Ensambles de Clasificadores

Eduardo Morales, Hugo Jair Escalante

INAOE

Contenido

Motivación

- Cuando las personas tienen que tomar decisiones difíciles, normalmente toman en cuenta la opinión de varios expertos, buscando mejorar sus decisiones
- En aprendizaje, cada modelo inducido se puede ver como un "experto", por lo que uno puede pensar en que se puedan producir predicciones más confiables si se combinan las predicciones de varios modelos
- Recientemente se han desarrollado técnicas para construir conjuntos de clasificadores cuyas decisiones son combinadas, de alguna forma, para clasificar nuevos ejemplos

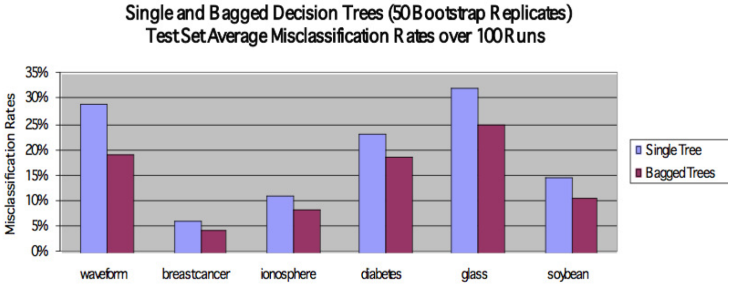
Motivación

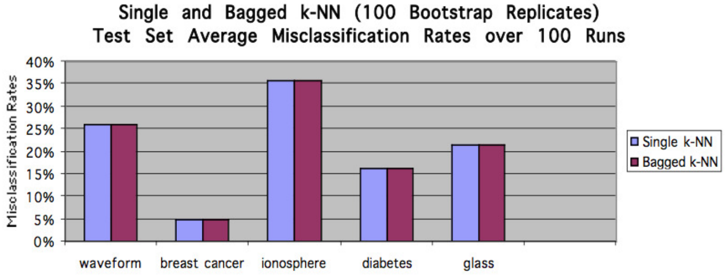
- No existe un clasificador dominante para todas las distribuciones de datos (*no free lunch theorem*) y de entrada no sabemos la distribución de la tarea
- Normalmente se prueban varios clasificadores y varias de sus variantes sobre una muestra "representativa" de los datos y se selecciona el mejor (de menor *error aparente*)
- El problema es que el error aparente puede ser diferente al error real (principalmente con pocos datos)

Motivación

- Lo que se ha encontrado es que, en general, los ensambles son mejores clasificadores que los clasificadores individuales que se usaron en su construcción
- Las condiciones necesarias y suficientes para esto son, que los clasificadores individuales:
  1. Tengan un *buen desempeño* $\approx$ errores menores al 50% (clasificación binaria) y
  2. Que sean *diversos* $\approx$ cometen diferentes errores (son independientes)

Motivación


Single and Bagged Decision Trees (50 Bootstrap Replicates)
Test Set Average Misclassification Rates over 100 Runs
Misclassification Rates
35%
30%
25%
20%
15%
10%
5%
0%
waveform
breastcancer
ionosphere
diabetes
glass
soybean
Single Tree
Bagged Trees


Single and Bagged k-NN (100 Bootstrap Replicates)
Test Set Average Misclassification Rates over 100 Runs
Misclassification Rates
40%
35%
30%
25%
20%
15%
10%
5%
0%
waveform
breast cancer
ionosphere
diabetes
glass
Single k-NN
Bagged k-NN

Motivación

- Si los errores de las hipótesis son menores al 50% y si son independientes, la probabilidad de que la clase mayoritaria cometa un error es equivalente al área de una distribución binomial donde más de la mitad de las hipótesis están mal
- Por ejemplo, para 21 hipótesis, cada una con 0.3 de probabilidad de error, el área de que 11 o más hipótesis estén al mismo tiempo mal es 0.026 (ver figura 1).
- Básicamente lo que se necesita entonces es construir clasificadores individuales con errores menores al 50% y que de alguna forma no estén correlacionados

Ensambles

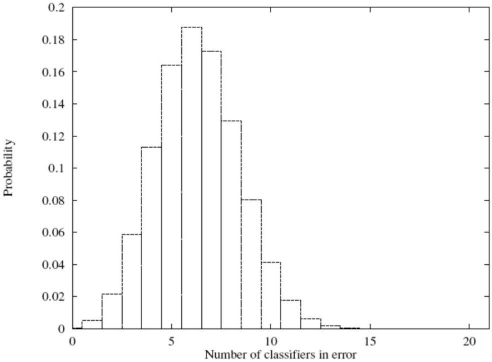


Figure: Distribución binomial de que 21 hipótesis cometan un error suponiendo que todas cometen errores en un 30% y que estos son independientes entre si.

¿Porqué funcionan los *ensambles* de clasificadores?

- Los ensambles permiten que errores no correlacionados de clasificadores individuales puedan eliminarse por votación mayoritaria
- Podemos, también, mencionar al menos otras tres razones relacionadas:
    1. *Estadística*
    2. *Computacional*
    3. *Representacional*

¿Porqué funcionan los *ensambles* de clasificadores?

1. *Estadística*: Los ejemplos de entrenamiento pueden no proporcionar información suficiente para seleccionar al mejor clasificador (y existen muchas posibles hipótesis de donde escoger).
   Con datos menores al número posible de hipótesis, el algoritmo puede encontrar muchas hipótesis equivalentes.
   Al "promediar" sus resultados se reduce el riesgo de seleccionar un mal clasificador.

¿Porqué funcionan los *ensambles* de clasificadores?

2. *Computacional*: Muchos algoritmos realizan un tipo de búsqueda local y pueden quedar atrapados en un mínimo local (e.g., gradiente descendente de NN o *hill climbing* de Árboles de decisión)
   Encontrar el árbol o la red neuronal más pequeños son problemas NP duros. Un ensamble puede dar una mejor aproximación

¿Porqué funcionan los *ensambles* de clasificadores?

3. *Representacional*: Nuestro espacio de búsqueda puede no contener a la función objetivo. Sin embargo, al formar un espacio pesado de hipótesis es posible expandir el espacio de representación de nuestras funciones.
   Por ejemplo, en las figuras 2 y 3 podemos ver que un clasificador de varios pedazos de clasificadores lineales es más expresivo que un solo clasificador lineal (figura 2) y como un conjunto de árboles de decisión se aproximan mejor a una diagonal (figura 3).

¿Porqué funcionan los *ensambles* de clasificadores?

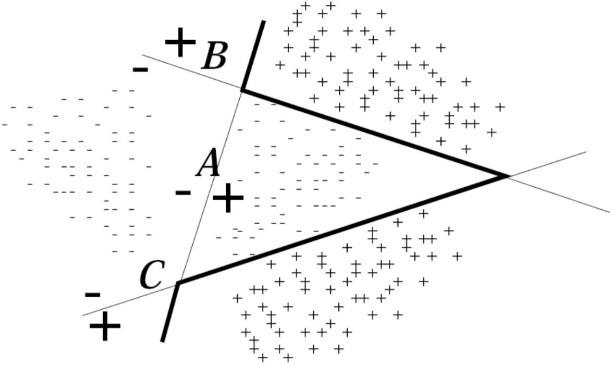


Figure: Un esamble de clasificadores lineales. La línea fuerte muestra la clasificación obtenida por voto mayoritario.

¿Porqué funcionan los *ensambles* de clasificadores?

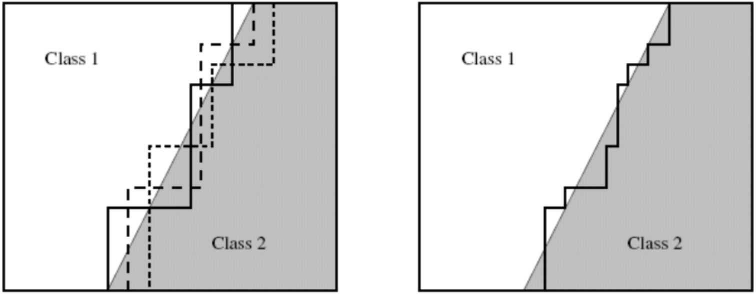
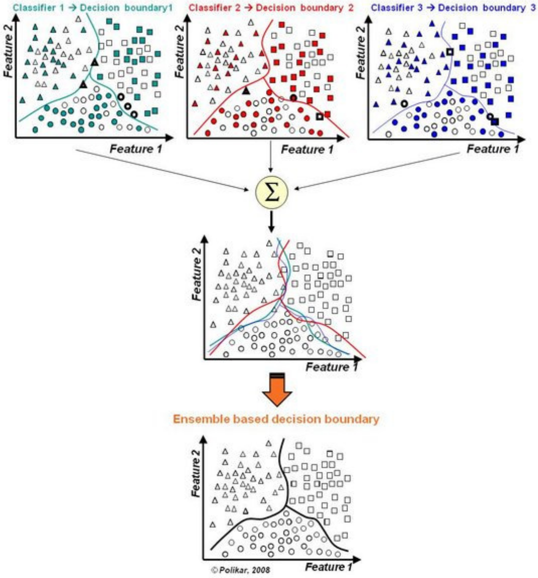
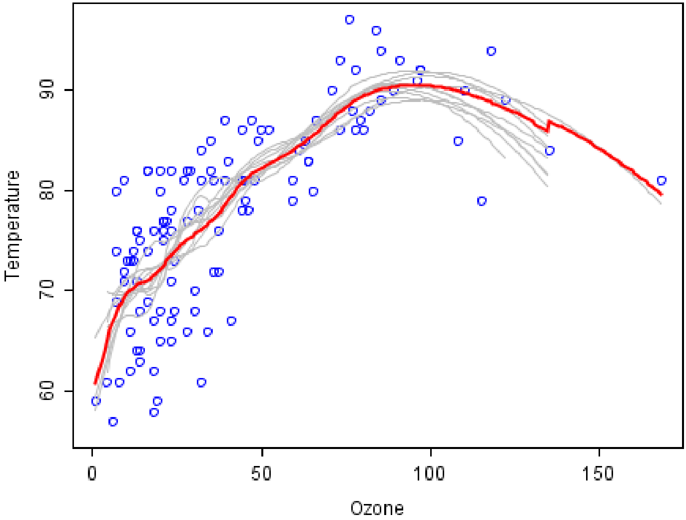


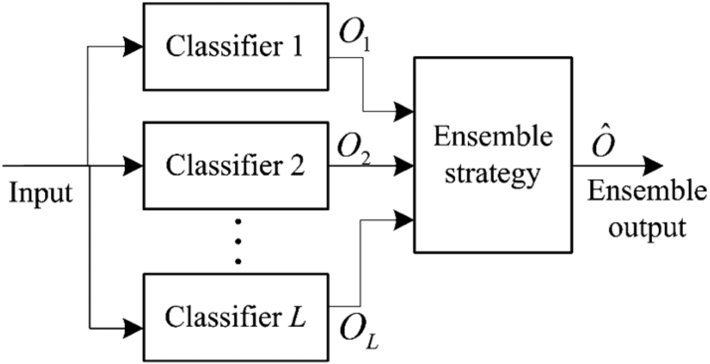
Figure: Tres árboles de decisión aproximando individualmente una línea diagonal (izquierda) y su combinación con voto simple (derecha).


Classifier 1 → Decision boundary1
Classifier 2 → Decision boundary 2
Classifier 3 → Decision boundary 3
Feature 2
Feature 1
Σ
Ensemble based decision boundary
©Polikar, 2008



Temperature
60
70
80
90
0
50
100
150
Ozone

Ensambles de Clasificadores



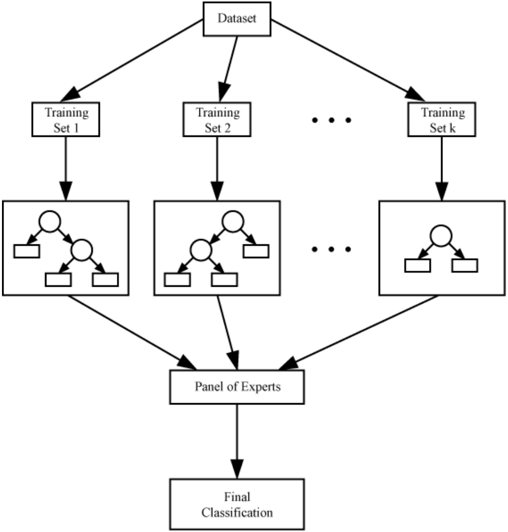
Ensambles de Clasificadores

- La idea básica es correr algoritmos de inducción varias veces y combinar los resultados de alguna forma para obtener un mejor resultado final
- La combinación de varios modelos se puede hacer de diferentes formas, la más simple es usar voto mayoritario (e.g., *bagging*), realizar un voto pesado (e.g., *boosting*) o usar un nuevo clasificador que decida cómo combinar esos resultados (e.g., *stacking*)
- La desventaja que presentan es que los resultados son difíciles de analizar

Algoritmos

- Existe una gran cantidad de algorítmos, la mayoría son variantes de:
  - Bagging
  - Boosting
  - Stacking

Bagging


Dataset
Training Set 1
Training Set 2
Training Set k
Panel of Experts
Final Classification

Bagging

- Bagging (*B*ootstrap *Agg*regat*ing*) genera clasificadores de varias muestras de los ejemplos
- Funciona especialmente para algoritmos de aprendizaje inestables (cambian mucho sus estructuras al cambiar un poco los ejemplos), por ejemplo, los árboles de decisión
- El error de combinar varios clasificadores se explica por lo que se conoce como *bias-variance decomposition*. El sesgo (*bias*) de cada clasificador está dado por su error intrínseco y mide qué tan bien un clasificador explica el problema. La varianza está dada por los datos que se usan para construir el modelo.

Bagging

- El error esperado total de clasificación está dado por la suma del sesgo y la varianza. Al combinar múltiples clasificadores se reduce el error esperado al reducir la varianza
- En la práctica un problema es que existe sólo un conjunto de entrenamiento y puede no ser fácil obtener más
- Lo que se hace es que se generan muestras de ejemplos *bootstrap*, muestreando uniformemente m instancias del conjunto de entrenamiento con reemplazo

Algoritmo de Bagging

- Se generan T muestras, $B_1, \ldots, B_T$ y se construye un clasificador C_i para cada muestra
- Se construye un clasificador final C^* de todos los C_1 a C_T cuya salida es la clase mayoritaria de los clasificadores
- Para cada muestra, un ejemplo tiene una probabilidad de $1 - (1 - 1/m)^m$ de ser seleccionado por lo menos una vez en las m veces que se selecciona una instancia
- Para valores grandes de m esto se aproxima a $1 - 1/e = 63.2\%$, por lo que cada muestra tiene aproximadamente un 63% de aparecer en los ejemplos de entrenamiento

Algoritmo de Bagging

for $i = 1$ to T **do**

  S' = muestra de S (i.i.d. con reemplazo)

  $C_i = Ind(S')$ $\{$construye clasificador$\}$

  $C^* = argmax_{y \in Y} \sum_{i:C_i(x)=y} 1$

end for

i.i.d = Independiente e identicamente distribuido

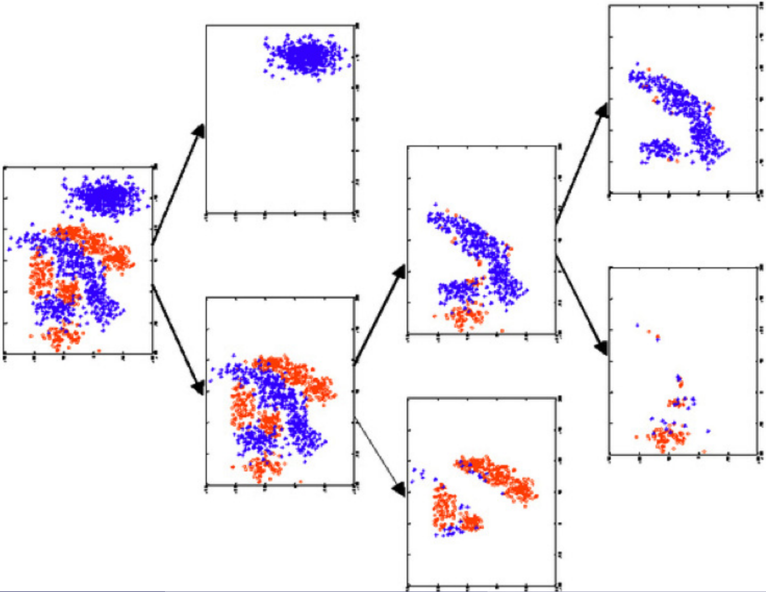
Bagging

- Bagging también se puede usar para clases con valores continuos, la diferencia es que la salida de los clasificadores se promedia. En este caso se reduce el valor esperado del error cuadrático medio.
- En general, se puede mejorar el resultado si se quita la opción de podado en los árboles de decisión, lo que los hace más inestables.
- Si los modelos nos regresan probabilidades, hace sentido combinar estas probabilidades, y lo que nos regresa *Bagging* es una probabilidad.

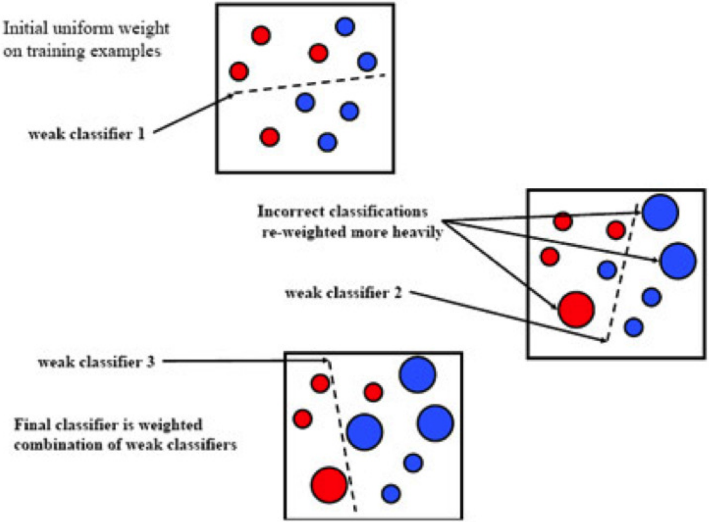
Bagging

- Los resultados de *bagging*, como ya lo mencionamos, son difíciles de interpretar. Una posibilidad es usar **MetaCost**
- La idea es construir un ensamble de clasificadores usando *Bagging*, usar la clasificación producida por el ensamble para re-etiquetar todos los datos y después aprender un clasificar único con esa re-etiquetación.
- El re-etiquetado de cada ejemplo se hace tratando de minimizar un costo esperado obtenido de las probabilidades arrojadas por *bagging*. Al final se obtiene un nuevo clasificador único que toma en cuentan los costos que ya se pusieron en las etiquetas.

Boosting



Boosting



$$H(x) = sign(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x))$$

Boosting

- Boosting y su variante más usada AdaBoost (*Ada*pting *Boost*ing) genera igual un conjunto de clasificadores, sin embargo, Adaboost los genera secuencialmente (*Bagging* los puede generar en paralelo)
- A todos los ejemplos, les asigna inicialmente un peso igual, pero cada vez que se genera un clasificador, se cambian los pesos de los nuevos ejemplos usados para el siguiente clasificador
- La idea es forzar al nuevo clasificador a minimizar el error esperado, para esto se les asigna más peso a los ejemplos mal clasificados y menos a los bien clasificados

Boosting

- Con esto, se busca crear modelos que se vuelvan “expertos” en los datos que no pudieron ser explicados por los modelos anteriores
- Después de cada interacción los pesos reflejan qué tan seguido las instancias han sido mal clasificadas por los clasificadores que se tienen hasta ese momento
- Se generan igual T clasificadores de muestras de ejemplos pesadas y el clasificador final se forma usando un esquema de votación pesado que depende del desempeño de cada clasificador en su conjunto de entrenamiento

Algoritmo de Boosting

$S' = S$ con pesos igual a 1
for $i = 1$ to T **do**
  $C_i = Ind(S')$ y $\epsilon_i = \frac{1}{m} \sum_{x_j \in S' : C_i(x_j) \neq y_j} peso(x)$
  if $\epsilon_i > 1/2$ {limitado a N veces} **then**
    cambia todos los pesos a 1 y regresa al principio
  end if
end for
$\beta_i = \epsilon_i / (1 - \epsilon_i)$
for cada $x_j \in S'$ **do**
  if $C_i(x_j) = y_j$ **then**
    $peso(x_j) = peso(x_j) \cdot \beta_i$
  end if
  Normaliza los pesos para que el peso total de S' sea m
end for
$C^*(x) = argmax_{y \in Y} \sum_{i : C_i(x) = y} log \frac{1}{\beta_i}$

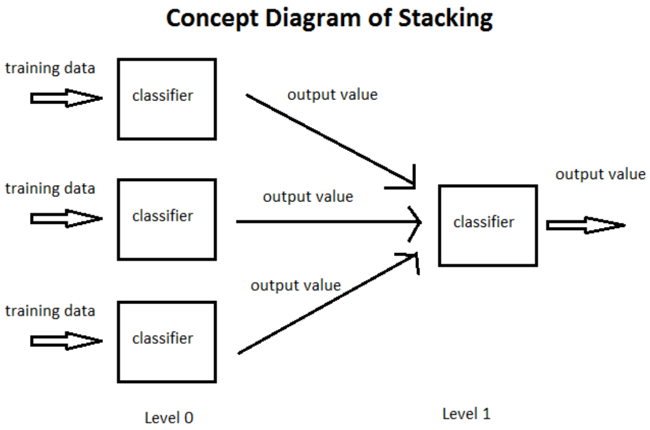
Boosting

- Este algoritmo requiere de clasificadores *débiles* que cambian su estructura con cambios en los datos y que no den errores mayores al 50%.
- El algoritmo se para cuando el error en los datos de entrenamiento pesados son mayores o iguales a 0.5 o cuando el error es cero (donde todos los pesos de las instancias se vuelven 0).
- Si no se pueden incorporar ejemplos pesados dentro del clasificador, se puede tener un efecto parecido por medio de un muestreo con reemplazo, seleccionando los ejemplos de acuerdo a su peso.

Algunos detalles

- Se puede tener problemas de *underflow*, por lo que es común eliminar ejemplos con pesos muy pequeños.
- En general, *AdaBoost* es mejor que *Bagging*, pero no es siempre uniformemente mejor que cada clasificador, mientras que *Bagging* si.
- *Bagging* sin *pruning* a veces reduce el error, *Boosting* sólo lo aumenta.
- En *Boosting*, si un clasificador tiene error cero, recibe recompensa infinita y es el unico ganador, por lo que generalmente se elimina.

Stacking


Concept Diagram of Stacking
training data
classifier
output value
training data
classifier
output value
training data
classifier
output value
classifier
output value
Level 0
Level 1

Stacking

- *Stacking* construye un conjunto de modelos usando diferentes algoritmos de aprendizaje.
- Una forma de combinar los clasificadores es usando voto mayoritario, sin embargo, esto hace sentido cuando los clasificadores se desempeñan en forma parecida.
- Para producir una clasificación utiliza un meta-algoritmo (*meta learner*) que aprende de acuerdo a las salidas de los clasificadores base (en lugar de voto mayoritario).
- En resumen, se construyen N clasificadores a partir de los datos usando algoritmos diferentes. Las salidas de los clasificadores se usan como atributos (por lo que se tienen tantos atributos para el meta-clasificador como clasificadores) de un nuevo clasificador.

Stacking

- Se puede estimar el desempeño de cada clasificador usando un conjunto de ejemplos de prueba y haciendo validación cruzada ($n-$*fold cross validation* o *leave-one-out cross validation*)
- Otra posibilidad es utilizar el valor de la probabilidad más alta de cada clasificador.
- Como meta-clasificador normalmente se utiliza algo simple.

Ensambles de Clasificadores

Las técnicas desarrolladas para ensambles de clasificadores las podemos dividir por:

1. Su arquitectura o topología
2. El ensamble de clasificadores; el tipo y número de clasificadores base
3. La forma de combinar los resultados

Arquitectura de Ensambles

1. Paralela: Múltiples clasificadores operan en paralelo y se usa una función de combinación para la salida de cada clasificador (e.g., *Bagging*)
2. Serial o Condicional: Los clasificadores se aplican en sucesión y cada clasificador produce un conjunto reducido de posibles clases (e.g., *Boosting*)
3. Híbridos: Por ejemplo, varios clasificadores en serie

Tipo de Clasificadores

Los ensambles de clasificadores se pueden dividir por los que:

1. Usan el mismo clasificador (la mayoría)
2. Combinan diferentes clasificadores (*Stacking*)

En general se prefieren clasificadores que den resultados probabilístas

Regla de Combinación

1. Función de integración (fusión): Todos los clasificadores contribuyen a la decisión final (clasificadores competitivos)
2. Función de selección: Se selecciona sólo un clasificador (o subconjunto) para la decisión final (clasificadores complementarios)
3. Se puede hacer una combinación de selección e integración

Ensambles de Clasificadores

Las técnicas desarrolladas de clasificación por votación o conjunta, las podemos dividir en tres grupos:

1. Los que cambian la distribución de los ejemplos de entrenamiento (*Boosting*),
2. Los que no cambian la distribución de los ejemplos (*Bagging*)
3. Los que combinan diferentes clasificadores (*Stacking*)

Combinación de Clasificadores

1. Voto mayoritario
2. Voto Bayesiano
3. Por "ranqueo"
4. Combinación lineal
5. Por producto
6. Pesada
7. Stacking
8. Seleccionador de regiones/clasificadores

Voto Bayesiano

- Desde el punto de vista bayesiano, lo que se quiere es tener la hipótesis más probable dado los datos:

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)}$$

- Cuando tenemos muchas hipótesis, podemos combinar esas hipótesis:

$$P(v_j \mid D) = \sum_{h_i \in H} P(v_j \mid h_i)P(h_i \mid D)$$

- Esto es, las probabilidades sobre cada valor que nos da cada hipótesis pesada por la probabilidad posterior que nos da la hipótesis dado los datos

Voto Bayesiano

- Esto es, por Bayes, proporcional a la probabilidad de los datos dada la hipótesis por la probabilidad de la hipótesis:
$$P(h \mid D) \propto P(D \mid h)P(h)$$
- Podriamos tener un enfoque completo bayesiano si pudieramos evaluar las expresiones anteriores para todas las posibles hipótesis
- Los ensambles se pueden ver como una aproximación tomando en cuenta las hipótesis “más probables”

Voto Bayesiano

- Por lo que una forma de construir ensambles es generar varias hipótesis y combinarlas desde un enfoque bayesiano
- La parte más idealizada del enfoque es $P(h)$ ya que muchas veces tanto el espacio de hipótesis (H) como $P(h)$ se seleccionan por razones computacionales

Combinación por “Ranqueo”

1. Transformar las salidas en un orden
2. Combinar usando por ejemplo el método de conteo de Borda

Valor	Clasif. 1	Clasif. 2	Clasif. 3
4	c	a	b
3	b	b	a
2	d	d	c
1	a	c	d

Combinación por "Ranqueo"

1. Lo que tenemos entonces es:
   $r_a = r_a^1 + r_a^2 + r_a^3 = 1 + 4 + 3 = 8$
   $r_b = r_b^1 + r_b^2 + r_b^3 = 3 + 3 + 4 = 10$
   $r_c = r_c^1 + r_c^2 + r_c^3 = 4 + 1 + 2 = 7$
   $r_d = r_d^1 + r_d^2 + r_d^3 = 2 + 2 + 1 = 5$
2. La clase ganadora es entonces b
3. Útiles con muchas clases, al combinar varios clasificadores, pero dependen de los números asociados

Otros Esquemas de Combinación

1. El promedio simple es la mejor opción con clasificadores con el mismo desempeño y misma correlación entre ellos
2. La combinación lineal:

$$p(x) = \sum_{i=1}^{n} w_i p_i(x)$$

3. Producto:

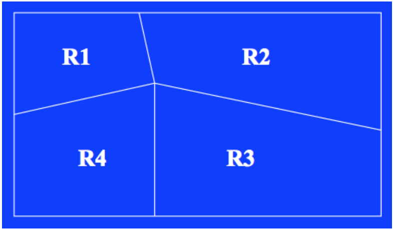
$$p(x) = \prod_{i=1}^{n} p_i(x)$$

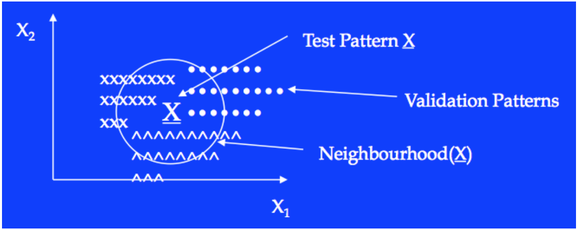
4. Otros esquemas: el máximo, el mínimo, la media
5. Esquemas pesados asociados al desempeño del clasificador, a la clase, tomados de la matriz de confusión, etc.

Combinación por Selección

1. Seleccionar el mejor clasificador
2. Definir regiones “de dominio”: Por *clustering* o basadas en desempeños locales
3. Seleccionar al clasificador

Combinación por Selección





Esquemas de Ensambles

Algunos esquemas generales para construir ensambles son:

1. Usando conocimiento del dominio
2. Manipulando los ejemplos de entrenamiento
3. Manipulando los atributos
4. Inyectando aleatoriedad
5. Manipulando las salidas

Manipulando los ejemplos de entrenamiento

- Ya vimos como una forma de construir ensambles es manipulando los ejemplos de entrenamiento para generar múltiples hipótesis
- Esto funciona cuando tenemos clasificadores inestables, esto es, algoritmos que pueden tener grandes cambios en sus resultados con pequeños cambios en los datos
- Árboles de decisión, redes neuronales y reglas de clasificación son inestables; Regresiones lineales y vecinos más cercanos tienden a ser estables

Manipulando los ejemplos de entrenamiento

Las formas más fáciles de manipular los ejemplos de entrenamiento son:

1. Muestreando con reemplazo los datos (*Bagging*)
2. Creando muestras disjuntas (*cross-validated committees*)
3. Pesando los datos (*AbaBoost*)

Manipulando los Atributos

- Otra forma de crear ensambles o clasificadores múltiples, es cambiando los atributos de entrada (*Random Subspace Method*).
- Esto funciona mejor cuando se tienen atributos redundantes
- Se puede hacer una selección aleatoria entre todos o entre los mejores N atributos (hay que definir N)
- Se pueden hacer transformaciones (e.g., aplicando Kernels)

Inyectando Aleatoriedad

- Finalmente, se puede introducir aleatoriedad a los algoritmos, esto es fácil en árboles de decisión en donde en lugar de seleccionar el mejor atributo en cada paso, se selecciona uno aleatoriamente dentro de los primeros N
- Otra posibilidad es seleccionar el mejor atributo dentro de un subconjunto aleatorio de atributos. Usando *bagging*, con esto se pueden construir varios árboles. Esto es lo que hace *Random Forest*
- Para determinar el número de atributos a seleccionar, muchas veces se usa $\sqrt{p}$ donde p es el número de atributos y para problemas de regresión $p/3$ con un mínimo de 5

Out of Bag (OOB) Score

- Sirve para validar el modelo producido por *Random Forest* cuando se tienen pocos ejemplos
- Al estar haciendo *Bagging* cada árbol se entrena con un subconjunto de los ejemplos, quedando otros ejemplos sin ver
- Cada ejemplo no visto por *N* clasificadores, se prueba en ellos y se clasifica usando voto mayoritario (de esos clasificadores)
- El promedio de clasificaciones correctos de todos los ejemplos no vistos por los árboles es el *OOB score*

Inyectando Aleatoriedad

- La aleatoriedad se puede introducir en los datos o en los clasificadores.
- *Wagging* (*Weight Aggregation*) funciona como *Bagging* pero introduce un ruido Gaussiano con media cero a cada peso de los ejemplos.
- Algunos algoritmos ya tienen incorporado un componente aleatorio, por ejemplo, la asignación inicial en las redes neuronales, por lo que se pueden hacer varias asignaciones iniciales y combinar los resultados usango *bagging*

Manipulando las Salidas

- Otra forma de crear ensambles es manipulando las salidas
- Lo más común es usar *error-correcting codes*, en donde un problema multiclase, se convierte en una combinación de muchos clasificadores binarios con una salida cambiada

Error-correcting output codes

- Se utilizan para mejorar el desempeño de clasificadores con clases múltiples
- Existen clasificadores que solo funcionan con clases binarias, como era originalmente con las máquinas de soporte vectorial (*support vector machines*)
- Para aplicarlos a más clases, se dividen los datos en conjuntos independientes de datos con dos clases

Error-correcting output codes

- Se puede hacer como sigue, crear un clasificador binario para cada clase, e.g., pertenece a una clase o no. Para 4 clases, tendríamos 4 clasificadores. Al final nos quedamos con el de la clase más probable.
- El riesgo es que un clasificador que tenga una probabilidad alta se puede "comer" a los demás.
- Otra forma de hacerlo es generar $n(n-1)/2$ clasificadores para todos los pares de clases y quedarse con la clase mayoritaria.

Error-correcting output codes

- Las técnicas de corrección de errores basados en códigos de salida generan una codificación para cada clase que permita fácilmente identificar errores en un clasificador.
- Se corre el algoritmo en cada una y los resultados se combinan
- El método funciona en general bien, tanto, que a veces se usa aunque los clasificadores funcionen bien con múltiples clases

Error-correcting output codes

- Por ejemplo, si tenemos 4 clases podemos generar los siguientes códigos:

Clase	Código
a	1111111
b	0000111
c	0011001
d	0101010

- El primer clasificador me predice 1 si la clase es *a* y 0 si es *b*, *c* o *d*. El segundo clasificador me predice 1 si la clase es *a* o *d* y 0 si es *b* o *c*, etc.
- Aquí en lugar de generar 4 clasificadores, uno para cada clase, se generan 7 clasificadores

Error-correcting output codes

- Cuando tengo un nuevo ejemplo me fijo a qué código se parece más usando distancia *Hamming*. Por ejemplo, si la salida de los 7 clasificadores con un nuevo ejemplo es: 1011111, es claro que el clasificador 2 fué el que se equivocó.
- Lo importante es que exista la mayor distancia *Hamming* entre cada renglón.
- Para k clases, cada código va a tener $2^{k-1} - 1$ bits. La primera clase se contruye con puros 1's. La segunda con 2^{k-2} 0's seguida de $2^{k-2} - 1$, 1's. La tercera, 2^{k-3}, 0's seguida de 2^{k-3}, 1's, seguida de 2^{k-3}, 0's, seguida de $2^{k-3} - 1$, 1's, así sucesivamente hasta el código de la última que tiene 0's y 1's alternados.

Variantes

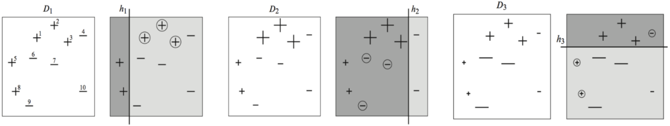
- *Option trees*: En lugar de producir muchos árboles, los árboles con opciones generan una sola estructura que representa en forma compacta varios árboles de decisión.
- Los árboles de opciones tienen dos tipos de nodos, los nodos de decisión y los nodos de opción
- En un nodo de decisión se sigue un camino, en un nodo de opción se siguen todos, por lo que los ejemplos ahora terminan en varias hojas
- Al final se combinan los resultados de cada hoja.

Variantes

- Una forma de crearlos es generar nodos de opción cada vez que se tienen medidas “parecidas” en ganancia de información. Para podar se toman los errores promedios de los nodos de opción.
- Otra forma de construirlos es ir añadiendo nodos, que se hace normalmente al hacer *boosting*, aquí los árboles se llaman *alternating decision trees*, los nodos de decisión, *splitter nodes* y los nodos de opción se llaman *prediction nodes*.

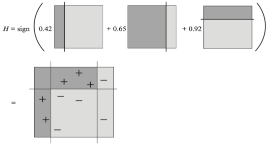
Gradient Boosting Machines

- Gradient Boosting = Gradient descent + Boosting



- En Adaboost se aproxima un modelo como una suma de modelos parciales pesados ($H(x) = \sum_t \alpha_t h_t(x)$)
- En cada paso se añade un clasificador débil que busca compensar las deficiencias de los modelos existentes
- Estas deficiencias son ejemplos con altos pesos

Gradient Boosting Machines



- En Gradient Boosting se hace el mismo proceso, pero las deficiencias se identifican por los gradientes
- La idea es construir árboles de regresión (CART) que aproximen las diferencias entre las salidas reales y las salidas del model actual
- Si $F(x)$ es una función inicial, que con diferentes datos tiene algunos errores $e(x)$ tales que $F(x_i) + e(x_i) = y_i \forall i \in N$
- Lo que nos gustaría es generar una función $h(x)$ tal que $h(x) = y - F(x)$

Gradient Boosting Machines

- Para esto podemos ajustar un árbol de regresión con los siguientes datos de entrenaminto: $(x_1, y_1 - F(x_1)), (x_2, y_2 - F(x_2)), \ldots, (x_N, y_N - F(x_N))$
- Los $y_i - F(x_i)$ se llaman *residuos* (donde no le va bien al modelo actual)
- Si al modelo resultante todavía no le va bien, añade otra función (árbol de regresión)

Gradient Boosting Machines

- Cuando queremos ajustar una función muchas veces podemos usar el gradiente de una función de pérdida (e.g., error cuadrático medio)
- Si la función de pérdida es: $L(y, F(x)) = (y - F(x))^2$
- Lo que queremos minimizar es: $J = \sum_i L(y_i, F(x_i))$

$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial \sum_i L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y$$

- Por lo que los residuos los podemos interpretar como gradientes negativos

$$y_i - F(x_i) = -\frac{\partial J}{\partial F(x_i)}$$

Gradient Boosting Machines

$$\begin{aligned} F(x_i) &= F(x_i) + h(x_i) \\ &= F(x_i) + y_i - F(x_i) \\ &= F(x_i) - \frac{\partial J}{\partial F(x_i)} \\ \Theta_i &= \Theta_i - \alpha \frac{\partial J}{\partial \Theta_i} x \end{aligned}$$

Entonces actualizamos nuestro modelo con los gradientes!

Algoritmo genérico

$$-g(x_i) = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = y_i - F(x_i)$$

Inicia con un modelo, e.g., $F(x) = \frac{\sum_i^N y_i}{N}$

Mientras no haya convergencia:

- Calcula los gradientes negativos $-g(x_i) = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$
- Ajusta un árbol de regresión h a estos valores $-g(x_i)$
- $F = F + \alpha h$

Algoritmo Genérico

- Lo importante de la formulación es que podemos usar otras funciones de pérdida sin cambiar el esquema general
- $L(y, F(x)) = |y - F(x)|$, su gradiente es: $-g(x_i) = \text{signo}(y_i - F(x_i))$
- Huber loss:

$$L(y, F(x) = \begin{cases} \frac{1}{2}(y - F(x))^2 & \text{si } |y - F(x)| \leq \delta \\ \delta(|y - F(x))| - \frac{\delta}{2} & \text{si } |y - F(x)| > \delta \end{cases}$$

Su gradiente es:

$$-g(x_i) = \begin{cases} y_i - F(x_i) & \text{si } |y - F(x)| \leq \delta \\ \delta(\text{signo}(y - F(x)) & \text{si } |y - F(x)| > \delta \end{cases}$$

Resumiendo

- Ajusta un modelo aditivo por pasos
- En cada paso añade un árbol de regresión para compensar las deficiencias del modelo actual
- Ahora las deficiencias son los gradientes negativos de una función de pérdida
- Se puede usar cualquier función de pérdida

GBM para Clasificación

- Si en lugar de una salida contínua queremos clasificar N clases, los GBM generan N funciones, una por cada clase:

$$P_{C1}(x) = \frac{e^{F_{C1}(x)}}{\sum_{i=1}^{N} e^{F_{Ci}(x)}}$$

$$P_{C2}(x) = \frac{e^{F_{C2}(x)}}{\sum_{i=1}^{N} e^{F_{Ci}(x)}}$$

$$\ldots$$

$$P_{CN}(x) = \frac{e^{F_{CN}(x)}}{\sum_{i=1}^{N} e^{F_{Ci}(x)}}$$

y se predice la clase más probable

Pasos del Algoritmo

1. Convierte la etiqueta de cada ejemplo en una distribución de probabilidad (e.g., Si el ejemplo 4 es de la clase 3 y hay 5 clases, $y_{C1}(x_4) = 0, y_{C2}(x_4) = 0, y_{C3}(x_4) = 1, y_{C4}(x_4) = 0, y_{C5}(x_4) = 0$
2. Calcula la distribución de probabilidad del modelo (inicialmente se puede tener una distribución uniforme)
3. Calcula las diferencias entre distribuciones, por ejemplo, usando la divergencia de Kullbak-Leibler (KL)
4. La meta es minimizar la función de pérdida, en este caso KL (de nuevo se obtiene su gradiente y se itera)

GBM para clasificar

- Ahora en lugar de una columna de parámetros que queremos optimizar del tamaño de los ejemplos, tenemos una matriz de parámetros de n clases y m ejemplos con sus respetivos gradientes
- Lo que se hace es ajustar n funciones de distribución, una por cada clase

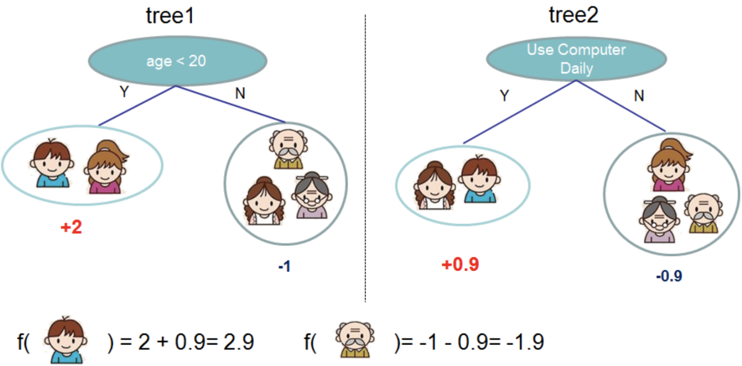
XGBoost (un GBM optimizado)

- Con n ejemplos, m atributos y $\mathbb{D} = \{(\vec{x}_i, y_i)\}$ datos, se construye un ensamble de K árboles de regresión (CART):

$$\hat{y}_i = \phi(\vec{x}_i) = \sum_{k=1}^{K} f_k(\vec{x}_i), f_k \in \mathcal{F}$$

  donde $\mathcal{F}$ es el espacio de posibles árboles de regresión
- En un ensamble de árboles de regresión (CART) la predicción se obtiene sumando los valores de cada árbol

XGBoost


tree1
age < 20
Y
N
+2
-1
tree2
Use Computer Daily
Y
N
+0.9
-0.9
f( ) = 2 + 0.9= 2.9
f( )= -1 - 0.9= -1.9

XGBoost

- Actualmente es común usar una función de perdida con un regularizador que penaliza los modelos complejos:

$$\mathcal{L}(\phi) = \sum_i l(y_i, \hat{y}_i) + \Omega_k(f_k)$$

  donde l es una función de pérdida diferenciable que mide la diferencia entre el valor predicho y el real

- Ω mide la complejidad del modelo:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2$$

  donde T es el número de hojas en el árbol y w son sus pesos

XGBoost

- Lo que hace *Boosting* es que añade incrementalmente un nuevo modelo f_t:

$$\begin{aligned} \hat{y}_i^{(0)} &= 0 \\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\ \ldots & \\ \hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \end{aligned}$$

- Para saber qué nuevo árbol añadir en cada paso, lo que buscamos es aquel que nos minimice la función de pérdida junto con el regularizador:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(\vec{x}_i)) + \Omega_k(f_t)$$

XGBoost

- Si pensamos como función de pérdida en el error cuadrático medio:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(\vec{x}_i)))^2 + \sum_{i=1}^{t} \Omega_k(f_t)$$

- Abusando de notación, expandiendo $(y_i - (\hat{y}_i^{(t-1)} + f_t(\vec{x}_i)))^2$, nos da:

$$= y^2 - y\hat{y} - yf - y\hat{y} + \hat{y}^2 + \hat{y}f - yf + \hat{y}f + f^2$$

$$= y^2 - 2y\hat{y} + \hat{y}^2 + 2(\hat{y} - y)f + f^2$$

- Lo que nos da:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} 2(\hat{y}_i^{(t-1)} - y_i)f_t(\vec{x}_i)) + f_t(\vec{x}_i)^2 + \Omega(f_t) + cte$$

XGBoost

- En general, no con todas las funciones de pérdida salen expresiones sencillas, por lo que se aproximan con una serie de Taylor de segundo orden:

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\vec{x}_i) + \frac{1}{2} h_i f_t^2(\vec{x}_i)) + \Omega_k(f_t) + cte$$

donde:

$$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

y

$$h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

- Quitando constantes podemos simplificar a:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\vec{x}_i) + \frac{1}{2} h_i f_t^2(\vec{x}_i)] + \Omega_k(f_t)$$

XGBoost

- La ventaja de esta forma es que depende de g_i y h_i y podemos usar cualquier función de pérdida
- Para el regularizador, definimos la definición de la función del árbol como: $f_t(x) = w_{q(x)}$, donde w es un vector con los valores de las hojas y $q(x)$ es una función que asigna cada dato a la hoja correspondiente.
- Considerando la definición de $f_t(x)$ y la definición de Ω la expresión anterior nos queda:

$$\begin{aligned} \tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n} [g_i w_{q(x)} + \frac{1}{2} h_i w_{q(x)}^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned}$$

donde $I_j = \{i | q(x_i) = j\}$ es el conjunto de instancias en la hoja j

XGBoost

- Definiendo $G_j = \sum_{i \in I_j} g_i$ y $H_j = \sum_{i \in I_j} h_i$, lo anterior nos queda:

$$\mathcal{L}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

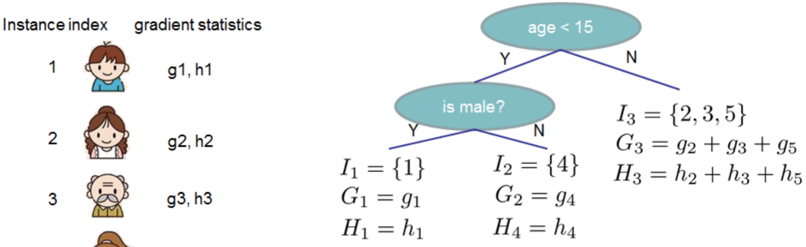
- Para una estructura fija (árbol $q(\vec{x})$) se puede calcular el peso óptimo de la hoja j como:

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

y calcular el valor óptimo correspondiente:

$$\mathcal{L}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

XGBoost



$$Obj = -\sum_j \frac{G_j^2}{H_j+\lambda} + 3\gamma$$

The smaller the score is, the better the structure is

XGBoost

- Esto se puede usar como una función para medir la calidad del árbol
- Cómo no se pueden enumerar todos los árboles y evaluar cuál es mejor, se usa una estrategia *greedy* que evalúa cada nodo a la vez y añade ramas a los árboles.
- Si I y D son las instancias en la rama izquierda y derecha respectivamente, la reducción en la función de pérdida (equivalente a contenido de información) después de dividir el nodo es:

$$\text{Ganancia} = \frac{1}{2}\left[\frac{G_I^2}{H_I+\lambda} + \frac{G_D^2}{H_D+\lambda} - \frac{(G_I+G_D)^2}{H_I+H_D+\lambda}\right] - \gamma$$

- Si la ganancia es menor que γ no se incluye

XGBoost

- Con atributos contínuos, se ordenan los datos (como en C4.5) y se busca el mejor corte
- Para acelerar el proceso, XGBoost usa un subconjunto aleatorio de atributos (ya sea de forma global o local), guarda información de los atributos ordenados, guarda estadísticas de cada atributo y paraleliza
- También comprimen y descomprimen datos y particionan los datos de forma horizontal (*shard*)
- Puede usar distintas funciones de costo y diferentes métricas de evaluación
- Sirve para regresión y para clasificación

Problemas generales de los ensambles

- Tratar adecuadamente el ruido para no compensar demasiado ejemplos erroneos.
- Se requiere más cómputo y memoria y métodos para eliminar redundancia y paralelizar.
- Obscurecen las decisiones tomadas.

Comentarios Finales

- Si el clasificador es inestable (e.g., árboles de decisión) entonces usar *Bagging*.
- Si el clasificador es estable y simple (e.g., Naive Bayes) entonces usar *Boosting*.
- Si el clasificador es estable y complejo (e.g., Redes Neuronales) entonces inyectar aleatoriedad
- Si se tienen muchas clases con clasificación binaria, entonces usar códigos de corrección de errores

Comentarios Finales

- *Bagging* y crear aleatoriedad ayudan a muestrear el espacio de hipótesis con un sesgo hacia buenos clasificadores
- *Boosting* ataca el problema de la representación, pero aumenta la posibilidad de realizar sobreajustes (con poco ruido *AdaBoost* tiene un buen desempeño, pero con mucho ruido puede sobreajustar mucho)
- Las razones por las cuales *AdaBoost* no sobreajusta de más todo el tiempo, tiene que ver con la forma en que se realiza, ya que el ajuste de pesos es uno a la vez.

Comentarios Finales

- Para bases de datos grandes, introducir aleatoriedad puede tener mejores resultados que *Bagging*, ya que con muchos datos es más probable que los conjuntos de entrenamiento de *Bagging* sean muy parecidos y por lo tanto los resultados también.
- Por otro lado, introducir aleatoriedad crea diversidad bajo cualquier condición, con el riesgo de crear clasificadores de baja calidad.
- Actualmente el área está dominada por Random Forest y XGBoost