Bias, Variância & *Ensembles*

- Em aulas anteriores vimos o conceito de *bias* de AM, o qual se constitui em certas suposições e escolhas efetuadas pelos indutores na busca de uma hipótese
- Nesta aula veremos que o *bias* e a variância podem ser vistos como componentes do erro de um classificador
- Assim, métodos que reduzam o *bias* ou a variância, ou ambos, são importantes em AM
- Um desses métodos, *ensembles*, consiste em induzir vários classificadores e então combiná-los em um único classificador final, através de um mecanismo de votação
- Em geral, *ensembles* mostraram ser eficazes quando aplicados a indutores que são instáveis, tais como indução de regras, árvores de decisão ou redes neurais

USP
José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
E-mail: augusto@usp.br
URL: http://dfm.ffclrp.usp.br/~augusto

Bias de AM

- Suposições adotadas pelos algoritmos de forma a generalizar os dados de treinamento são denominadas biases de Aprendizado de Máquina
- Dois tipos de *Biases* de AM
  - absoluto
  - relativo

Bias de AM

- *Bias* absoluto
  - o indutor assume que a função a ser aprendida é um elemento de um determinado conjunto de funções (espaço restrito de hipóteses)
  - Ex: o algoritmo *perceptron* restringe a busca somente no espaço de funções lineares enquanto árvores de decisão traçam hiperplanos paralelos aos eixos
- *Bias* relativo
  - o indutor tem uma ordem de preferência nas hipóteses, assumindo que a função a ser aprendida é mais similar a um conjunto de funções do que de outro
  - Ex: os algoritmos de indução de árvores de decisão consideram árvores menores antes de considerar as árvores maiores. Se esses algoritmos encontram uma árvore pequena que classifica corretamente os dados de treinamento então uma árvore maior não é considerada

Bias e Variância Estatísticos

- *Bias* estatístico
  - O bias estatístico captura a idéia de erro sistemático para uma dada amostra
  - Ex: se função a ser aprendida é uma onda senoidal f(x) = sin(x) e o algoritmo de aprendizado constrói funções lineares h(x)=ax+b, então haverá erros sistemáticos, a cada subida e descida da onda senoidal
- Variância
  - A variância captura as variações aleatórias no algoritmo, de uma amostra para outra
  - A variação pode ser devida à variação do conjunto de treinamento, de ruído aleatório nos dados ou pelo comportamento aleatório do próprio algoritmo de aprendizado, tais como os valores iniciais aleatórios que são freqüentemente utilizados em redes neurais *backpropagation*

Relação entre Biases de AM e Bias e Variância Estatísticos

Bias Estatístico	Variância Estatística	Biases de Aprendizado Absoluto	Relativo
grande	pequena	apropriado	muito forte
pequeno	pequena	apropriado	bom
pequeno	grande	apropriado	muito fraco
grande	pequena	inapropriado	muito forte
grande	média	inapropriado	bom
grande	grande	inapropriado	muito fraco

Princípio Fundamental da Decomposição

- O erro de um classificador h pode ser decomposto:
  - erro mínimo, que pode ser obtido pelo classificador ideal: o limite inferior no erro esperado de qualquer indutor;
  - *bias*, que mede quão aproximadamente o indutor erra em média;
  - variância, que mede quanto os erros do indutor vão variar
- erro(h) = erro(f) + bias(h) + variância(h)
  - Em classificação, f = B* (classificador de Bayes)

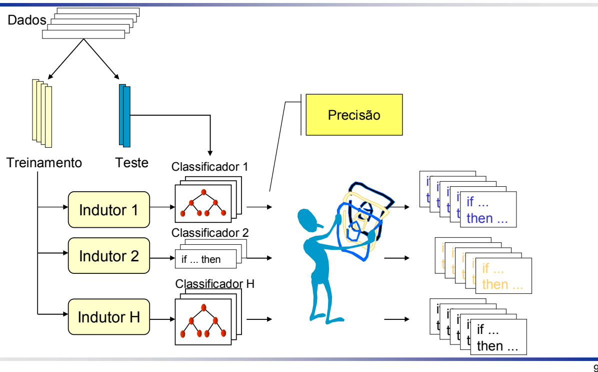
Princípio Fundamental da Decomposição

- Em geral, um indutor constrói divisões no espaço da descrição, que podem ser consideradas como uma família de funções H
- Cada indutor tenta selecionar o melhor classificador h, usando o conjunto de treinamento e o conjunto das funções H
- Por exemplo, se a família de funções H, que podem ser geradas por um algoritmo de aprendizado, é um pequeno conjunto de funções lineares e a função objetivo f é não linear, então o *bias* de h será grande
- Por outro lado, uma vez que um número pequeno de parâmetros são estimados a partir do pequeno conjunto H, a variância de h será pequena
- Mas se H é uma grande família de funções, como as funções representadas por árvores de decisão ou redes neurais, então o *bias* é usualmente pequeno (uma vez que é quase sempre possível aproximar a função f por alguma h pertencente à H), mas a variância pode ser grande (já que muitos parâmetros podem ser ajustados)

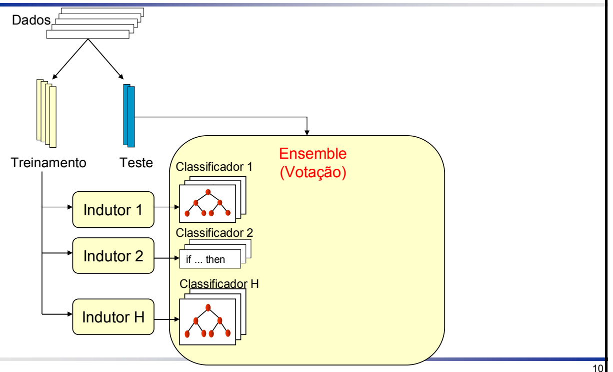
Redução do Erro

- Reduzir *bias* e variância estatísticos
- Combinação de classificadores: *ensembles*
- Um *ensemble* h* consiste num conjunto de classificadores $\{h_1, h_2, ..., h_L\}$ cujas previsões são combinadas para prever a classe de um novo exemplo
- Normalmente, a classe prevista com maior freqüência por $\{h_1, h_2, ..., h_L\}$ é a classe prevista pelo classificador final h* (voto majoritário)
- Em geral, um *ensemble* é mais preciso do que qualquer um dos classificadores que o compõe

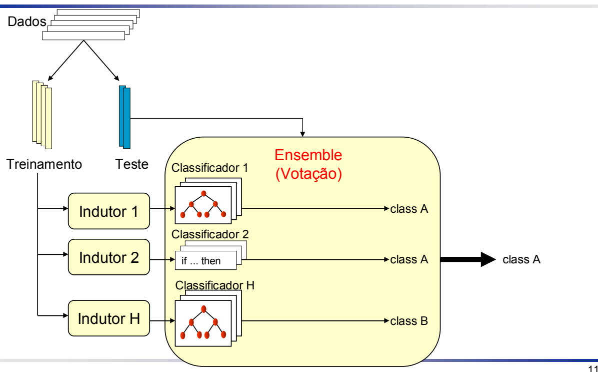
Ensembles



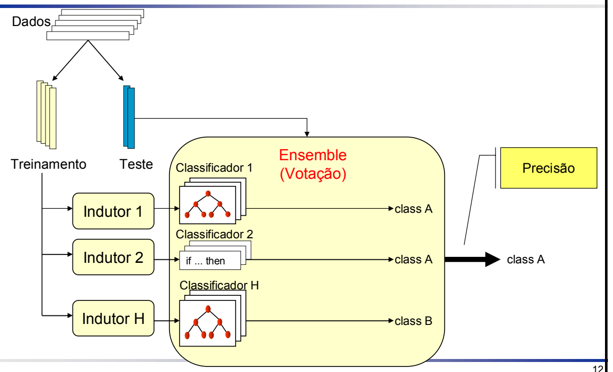
Ensembles



Ensembles



Ensembles



Ensembles

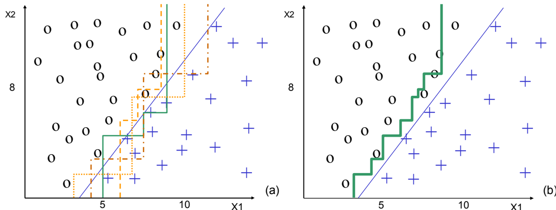
- Cada classificador no *ensemble* deve discordar de um outro
- Ex: Suponha *ensemble* $\{h_1, h_2, h_3\}$ e um novo exemplo x a ser classificado
  - Se os três classificadores são idênticos, então quando $h_1(x)$ erra, $h_2(x)$ e $h_3(x)$ também erram
  - Todavia, se os erros dos três classificadores não estão correlacionados, então quando $h_1(x)$ está errado, $h_2(x)$ e $h_3(x)$ podem estar certos
  - Se isso ocorrer, através do voto majoritário, o novo exemplo x será corretamente classificado pelo *ensemble*

Necessidade de Ensembles

- O espaço de hipóteses H pode não conter a função objetivo f
- Ao invés disso, H pode incluir várias aproximações igualmente boas para f
- Pela combinação dessas aproximações é possível representar classificadores que estejam fora de H
- Um fato importante é que *ensembles* são geralmente mais precisos que os classificadores individuais que o compõe

Necessidade de Ensembles

- A região real de separação de classes, indicada pela linha diagonal, é aproximada de formas diferentes por quatro árvores de decisão em (a)
- A combinação das quatro árvores, num ensemble, fornece uma região que se aproxima mais da diagonal em (b)



Métodos para Criar Ensembles

- Stack
- Window
- Bagg
- Wagg
- Boost
- Arc

Stack

- *Stacking* utiliza um meta-indutor no lugar do esquema de votação
- Cada exemplo de treinamento é classificado por um indutor nível 0, sendo, preferencialmente, cada indutor nível 0 diferente dos demais
- A entrada para o meta-indutor (indutor de nível 1) são as classificações preditas por cada um dos classificadores nível 0
- Assim, o número de atributos do indutor nível 1 é igual ao número de indutores nível 0, cada atributo significando um indutor nível 0

Window

- Seleciona um subconjunto de exemplos (*window*) do conjunto de treinamento e gera uma hipótese
- Essa hipótese é usada para classificar os exemplos remanescentes no conjunto de treinamento
- Se existir exemplos não classificados, eles são incluídos na janela original e uma segunda hipótese é gerada
- O processo se repete até que a hipótese correspondente à janela atual classifique todos os exemplos fora da janela

Algoritmo Windowing

Algorithm 1 Windowing

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm
W: the initial window size, $W := \max\{0.2n, 2\sqrt{n}\}$ in (Quinlan, 1988)
I: the increment to window, $I := \max\{0.2W, 1\}$ in (Quinlan, 1988)

1: **procedure** window(n,Instances,Inducer,L,W,I)
2: window_l := sample(W,Instances)
3: $L := (n - W)/I$ *// number of windows*
4: **for** $l := 1$ **to** L **do**
5: h_l := Inducer(window_l)
6: $\text{error}(h_l) := \sum_{x_i \notin \text{window}_l} \| h_l(x_i) = y_i \|$ *// error on instances outside the window*
7: **if** $\text{error}(h_l) = 0$ **then**
8: **exit loop** *// all instances correctly classified*
9: **end if**
10: $\text{window}_{l+1} := \text{window}_l$ + *include almost I misclassified instances by h_l from* Instances
11: **end for**
12: $h^*(x) := h_l(x)$
13: **return** h^*

Bagg (Bootstrap Aggregation)

- Gera uma amostra *bootstrap* S_1 e a hipótese correspondente h_1
- Repete o processo L vezes, gerando amostras S_2, S_3, ..., S_L e induzindo hipóteses h_2, h_3, ..., h_L, respectivamente
- Todos os classificadores são combinados em um único classificador h*, utilizando voto majoritário
- *Bagging* é o mais eficaz nos algoritmos instáveis, onde mudanças pequenas no conjunto de treinamento produzem grandes mudanças nos classificadores gerados

Algoritmo Bagging

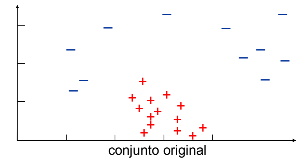
Algorithm 2 Bagging

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm
L: the number of bagging classifiers

1: **procedure** bagg(n,Instances,Inducer,L)
2: **for** $i := 1$ **to** n **do**
3: $p_i := 1/n$ *// initialize normalized weights*
4: **end for**
5: **for** $l := 1$ **to** L **do**
6: S_l := bootstrap_sample(n,p,Instances)
7: h_l := Inducer(S_l)
8: **end for**
9: $h^*(x) := \arg\max_{y \in \{C_1, C_2, \ldots, C_k\}} \sum_{l=1}^{L} \| h_l(x) = y \|$
10: **return** h^*

Bagg

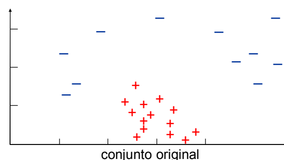
- Obtenha uma amostra *bootstrap*
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes



Bagg

- Obtenha uma amostra *bootstrap*
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes

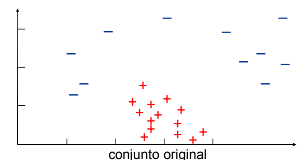
conjunto original



Bagg

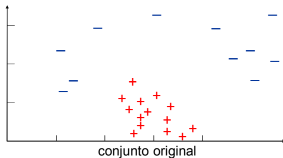
- Obtenha uma amostra *bootstrap*
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes
- Obtenha um classificador h_l

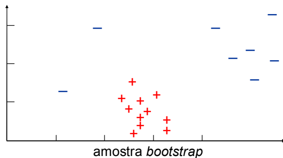
conjunto original



Bagg

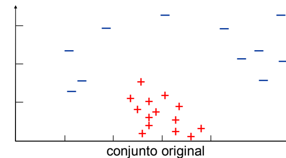
- **Obtenha uma amostra *bootstrap***
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes
- Obtenha um classificador h_l
- (Repetir)

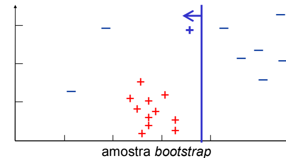




Bagg

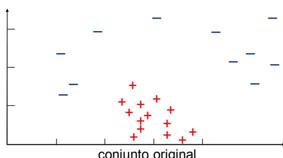
- Obtenha uma amostra *bootstrap*
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes
- **Obtenha um classificador h_l**
- (Repetir)

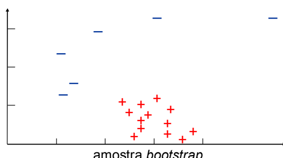




Bagg

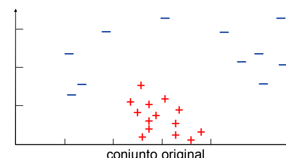
- **Obtenha uma amostra *bootstrap***
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes
- Obtenha um classificador h_l
- (Repetir)

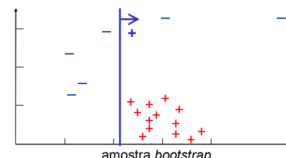




Bagg

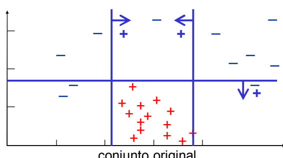
- Obtenha uma amostra *bootstrap*
  - Em uma amostra boostrap, alguns dos exemplos originais aparecem 0, 1 ou repetidas vezes
- **Obtenha um classificador h_l**





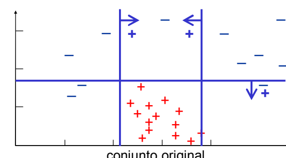
Bagg

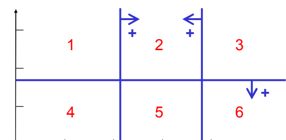
- Ao final, os classificadores são combinados pelo voto majoritário entre eles



Bagg

- Ao final, os classificadores são combinados pelo voto majoritário entre eles
- Novos exemplos que se encontrem nas regiões numeradas são classificados como:
  - Região 1 [1+,2-]: -
  - Região 2 [2+,1-]: +
  - Região 3 [1+,2-]: -
  - Região 4 [2+,1-]: +
  - Região 5 [3+,0-]: +
  - Região 6 [2+,1-]: +





Wagg (Weight Aggregation)

- *Wagging* é similar a *bagging* mas ele altera os pesos dos exemplos do conjunto de treinamento, ao invés de amostrar
- O método repetidamente perturba o conjunto de treinamento como *bagging* o faz, mas ao invés de tirar amostras, *wagging* adiciona ruído gaussiano à cada peso com média zero e desvio padrão fixado normalmente em dois
- Em cada iteração, *wagging* inicia com todos os pesos iguais e então ruído é adicionado aos pesos e uma hipótese é induzida

Boost

- *Boost* foi introduzido por Shapire (1990) como um método de melhorar o desempenho de algoritmos
- Após algumas melhorias, o algoritmo AdaBoost (Adaptative Boosting) foi introduzido em 1995, também denominado AdaBoost.M1
- O algoritmo de boosting gera classificadores sequencialmente enquanto *bagging* pode gerá-los em paralelo
- A cada exemplo de treinamento é associado um peso
- A cada iteração de *boosting*, um classificador é extraído a partir dos exemplos ponderados e cada exemplo é reponderado caso ele tenha sido classificado incorretamente

Boost

- Antes de induzir o primeiro classificador, todos os exemplos são equiprováveis, ou seja, cada exemplo tem um mesmo peso associado
- Após induzir o primeiro classificador, *boosting* altera os pesos dos exemplos de treinamento, repetindo esse ciclo L vezes
- O classificador final é formado utilizando voto ponderado no qual o peso de cada classificador depende de seu desempenho no conjunto de treinamento

Boost

- O algoritmo AdaBoost requer um indutor cujo erro seja limitado por uma constante estritamente menor que 1/2
- Algumas implementações utilizam *resampling* uma vez que alguns indutores não são capazes de suportar exemplos ponderados

Boost

Algorithm 3 Boosting (AdaBoost.M1)

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm accepting instances weighting
L: the number of boosting classifiers

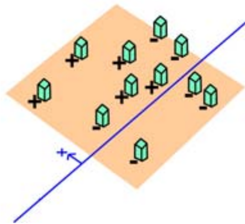
1: **procedure** boost(n,Instances,Inducer,L)
2: **for all** $i := 1$ **to** n **do**
3: $w_i := 1/n$ *// initialize the weights*
4: **end for**
5: **for** $l := 1$ **to** L **do**
6: **for** $i := 1$ **to** n **do**
7: $p_i := w_i/(\sum_{j=1}^{n} w_j)$ *// normalize the weights*
8: **end for**
9: $h_l :=$ Inducer(Instances,p)
10: $\text{error}(h_l) := \sum_{i=1}^{n} p_i \parallel h_l(x_i) \neq y_i \parallel$ *// compute the hypothesis error*
11: **if** $\text{error}(h_l) > 1/2$ **then**
12: $L := l - 1$
13: **exit loop**
14: **end if**
15: $\beta_l := \text{error}(h_l)/(1-\text{error}(h_l))$
16: **for** $i := 1$ **to** n **do**
17: **if** $h_l(x_i) = y_i$ **then**
18: $w_i := w_i \beta_l$ *// compute new weights*
19: **end if**
20: **end for**
21: **end for**
22: $h^*(x) := \arg\max_{y \in \{C_1, C_2, \ldots, C_k\}} \sum_{l=1}^{L} \log(1/\beta_l) \parallel h_l(x) = y \parallel$
23: **return** h^*

Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)

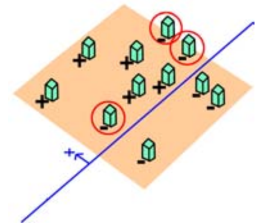
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l



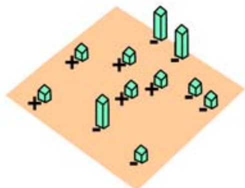
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l
- Aumente os pesos do exemplos de treinamento classificados incorretamente



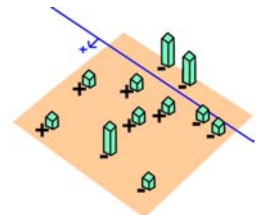
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l
- Aumente os pesos do exemplos de treinamento classificados incorretamente
- (Repetir)



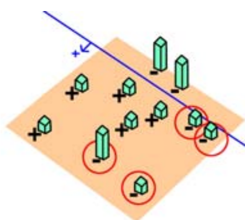
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- **Obtenha um classificador h_l**
- Aumente os pesos do exemplos de treinamento classificados incorretamente
- (Repetir)



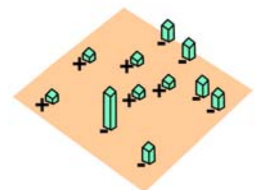
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l
- **Aumente os pesos do exemplos de treinamento classificados incorretamente**
- (Repetir)



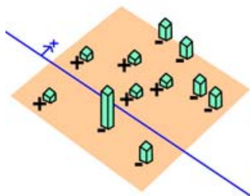
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l
- Aumente os pesos do exemplos de treinamento classificados incorretamente
- **(Repetir)**



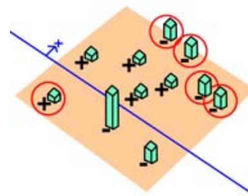
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- **Obtenha um classificador h_l**
- Aumente os pesos do exemplos de treinamento classificados incorretamente
- (Repetir)



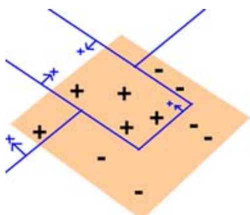
Boost

- Inicie com uma distribuição uniforme de pesos nos exemplos de treinamento
  - (Os pesos indicam quais exemplos são importantes)
- Obtenha um classificador h_l
- **Aumente os pesos do exemplos de treinamento classificados incorretamente**



Boost

- Ao final, faça (cuidadosamente) uma combinação ponderada dos classificadores obtidos em todas as iterações



Arc (Adaptively Resample and Combine)

- Arcing é um termo definido por Breiman (1996a) para descrever a família de algoritmos que amostram e combinam de forma adaptativa
- O algoritmo arc-x4 demonstrou-se tão preciso quanto AdaBoost, sem utilizar a ponderação para compor o classificador final
- Assim, o poder do algoritmo AdaBoost é derivado da reponderação dos exemplos e não da combinação final

Arc (Adaptively Resample and Combine)

Algorithm 5 Arcing (arc-x4)

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm accepting instances weighting
L: the number of arcing classifiers

1: **procedure** bagg(n,Instances,Inducer,L)
2: **for all** $i := 1$ **to** n **do**
3: $p_i := 1/n$ *// initialize the weights*
4: $m_i := 0$ *// number of misclassification of instance i by classifiers $1, 2, \ldots, l$*
5: **end for**
6: **for** $l := 1$ **to** L **do**
7: $S_l :=$ bootstrap_sample(n,p,Instances)
8: $h_l :=$ Inducer(S_l)
9: **for** $i := 1$ **to** n **do**
10: $m_i := m_i + \| h_l(x_i) \neq y_i \|$ *// update misclassifications*
11: **end for**
12: **for** $i := 1$ **to** n **do**
13: $p_i := (1 + m_i^4)/(\sum_{j=1}^{n} (1 + m_j^4))$ *// compute new normalized weights*
14: **end for**
15: **end for**
16: $h^*(x) := \arg\max_{y \in \{C_1, C_2, \ldots, C_k\}} \sum_{l=1}^{L} \| h_l(x) = y \|$
17: **return** h^*

Ensembles: Problemas

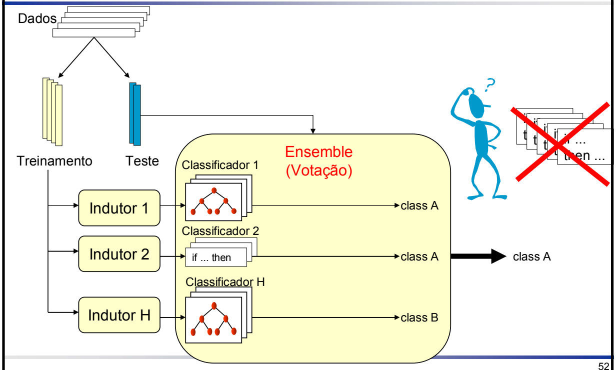
- O classificador final h* é muito grande
- O classificador final h* tende a ser redundante
- O classificador final h* é difícil de ser interpretado pelo cérebro humano
  - Se cada classificador que compõe é simbólico (árvore ou regras) não é possível definir qual regra foi aplicada para encontrar a classe do novo exemplo (foi efetuada uma votação)

Ensembles: Problemas

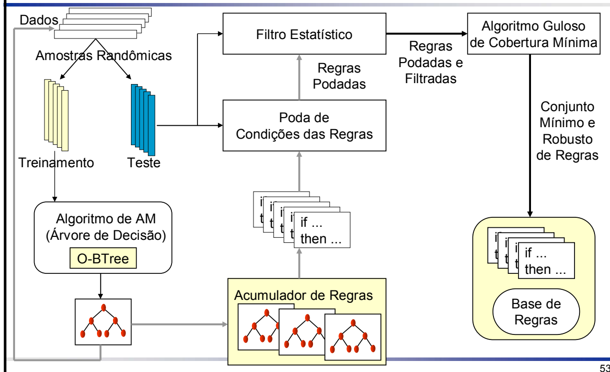
- "Algum método é necessário para converter uma combinação de árvores (ou outras hipóteses mais complexas) numa hipótese menor equivalente. Estas árvores são muito redundantes; como podemos remover essa redundância enquanto ainda reduzindo bias e variância?"

(Dietterich, 1997)

Ensembles: Problemas



Sistema Ruler (Fayyad, 96)



Sistema Xruler