Construction interactive d'un arbre de décision - Projet FoilProp 2020

Partie I - FOILP1

Le but de cette partie est de comprendre et réaliser un algorithme qui construit un ensemble de règle couvrant tous les positifs et ne couvrant aucun négatif. Afin de tester notre programme, nous avons eu accès à plusieurs jeux d'essai.

Le choix d'implémentation :

Afin d'implémenter l'algorithme FOILP1, nous avons choisi d'opter pour de la programmation objet avec JAVA. Ce dernier est un bon choix car il y a une bibliothèque Weka qu'on peut exploiter afin de nous faciliter la tâche sur la lecture du fichier d'entrée.

L'algorithme :

L'algorithme qu'on va utiliser pour cette partie du projet se nomme *Algorithme de couverture séquentielle.*

Les classes utilisées :

Grâce à la bibliothèque Weka, nous avons eu accès à plusieurs classes afin de nous faciliter la lecture du fichier.

La classe Instances :

Cette classe prend en paramètre un Reader (ici le reader est un BufferReader du fichier de test passé en argument) et donc le fichier *arff*.
Grâce à cette classe, on peut énumérer les instances ou les attributs du fichier avec les méthodes enumerateInstances() et enumerateAttributes().

La classe Instance :

La classe Instance représente simplement une instance du fichier. Grâce à différentes méthodes, on peut accéder à la liste des attributs ainsi que la valeur, soit sur un format numérique, soit sur un format String.

La classe Attribute :

La classe Attribute représente un attribut du fichier arff. Comme pour Instance, avec différentes méthodes, on peut accéder à la liste des valeurs possibles, son nom etc..

Nous avons eu besoin de créer des classes afin d'implémenter le plus clairement possible notre algorithme.

La classe Literal :

Cette classe a comme attribut de classe :

- Un attribut de type String
- Une valeur de type String
- Le nombre d'itération de type entier
- Et son gain de type double

Cette classe permet de représenter la paire Attribut/Valeur, par exemple cheveux -> blond, cheveux étant l'attribut et blond étant sa valeur.

La classe EnsembleInstance :

Cette classe a comme attribut de classe :

- Un arrayList d'instances positives
- Un arrayList d'instances négatives

La classe EnsembleInstance permet de séparer les instances positives et les instances négatives avec un ensemble de méthode pour y accéder.

La classe Regle :

Cette classe a comme attribut de classe :

- Un arrayList de littéraux

La classe Regle permet de construire une règle spécifique (de la forme "SI Literal ET Literal ALORS Literal"). Il y a aussi une fonction nommée instanceVerifieRegle qui prend en paramètre une Instance et qui teste si la règle vérifie l'instance en question.

Les tests effectués :

Nous avons effectué différents tests afin de voir si nous obtenons des règles cohérentes.

Coup de soleil :

Voici les résultats sur le jeu de données *coup_de_soleil.arff*

```
--------FOILP1------
Rules :
R1 : SI creme = non ET cheveux = blond ALORS coup_de_soleil = oui, Positives=2
R2 : SI cheveux = roux ALORS coup_de_soleil = oui, Positives=1

positive covered : 3
```

On voit que notre programme affiche deux règles pour le jeu de données *coup_de_soleil.arff*, le premier couvre deux positifs et le deuxième couvre le dernier positif restant.

Météo :

Voici les résultats sur le jeu de données *weather.nominal.arff*

```
--------FOILP1------
Rules :
R1 : SI outlook = overcast ALORS play = yes, Positives=4
R2 : SI humidity = normal ET windy = FALSE ALORS play = yes, Positives=4
R3 : SI temperature = mild ET humidity = normal ALORS play = yes, Positives=2
R4 : SI windy = FALSE ET outlook = rainy ALORS play = yes, Positives=3

positive covered : 9
```

Pour l'exemple *weather.nominal.arff,* on remarque que le programme nous retourne quatre règles différentes qui couvre l'ensemble des positifs. Le problème est qu'un exemple peut être couvert par deux ou plusieurs règles. Nos règles sont donc trop générales et il faudrait donc "spécialiser" les règles afin qu'un exemple soit couvert par le moins de règles possibles. Il est aussi possible que dans un jeu de données, il existe des règles très spécifiques qui peuvent remettre en question un ensemble de règle. Pour cela nous avons donc la seconde partie du projet.

Partie II - FOILP2

Introduction

Nous avons vu dans la première partie que l'algorithme FOILP1 permettait d'avoir des règles plus "globales" dans le sens où nos règles contiennent peu d'attributs. Cela nous permettrait d'obtenir des règles valides dans 100% des cas. Mais le problème est que les règles générées par le premier algorithme pouvaient couvrir le même exemple. La marge d'erreur dans les données n'est pas prise en compte.

Le choix d'implémentation

Les consignes étaient de modifier légèrement l'algorithme précédent pour atteindre le but escompté. Cependant, nous avons préféré remettre le projet à plat pour repartir sur des bases neuves, néanmoins le principe de l'algorithme foil, lui, reste inchangé.

L'algorithme

Dans cette version de l'algorithme FOIL, nous l'avons adapté pour un arbre de décision. Dans un premier temps, nous calculons le gain de chaque attribut afin de déterminer qui pourra au mieux séparer les cas positifs (un cas est positif si le dernier attribut du cas a comme valeur la première des valeurs possible) des cas négatifs. Puis nous commençons par créer *n fils* (*n* étant le nombre de littéraux pour l'attribut choisi, celui qui a le meilleur gain) et enfin récursivement, on applique de nouveau l'algorithme sur les attributs restants tout en ayant appliqué le littéral ayant le plus gros gain.

Cette méthode ne nous laissera que des feuilles pures avec des règles assez spécifiques. Pour obtenir des règles plus généralistes, nous avons ajouté deux filtres :

- *entropie_THRESHOLD (par défaut 0.55 qui correspond à l'entropie d'une règle ayant 7 instances positives et une instance négative)* nous permet de définir l'entropie minimal requis pour considérer une règle. Si l'attribut analysé a un entropie inférieur à la valeur définie par l'utilisateur, nous considérons l'arborescence des littéraux comme formant une règle. De cette façon, nous faisons en sorte d'accepter qu'une règle couvre une partie d'instances négatives que nous jugeons plus ou moins négligeable. Dans la partie tests effectués, nous verrons la différence d'exécution si nous changeons l'entropie. Donc pour résumer, plus nous augmentons cet indicateur, plus nous acceptons d'avoir des instances négatives. Dans l'autre sens, plus nous diminuons plus nous souhaitons nous rapprocher de feuilles pures. Cette valeur a été mise à 0.55 de base, ce qui correspond à l'entropie d'une règle ayant 7 instances positives et une instance négative.

- *MINIMUM_INSTANCES (par défaut 0.005 soit 0.5%)* nous permet d'enlever des feuilles, même pures, qui ne représentent que peu de cas. Plus nous diminuons cette valeur plus nous allons garder de feuilles, donc couvrir plus de cas, mais aussi augmenter le nombre de règles. Cette valeur est initialisée à 0.005 (=0.5%). Une feuille devra contenir au moins N*0.005 instance pour être considérée comme règle, avec N le nombre total d'instance.

Les tests effectués :

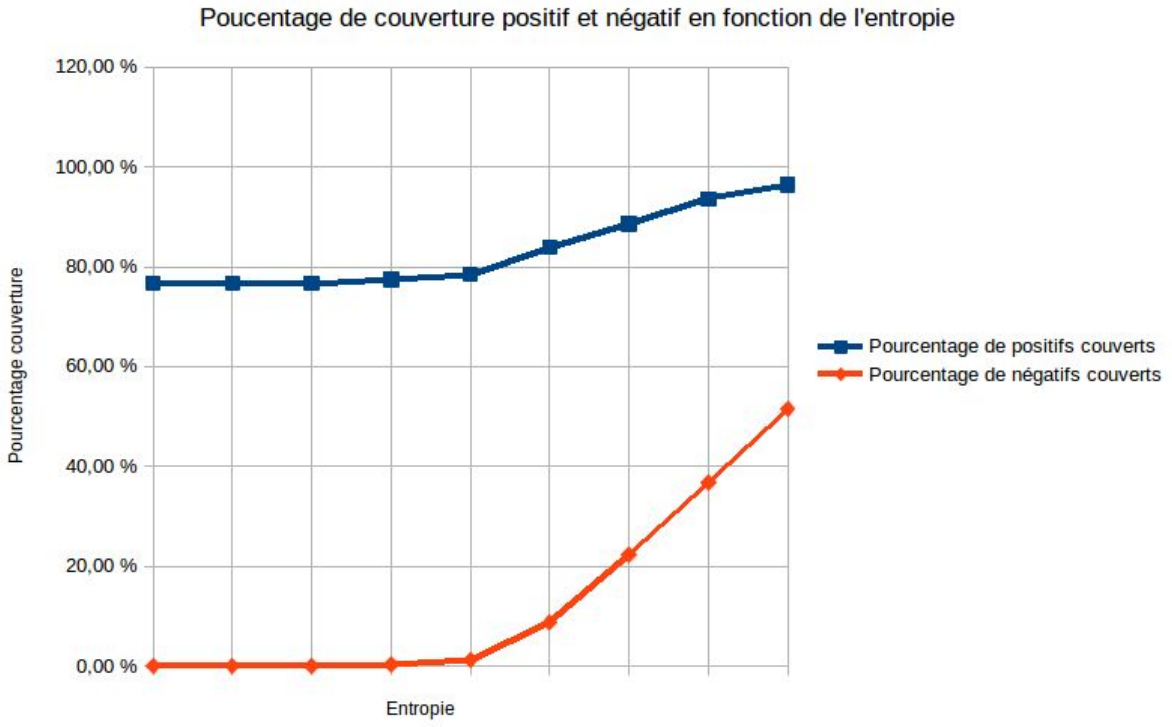
Nous avons fait effectué des tests sur des plus grands jeux d'essais tout en comparant les résultats de FOILP1 et FOILP2.

Tic-tac-toe :

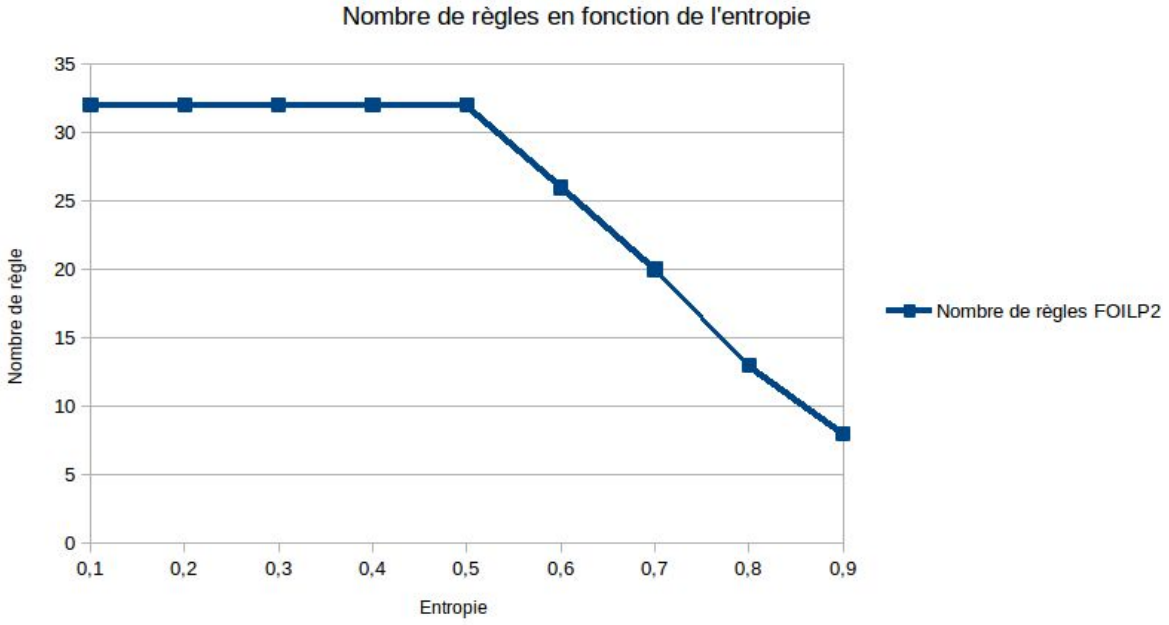
Voici les résultats sur le jeu de données *tic-tac-toe.arff*, avec les filtres par défaut :

```
positive covered : 491/626(78.4345%, we would like this to be close to 100% to cover most of positive instances)
negative covered : 4/332(1.2048193%, we would like this to be close to 0% to cover no negative instances)
```

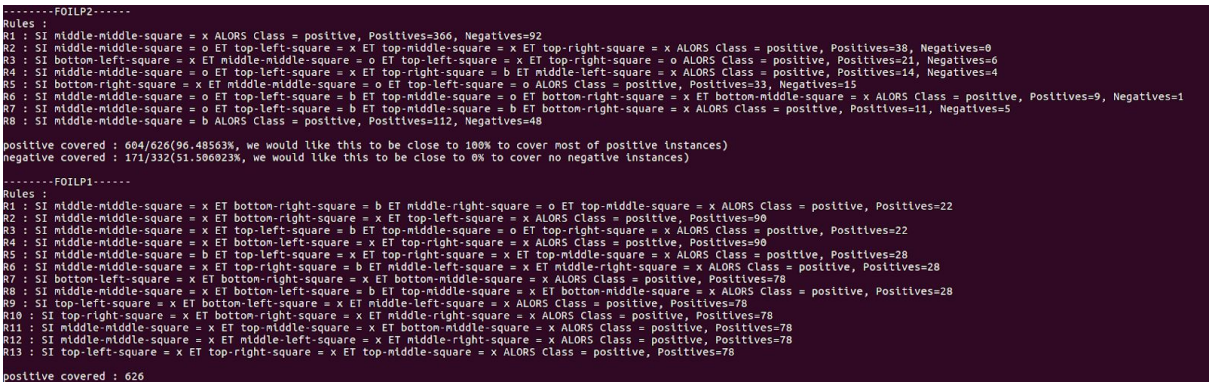
Il y a 32 règles qui ont été générées par l'algorithme FOILP2. L'affichage ci-dessus montre que 491 instances positives ont été couvertes par les 32 règles. Il y a 78% des cas positifs couverts. 4 instances négatives ont été couvertes par les 32 règles soit 1.2%. Nous pouvons faire varier l'*entropie_THRESHOLD* afin de constater la progression de la couverture des cas positifs et des cas négatifs.



Ce graphique montre le pourcentage de couverture des cas positifs et négatifs en fonction de l'entropie pour l'exemple *tic-tac-toe.arff.* Nous remarquons que pour cet exemple, le pourcentage de couverture reste fixe de l'entropie 0.1 jusqu'à l'entropie 0.4. Dans cette tranche, il n'y a pas de cas négatif couvert par nos règles. En revanche à partir d'un entropie de 0.5, nous avons une croissance du pourcentage de négatifs couverts en dépassant les 50 % (soit 171 sur 332 cas négatifs). Nous pouvons noter une augmentation importante en ce qui concerne le pourcentage de positifs couverts allant jusqu'à plus de 96 % (soit 604 sur 626). Ce graphique montre donc l'impact du filtre que nous avons mis en place. Donc plus l'entropie est faible, plus nous aurons des feuilles "pures", c'est à dire l'ensemble des règles qui couvrent très peu ou pas de cas négatifs.



Le graphique ci-dessus montre le nombre de règles en fonction de l'entropie. Nous remarquons que, comme dans le premier graphe, les changements s'effectuent à partir d'une entropie de 0.5. Entre 0.1 et 0.4 nous avons toujours 32 règles. Puis il y a une décroissance conséquente, pour un entropie 0.9 nous avons seulement 8 règles. Nous obtenons moins de règles car nous acceptons de couvrir un certain nombre de cas négatifs.



Le résultat ci-dessus est la comparaison entre l'algorithme FOILP2 et FOILP1 avec un entropie de 0.9. Nous remarquons qu'il y a une différence sur le nombre de règles. De plus, FOILP1 couvre l'ensemble des cas positifs (avec beaucoup de redondance) alors que FOILP2 couvre 96.5 % des cas positifs avec plus de 50 % de cas négatifs couverts. Ici FOILP2 est plus général.

Ci-dessus, nous avons la comparaison entre FOILP2 ET FOILP1 avec l'entropie par défaut (0.55). Nours remarquons qu'il y a plus de règles dans FOILP2 que dans FOILP1. Nous pouvons donc dire que FOILP2 est plus spécialisé.

Mushroom valid :

Voici les résultats sur le jeu de données *mushroom_valid.arff*, avec les filtres par défaut :

```
Rules :
R1 : SI spore-print-color = k ET odor = a ET gill-color = k ALORS poisonous = 0, Positives=11, Negatives=0
R2 : SI spore-print-color = n ET odor = a ET cap-surface = y ET gill-color = n ALORS poisonous = 0, Positives=13, Negatives=1
R3 : SI cap-color = y ET odor = a ET gill-color = g ALORS poisonous = 0, Positives=11, Negatives=0
R4 : SI odor = a ET gill-color = w ALORS poisonous = 0, Positives=30, Negatives=2
R5 : SI odor = l ALORS poisonous = 0, Positives=109, Negatives=11
R6 : SI spore-print-color = k ET odor = n ALORS poisonous = 0, Positives=363, Negatives=39
R7 : SI spore-print-color = n ET odor = n ALORS poisonous = 0, Positives=325, Negatives=40
R8 : SI spore-print-color = w ET cap-color = c ET odor = n ALORS poisonous = 0, Positives=11, Negatives=1

positive covered : 873/1000(87.3%, we would like this to be close to 100% to cover most of positive instances)
negative covered : 94/693(13.564214%, we would like this to be close to 0% to cover no negative instances)
```

Dans cet exemple, nous avons fait en sorte que le cas valide soit poisonous = 0 pour que ce soit plus pratique. (Nous voulons les champignons qui ne sont pas empoisonnés). On remarque que nous avons “seulement” 8 règles sur ce grand jeu de données et que nous couvrons 87.3 % des cas positifs et 13.56 % des cas négatifs, ce qui pourrait expliquer pourquoi nous avons seulement 8 règles.

```
Rules :
R1 : SI spore-print-color = k ET odor = a ET gill-color = k ALORS poisonous = 0, Positives=11, Negatives=0
R2 : SI cap-color = y ET odor = a ET gill-color = g ALORS poisonous = 0, Positives=11, Negatives=0
R3 : SI spore-print-color = n ET odor = a ET gill-color = w ALORS poisonous = 0, Positives=16, Negatives=0
R4 : SI odor = l ET population = n ET gill-color = k ALORS poisonous = 0, Positives=10, Negatives=0
R5 : SI odor = l ET population = n ET gill-color = g ALORS poisonous = 0, Positives=12, Negatives=0
R6 : SI odor = l ET population = s ET gill-color = n ALORS poisonous = 0, Positives=10, Negatives=0
R7 : SI spore-print-color = k ET odor = n ET gill-color = k ET population = a ALORS poisonous = 0, Positives=13, Negatives=0
R8 : SI spore-print-color = k ET odor = n ET cap-shape = x ET cap-surface = f ET gill-color = n ALORS poisonous = 0, Positives=23, Negatives=0
R9 : SI spore-print-color = k ET stalk-color-below-ring = g ET odor = n ET cap-shape = f ET gill-color = n ALORS poisonous = 0, Positives=9, Negatives=0
R10 : SI spore-print-color = k ET stalk-surface-below-ring = f ET odor = n ET gill-color = h ALORS poisonous = 0, Positives=13, Negatives=0
R11 : SI spore-print-color = k ET stalk-color-below-ring = p ET odor = n ET cap-surface = y ET gill-color = p ALORS poisonous = 0, Positives=13, Negatives=0
R12 : SI spore-print-color = k ET stalk-color-below-ring = w ET odor = n ET gill-color = p ET population = a ALORS poisonous = 0, Positives=13, Negatives=0
R13 : SI spore-print-color = k ET stalk-color-below-ring = w ET odor = n ET gill-color = p ET population = y ALORS poisonous = 0, Positives=16, Negatives=0
R14 : SI spore-print-color = k ET stalk-color-above-ring = p ET stalk-color-below-ring = g ET odor = n ET gill-color = u ALORS poisonous = 0, Positives=10, Negatives=0
R15 : SI spore-print-color = k ET odor = n ET cap-color = n ET gill-color = w ALORS poisonous = 0, Positives=28, Negatives=0
R16 : SI spore-print-color = k ET odor = n ET cap-color = g ET gill-color = w ALORS poisonous = 0, Positives=24, Negatives=0
R17 : SI spore-print-color = n ET odor = n ET cap-color = n ET cap-surface = s ET population = a ALORS poisonous = 0, Positives=9, Negatives=0
R18 : SI spore-print-color = n ET odor = n ET population = s ET gill-color = k ALORS poisonous = 0, Positives=11, Negatives=0
R19 : SI spore-print-color = n ET odor = n ET population = s ET gill-color = p ALORS poisonous = 0, Positives=12, Negatives=0
R20 : SI spore-print-color = n ET odor = n ET population = v ET gill-color = n ALORS poisonous = 0, Positives=24, Negatives=0
R21 : SI spore-print-color = n ET stalk-color-below-ring = p ET odor = n ET population = v ET gill-color = u ALORS poisonous = 0, Positives=11, Negatives=0
R22 : SI spore-print-color = n ET stalk-color-above-ring = g ET odor = n ET cap-color = n ET population = y ALORS poisonous = 0, Positives=14, Negatives=0
R23 : SI spore-print-color = n ET stalk-color-above-ring = p ET odor = n ET cap-color = n ET population = y ALORS poisonous = 0, Positives=16, Negatives=0
R24 : SI spore-print-color = n ET odor = n ET cap-color = e ET cap-surface = f ET population = y ALORS poisonous = 0, Positives=23, Negatives=0
R25 : SI spore-print-color = w ET odor = n ET cap-color = c ET population = v ALORS poisonous = 0, Positives=11, Negatives=0

positive covered : 363/1000(36.3%, we would like this to be close to 100% to cover most of positive instances)
negative covered : 0/693(0.0%, we would like this to be close to 0% to cover no negative instances)
```

Nous avons de nouveau exécuté l’algorithme sur le jeu *mushroom_valid.arff* mais cette fois-ci en mettant un entropie de 0.1. Nous pouvons voir que nous avons maintenant 25 règles qui sont uniquement des feuilles pures (elles ne couvrent aucun cas négatif). Les règles sont donc plus “spécialisés”, elles ont par conséquent beaucoup d’attributs. Cependant, le problème est que nous couvrons seulement 36.3 % des cas positifs malgré le fait que nous ne couvrons aucun cas négatif.

```
Rules :
R1 : SI odor = a ALORS poisonous = 0, Positives=100, Negatives=17
R2 : SI odor = l ALORS poisonous = 0, Positives=109, Negatives=11
R3 : SI odor = n ALORS poisonous = 0, Positives=719, Negatives=111

positive covered : 928/1000(92.8%, we would like this to be close to 100% to cover most of positive instances)
negative covered : 139/693(20.05772%, we would like this to be close to 0% to cover no negative instances)
```

Quand nous exécutons l’algorithme avec un entropie de 0.9, on remarque que nous n’obtenons plus que 3 règles. Ces dernières sont très générales (il n’y a qu’un seul attribut par règle). Le pourcentage de positifs couverts est très élevé (92.8 %) comparé à un entropie de 0.1 et le pourcentage de négatifs couverts est, quant à lui, de 20 %. Sur les 693 cas négatifs, nous en couvrons 139, c’est un risque à prendre si nous voulons couvrir le maximum de cas positifs.

FOILP1 n’arrive pas à aller jusqu’au bout.

Critique du projet

Grâce à ce projet, nous avons pu comparer deux algorithmes permettant de couvrir le maximum de cas positifs dans un exemple. FOILP1 va, dans tous les cas, toujours couvrir 100 % des cas mais le problème de celui-ci est qu'il y aura beaucoup de redondances dans des jeux d'essais assez conséquent (comme *mushroom_valid.arff* par exemple). Pour essayer d'y remédier, nous avons donc l'algorithme FOILP2 qui, grâce à deux paramètres, va pouvoir essayer de spécialiser les règles afin d'éviter les redondances. Le soucis de ce dernier est que nous risquons de couvrir aussi des cas négatifs si nous augmentons trop l'entropie. C'est donc un risque à prendre en compte.

La conclusion est qu'il faut adapter notre jeu de données en fonction de nos attentes. Est-ce que nous voulons couvrir le maximum de cas possibles tout en ayant le moins de règles possibles ? Cela dépend donc de notre utilisation.