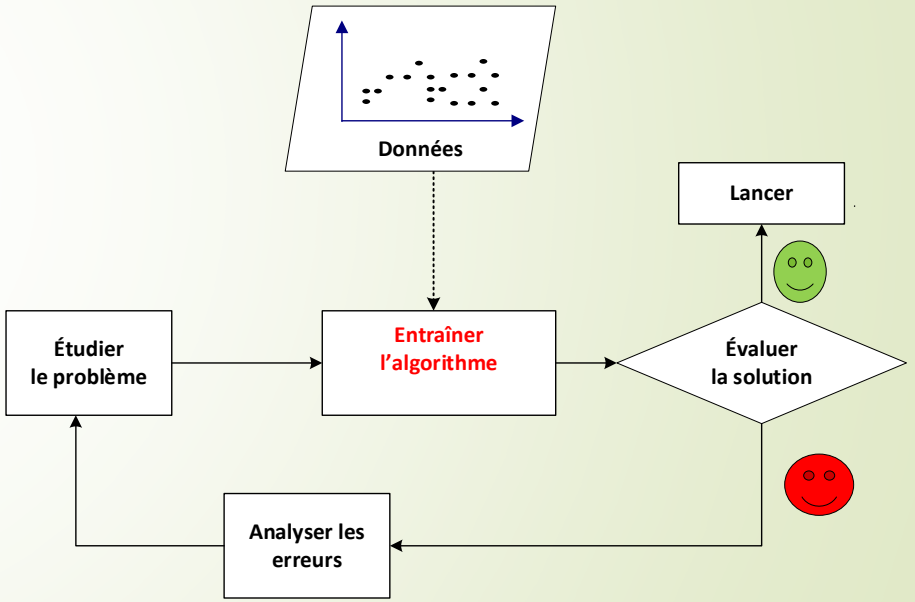
PLAN

1. Introduction à l'apprentissage supervisé : classification
2. Construction Séquentielle d'un arbre de décision : C4.5 (classique)
3. Construction Parallèle d'un arbre de décision : Algorithme SPRINT
4. Construction Distribuée d'un arbre de décision : MapReduce-C4.5
5. Démonstration avec **PySpark : Détection d'intrusion KDDCup**
6. Devoir #1 à venir...

Classification : une problématique importante de l'apprentissage automatique

- Un problème de classification est caractérisé :
  - Données en entrée : jeu d'entraînement de l'algorithme d'apprentissage;
  - Chaque donnée, appelée observation, est définie par un nombre d'attributs ou variables;
  - Les attributs sont soient continus quand les valeurs d'attributs sont ordonnées ou soient catégoriels (nominales) quand il n'y a pas une relation d'ordre entre les valeurs.
  - Un des attributs catégoriels est appelé le label de classe ou l'attribut de classification.

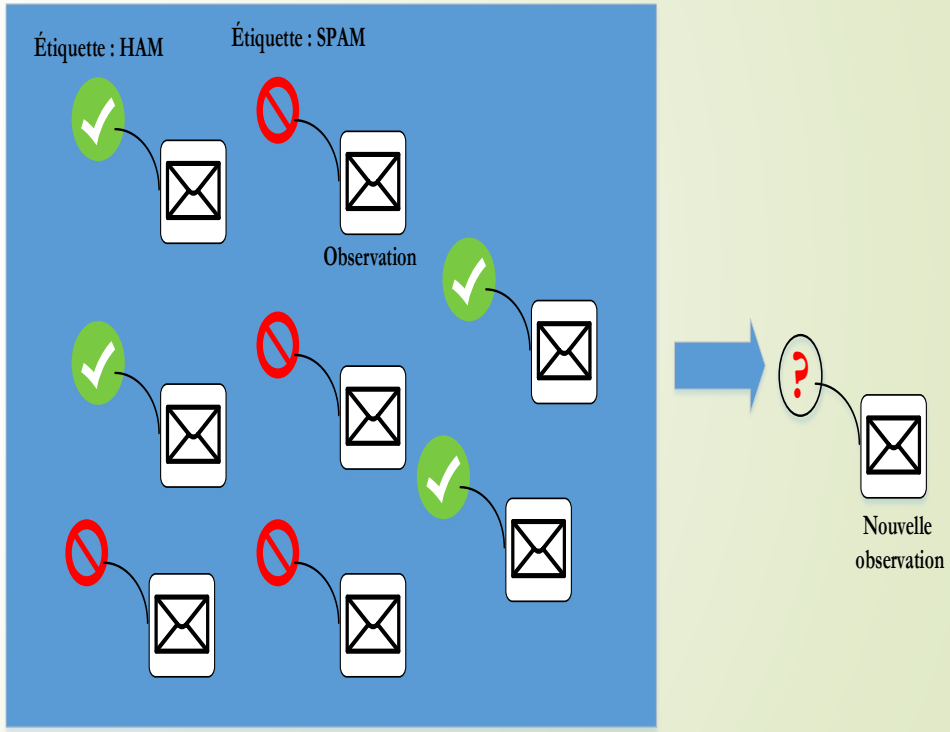
Schéma de l'entrainement



Objectif de la classification

- L'objectif est d'utiliser le jeu d'entrainement pour construire un modèle du label de classe qui sera basé sur les autres attributs.
  - L'idée est d'utiliser le modèle pour pouvoir classer de nouvelles données qui ne font pas partie du jeu d'entrainement.
- Domaine d'application :
  - Détection de fraudes;
  - Marché de détails et le marketing
  - ...

- Exemple de classification de spam



Algorithmes de classification

- Plusieurs modèles de classifications ont été proposés :
  - Réseaux de neurones
  - Algorithmes génétiques
  - Arbres de décisions,...
- Les arbres de décisions sont les plus populaires puisqu'ils obtiennent des taux de détection raisonnables et ils sont moins coûteux en termes de ressources de calcul.
- Les algorithmes les plus courant sont le C4.5 [Quinlan 93], SLIQ [Mehta et al. 96] basé sur ID3 [Quinlan 93].

Règles de classification séquentielle : cas du C4.5 (1)

- Soit T un jeu d'entrainement ayant $\{C_1, C_2, \ldots\ldots \quad , \ldots . C_k\}$classes
- Règle 1 :
  - T contient que des données appartenant à une seule classe C_i. L'arbre de décision pour T est une feuille annotée avec la classe C_i
- Règle 2 :
  - T contient des données appartenant à une **mixture** de classes. Un test est choisi selon une certaine mesure, et basé sur le meilleur attribut qui a un ou plusieurs valeurs$\{O_1, O_2, \ldots, O_n\}$
  - Il faut noter que dans la plupart des implémentations, $n = 2$ et on génère un arbre binaire.
  - T est partitionné en sous-ensemble$\{T_1, T_2, \ldots, T_i, \ldots., T_n\}$ où T_i contient les données dont l'attribut de test choisi à la valeur O_i
  - L'arbre de décision consiste en un nœud annoté avec le test choisi (attribut) et chaque branche dérivée sera associée à chaque valeur O_i de l'attribut de test
  - Le même mécanisme est appliqué récursivement pour chaque sous-ensemble Ti.

Règles de classification séquentielle : cas du C4.5 (2)

- **Règle 3 :**
  - T ne contient aucune donnée. L'arbre de décision pour T est une feuille, mais la classe qui lui sera associée sera déterminée à partir d'autres informations autre que T.
    - On utilise la classe majoritaire au niveau du nœud parent.
    - Une des mesures du choix du meilleur attribut est basée sur l'indice **du gain d'information** qui utilise l'entropie (l'incertitude sur la classification).
    - L'entropie d'un ensemble S d'individus contenant n classes:

$$E(S) = -\sum_{i} p_i log_2 p_i$$

    - p_i est la proportion d'individus de la classe i (la distribution par rapport à la classe)

Exemple de classification séquentielle : Tournoi de Tennis : 2 classes (PlayTenis: Y/N) et 4 attributs

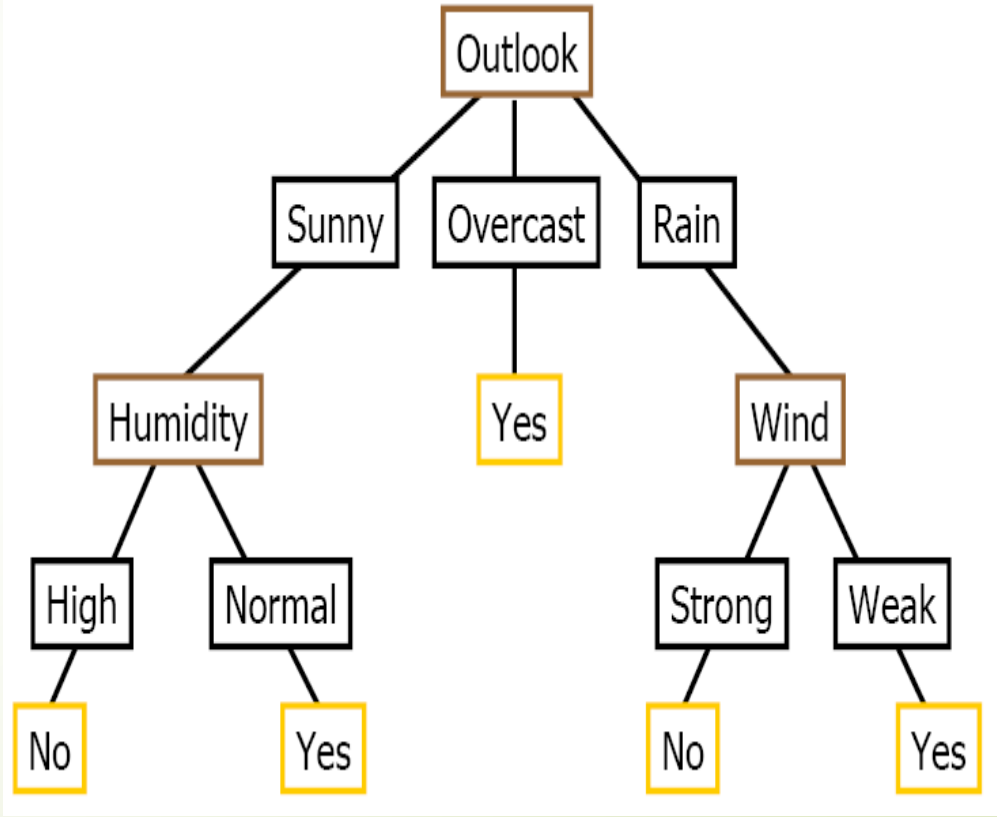
Day	Outlook	Temp.	Humidit Y	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Weak	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cold	Normal	Weak	Yes
D10	Rain	Mild	Normal	Strong	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Exemple de classification séquentielle : Tournoi de Tennis (3)

- La figure montre, l'application des règles 2 et 3 pour apprendre l'arbre de la figure.
- Le choix est basé sur le gain d'information (entropie) des attributs.
- L'entropie est calculée en utilisant l'information sur la distribution des classes.

Attribute Value	Class	
	Play	Don't Play
sunny	2	3
overcast	4	0
rain	3	2

Table 2: Class Distribution Information of Attribute *Outlook*



Exemple de classification séquentielle : Tournoi de Tennis (3)

- La distribution de la classe pour un attribut numérique implique des tests binaire pour chaque valeur possible (voir le tableau à droite, le cas de l'attribut Humidité qui est trié).
- Une fois que tous les distributions de classe pour chaque attribut sont collectées, on procède pour le calcul du gain
- On choisit l'attribut ayant **le gain maximal** comme nœud de test.

Outlook	Temp(F)	Humidity(%)	Windy?	Class
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Attribute Value	Binary Test	Class	
		Play	Don't Play
65	≤	1	0
	>	8	5
70	≤	3	1
	>	6	4
75	≤	4	1
	>	5	4
78	≤	5	1
	>	4	4
80	≤	7	2
	>	2	3
85	≤	7	3
	>	2	2
90	≤	8	4
	>	1	1
95	≤	8	5
	>	1	0
96	≤	9	5
	>	0	0

Gain d'information (1)

- Sélectionner l'attribut avec le plus grand gain d'information
- Soient P (p.e jouer au tenis) et N (ne pas jouer au tenis) deux classes et S un ensemble (dataset) avec p individus de la classe P et n individus de la classe N
- L'information nécessaire pour déterminer si un individu pris au hasard fera partie de P ou de N est donnée par l'entropie

$$E(S) = I(p,n) = -\frac{p}{p+n} log_2\left(\frac{p}{p+n}\right) - \frac{n}{p+n} log_2\left(\frac{n}{p+n}\right)$$

Gain d'information (2)

- Soient les ensembles $\{S_1, S_2, \ldots . S_v\}$ formant une partition de l'ensemble S de données en utilisant l'attribut A
- Toute partitions S_i contient p_i individus de la classe P et n_i individus de la classe N.
- L'entropie ou l'information nécessaire pour classifier les individus dans le sous-arbre S_i est

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(p_i, n_i)$$

- Le gain d'information généré par le partitionnement selon l'attribut A est :

$$G(A) = I(p, n) - E(A)$$

- Choisir l'attribut qui maximise le gain d'information.

Gain d'information (3)

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	false	N
sunny	hot	high	true	N
overcast	hot	high	false	P
rain	mild	high	false	P
rain	cool	normal	false	P
rain	cool	normal	true	N
overcast	cool	normal	true	P
sunny	mild	high	false	N
sunny	cool	normal	false	P
rain	mild	normal	false	P
sunny	mild	normal	true	P
overcast	mild	high	true	P
overcast	hot	normal	false	P
rain	mild	high	true	N

Hypothèses :

- Classe ***P*** : jouer_tennis = "oui"
- Classe ***N*** : jouer_tennis = "non"
- Information nécessaire pour classer un exemple donné est :

$$I(p,n) = I(9,5) = 0.940$$

Gain d'information: exemple

Calculer l'entropie pour l'attribut *outlook* :

outlook	p_i	n_i	$I(p_i, n_i)$
sunny	2	3	0,971
overcast	4	0	0
rain	3	2	0,971

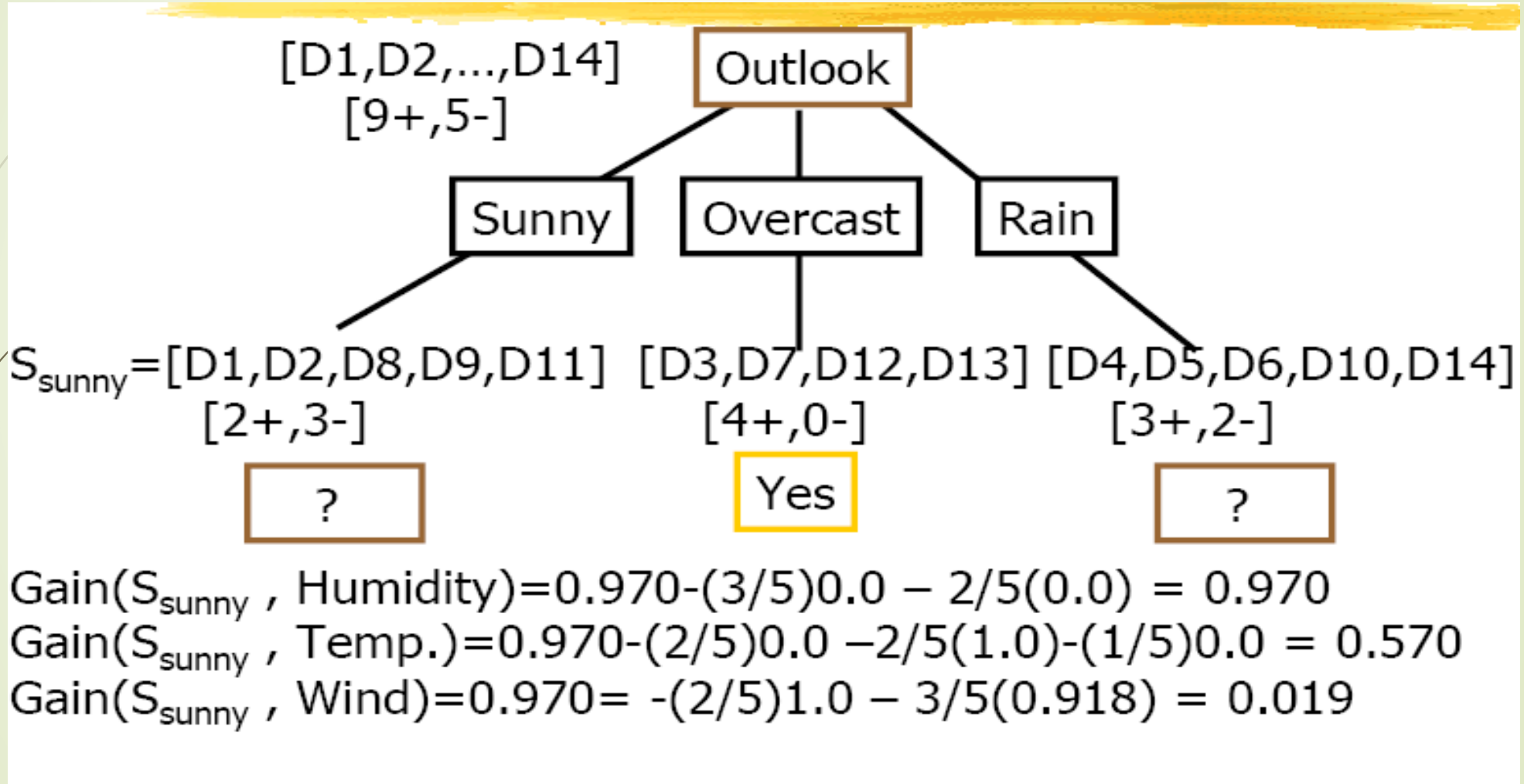
On a $$E(outlook) = \frac{5}{14} I(2,3) + \frac{4}{14} I(4,0) + \frac{5}{14} I(3,2) = 0.694$$

Alors $$Gain(outlook) = I(9,5) - E(outlook) = 0.246$$

De manière similaire

$$Gain(temperature) = 0.029$$
$$Gain(humidity) = 0.151$$
$$Gain(windy) = 0.048$$

Sélection du meilleur attribut : outlook

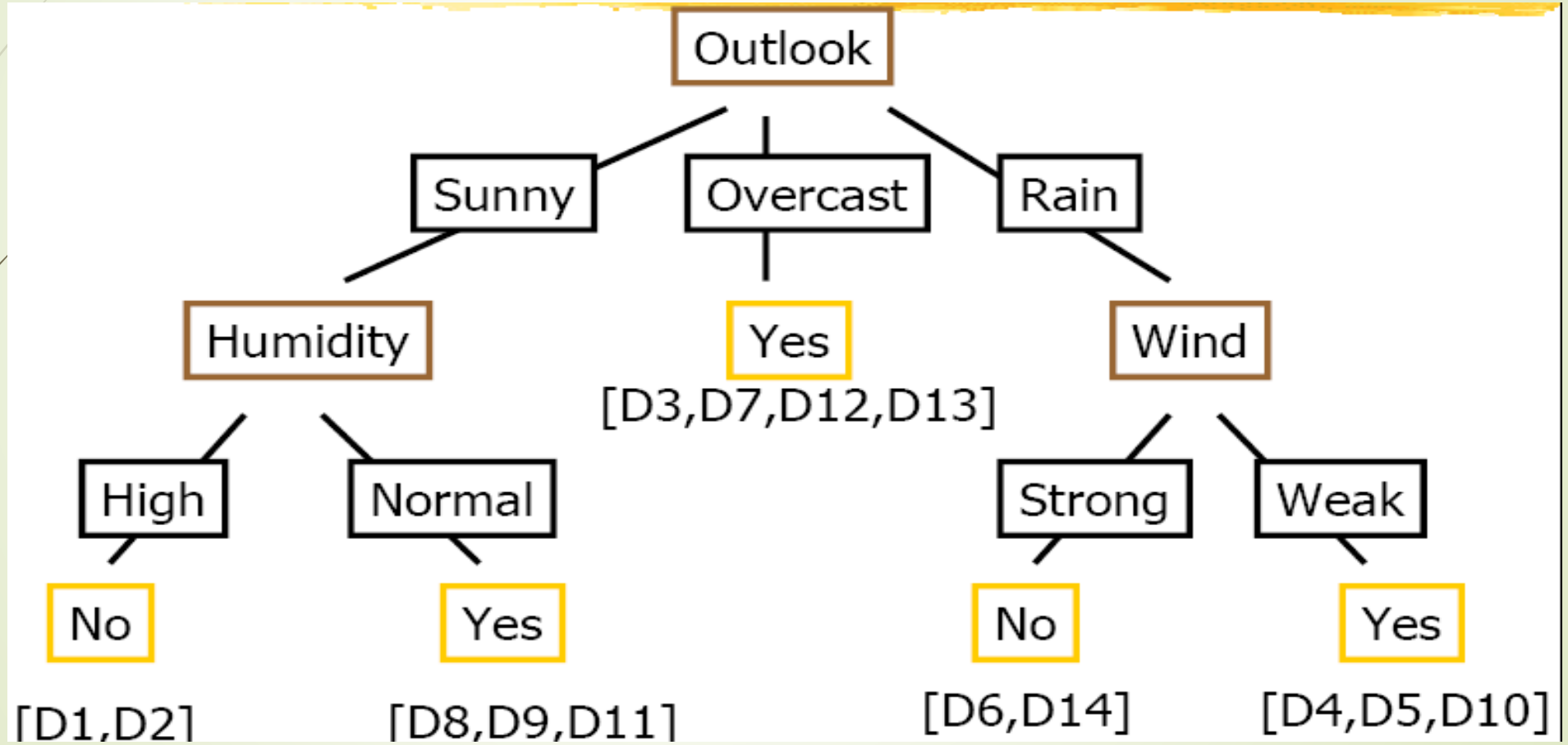


$Gain(S_{sunny}, Humidity)=0.970-(3/5)0.0 - 2/5(0.0) = 0.970$

$Gain(S_{sunny}, Temp.)=0.970-(2/5)0.0 -2/5(1.0)-(1/5)0.0 = 0.570$

$Gain(S_{sunny}, Wind)=0.970= -(2/5)1.0 - 3/5(0.918) = 0.019$

Arbre de classification appris à partir du dataset de 14 tournois de tenis.

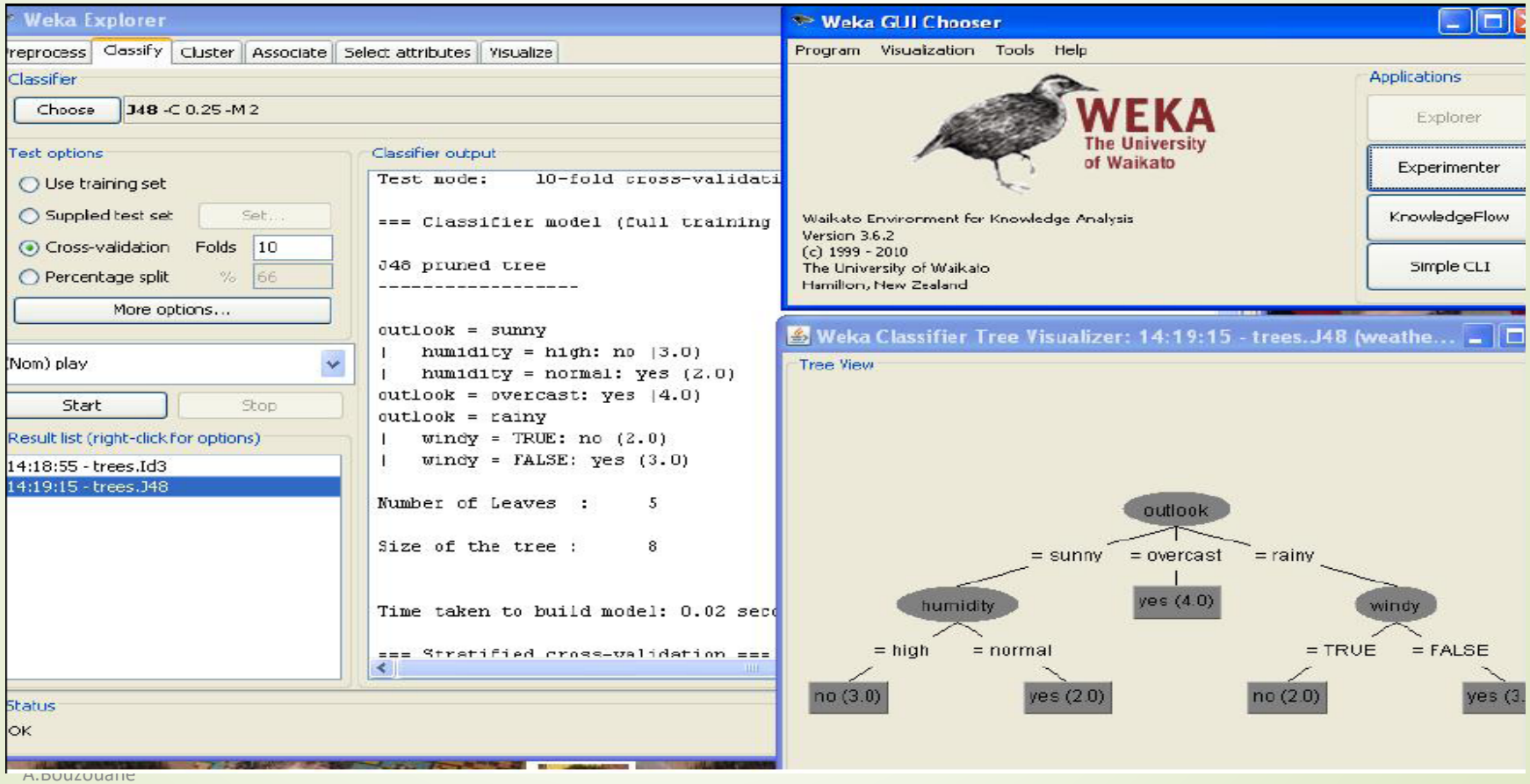


Algorithme ID3 (précurseur) du C45

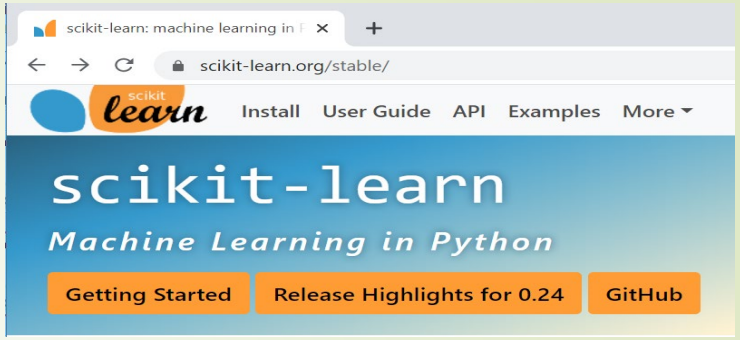
Algorithme ID3

INPUT : *X ensemble des exemples*
A= $\{a_{j=1,\dots,p}\}$ p attributs restants
OUTPUT : nœud racine de l'arbre
Créer un nœud racine
Si tous les éléments de *X* sont positifs alors
racine.etiquette = +, return racine
fin si
Si tous les éléments de *X* sont négatifs alors
racine.etiquette = - , return racine
fin si
Si A = vide alors
racine.etiquette = classe majoritaire de *X*, return racine
fin si
$a^* = \arg\max_a gain(X, a)$
racine.etiquette = a*
pour toutes les valeurs v_i de a* faire
ajouter une branche avec la valeur v_i
former $X_{a^*= v_i}$ *dans X* dont l'attribut a* = v_i
Si $X_{a^*= v_i}$ = *vide alors*
feuille étiquetée = classe majoritaire de *X*
Sinon
ID3($X_{a^*= v_i}$, A-{a*})
fin si
fin pour
return racine

Démonstration avec Weka, (J4.8 : C4.5 en java) https://www.cs.waikato.ac.nz/ml/weka/



Démonstration avec scikit-learn, (DecisionTreeClassifier) https://scikit-learn.org/stable/



jupyter ArbreDecisionWeatherNominal (autosaved)

File Edit View Insert Cell Kernel Help

Run Code

Out[47]: (14, 5)

In [48]: `df.head()`

Out[48]:

	outlook	temperature	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes

jupyter ArbreDecisionWeatherNominal (autosaved)

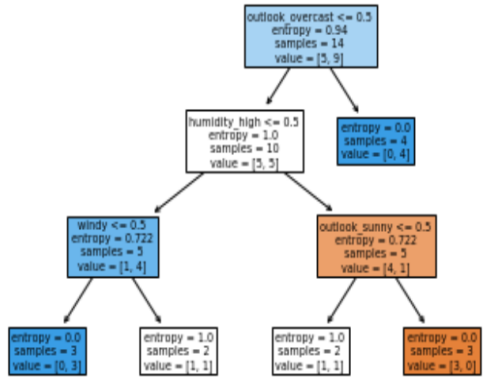
Logout

File Edit View Insert Cell Kernel Help

Trusted Python 3

Run Code

```
Text(299.075,28.055,'entropy = 0.0\nsamples = 3\nvalue = [3, 0]'),
Text(256.35,140.275,'entropy = 0.0\nsamples = 4\nvalue = [0, 4]')]
```



Bilan de la classification séquentielle

- L'algorithme C4.5 génère un arbre de décision-classification pour le jeu de données en partitionnant récursivement ces données.
  - L'arbre grossit en utilisant une stratégie en profondeur d'abord.
- L'algorithme considère tous les tests possibles qui peuvent segmenter les données.
- Pour un attribut discret (nominal), un test est considéré en prenant en compte les valeurs possibles et les distributions de classes pour chaque valeur de l'attribut.
- Pour un attribut continu, des tests binaires impliquant chaque valeur distincte de l'attribut est considéré:
  - Dans le but de partager d'une manière efficiente, le gain d'information de tous les tests binaires, les données associées à un nœud **sont triées** et que les calculs du gain pour chaque test binaire (seuil de coupure) sont réalisés en une seule lecture des données.
  - Ce processus est répété pour chaque attribut continu.

Classification de données massives

- Une manière « naïve » de réduire la complexité de calcul (traitement) pour la construction d'un classeur (arbre de décision) en utilisant un très volume de données (bigData), est de recourir simplement à la méthode de construction par échantillonnage des données.
  - Une telle méthode n'engendre pas le même taux de détection qu'un classeur entrainé sur toutes les données et c'est coûteux en temps!

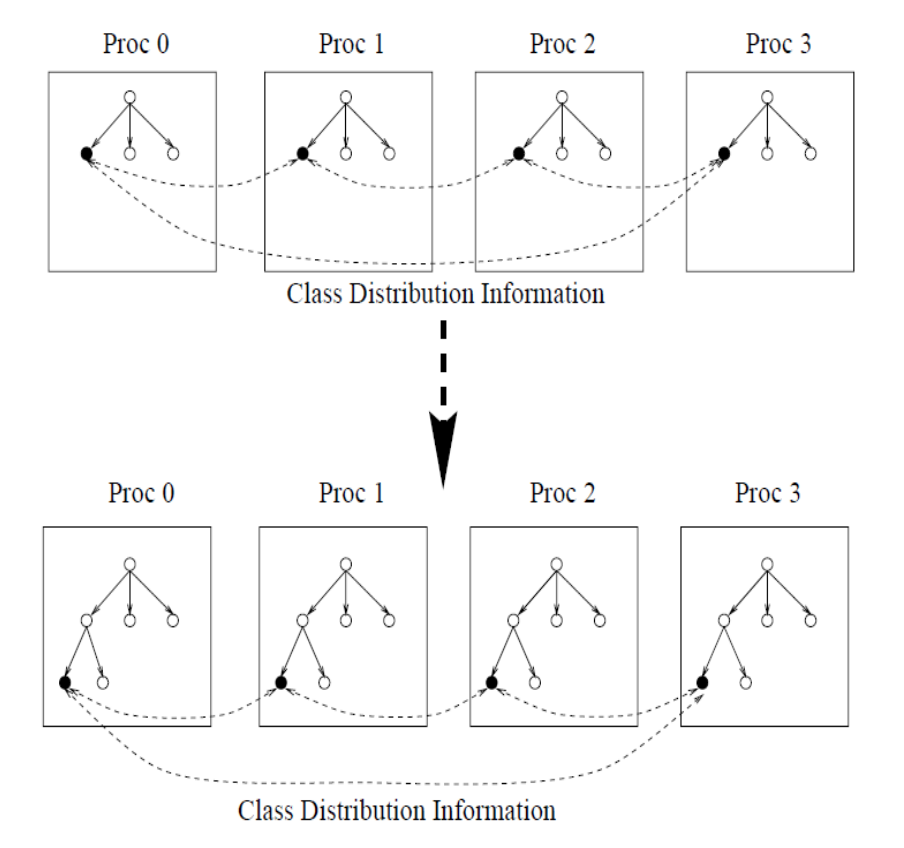
- Étant donné le contexte du BigData, il est non seulement recommandé de concevoir des algorithmes d'apprentissage performant en termes de précision et en temps de calcul mais qu'ils facilitent aussi le passage à l'échelle.
  - Dans le but d'obtenir un taux de détection qui avoisine la réalité en un temps raisonnable,
  - Une des pistes est de tenter une approche par la « parallélisation » de ces algorithmes d'apprentissage !

Parallélisation d'un classeur basé sur un arbre de décision

- Un algorithme de construction d'arbre de décision est naturellement parallélisable.
  - Une fois un nœud est généré, tous les autres nœuds enfants de ce nœud peuvent être générés d'une manière concurrentielle.
  - le traitement pour la génération des nœuds successeurs peut être décomposé en effectuant la **décomposition des données** du jeu d'entraînement.
- **Néanmoins la parallélisation d'algorithmes de classification soulève des défis** :
  1. La forme de l'arbre appris est irrégulière et déterminée uniquement à l'exécution
     - La charge de calcul associée au nœud est variable et dépendant des données, par conséquent, n'importe quel schéma statique d'allocation de charges souffrira du problème de la répartition de la charge de calcul –load balancing-
  2. Si les données sont partitionnées et allouées dynamiquement aux processeurs traitant les nœuds successeurs générés, il y aura un coût concernant le mouvement des données : **problème de localité des données**

Classification parallèle : construction d'un arbre de décision

- On assume N données (BigData) d'entraînement distribuées aléatoirement sur P processeurs, tel que chaque processeur disposera de N/P données.
- Tous les processeurs contribuent, d'une manière synchrone, à la construction d'un arbre de décision en se communiquant les informations de distributions des données/classes locales.
  - La figure montre que la racine de l'arbre a été étendu et le prochain nœud courant est le nœud le plus à gauche (cercle plein, figure en haut).
  - Les 4 processeurs coopèrent pour étendre à nouveau ce nœud pour avoir deux autres nœuds (les plus à gauche, figure de bas)
  - Un des nœuds est sélectionné de nouveau pour être le nœud courant, et de nouveau, les 4 processeurs coopèrent pour étendre ce nœud



Pseudo-code de classification parallèle

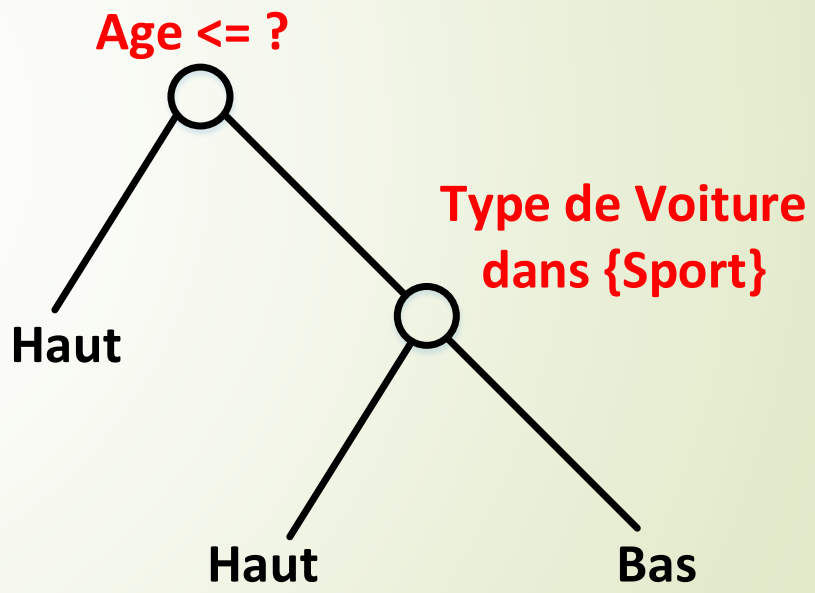
1. Sélectionner un nœud à étendre selon la stratégie d'expansion d'un arbre de décision (profondeur d'abord ou largeur d'abord);
2. Identifier ce nœud comme étant le nœud courant. Initialement, la racine de l'arbre est le nœud courant;
3. Pour chaque attribut, collecter les informations de distribution de classes des données locales au nœud courant;
4. Échanger les informations de distributions locales à travers les processeurs;
5. Calcul simultané des gains d'informations de chaque attribut au niveau de chaque processeur
6. Sélectionner le meilleur attribut à travers un échange d'informations entre les processeurs.
7. Dépendamment du facteur de branchement de l'arbre désiré, créer des nœuds successeurs relativement aux valeurs d'attributs, et segmenter les données locales associées à chaque processeur.
8. Répéter les étapes 1 à 7 jusqu'à ce qu'il n'y a plus de possibilité d'extension de nœuds.

Exemple : prédiction du risque d'une police d'assurance.

- Soit le jeu d'entraînement suivant où idE est l'identificateur d'individu

idE	Age	TypeVoiture	Classe (Risque)
0	23	Familial	Haut
1	17	Sport	Haut
2	43	Sport	Haut
3	68	Familial	Bas
4	32	Camion	Haut
5	20	Familial	Bas
6	75	Sport	Haut
7	18	Sport	Bas
8	24	Sport	Haut
9	36	Camion	Bas
10	56	Familial	Haut
11	55	Familial	Bas

- l'arbre recherché :



Cas de l'algorithme SPRINT [J. Shafer et al.] dédié pour la construction parallèle

- Chaque attribut est maintenu dans une structure de liste où chaque entrée est un triplet (valeur attribut, classe de l'individu, identificateur de l'individu ou l'enregistrement) :
  - attA={(val1,cla1, id1), (val2,cla2, id2),.....(valk,clak,idk)}
  - Quand le meilleur attribut de segmentation du nœud est déterminé, les listes des attributs associées au nœud seront segmentées.
- Une table de hachage, du même ordre que les données d'apprentissage, est utilisée pour faire la correspondance entre les ids des individus et où chaque individu sera propagé selon la décision de segmentation associé au nœud.
- Chaque entrée dans la liste d'attribut est déplacée sur un nœud de l'arbre de classification selon l'information recensée de la table de hachage.

- **L'ordre d'apparition dans les listes d'attributs est maintenu comme celui du pré-tri effectué au départ.**

Sprint (2)

- Avant tout traitement, les données sont **triées** relativement aux attributs de type numérique et organisées sous forme d'une liste :{ (valAtt, Classe, id)}:
- Supposons que nous avons **2 processeurs**, chacun reçoit tous les attributs (age, typeVoiture, risque) et ainsi que les données qui sont réparties en 2 blocks, comme le montre les deux tableaux suivants.
  - On suppose que le nœud racine est le nœud numéroté 0

PROCESSEUR 1

Liste Attribut : Age				Liste Attribut : Type Voiture		
Age	Classe	idE		TypeVoiture	Classe	idE
17	Haut	1		Familial	Haut	0
18	Bas	7		Sport	Haut	1
20	Haut	5		Sport	Haut	2
23	Haut	0		Familial	Bas	3
24	Haut	8		Camion	Bas	4
32	Bas	4		Familial	Haut	5

PROCESSEUR 2

Liste Attribut : Age				Liste Attribut : Type Voiture		
Age	Classe	idE		TypeVoiture	Classe	idE
36	Bas	9		Sport	Haut	6
43	Haut	2		Sport	Bas	7
55	Bas	11		Sport	Haut	8
56	Haut	10		Camion	Bas	9
68	Bas	3		Familial	Haut	10
75	Haut	6		Familial	Bas	11

Sprint (3) : Chaque processeur maintient des informations sur les distributions des attributs/classes au niveau du nœud racine 0

PROCESSEUR 1			
Age	Test binaire	Classe	
		Haut	Bas
17	<=	1	0
	>	6	5
18	<=	1	1
	>	6	4
20	<=	3	1
	>	5	4
23	<=	3	1
	>	4	4
24	<=	4	1
	>	3	4
32	<=	4	2
	>	3	3

Type de Voiture	Classe	
	Haut	Bas
Familial	2	1
Camion	0	1
Sport	2	0

PROCESSEUR 2			
Age	Test binaire	Classe	
		Haut	Bas
36	<=	4	3
	>	3	2
43	<=	5	3
	>	2	2
55	<=	5	4
	>	2	1
56	<=	6	4
	>	1	1
68	<=	6	5
	>	1	0
75	<=	7	5
	>	0	0

Type de Voiture	Classe	
	Haut	Bas
Familial	1	1
Camion	0	1
Sport	2	1

Sprint (4) : chaque processeur exécute le code suivant

Appel Initial : Partition (Jeu d'entrainement)

Partition (Données S)

Si (toutes les données de S sont de la même classe alors retour

Pour chaque attribut A faire

Évaluer le partitionnement sur l'attribut A

Utiliser le meilleur attribut pour segmenter S on S1 et S2

Partition (S1)

Partition(S2)

Sprint(5): Calcul du Gain d'information, en utilisant l'entropie = $\sum_i p_i \, log_2 p_i$

- p_i est la proportion d'individus de la classe i. Nous avons 2 classes : Risque Haut et Risque Bas dont 7 de la classe Haut et 5 de la classe Bas
- Supposons que le meilleur split a été trouvé par le processeur 1, donné par l'attribut age <=23.
- Entropie de départ :

$$I(7+,5-)= -7/12 log_2(7/12) - 5/12 log_2(5/12) = 0.97$$

Au niveau du processeur 1 :

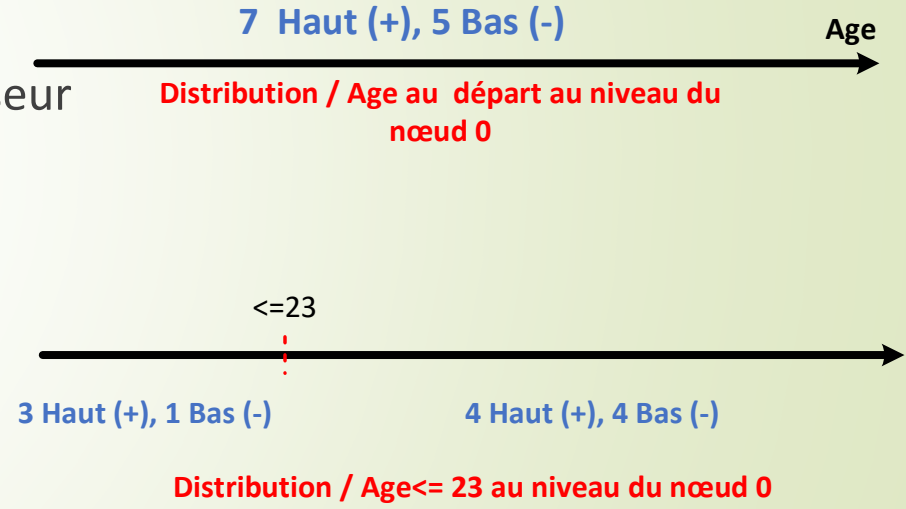
$$E(Age<=23) = 4/12 \, I(3+,1-) + 8/12 I(4+,4-) = 0.80$$

$$Gain(Age<=23) = 0.97 - 0.8 = 0,17$$

Au niveau du processeur 2 :

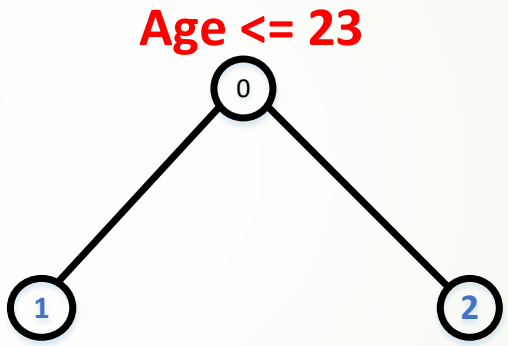
$$E(Age<=56) = 10/12 \, I(6+,4-) + 2/12 \, I(1+. 1-) = 0.95$$

$$Gain(Age<=56) = 0.97 - 0,95 = 0.02$$



Sprint (6): Résultat de la segmentation

- À partir de là, nous allons étendre le nœud racine 0 de l'arbre pour obtenir deux nouveaux nœuds successeurs (arbre binaire) de nœuds numérotées 1, 2



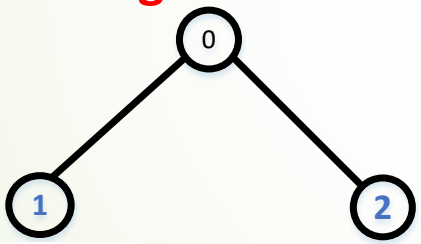
- Comment les données des blocks se répartiront-elles au niveau de chaque processeur ?

Sprint (7) : résultat de la segmentation

- Les listes d'attributs Age seront subdivisées relativement au seuil 23, en deux sous listes d'attributs Age qui seront associées respectivement aux nœuds 1 et 2, à travers les 2 processeurs, comme le montre suivante, :

Processeur 1

Age <= 23

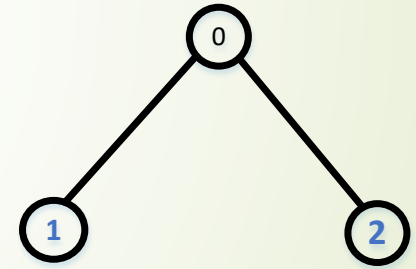


Age	Classe	idE
17	Haut	1
18	Bas	7
20	Haut	5
23	Haut	0

Age	Classe	idE
24	Haut	8
32	Bas	4

Processeur 2

Age <= 23



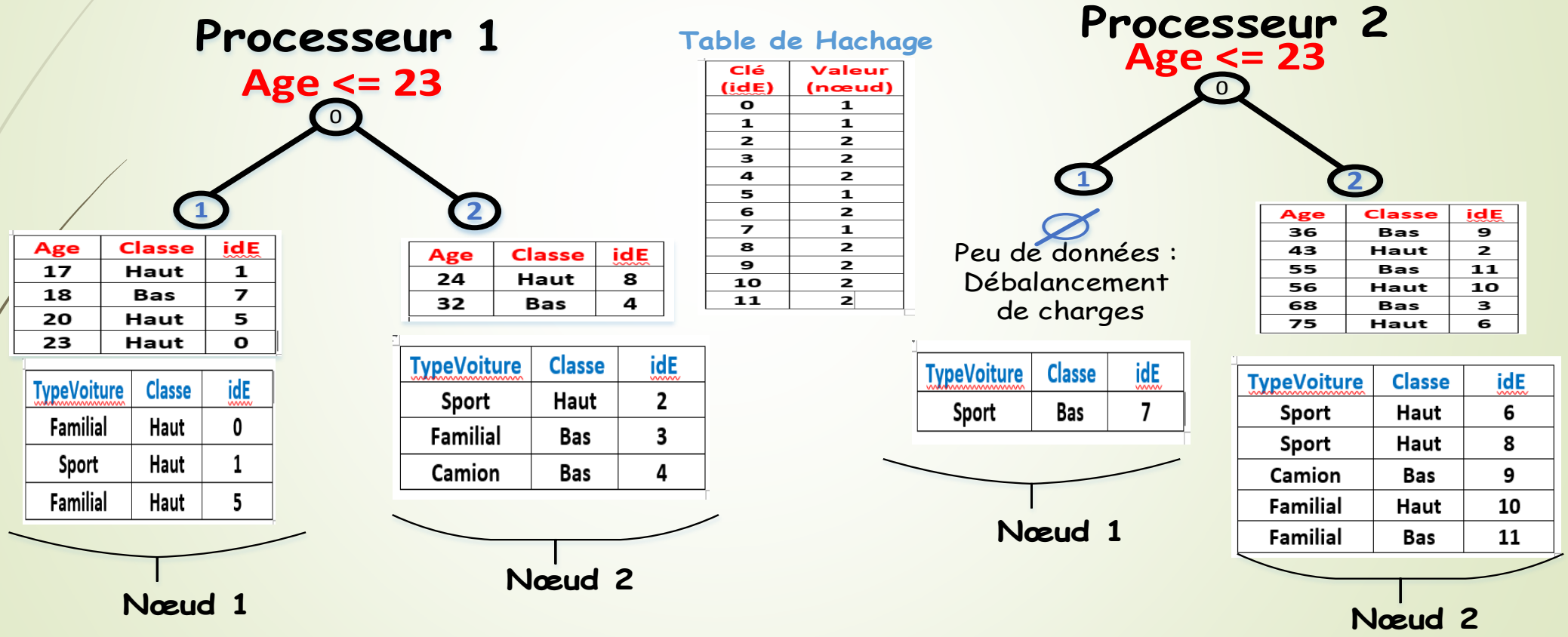
Pas de données : Débalancement de charges

Age	Classe	idE
36	Bas	9
43	Haut	2
55	Bas	11
56	Haut	10
68	Bas	3
75	Haut	6

Sprint (8): Construction de la table de hachage

- Mais comment les autres listes d'attributs seront distribuées, Type de Voiture, à travers les nœuds des processeurs ?
- Chaque processeur construit une table de hachage dont la clé est l'identité de l'enregistrement et la valeur est une référence vers un des nœuds de l'arbre : {(idE, noeud)}
- Suite à la subdivision au nœud de test (Age<=23), chacun des processeurs a une information sur la localisation d'un sous ensemble des individus.
  - Le processeur 1 sait que les individus dont les idEs ={1,7,5,0} sont associés au nœud 1 et les idEs ={8,4} sont associés au nœud 2.
    - Il communiquera ces informations (idE, nœud) au processeur 2.
  - Idem, le processeur 2 sait que les individus dont les idEs ={9, 2, 11, 10, 3, 6} sont associés au nœud 2, il communiquera cette information au processeur 1.

Sprint (9) : Utilisation de la table de hachage pour répartir les listes d'attributs



Bilan de la parallélisation (1)

- L'avantage :
  - L'algorithme ne requiert pas un mouvement des données
- L'inconvénient :
  - L'algorithme souffre d'une forte communication et du problème de la répartition de charges de calcul entre les processeurs.
- Pour chaque nœud de l'arbre de décision, après avoir collectées les informations de distributions de classes des données locales, tous les processeurs ont besoin de synchroniser et d'échanger les informations de distributions
  - Au niveau des nœuds moins profonds, la communication est relativement moins forte car le nombre de données à traiter (moins de subdivision) est relativement large
  - Plus l'arbre grossit et devient plus profond, plus il y a de la segmentation (partitions) de données, et donc le nombre de données associées à chaque nœud diminue, et comme conséquence, le temps de calcul des nouvelles distributions pour chaque nouveau nœud diminue.
  - Si la moyenne du facteur de branchement de l'arbre de décision est k, alors le nombre de données au niveau de chaque nœud enfant est en moyenne de 1/k du nombre de données du nœud parent. Toutefois, la communication ne diminue pas pour autant ! Du moment que le nombre d'attributs diminue de 1, à chaque fois
  - Par conséquent, la communication domine au niveau du temps de traitement global.

Bilan (2)

- Même si chaque processeur démarre avec le même nombre de données, la quantité de données associée à un même nœud d'un arbre de décision peu varié substantiellement d'un processeur à un autre.
- Par exemple :
  - Le processeur 1 pourrait avoir toutes les données au niveau du nœud A et aucune donnée au niveau du nœud B. Inversement, le processeur 2 pourrait avoir toutes les données au niveau du nœud B et aucune donnée au niveau du nœud A.
  - Quand le nœud A sera sélectionné comme nœud courant, le processeur 2 n'aura aucune tâche (travail) à effectuer. Similairement quand le nœud B sera sélectionné comme nœud courant, le processeur 1 n'aura aucune tâche à effecteur.

Modèle Map-Reduce : modèle de programmation distribué

- Introduit, initialement, par Google pour exploiter le traitement en cluster dans l'organisation des fichiers à large-échelle : Système de Fichiers Distribué (SFD).
- Un SFD est caractérisé :
  - Les fichiers peuvent être énormes dont la taille est possiblement de l'ordre de téraoctets. Si vous avez des fichiers de « petites » tailles, il n'y a pas lieu de recourir à un SFD
  - Les fichiers sont rarement mis à jour. Ils sont souvent lus comme des données pour un certain calcul. De temps en temps (dés fois), des données sont ajoutées aux fichiers.
    - Un système de réservation de billets d'avion n'est pas souhaitable pour un SFD même si les données sont énormes, à cause de la fréquence de la mise à jour !
  - Les fichiers sont divisés en blocks (chunks), généralement de 64Mo et qui sont répliqués sur plusieurs machines différentes (tolérance aux pannes)
  - Pour retrouver les chunks d'un fichier, un autre « petit » fichier appelé master node (name node) est associé au ficher. Lui aussi est répliqué.

MapReduce (1) : fonctions Map et Reduce

- Pour gérer un traitement à grande-échelle, vous avez juste besoin d'écrire deux fonctions **Map** et **Reduce** matérialisant le traitement en question.
  - L'exécution parallèle et la coordination des tâches qui exécutent les fonctions Map ou Reduce sont gérées par le système (built-in).

- En résumé, un traitement MapReduce s'exécute de la manière suivante :

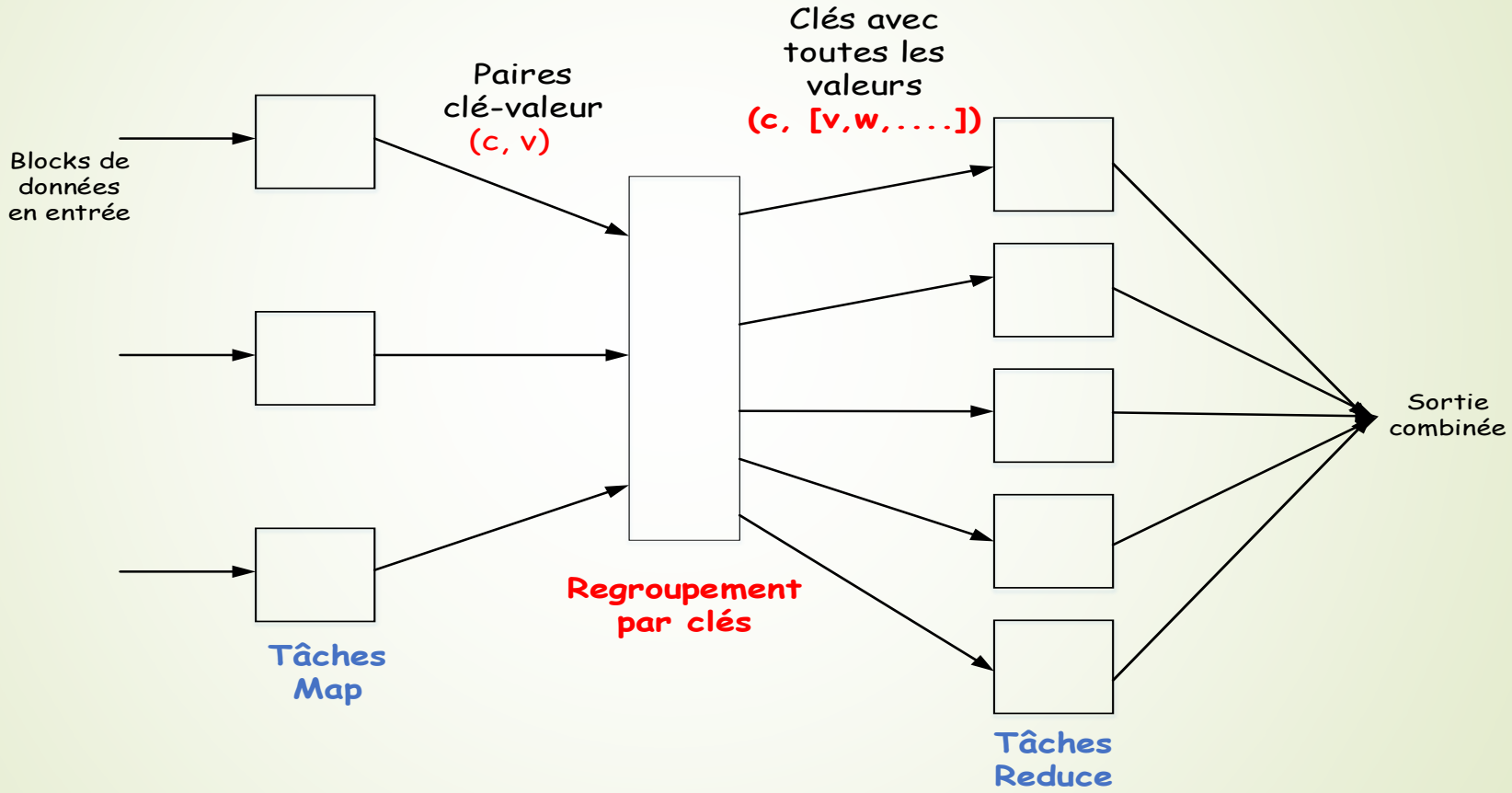
1. Chaque **tâche Map** lui est associée un ou plusieurs blocs de données du SFD:
   - Son rôle consiste à transformer chaque bloc de données en **une séquence de paires clé-valeur**
   - La manière dont les paires clé-valeur sont produites à partir des données d'entrée est déterminée par le code écrit par l'utilisateur pour la fonction **Map()**

MapReduce (2) : fonctions Map et Reduce

2. Les paires clé-valeur de chaque tâche Map sont collectées par le contrôleur Maître et sont triées par rapport à la clé (shuffle). Ces clés sont réparties entre les tâches Reduce:
   - Par conséquent, toutes les paires clé-valeur ayant la même clé sont dirigées vers la même tâche Reduce

3. À un instant t, les tâches Reduce effectuent le traitement sur une clé donnée, et combinent toutes les valeurs associées à cette clé dans une certaine manière.
   - La manière de combiner les valeurs est déterminée par le code écrit par l'utilisateur pour la fonction Reduce()

Archétype du traitement MapReduce



Les tâches Map

- Nous visualisons les fichiers d'entrée pour une tâche Map comme consistant d'éléments qui peuvent être, par exemple, de type tuple (liste, enregistrement) ou document.
  - Un bloc est une collection d'éléments dont aucun élément n'est dupliqué
- Techniquement, toutes les entrées aux tâches Map et toutes les sorties des tâches Reduce sont sous la forme d'une paire clé-valeur.
  - Les clés des entrées sont (souvent des index des enregistrements,...) non pertinentes et souvent ignorés
- La fonction Map() prend un élément d'entrée comme un argument et produit zéro ou plusieurs paires clé-valeur.
  - Les types des clés, valeurs sont arbitraires individuellement
  - Les clés ne sont pas les clés au sens usuel, elles ne doivent pas être, par exemple, unique
  - La tâche Map peut produire plusieurs paires clé-valeur avec la même clé, même à partir du même élément.

Exemple d'un calcul MapReduce : comptage de mots d'un répertoire de documents.

- Calculer le nombre d'occurrence de chaque mot à partir d'une collection de documents (téraoctets)
  - Le fichier d'entrée est l'archive de documents
  - Chaque document est un élément.
- La fonction Map() pour cet exemple utilisera des clés (de type string) qui représenteront les mots et les valeurs qui seront de type entier.
- La tâche Map lit un document :
  - Segmentation (divise) en une séquence de mots $w_1, w_2, \dots, w_n,$
  - Émission en sortie d'une séquence de paires clé-valeur où la valeur est réduite à 1.

$$(w_1, 1), (w_2, 1), \dots, (w_n, 1)$$

  - Une seule tâche Map pourra typiquement traiter plusieurs documents – tous les documents sont contenus dans un ou plusieurs blocks-
    - À partir de là, la sortie d'une tâche Map peut avoir plusieurs séquences de paires clé-valeur.
    - Notez si un mot w apparaît m fois dans tous les documents assignés à un processus Map donné, alors il y aura m paires clé-valeur de la forme $(w, 1)$ en sa sortie.

Regroupement (Mélange, Shuffle) par la clé

- Aussitôt que toutes les tâches Map se terminent avec succès, les paires clé-valeur sont regroupées par clé, et les valeurs associées avec chaque clé, sont mises dans une liste de valeurs.
  - Le regroupement effectué par le système, relativement à ce que les tâches Map et Reduce font.
- le processus Contrôleur-Maitre connait combien de tâches Reduce qu'il y aura, mettons r tâches.
  - Le contrôleur-maitre dispose d'une fonction de hachage qui est appliquée aux clés pour produire un numéro de bloc (bucket) compris entre $0\ et\ r-1$
  - Chaque clé produite en sortie par la tâche Map est « hachée » et sa paire clé-valeur est placée dans un des r fichiers locaux. Chaque fichier est destiné pour une tâche reduce.
    - **Avec Hadoop, on transfère les résultats dans des fichiers (E/S) sur disque physique d'où l'inefficacité**
    - **Avec Spark, on travaille directement en mémoire, d'où la rapidité du traitement**

Regroupement par la clé : shuffle (2)

- Pour effectuer le regroupement avec la clé et la distribution aux tâches Reduce, le contrôleur Maitre :
  - fusionne les (fichiers dans le cas de Hadoop ou blocs dans le cas de Spark) de chaque tâche Map et qui sont destinés pour une tâche Reduce particulière
  - achemine le fichier fusionné sous forme d'une séquence de paires clé-liste-valeurs.
  - Ainsi, pour chaque clé c, l'entrée de la tâche Reduce qui gère la clé c est une paire de la forme : $(c, [v_1, v_2, \dots v_n])$ où (c, v_1), (c, v_2), …. , (c, v_n) sont toutes les paires avec la clé c provenant de toutes les tâches Map.

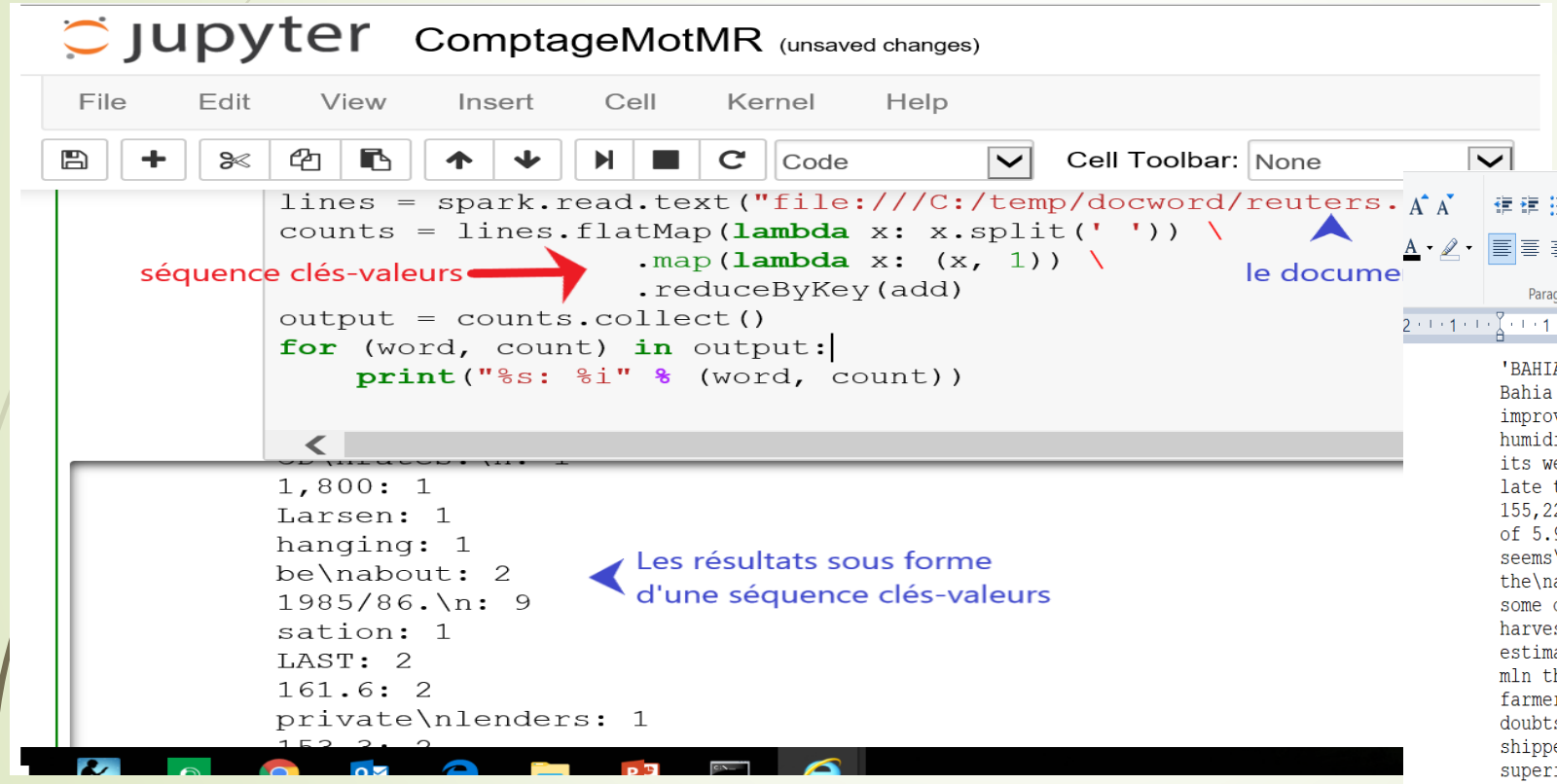
Tâches Reduce

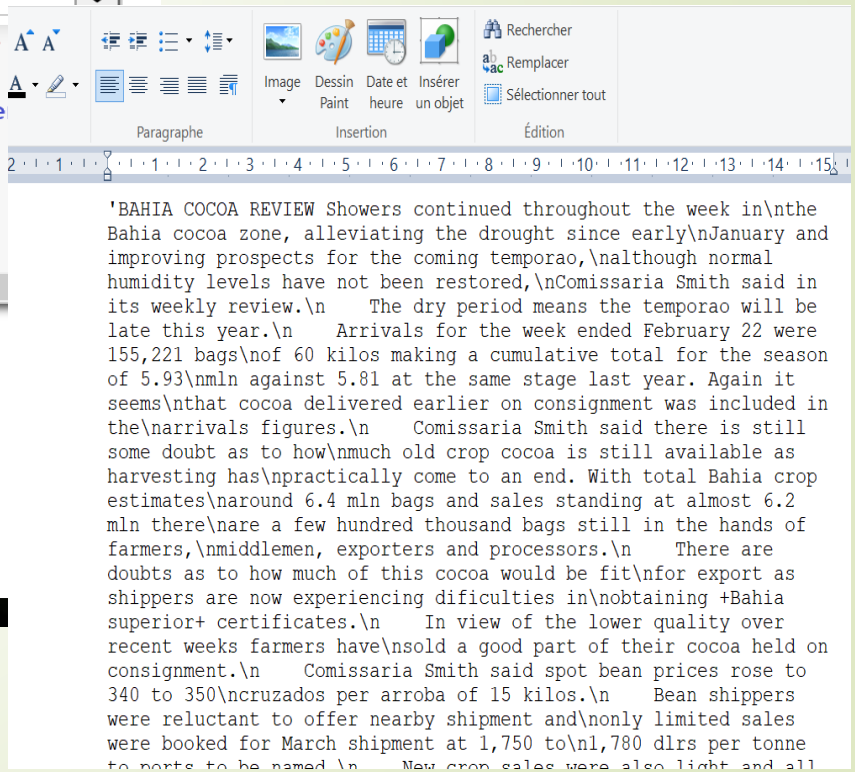
- L'argument d'une fonction Reduce() est une paire consistant en une clé et une liste de valeurs qui lui est associée.
  - La sortie de la fonction Reduce est une séquence de 0 ou plusieurs paires de clé-valeur
  - Ces paires clé-valeur peuvent être de différents types par rapport aux paires émises par la tâche Map
  - Souvent, on utilise le terme **Réducteur** (reducer) pour faire référence à l'application de la fonction reduce sur la clé et la liste des valeurs.
    - Un exemple de réducteur : la somme. La moyenne,...une opération qui produit en sortie, une paire clé-valeur.
- Les sorties produites par l'ensemble des tâches Reduce sont fusionnées dans un seul fichier (block).

Exemple : comptage des mots

- La sortie d'un Réducteur consiste en un mot et une somme.
- Alors les sorties de toutes les tâches Reduce est une séquence de paires (w, m) où w est le mot qui apparait au moins une fois dans tous les documents et m est le total des occurrences de w dans tous les documents.

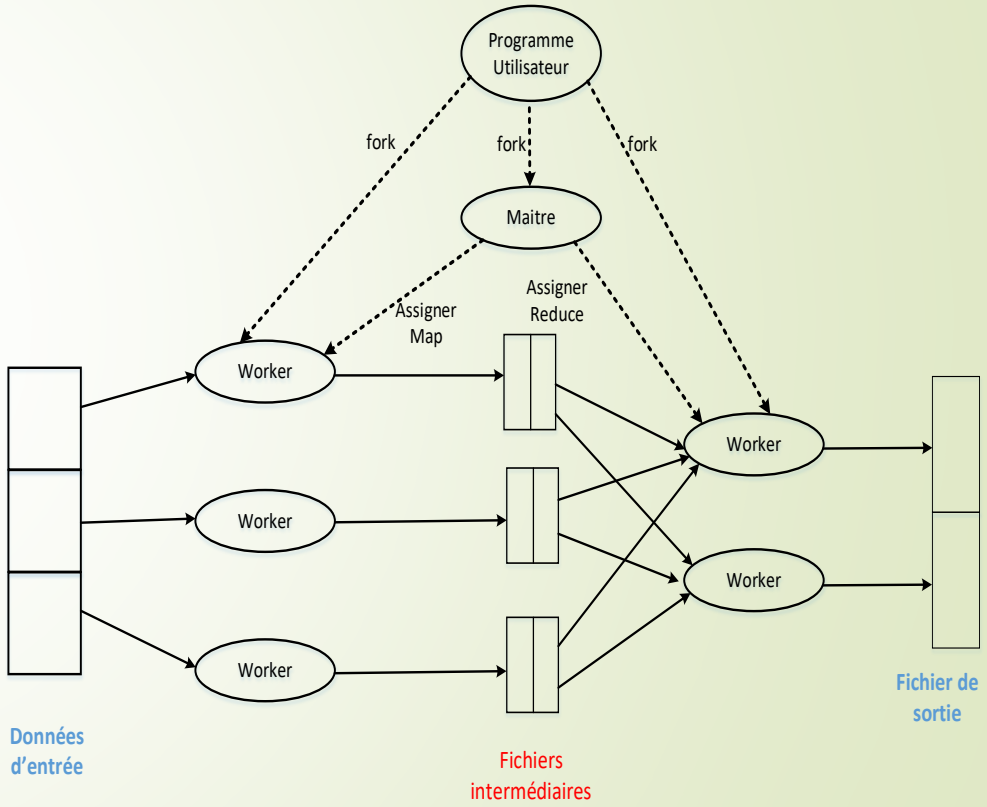
Démonstration en pyspark: comptage de mots d'un document (1Mo)





Détails sur l'exécution de MapReduce

- Le programme utilisateur lance en simultané « fork » un processus « contrôleur Maitre ou Driver » et un certain nombre de processus « worker ou exécuteur » sur différents nœuds.
  - Un worker gère soient des tâches Map « Map-worker » soient des tâches Reduce « Reduce-worker » mais pas les deux.
  - Le contrôleur « Maitre » a plusieurs responsabilités :
    1. Création d'un certain nombre de tâches Map et un certain nombre de tâches Reduce. Ce nombre est fixé par défaut ou indiqué par l'utilisateur.
    2. Ces tâches seront assignées au processus « worker » par le contrôleur Maitre



Détails sur l'exécution de MapReduce

- Il est raisonnable de créer une tâche Map pour chaque bloc du fichier d'entrée, mais on peut souhaiter créer très peu de tâches Reduce :
  - La raison principale pour la limitation du nombre de tâches Reduce, c'est qu'il est nécessaire pour chaque tâche Map de créer un fichier intermédiaire (block) pour chaque tâche Reduce, et s'il y en a beaucoup de tâches Reduce, il y aura énormément de fichiers intermédiaires à créer.
- Le contrôleur Maitre garde une trace des statuts de chaque tâche Map et Reduce : attente, exécution sur un Worker particulier, complété
- Le processus Worker rapporte le statut de la tâche au Maitre quand il termine une tâche, et une nouvelle tâche est planifiée par le Maitre pour ce Worker.
- Chaque tâche Map lui est assignée un ou plusieurs blocks de fichiers d'entrée et exécute le code écrit par l'utilisateur en utilisant cette entrée.
- La tâche Map crée un fichier intermédiaire (bloc) pour chaque tâche Reduce sur le disque local du Worker qui exécute la tâche Map, dans le cas de Spark, il opère via la mémoire
- Le contrôleur Maitre est informé de l'emplacement et la taille du fichier intermédiaire et la tâche Reduce destinataire du fichier.
- Quand la tâche Reduce est assignée par le Maitre à un processus worker, il fournit à la tâche le fichier d'entrée
- La tâche Reduce exécute le code écrit par l'utilisateur et écrit ses soties dans un fichier qui fera partie du SFD.

Un aperçu de Distribution de l'algo. ID3 avec MapReduce

1. Répartir les données massives sous forme de blocs exploitables par les nœuds Worker;
   - Il y aura autant de tâches Map qu'il y a de blocs
2. Chaque tâche Map consistera à calculer les distributions des classes par attribut relativement aux blocs de données
   - La sortie est une séquence de paires clé-valeur dont la clé est l'attribut et la valeur est un vecteur de distributions des classes :

     (attr, [(val1, c1, c2,...ck), (val2, c1, c2,...ck),....,(valm, c1,c2,...,ck)]

   - Le calcul des distributions se feront en parallèle sur les différents blocks
3. Chaque tâche Reduce calculera les gains d'informations par attribut:
   - La sortie est une séquence de paires clé-valeur dont la clé est l'attribut et la valeur est le gain d'information) distributions des classes : (attr, gain)
   - Le calcul des gains se feront en parallèle
4. Le choix de l'attribut de division associé à un nœud sera confié à un nœud Maitre
5. La construction de l'arbre de décision consiste en des appels récursifs (itérations) sur les workers Map et Reduce

MapReduce-Tree Algorithm [V. Purdila et.al]

L'algorithme est composé de trois sections :

1. Section « contrôleur-Maitre » :
   - Exécute récursivement la fonction d'induction de l'arbre **ID3()** en utilisant les paramètres suivants :
     - **Filtres** : une liste de paires (attribut, valeur) pour un attribut nominal ou (attribut, valeur, signe) pour un numérique afin de filtrer les données à un certain niveau de l'arbre.
     - **Chemin d'entrée** : localisation du fichier (bloc) de données d'apprentissage
     - **ClassePlusCommune** : la classe majoritaire à l'itération précédente
     - **Attributs** : liste des attributs à considérer pour la segmentation d'un nœud
2. Section tâche Map :
   - **La fonction Map(clé, donnée)**
3. Section tâche Reduce
   - **La fonction Reduce(attribut, vecteur de distributions des classes)**

MapReduce-Tree Algorithm (2)

```
1  ID3(Filtres, CheminEntree, ClassePlusCommune,Attributs)
2      #noeud est un noeud de l'arbre de décision
3      noeud = {creer un nouveau noeud}
4      if (attributs est vide) then
5          labeliser le noeud avec ClassePlusCommune
6      return noeud
7      endif
8  # res est une structure de données contenant le résultat. recProcessed est le nombre de données traitees
9      res = runHadoopJob (CheminEntree, Filtres, Attributs) //lancement des workers map et redude
10     if (res.recProcessed == 0 or res.classEntropy == 0) then
11         labeliser le noeud avec res.ClassePlusCommune
12     return noeud
13     endif
14     maxAttr = {attribut avec le maximum de gain}
15     if (maxAttr est un attribut nominal ) then
16         foreach(v in maxAttr.values)
17             if (Filtres contient maxAttr) then
18                 Filtres[maxAttr].value = v
19             else
20             Filtres.Add(maxAttr,v)
21             endif
22             supprimer maxAttr des Attributs
23             noeud.noeuds.Add(ID3(Filtres,CheminEntree, res.ClassePlusCommune, Attributs))
24         endfor
25     else
26         Filtres.Add(maxAttr, maxAttr.threshold, -1)
27         noeud.noeuds.Add(ID3(Filtres,CheminEntree, res.ClassePlusCommune, Attributs))
28         Filtres.Add(maxAttr, maxAttr.threshold, 1)
29         noeud.noeuds.Add(ID3(Filtres,CheminEntre, res.ClassePlusCommune, Attributs))
30     endif
31 return noeud
```

MapReduce-Tree Algorithm (3)

```
1  map(cle, pattern)
2      Input: cle = le numéro de l'enregistrement dans le fichier de données
3             pattern = enregistrement ou observation ou donnée
4
5  #Filtres est un ensemble de filtres lus à partir du setup lors du lancement de la tache Map
6  #Attributs is est l'ensemble des attributes de la fonction setup
7  #distrib[][] est un tableau qui contient la distribution des classes pour chaque attibut
8
9  if (acceptRecord(pattern, Filtres)) then
10     distrib[][] = { calculer la distribution des classes pour chaque attribut des Attributs }
11     foreach(attr in Attributs)
12         emit(attr, distrib[attr])
13     endfor
14 endif
```

MapReduce-Tree Algorithm [4]

```
reduce(attr, ldistrib)
    Input: attr = un attribut
    ldistrib = une liste de vecteurs des distributions de classes
val est un vecteur de 0
ret est une structure de {recProcessed, classEntropy, threshold, ClassePlusCommune, informationGain}
foreach(distrib in ldistrib)
    val[] = val[] + distrib[] #sommation de l'ensemble des distributions
endfor
ret.recProcessed = calculateTotalRecords(val) # le nombre d'indvidus traités au niveau du noeud
ret.classEntropy = calculateClassEntropy(val) # calcul de l'entropie du dataset au niveau du noeud
ret.threshold = calculateThreshold(attr, val) # seui dans le cas d'un attribut numérique
ret.mostCommonClass = calculateMostCommonClass(val) # calcul de la classe majoritaire au sein du dataset
ret.informationGain = calculateInformationGain(val) # calcul du gain d'information de l'attribut
emit(attr,ret) # générer une paire clé-valeur
```

Illustration à l'aide de l'exemple : polices d'assurances

- Soit le jeu de données sur la classification du risque d'une police d'assurances :
  - Filtres = {}
  - Attributs = {Age, TypeVoiture}
  - ClassePlusCommune={}
  - Création d'un nœud racine : 0
- On suppose que nous avons 2 nœuds worker et un nœud Maitre.
  - La répartition des données, est effectuée horizontalement, sur 2 blocs

idE	Age	TypeVoiture	Classe (Risque)
0	23	Familial	Haut
1	17	Sport	Haut
2	43	Sport	Haut
3	68	Familial	Bas
4	32	Camion	Haut
5	20	Familial	Bas
6	75	Sport	Haut
7	18	Sport	Bas
8	24	Sport	Haut
9	36	Camion	Bas
10	56	Familial	Haut
11	55	Familial	Bas

Suite exemple (2) : répartition des données en 2 blocks

Block du nœud Worker 1

IdE	Age	TypeVoiture	Classe
0	23	Familial	Haut
1	17	Sport	Bas
2	43	Sport	Haut
3	68	Familial	Bas
4	32	Camion	Haut
5	20	Familial	Bas

Block du nœud Worker 2

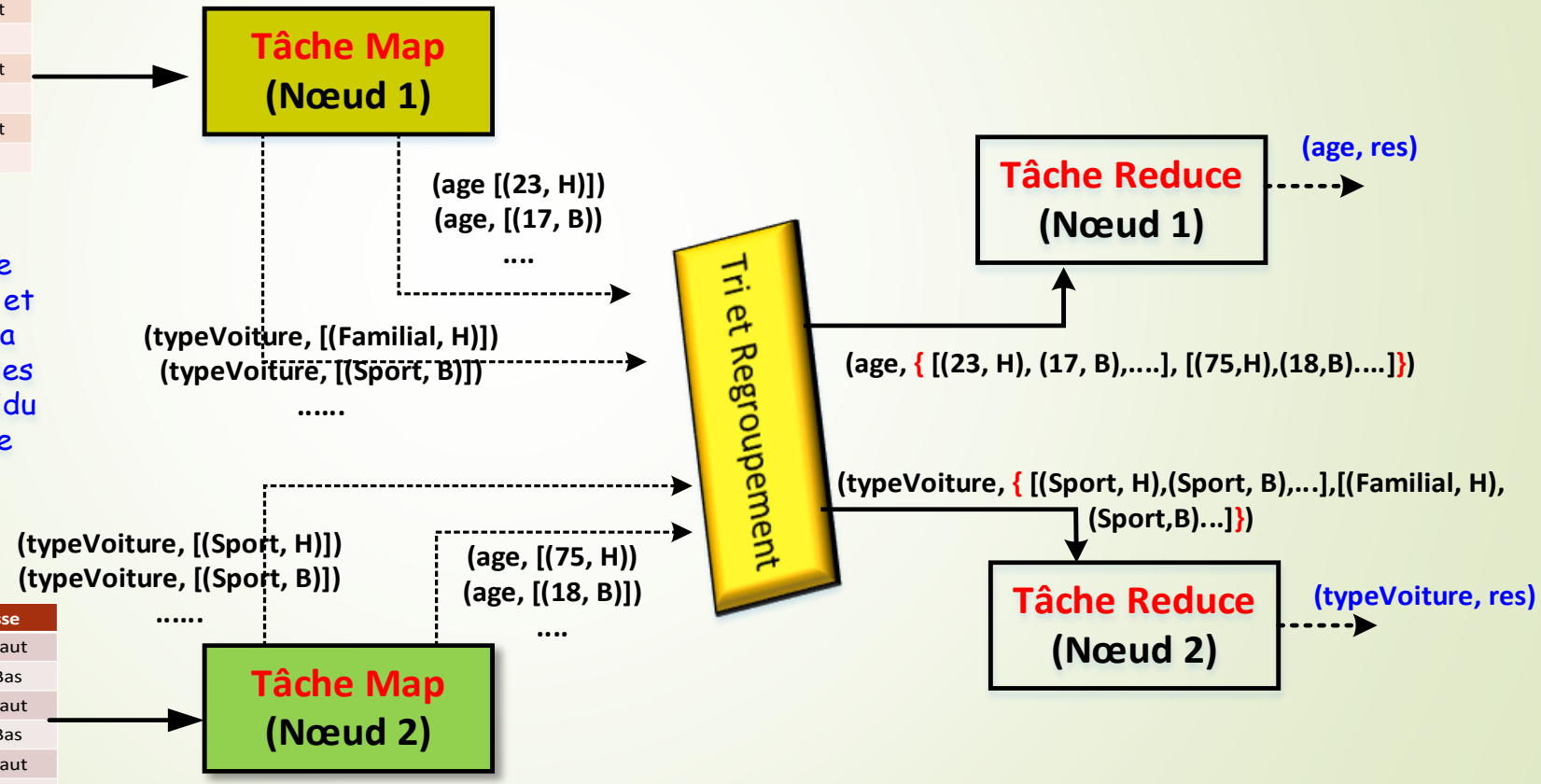
IdE	Age	TypeVoiture	Classe
6	75	Sport	Haut
7	18	Sport	Bas
8	24	Sport	Haut
9	36	Camion	Bas
10	56	Familial	Haut
11	55	Familial	Bas

Suite exemple (3) : Appel de ID3()→ 2 tâches Map et 2 tâches Reduce en parallèle

IdE	Age	TypeVoiture	Classe
0	23	Familial	Haut
1	17	Sport	Bas
2	43	Sport	Haut
3	68	Familial	Bas
4	32	Camion	Haut
5	20	Familial	Bas

Lecture séquentielle des enregistrements et appel respectif de la fonction Map() avec les arguments ide, individu pour chaque ligne de chaque block 1, 2

IdE	Age	TypeVoiture	Classe
6	75	Sport	Haut
7	18	Sport	Bas
8	24	Sport	Haut
9	36	Camion	Bas
10	56	Familial	Haut
11	55	Familial	Bas



Suite exemple (4) : déroulement

- Lors de l'appel d'ID3({}, chemin,{},{Age, TypeVoiture})
- La tâche Map du nœud 1 :
  - pour chaque ligne du block, il y aura un appel de la fonction Map(#ligne, individu) qui produira une séquence de paires clé-valeur. Le but est de briser chaque ligne en clés-valeurs.
  - Exemple :
    - La première ligne est composé de l'individu (23, Familial, Haut). La Map() produira la séquence suivante : (age, [(23, Haut)]), (typeVoiture, ([Familial,Haut]). Chaque valeur d'une clé représente une distribution par rapport à la classe.
    - Le même traitement se poursuit pour les autres lignes.
- Idem pour la tâche Map du Nœud 2 en manipulant en parallèle le bloc.
- Une fois ces deux tâches sont terminées, le système (Hadoop ou Spark) active le tri et le regroupement dans l'ensemble des séquences de paires clés-valeurs produites par les 2 tâches Map. Le résultat est une séquence de paire clé-valeur où les distributions associées à une même clé sont regroupées.
  - La valeur de chaque clé est une liste de vecteurs de distributions des classes
- Ensuite, chaque paire clé-valeur est acheminée vers la tâche Reduce appropriée en utilisant une fonction de hachage sur la clé pour localiser le Reducer dédié

Suite exemple (5) : déroulement

	Haut	Bas
Familial	2	3
Sport	3	2
Camion	1	1

- Chaque tâche Reduce a pour but :

1. Sommer les distributions pour chaque clé.

Par exemple, sur le nœud-worker 2, l'attribut TypeVoiture aura la distribution suivante, comme le montre la table.

2. Mise à jour de la structure ret :

- ret.redProcessed = 12 individus traités dont 7 de la classe haut et 5 de la classe bas
- ret.classEntropy = $-7/12 \log_2(7/12) -5/12 \log_2(5/12)= 0.97$
- ret.threshold = {} c'est un attribut nominal.
- ret.ClassePlusCommune = Haut.
- ret.informationGain= $0.97 – [5/12 I(2,3) + 5/12 I(3,2) + 2/12 I(1,1)] = 0.97-0,96 =0.01$
- Émission de la paire (typeVoiture, ret)

Suite exemple (6) :déroulement

- Idem pour la tâche Reduce qui traite l'attribut l'age :

1. Sommer les distributions pour la clé Age.

Par exemple, sur le nœud-worker1, l'attribut Age aura la distribution suivante:

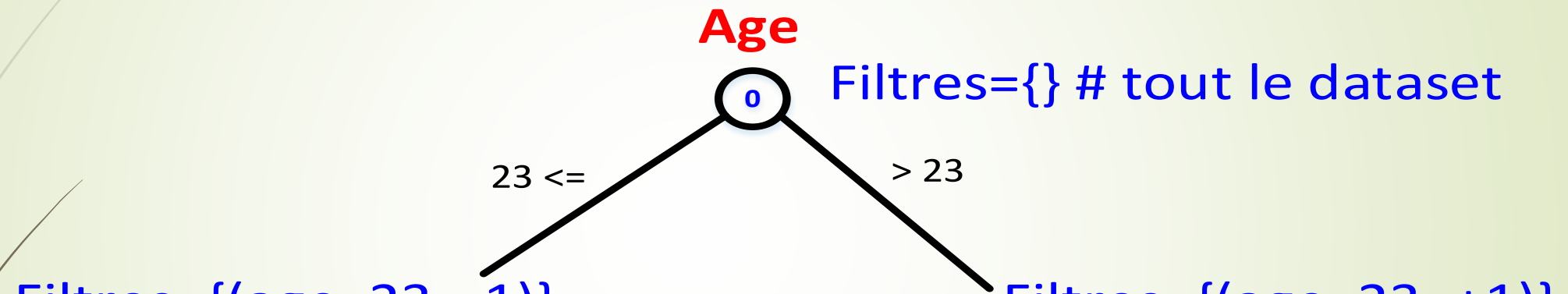
2. Mise à jour de la structure ret :

- ret.redProcessed = 12 individus traités dont 7 de la classe haut et 5 de la classe bas
- ret.classEntropy = $-7/12 \log_2(7/12) -5/12 \log_2(5/12)= 0,97$
- ret.threshold = 23 .
- ret.ClassePlusCommune = Haut.
- ret.informationGain= 0,17
- Émission de la paire (age, ret)

Age	Test (-/+)	Haut	Bas
17	- (<=)	1	0
	+ (>)	6	5
18	-	1	1
	+	6	5
20	-	2	1
	+	5	5
23	-	3	1
	+	4	4
.....les autres valeurs			

Suite Exemple (7)

- Le gain maximal est donné par l'attribut age, maxAttr={Age}



Filtres={(age, 23, -1)}

On considère que les individus dont l'age <= 23, Il y aura un autre appel ID3(Filtres, Chemin, {Haut}, {Age, TypeVoiture}) pour construire le prochain nœud de l'arbre (1)

Filtres={(age, 23, +1)}

On considère que les individus dont l'age > 23 , Il y aura un autre appel ID3(Filtres, Chemin, {Haut}, {Age, TypeVoiture}) pour construire le prochain nœud de l'arbre (2)

Demo avec pyspark : détection d'intrusion, 500K individus

IP[y]: Notebook KDDCupPySpark_Abdenour2019 Last Checkpoint: Nov 01 11:33 (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [10]:
```
import urllib
```

In [11]:
```
f = urllib.urlretrieve ("http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data.gz", "kddcup.data.gz
```

In [12]:
```
data_file = "./kddcup.data.gz"
raw_data = sc.textFile(data_file)

print "Train data size is {}".format(raw_data.count())
```

```
Train data size is 4898431
```

In [20]:
```
raw_data.first()
```

Out[20]:
```
u'0,tcp,http,SF,215,45076,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,
0.00,0.00,0.00,normal.'
```

In [21]:
```
ft = urllib.urlretrieve("http://kdd.ics.uci.edu/databases/kddcup99/corrected.gz", "corrected.gz")
```

In [22]:
```
test_data_file = "./corrected.gz"
test_raw_data = sc.textFile(test_data_file)

print "Test data size is {}".format(test_raw_data.count())
```

```
Test data size is 311029
```

In [23]:
```
from pyspark.mllib.regression import LabeledPoint
from numpy import array

csv_data = raw_data.map(lambda x: x.split(","))
test_csv_data = test_raw_data.map(lambda x: x.split(","))
```

IP[y]: Notebook KDDCupPySpark_Abdenour2019-uneAutre (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

```
print "Prediction made in {} seconds. Test accuracy is {}".format(round(tt,3), round(test_accuracy,4))
```

```
Prediction made in 27.803 seconds. Test accuracy is 0.915
```

In [31]:
```
print "Learned classification tree model:"
print tree_model.toDebugString()
```

```
Learned classification tree model:
DecisionTreeModel classifier of depth 4 with 23 nodes
  If (feature 22 <= 88.5)
   If (feature 38 <= 0.7949999999999999)
    If (feature 36 <= 0.49)
     If (feature 34 <= 0.9550000000000001)
      Predict: 0.0
     Else (feature 34 > 0.9550000000000001)
      Predict: 1.0
    Else (feature 36 > 0.49)
     If (feature 2 in {0.0,42.0,24.0,20.0,46.0,57.0,60.0,44.0,27.0,12.0,7.0,3.0,18.0,67.0,43.0,26.0,55.0,58.0,36.0,4.0,47.0,15.
0})
      Predict: 0.0
     Else (feature 2 not in {0.0,42.0,24.0,20.0,46.0,57.0,60.0,44.0,27.0,12.0,7.0,3.0,18.0,67.0,43.0,26.0,55.0,58.0,36.0,4.0,47.
0,15.0})
      Predict: 1.0
   Else (feature 38 > 0.7949999999999999)
    If (feature 3 in {10.0,1.0,9.0,3.0,4.0})
     Predict: 0.0
    Else (feature 3 not in {10.0,1.0,9.0,3.0,4.0})
     Predict: 1.0
  Else (feature 22 > 88.5)
   If (feature 5 <= 2.0)
    If (feature 11 <= 0.5)
     Predict: 1.0
    Else (feature 11 > 0.5)
```

Bilan

- Le modèle MapReduce est très adapté pour l'apprentissage distribué d'un arbre de décision.
- Contrairement à l'algorithme parallèle SPRINT, il n'y a pas de communication directe de résultats intermédiaires et de distributions des classes entre les nœuds.
- Généralement, la performance est très proche de la réalité du moment qu'on utilise un large dataset et le passage à l'échelle est garantit avec MapReduce.
- Dans le cas de Spark, il n'y a pas d'écriture sur disque, par conséquent, il est plus rapide que Hadoop (~100 fois plus).

Bibliographie

- Parallel formulations of decision-tree classifications algorithms, A. Srivastava et al., IBM
- SPRINT : A scalable parallel classifier for data mining, J. Shafer et al., IBM, VLDB Conference, 1996
- MR-Tree : A scalable MapReduce Algorithm for building decision tree, V. Purdila et al., Journal of applied computer & mathematics, 16:8, 2014