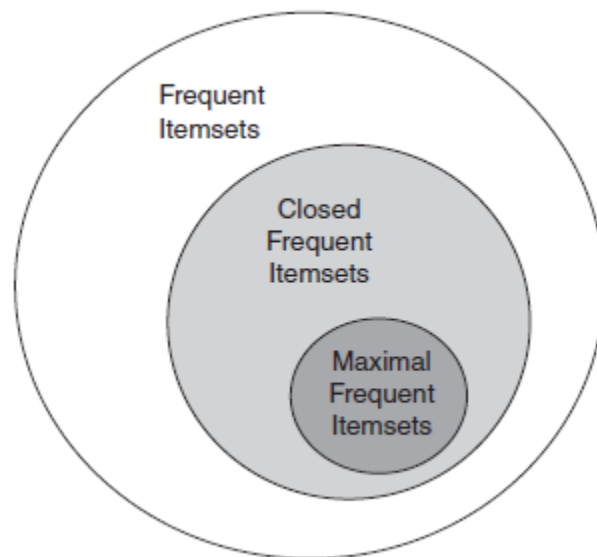


Figure 27: Si un ensemble est maximal, il est forcément fermé car sinon, il aurait un ensemble maximal comme super-ensemble (contradiction).

9 Optimiser Apriori

Bien que déjà pas mal, l'algorithme Apriori fait beaucoup d'accès à la base de données quand il compte les support count. De plus, lorsque les largeurs de transaction augmentent, le temps de calcul augmente aussi de manière significative. Nous allons donc voir quelles sont les améliorations que nous pouvons apporter à l'ordre dans lequel les itemsets sont parcourus lors de la sélection des itemsets fréquents..

9.1 Traverser un treillis

Le treillis formé par les itemsets de l'ensemble des transactions peut être traversé afin d'y trouver les itemsets fréquents. Tout repose alors sur la technique de recherche utilisée. Quelques unes de ces stratégies sont expliquées ci-dessous.

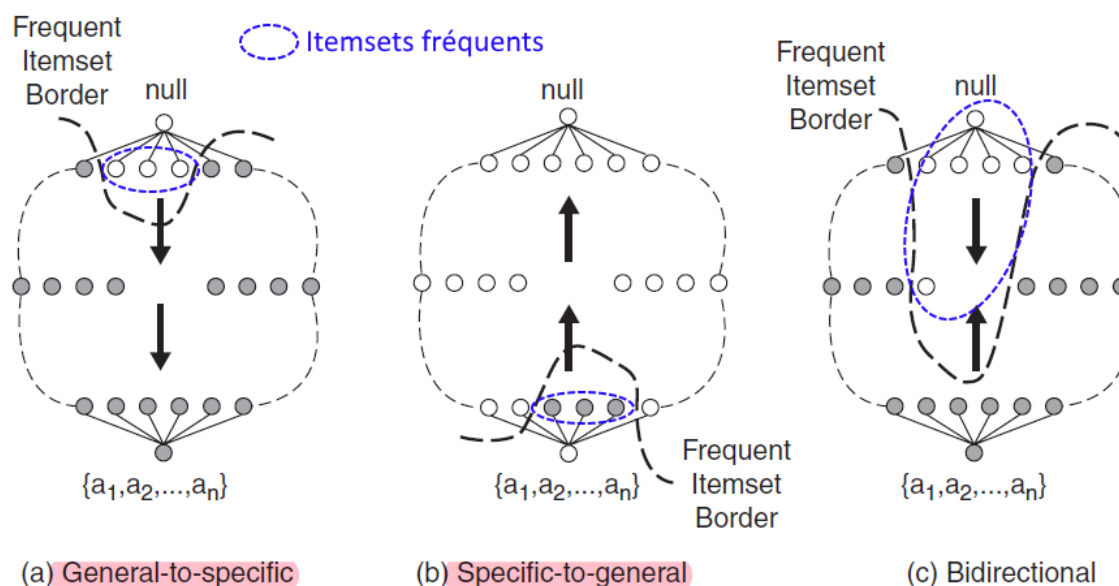


Figure 28

9.1.1 General-To-Specific

On démarre de petits itemsets (1 élément, puis 2, ...) pour terminer dans les grands itemsets. Cette méthode est efficace lorsque les itemsets fréquents sont petits. Dès lors, la frontière fréquents/non-fréquents se trouve en haut du treillis (cf. figure 28).

9.1.2 Specific-To-General

À l'inverse de la recherche précédente, on démarre cette fois des grand itemsets pour terminer dans les plus petits. Cette méthode est efficace lorsque les itemsets fréquents sont grands,

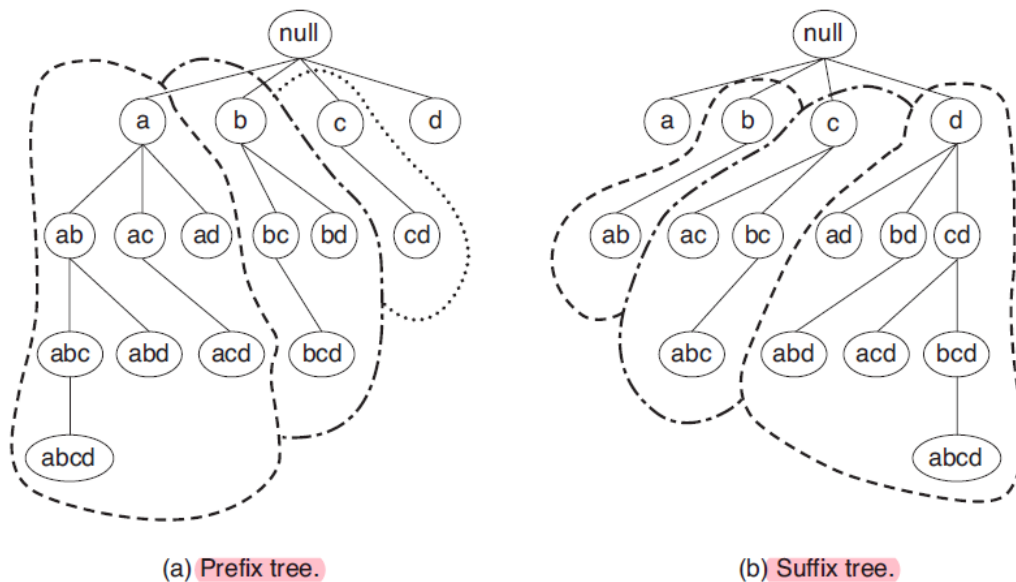
et donc généralement **quand on a de grandes transactions**. Dès lors, la frontière fréquents/non-fréquents se trouve en bas du treillis (cf. figure 28).

9.1.3 Bidirectionnel

On peut combiner les deux techniques précédentes, mais ça demande plus d'espace mémoire pour l'exécuter. Utile pour les transactions de tailles très variantes (beaucoup de grandes transaction et beaucoup de petites).

9.1.4 Classes d'équivalence

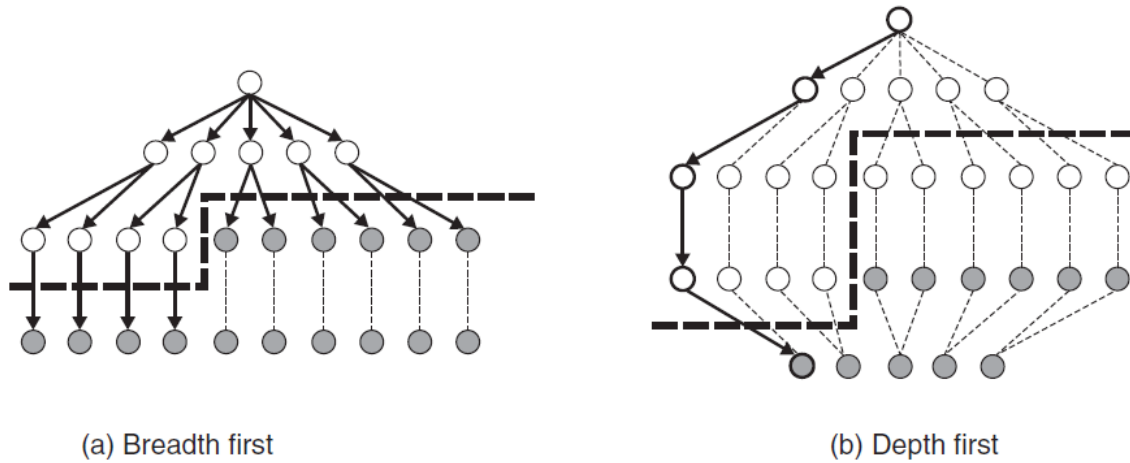
Pour parcourir le treillis plus efficacement, les objets peuvent y être regroupé par classes d'équivalence, c'est-à-dire par une propriété que tout le groupe a en commun, comme un **préfixe** ou un **suffixe**. **On peut noter que l'algorithme *Apriori* considérait les classes d'équivalence de taille**. On y prenait donc d'abord tous les itemsets de taille 1, puis on passait à ceux de taille 2, ...



Equivalence classes based on the prefix and suffix labels of itemsets.

Figure 29

9.1.5 Profondeur / Largeur



Breadth-first and depth-first traversals.

Figure 30

L'algorithme *Apriori* traverse les itemsets en largeur, considérant les tailles inférieures et finissant dans les grandes tailles.

L'exploration en profondeur quand à elle est souvent utilisée en pratique car elle permet un grand élagage. On peut élaguer de deux manières, nous prendrons comme exemple la figure 31.

1. Si un itemsets X est trouvé fréquent, alors, tous les sous-arbres ne contenant que des sous-ensembles de X peuvent être ignorés pour la suite de l'exploration. Cet élagage utilise l'ordre lexicographique utilisé dans l'arbre. C'est l'exemple avec $\{b, c, d, e\}$. Par contre si $\{a, b, c\}$ est trouvé fréquent, on ne peut pas élaguer le sous-arbre $\{a, c\}$ car celui-ci contient des ensembles, comme $\{a, c, d, e\}$ par exemple, qui ne sont pas des sous-ensembles de $\{a, b, c\}$.
2. Si le support de $\{a, b, c\}$ est le même que le support de $\{a, b\}$, on peut ignorer les sous-arbres qui ont abd et abe comme racine car on est certains qu'ils ne contiendront pas d'itemsets fréquents maximaux.

On peut le prouver. En effet, si $\{a, b, c\}$ et $\{a, b\}$ ont le même support count, ça veut dire que dans une transaction, $\{a, b\}$ n'apparaît jamais sans c . On a dès lors qu' $\{a, b, d\}$ a le même support que $\{a, b, c, d\}$ qui aura été exploré dans le sous-arbre de racine $\{a, b, c\}$ de par l'ordre lexicographique respecté.

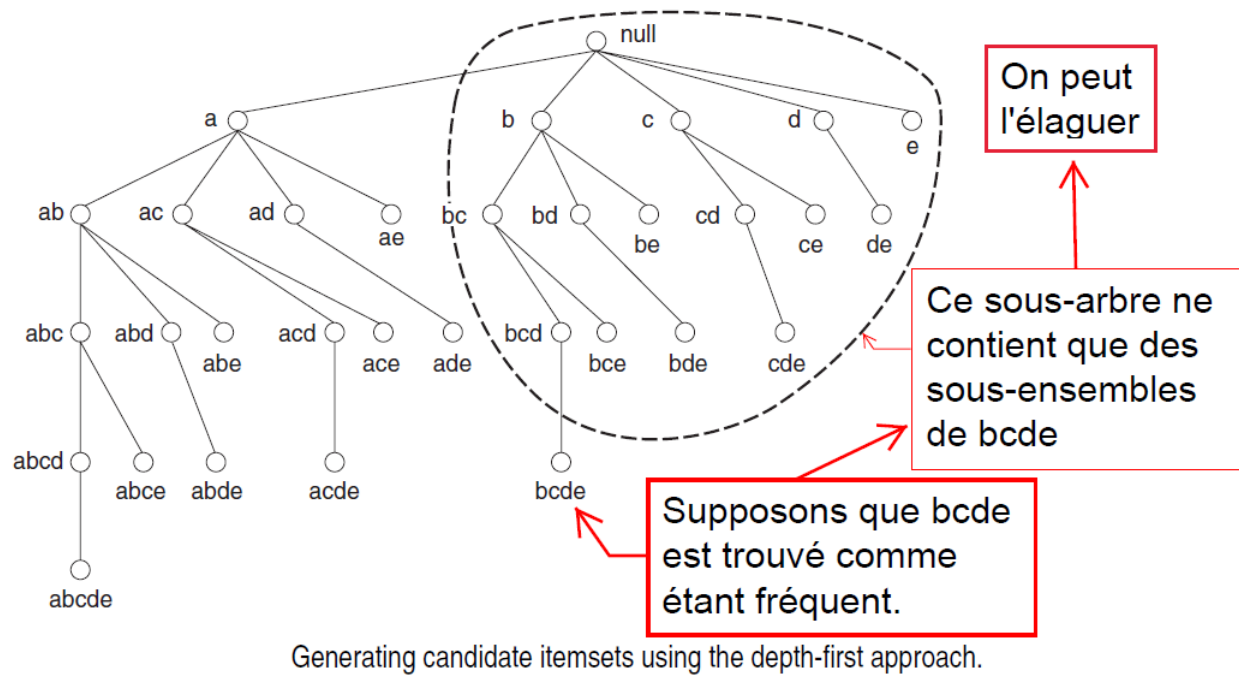


Figure 31

9.2 Représentation des transactions

On peut aussi chercher à optimiser la structure de données contenant les transactions. On distingue deux types principaux. Ils sont représentés sur la figure 32. La structure de gauche est généralement utilisée mais celle de droite a quelques avantages. En effet, la **structure verticale** stocke, pour chaque objet, la liste des transactions dans lesquels l'objet est présent. Cependant, la liste des ID de transactions est parfois trop grandes que pour être stockée de cette manière.

Horizontal Data Layout		Vertical Data Layout				
TID	Items	a	b	c	d	e
1	a,b,e	1	1	2	2	1
2	b,c,d		2	3	4	3
3	c,e			4	5	6
4	a,c,d			5	7	
5	a,b,c,d			6	8	
6	a,e			7	10	
7	a,b			8		
8	a,b,c			9		
9	a,c,d					
10	b					

Horizontal and vertical data format.

Figure 32

En partant d'une telle structure, il est aisé de définir des relations entre les objets et les itemsets dans lesquels ils sont contenus.

cover(i)

Soit un item i , $cover(i)$ = l'ensemble des transaction qui contiennent l'objet i . Exemple, dans la figure 32, $cover(e) = \{1, 3, 6\}$.

conditional cover, cocov($[s]$)

Soit un itemset s , et l'ensemble des objets I ;

$$cocov[s] = \{(i, T) | i \in I, \text{ et } i \text{ précède tous les él. de } s \text{ dans l'ordre lex., et } T = cover(i \cup s)\}$$

Exemple, dans la figure 32,
 $cocov[c] = \{(a, \{4, 8, 9\}), (b, \{2, 5, 8, 9\})\}$
 $cocov[be] = \{(a, \{1\})\}$

On peut noter que pour $j \cup s$ un itemset, $j \in I$, alors $\forall i \in I$, $i \prec j$, on a

$$cocov[j \cup s](i) = cocov[s](j) \cap cocov[s](i)$$

10 Algorithme FP-Growth

Cet algorithme utilise une structure de donnée un peu qui s'appelle le FP-Tree.

10.1 FP-Tree

Construire un FP-Tree comporte plusieurs étapes

1. On compte tout d'abord les support count des itemsets de taille 1, afin de **trier les objets par ordre décroissant** de support et pour écarter les non-fréquents.
2. On passe ensuite une deuxième fois sur les transactions pour construire les branches du FP-Tree, c'est ce qui est montré sur la figure 33. Cette étape **trie les transactions par préfixe commun, en coupant dans l'ordre des objets de l'étape 1.**

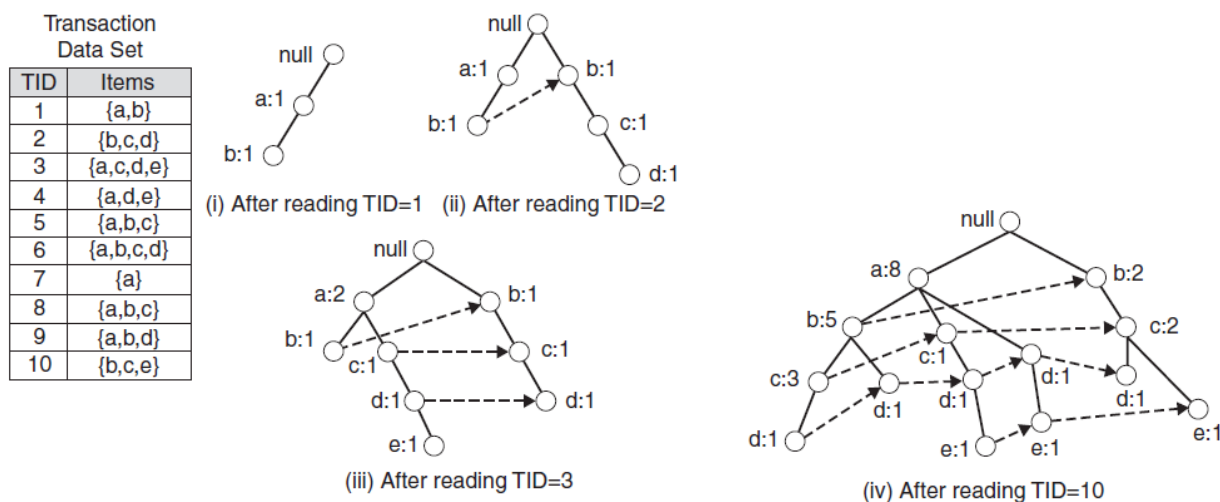


Figure 33

On peut noter que l'ordre décroissant des objets de l'étape 1. ne garanti pas une taille minimum de l'arbre. De plus, dans le pire des cas, cette représentation prend plus de place que le dataset non compressé (pointeurs, compteurs, ...), mais en général, il le compressé tout de même.

10.2 Génération d'itemsets fréquents

L'avantage de cette algorithme est de pouvoir déduire les itemsets fréquents directement depuis la structure de donnée.

Décomposition en sous-problème On commence par explorer le FP-Tree de manière "bottom-up", donc, dans l'exemple, de e à a , et on le divise en sous-problème, comme l'illustre la figure 34. C'est ici que les pointeurs entre objets vont être utiles. En effet, dès qu'on tombe sur un e dans l'arbre, on peut suivre les pointeurs pour trouver tous les autres e .

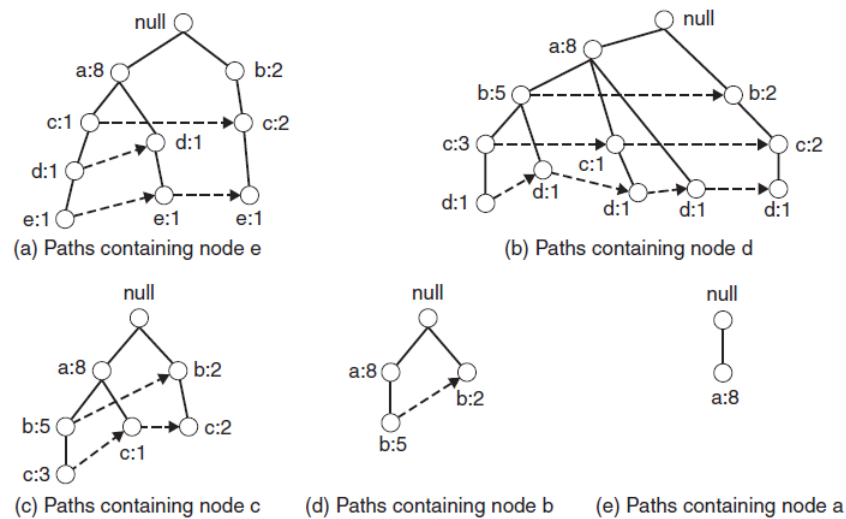


Figure 34

Division en FP-Tree conditionnel On va ensuite regarder, suffixe par suffixe, quel itemsets sont fréquents pour chaque sous-problème. L'obtention d'un FP-Tree conditionnel se fait par *cocov*, défini précédemment, les ID de transaction étant remplacés par des chemins de l'arbre. Ceci est illustré à la figure 35. On peut y remarquer que *b* est supprimé de l'arbre (b). Ceci s'explique par le fait que *b* ne se retrouve qu'une seule fois dans l'arbre (a) et le seul chemin partant de lui arrivant en *e* a un support count de 1, on en conclut donc que tout chemin partant de *b* vers *e* est non fréquent. Comme chaque sous-problème est disjoint, on ne génère pas d'itemsets dupliqués.

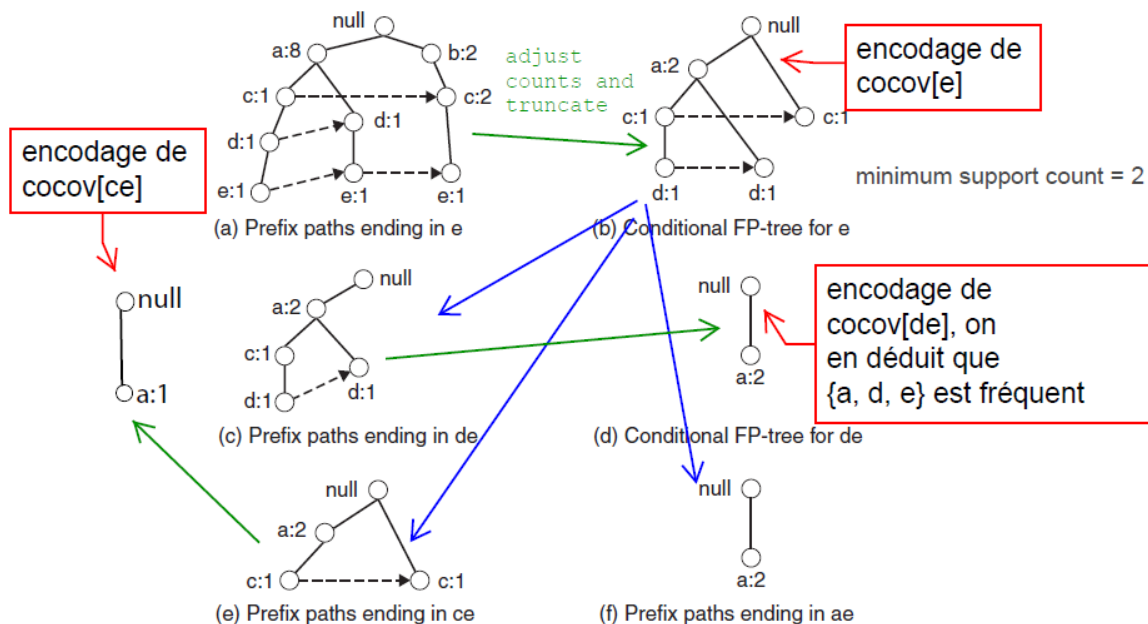


Figure 35

Part III

Analyse par clustering