Arbre de décision – Evaluation de l'apprentissage - Elagage

Sources du cours et plan

Cours n°2 d'Aomar Osmani, Apprentissage Symbolique, MICR 2006-2007

Decision Trees, Occam's Razor, and Overfitting, William H. Hsu
Department of Computing and Information Sciences, KSU http://www.kddresearch.org

Cours de Machine Learning, Merlin Holzapfel & Martin Schmidt

Plan :

Biais d'apprentissage des arbres de décision
Evaluation empirique
Sur-apprentissage
Techniques d'élagage

Biais d'apprentissage d'algorithmes de calcul d'arbres de décision

Biais en apprentissage [Mitchell, 97]

- Biais de Preference / Biais de Langage
  - Biais de préférence
    - Dynamique: ordonne les hypothèses de l'espace de recherche: *heuristique de recherche*
  - Language bias
    - Souvent statique, s'implémente à travers le langage d'hypothèses (langage cible)
    - Restriction a priori *de l'espace de recherche*

Rasoir d'Occam: un biais de préférence

- Rasoir d'Occam: pour
  - Moins d'hypothèses courtes que d'hypothèses longues
    - 2 fois moins de chaînes de bits de longueur n que de longueur n+1, $n \geq 0$
    - Si une hypothèse courte colle aux données, cela a moins de chance d'être une coïncidence
  - Justification / compromis
    - Toutes autres choses étant égales par ailleurs, des modèles complexes semblent se généraliser moins bien
- Rasoir d'Occam: contre
  - *taille*(*h*) repose sur *H* - définition circulaire?
  - Une représentation interne à l'algorithme définit quelles *h* sont courtes – arbitraires ?
  - Il y a beaucoup de manières de définir de petits ensembles d'hypothèses

Evaluation empirique des systèmes d'apprentissages

Evaluation empirique des hypothèses produites

- Evaluation / ensemble d'apprentissage
- Evaluation de la prédictivité des hypothèses produites : ensemble d'exemples séparés en ensemble d'apprentissage A et ensemble test T.
  - Apprentissage d'une hypothèse H avec A
  - Evaluation de la prédiction de H sur T

Estimation de l'erreur réelle d'une hypothèse

- Matrice de confusion ou de contingence

		Prédite	
		Positif	Négatif
Actuelle	Positif	TP	FN
	Négatif	FP	TN

TP : nombre de positifs classés positifs

FP : nombre de négatifs classés positifs

TN : nombre de négatifs classés négatifs

FN : nombre de positifs classés négatifs

Estimation de l'erreur réelle d'une hypothèse

- Erreur(h) = 100-100*(TP+TN)/(TP+TN+FP+FN)
- Exemple :

		Prédite	
		Positif	Négatif
Actuelle	Positif	400	100
	Négatif	200	300

- Erreur(h) = 100-100*(TP+TN)/(TP+TN+FP+FN) = 70%

Découpage des données

Données d'apprentissage

utilisées par les algorithmes d'apprentissage pour générer les classeurs

Données d'optimisation ou de validation

utilisées pour optimiser les paramètres de ces classeurs

Données de test

utilisées pour calculer les taux d'erreurs sur classeurs optimisés

Ensemble test

- Pour prédire les performance d'un classeur sur les données non observées, on a besoin de données non utilisées pour la construction de ce classeur :
  - l'ensemble test doit être indépendant de l'ensemble d'apprentissage
  - les deux ensembles doivent avoir des exemples représentatifs du problème

Prédiction des performances

Quelques méthodes :

- Hold-out
- hold-out stratifié
- hold-out répété
- Validation croisée
- Leave-one-out
- ...

Hold-out

- Un sous-ensemble pour le test, le reste pour l'apprentissage/validation (1/3-2/3)
- Stratifié : équilibre des classes respecté
- Répété : l'erreur est la moyenne sur plusieurs hold-out (estimation de la variance de l'estimateur)
- Dilemme
  - Pour obtenir un bon classeur, utiliser le plus de données possible
  - Pour avoir une bonne estimation de l'erreur, utiliser le plus de données de test possible

k-validation croisée (k-fold cross-validation)

- Décider du nombre k fixe de partitions des exemples
- Couper les données en k partitions égales
- Utiliser une partition pour le test, les k-1 autres pour l'apprentissage
- Répéter le processus k fois (par permutation circulaire)
- Erreur = moyenne des erreurs sur les k partitions

k-validation croisée (k-fold cross-validation)

- Souvent stratifiée
- 10 partitions très utilisées en pratique (bon compromis entre le nombre d'exemples pour l'apprentissage et pour le test)

Leave-one-out

Cas particulier de la k-validation croisée où k = nombre d'exemples

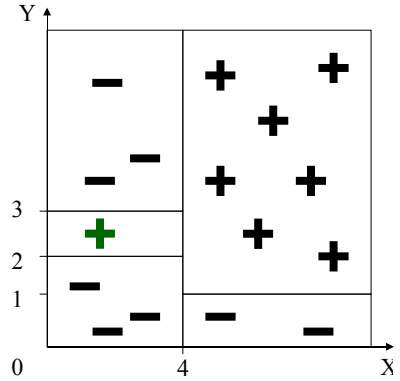
- Avantage :
  - chaque itération utilise un nombre élevé de données pour la phase d'apprentissage
  - déterministe : pas de variance
- Inconvénients :
  - algorithme d'apprentissage exécuté n fois
  - pb de garantie de la stratification des exemples

=> utilisé quand n < 100

Sur-apprentissage "over-fitting"

Sur-apprentissage - Exemple

ENTRÉE :



Les règles apprises

$$R_1 : (X > 4 \wedge Y > 1)$$

$$R_2 : (X \leq 4 \wedge Y \leq 3 \wedge Y > 2)$$

Arbre de décision et sur-apprentissage

- Si bruit dans les exemples ou trop peu d'exemples,
  - les performances des hypothèses produites peuvent se dégrader sur l'ensemble de test, tout en restant excellentes sur l'ensemble d'apprentissage -> calage ou « overfitting »
  - Plus précisément: l'hypothèse *h* sur-apprend étant donné un ensemble d'apprentissage *D* si ∃ une autre hypothèse *h'* telle que $erreur_D(h) < erreur_D(h')$ mais $erreur_{test}(h) > erreur_{test}(h')$
  - Causes: ensemble d'apprentissage trop petit; bruit; coïncidence

Sur-apprentissage en apprentissage supervisé

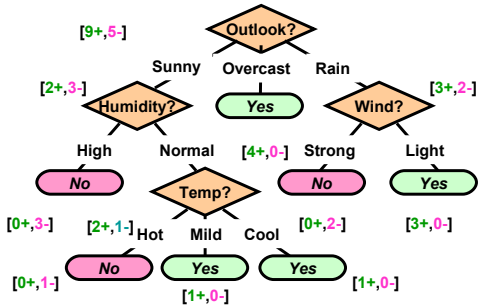
- Elagage : on perd de la précision sur l'ensemble d'apprentissage, mais on gagne en terme de prédiction
  - A priori
    - **Sélectionner les attributs *pertinents* (i.e., utile à la construction du modèle). Pb: demande une mesure permettant de prédire quels attributs sont pertinents**
  - Pré-élagage :
    - **décide ou non de continuer à développer un certain noeud. On sélectionne un ensemble de validation V, et on stoppe quand la performance de *h* se dégrade sur V**
  - Post-élagage
    - **Construit le modèle complet, puis élague les éléments qui contribuent au sur-apprentissage**

Arbre de décision - exemple

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
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	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D14	Rain	Mild	High	Strong	No
D13	Overcast	Hot	Normal	Weak	Yes

Sur-apprentissage dans les arbres de décisions: exemple

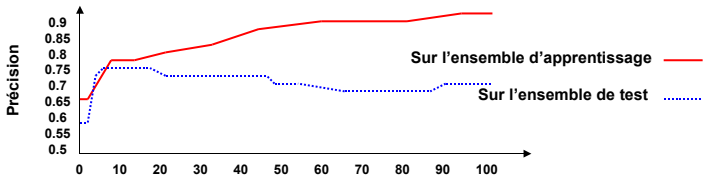
- **Arbre de décision construit pour la classe *PlayTennis:***



- **Supposons qu'arrive le nouvel exemple:** <*Sunny*, *Hot*, *Normal*, *Strong*, +>
- **L'arbre construit lui assigne une classe incorrecte**
- **Comment l'arbre doit-il être mis à jour?**

Apprentissage d'arbre de décision : Comment éviter le sur-apprentissage

- A priori
  - Filtre d'attribut, développement de wrapper pour la sélection de sous-ensembles d'attributs pertinents.
- Pré-élagage
  - Arrête le développement de l'arbre avant que tous les exemples de l'ensemble d'apprentissage soient bien classés



D'après (Mitchell, 97), p67

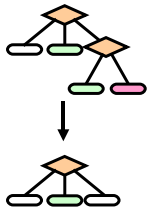
- Post-élagage: Construire l'arbre complet et enlever les noeuds qui sont non nécessaires

Apprentissage d'arbres de décision et sur-apprentissage

- Méthodes pour évaluer les sous-arbres à élaguer (pre ou post-pruning):
  - Validation croisée: sélectionner un ensemble de validation pour évaluer l'utilité de T
  - Test statistique: évaluer si la régularité observée a une chance d'être due au hasard (test du chi2 ou de fisher).
  - Minimum Description Length (MDL)
    - Complexité additionnelle de *T* plus grande que celle de retenir des *exceptions*?
    - Compromis: coder le *modèle* contre coder l'erreur résiduelle

Reduced-Error Pruning (REP)

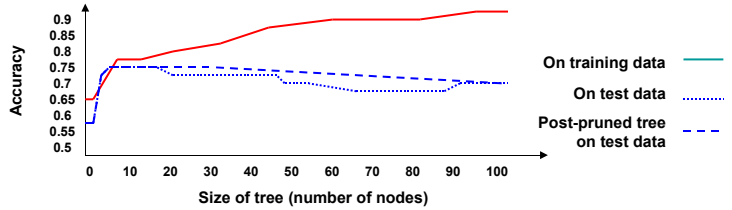
- Post-élagage,approche par Validation Croisée
  - Enlever de sous-arbre de racine *noeud*
  - *Noeud* devient une feuille (avec pour étiquette la classe majoritaire des exemples de ce noeud)



- Algorithme *Reduced-Error-Pruning* (*D*)
  - Partitionner *D* en D_{train} (training / "growing"), $D_{validation}$ (validation / "pruning")
  - Construire l'arbre complet *T* en utilisant *ID3* sur D_{train}
  - JUSQU'A CE QUE la précision sur $D_{validation}$ baisse DO
    FOR chaque noeud interne *candidat* de *T*
      ***Temp*[*candidat*] ← Elague(*T*, *candidat*)**
      ***Précision*[*candidat*] ← *Test* (*Temp*[*candidat*], $D_{validation}$)**
    T ← *T'* ∈ *Temp* avec la meilleure précision (algorithme glouton)
  - RETURN *T (élagué)*

Effets de Reduced-Error Pruning

- Reduction de l'erreur sur l'ensemble de test



  - *NB*: attention, $D_{validation}$ est différent de D_{train} er D_{test}
- Pros and Cons
  - +: Produit le plus petit sous-arbre de T le plus précis
  - -: utilise moins de données pour construire *T*
    - Etant donné le nb des exemples disponibles, peut-on se permettre d'utiliser un $D_{validation}$?

Elagage de règles

- Méthode très fréquemment utilisée
  - Variantes utilisées dans *C4.5*…
  - Indépendant de la manière don't les règles ont été obtenues
- Algorithme *Rule-Post-Pruning* (*D*)
  - Construire un arbre de décision *T* sans élagage
  - Convertir *T* en un ensemble de règles équivalentes (une pour chaque chemin de la racine à une feuille)
  - Elague (généralise) chaque règle indépendemment en effaçant toute les préconditions don't l'effacement améliore l'erreur apparente
  - Trier les règles obtenues par leur précision
  - Les appliquer dans l'ordre à D_{test}

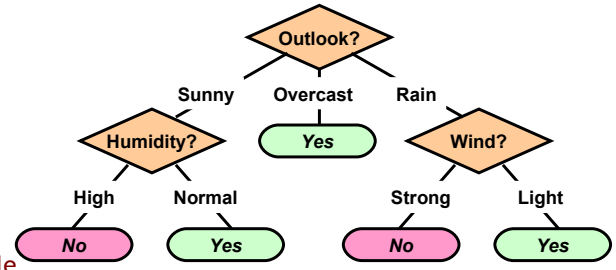
Critères d'évaluation de règles

- ***Spécificité*** $\frac{VP}{VP + FN}$
- ***Sensibilité*** $\frac{VN}{FP + VN}$
- ***Rappel*** $\frac{VP}{VP + FN}$
- ***Précision*** $\frac{VP}{VP + FP}$

Réel / *Estimé*	+	-
+	VP	FP
-	FN	VN

Convertir un arbre de décision en règles

- Syntaxe d'une règle :
  - Prémisses : préconditions (conjonction des tests de la racine à une feuille)
  - Conclusion : étiquette de classe



- Exemple
  - SI (Outlook = Sunny) ∧ (Humidity = High) ALORS PlayTennis = No
  - SI (Outlook = Sunny) ∧ (Humidity = Normal) ALORS PlayTennis = Yes
  - …

Avantage des ensembles de règles vs. Arbres de décision

- Avantages
  - Facile à comprende
  - Souvent de meilleurs résultats que les arbres de décision
  - représentable en logique d'ordre un
  - possibilité d'ajouter de la connaissance du domaine
- Désavantages
  - Pb si la taille de l'ensemble d'apprentissage augmente
  - Pb si bruit dans les données

Incremental Reduced Error Pruning - IREP

- Fürnkranz & Widmer (1994)
- Application itérative de REP
- Comment IREP fonctionne:
  - Apprentissage et élagage „mêlés"
  - Les exemples non couverts sont séparés aléatoirement en „grow" et „test"
  - Stratégie séparer pour régner : construit une règle en appliquant une recherche gloutonne
  - Dès qu'une règle est apprise, elle est immédiatement élaguée
    → pas de sur-apperntissage

L'implémentation d'IREP par Cohen

- Construit des règles tant qu'il existe des positifs non couverts
  - Les exemples non couverts jusqu'à présent sont séparés en growing set(2/3) et pruning set(1/3)
  - Construit une règles à partir du growing set
  - Elague immédiatement la règle:
    - Efface une séquence finale de conditions qui maximise la **fonction *v (s'arrête si plus d'amélioration)*:**

$$v(Rule, prunePos, pruneNeg) \equiv \frac{p+(N-n)}{P+N}$$

$$v*(Rule, prunePos, pruneNeg) \equiv \frac{p-n}{p+n}$$

  - N/P: nbre d'exemples +/- dans le pruning set
  - n/p: nbre d'exemples +/- dans le pruning set couvert par la règles
  - Ajouter la règle élaguée à l'ensemble de règles résultat
  - Efface tous les exemples couverts par la règle (p/n)

Ripper (Cohen 1995)

- 2 types de boucles dans Ripper :
  - Boucle externe : ajouter une règle à la fois à la base de règle courante
  - Boucle interne : ajouter une condition à la fois à la règle courante

Le pseudocode de la boucle externe de Ripper est décrite au transparent suivant

O(Nlog²N)

```
Ripper(Pos,Neg,k)
  RuleSet ← LearnRuleSet(Pos,Neg)
  For k times
    RuleSet ← OptimizeRuleSet(RuleSet,Pos,Neg)
LearnRuleSet(Pos,Neg)
  RuleSet ← ∅
  DL ← DescLen(RuleSet,Pos,Neg)
  Repeat
    Rule ← LearnRule(Pos,Neg)
    Add Rule to RuleSet
    DL' ← DescLen(RuleSet,Pos,Neg)
    If DL'>DL+64
      PruneRuleSet(RuleSet,Pos,Neg)
      Return RuleSet
    If DL'<DL DL ← DL'
      Delete instances covered from Pos and Neg
  Until Pos = ∅
  Return RuleSet
```

DL: description length


```
PruneRuleSet(RuleSet,Pos,Neg)
  For each Rule ∈ RuleSet in reverse order
    DL ← DescLen(RuleSet,Pos,Neg)
    DL' ← DescLen(RuleSet-Rule,Pos,Neg)
    IF DL'<DL Delete Rule from RuleSet
  Return RuleSet
OptimizeRuleSet(RuleSet,Pos,Neg)
  For each Rule ∈ RuleSet
    DL0 ← DescLen(RuleSet,Pos,Neg)
    DL1 ← DescLen(RuleSet-Rule+
     ReplaceRule(RuleSet,Pos,Neg),Pos,Neg)
    DL2 ← DescLen(RuleSet-Rule+
     ReviseRule(RuleSet,Rule,Pos,Neg),Pos,Neg)
    If DL1=min(DL0,DL1,DL2)
      Delete Rule from RuleSet and
        add ReplaceRule(RuleSet,Pos,Neg)
    Else If DL2=min(DL0,DL1,DL2)
      Delete Rule from RuleSet and
        add ReviseRule(RuleSet,Rule,Pos,Neg)
  Return RuleSet
```


Ripper Algorithm

- Dans Ripper, les règles sont construites par ajour de conditions. On recherche la condition qui maximize une mesure de gain d'information

$$Gain(R',R) = s \cdot (\log_2 \frac{N'_+}{N'} - \log_2 \frac{N_+}{N})$$

  - *R* : la règle initiale
  - *R'* : la règle candidate après ajout d'une condition
  - *N* (*N'*):le nombred'exemple couverts par *R* (*R'*)
  - N_+ (N'_+): le nombre de vrais positifs couverts par *R* (*R'*)
  - *s* : le nombre de vrais positifs couverts par *R* et *R'* (after adding the condition)
- L'ajout se fait jusqu'à ce qu'il n'y ait plus d'exemples négatifs couverts ou critère heuristique MDL (Minimal Description Length)
- RIPPER: Performance comparable à C4.5 rules, tout en étant bien plus efficace, en particulier sur des données bruitées

References

William W. Cohen, Fast Effective Rule Induction, 1995
William W. Cohen , Efficient Pruning Methods,1993
J. Fürnkranz & G. Widmer, Incremental Reduced Error Pruning, 1994
T. Mitchell, Machine Learning, sections 3.6-3.8