6. Modelos de función de base adaptativa

6.1. Introducción

Esta sección se refiere a una familia general de modelos que llamaremos modelos de **función de base adaptativa**, que tienen la forma:

$$f(x) = w_0 + \sum_{k=1}^{K} w_k \phi_k(x)\,. \tag{6.1}$$

A la función $\phi_k(x)$ se le dice la m-ésima función de base, la cual variará en función de los datos. Esta familia general de modelos incluye los modelos a estudiar en está sección: árboles, bosques, modelos basados en bagging y boosting, como también las redes neuronales y las sumas generales de modelos (Murphy, 2022).

6.2. Árboles

6.2.1. Motivación con caso regresión

Antes de definir árboles de manera formal, construiremos una intuición para el caso de regresión. Consideremos una función de una variable. Una manera de aproximar tal función es hacer una interpolación usando funciones constantes. Una primera idea puede ser aproximar la función entera simplemente por su media.

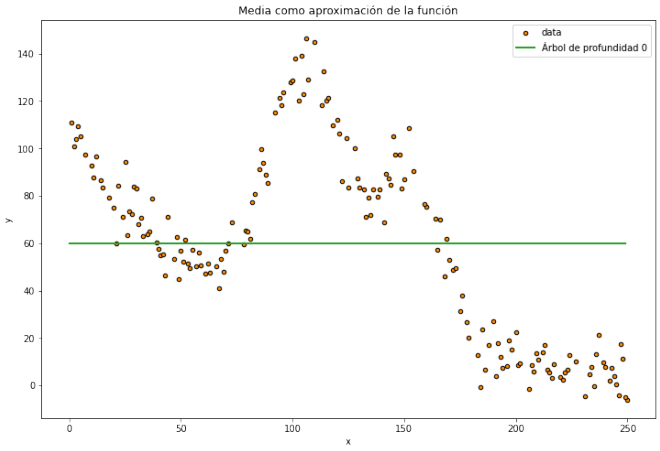

Fig. 30. Aproximación de una función usando su media (árbol trivial no ajustado).

Otra manera, más bien voraz, de enfrentar el problema es realizar la interpolación con todos los puntos. Esto resultará con seguridad en un sobreajuste de los datos.

Una manera más inteligente consiste en agrupar ciertos puntos y hacer una interpolación usando la media de estos. El desafío es encontrar una partición conveniente del dominio de los datos de modo que al predecir un punto de nuestra función, tomar la media de los datos de entrenamiento para el subconjunto correspondiente resulte en una buena aproximación.

6.2.2. Algoritmo *CART*

Los árboles (de regresión) corresponden a ejecutar lo anterior de manera recursiva. Primero, necesitamos una algún criterio que nos indique si realizar un corte vale o no la pena. Para esto podemos definir el costo para un conjunto D de datos como sigue:

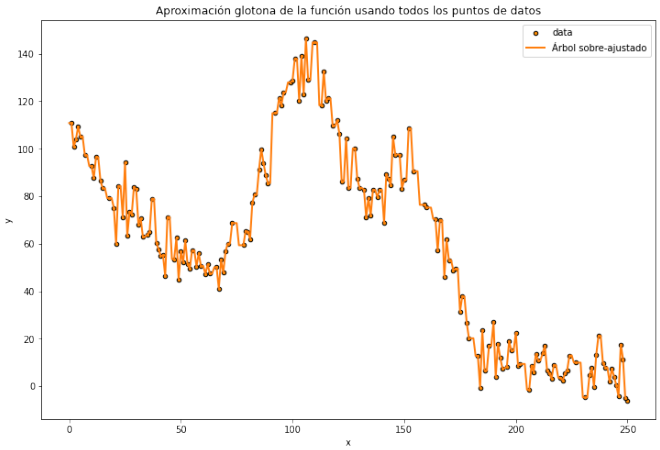

Fig. 31. Aproximación de una función usando todos los puntos de entrenamiento (árbol sobreajustado).

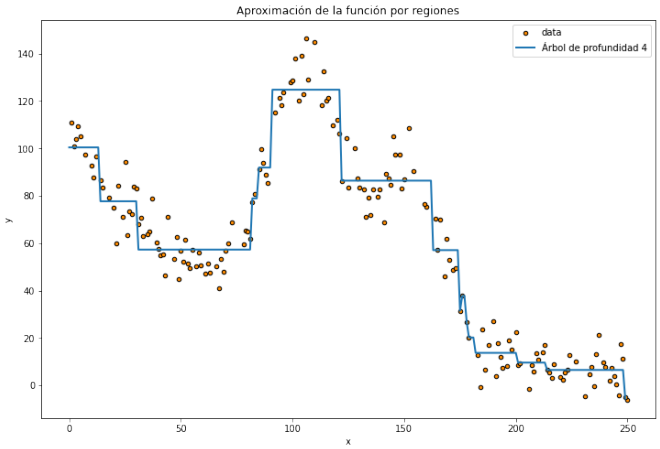

Fig. 32. Aproximación de una función con un árbol de profundidad adecuada.

$$cost(D) = \sum_{i \in D} (y_i - \bar{y})^2 \,, \tag{6.2}$$

con $\bar{y} = \frac{1}{|D|} \sum_{i \in D} y_i$. Notemos como este costo es proporcional a la varianza empírica del conjunto D, con lo cual pedir bajo costo en los grupos de una partición se traduce en pedir que los datos estén cercanos a la media de tal subconjunto. Con esto podemos armar un algoritmo de base recursivo que genere un árbol de regresión.

Algoritmo 1 Ajuste de árboles (CART)

function AJUSTEARBOL($nodo, D, profundidad$)
  $j^*, t^* = \arg\min_{j \in \{1,\dots,m\}, t \in \Gamma_j} [cost(D_L(j,t)) + cost(D_R(j,t))]$
  $D_L(j,t) = \{(x^{(i)}, y^{(i)}) : x_j^{(i)} \leq t\}$, $D_R(j,t) = \{(x^{(i)}, y^{(i)}) : x_j^{(i)} < t\}$
  if criterio_de_parada($costo, profundidad, D_I, D_D$) **then**: **return** nodo
  else
    $nodo$.izquierda = ajusteArbol($nodo, D_I, profundidad + 1$)
    $nodo$.derecha = ajusteArbol($nodo, D_R, profundidad + 1$)
  return nodo

Nótese que esto no necesariamente encontrará el árbol binario óptimo. Se prefiere este método voraz pues ajustar un árbol binario óptimo es un problema NP completo. En particular notemos como vamos separando coordenada por coordenada, lo cual nos hace ganar en interpretabilidad. Por otro lado, el criterio de parada lo discutiremos más adelante.

Siguiendo el algoritmo anterior obtendremos una partición. Podemos enumerar los nodos de 1 hasta K, con lo cual recuperamos la forma de base adaptativa:

$$f(x) = \mathbb{E}[y|x] = \sum_{k=1}^{K} w_k \phi_k(x)\,, \tag{6.3}$$

con $\phi_k(x) = \mathbf{1}_{D_k}(x)$ y $w_k = \frac{1}{|D_k|}\sum_{x^{(i)}\in D_k} y^{(i)}$. Como se señaló en la intuición, escogeremos la media de los datos en el subconjunto como predicción. Esto se justifica pues aquel valor es el que minimiza el error cuadrático. Podríamos también aprender un modelo simple (por ejemplo un regresor de mínimos cuadrados) de manera local en cada partición, sin embargo como hemos escogido la partición de manera que minimice la varianza, es razonable pensar que la media será una aproximación suficientemente buena.

6.2.3. Criterios de corte para clasificación

En el algoritmo CART hicimos uso de la función costo, que nos mostraba cuánto variaban los miembros de un intervalo respecto de la media. Podemos generalizar este mismo principio para clasificación, donde intentaremos caracterizar la impureza de etiquetas para un conjunto de manera adecuada. Primero tomemos el vector de probabilidades de pertenencia a una clase, condicionado a estar en un nodo. Sea $\mathcal{C}$ el conjunto de clases,

$$\hat{\pi}_c(D) = \frac{1}{|D|}\sum_{x^{(i)}\in D} \mathbf{1}_{y=c}(y^{(i)}) \qquad ,\forall c \in \mathcal{C}\,. \tag{6.4}$$

Usando esto, predecir la probabilidad de que un punto pertenezca a cada clase estará dado por el vector de fracciones empírica $\hat{\pi}$ correspondiente al nodo al cual pertenezca el dato en cuestión. Usemos este mismo vector para definir los criterios de impureza por nodo (ignoraremos la dependencia de D en el vector de probabilidades para simplificar la notación) (Breiman y cols., 1984).

- **Tasa de error**
  Sea $\hat{y} = \arg\max_{c\in\mathcal{C}} \hat{\pi}_c$ la clase más probable. El error estará dado por

  $$cost(D) = \frac{1}{|D|}\sum_{x^{(i)}\in D} \mathbf{1}_{y=\hat{\mathbf{y}}}(y^{(i)}) = 1 - \hat{\pi}_{\hat{y}} \tag{6.5}$$

  El problema de este criterio es su poca sensibilidad a cambios en el vector de probabilidad. Los siguientes dos criterios mejoran esta situación.

- **Gini**
  Corresponde a la tasa de error esperado:

  $$cost(D) = \sum_{c\in\mathcal{C}} \hat{\pi}_c(1-\hat{\pi}_c) = 1 - \sum_{c\in\mathcal{C}} \hat{\pi}_c^2 \tag{6.6}$$

- **Entropía**
  También llamada log-pérdida y muchas veces denotada por $H(\hat{\pi})$, esta métrica está dada por:

  $$cost(D) = -\sum_{c\in\mathcal{C}} \hat{\pi}_c log(\hat{\pi}_c)\,. \tag{6.7}$$

  Esta elección de perdida tiene justificación en la Teoría de la Información. En particular, su uso como criterio de corte equivale a la minimización de la entropía cruzada.

Consideremos el ejemplo de clasificación binaria. Notemos que para las tres el máximo está cuando tenemos 50/50 de datos para cada clase en el nodo en cuestión, que es justamente el caso de mayor heterogeneidad de los datos. Por el contrario, los valores son cero cuando el conjunto tiene miembros de una sola clase. La sensibilidad antes mencionada se desprende de esto. En el caso de clases impuras, el error de clasificación está siempre por debajo de los otros criterios, que castigan más fuertemente la impureza.

6.2.4. Evitar sobreajuste: detención temprana y poda

Una primera estrategia para evitar el sobreajuste del árbol es la **detención temprana**, i.e., evitar que se siga particionando el espacio. A continuación mencionaremos algunos de los varios criterios. Estos se aplican como un criterio a verificar antes de seguir partiendo nodos en el algoritmo 1 (CART):

- Máxima profundidad: se puede imponer una profundidad máxima, de modo que una vez alcanzada esta no se separe más el nodo en cuestión.
- Número mínimo en nodo: si un nodo contiene muy pocos datos entonces partirlo puede resultar en nodos con muy pocos datos (a este fenómeno se le llama fragmentación de datos). Podemos considerar o bien un entero o bien un porcentaje mínimo de los datos totales.
- Mínima reducción de costos: Dados lados izquierdos y derechos de un corte, consideremos la reducción en costo como:
$$\Delta = cost(D) - \left[\frac{|D_L|}{|D|}cost(D_L) + \frac{|D_R|}{|D|}cost(D_R)\right] \tag{6.8}$$
Esta métrica de reducción de costos normalizada nos permite definir un criterio de parada donde no cortaremos el nodo si los nodos resultantes no inducen una reducción significativa.

Los criterios anteriores inducen hiperparámetros que pueden ser escogidos en una búsqueda de grilla.

La **poda** es otra estrategia para evitar el sobreajuste y disminuir la complejidad del árbol. Esta consiste en ajustar un árbol uno (posiblemente con criterios de parada anteriormente expuestos) y luego podarlo, que corresponde a elegir un sub-árbol (i.e., "podar" nodos). El conjunto de todos los subárboles de un árbol de decisión es potencialmente muy elevado. Es por esto que se elegirá un conjunto adecuado de subárboles, de modo que sea razonable comparar su rendimiento para elegir la mejor opción.

Primero consideraremos una métrica $R(T)$ que nos de una noción de costo para un árbol T. Típicamente se usará la suma de las tasas de errores (definidas anteriormente para clasificación y regresión) sumadas para cada hoja. Consideremos que el tamaño de un árbol T es su número de hojas y denotemos aquello por $|T|$. Lo anterior nos permite definir una métrica que incorpora tanto el error como el tamaño de un árbol:

$$R_\alpha(T) = R(T) + \alpha|T| \tag{6.9}$$

Esta expresión se puede interpretar como agregar una penalización por complejidad si pensamos α como costo en complejidad de un nodo terminal. Notemos que a medida que se aumenta α más estaremos prefiriendo un árbol con menos hojas, por ende la métrica es sensible a tal hiperparámetro. Usaremos este hecho para construir un algoritmo que seleccione árboles adecuados de los cuales seleccionar el mejor, donde la idea se resume a continuación:

Dado el árbol original T_0, el objetivo será construir una sucesión de árboles $T_0, T_1, \ldots, T_m$, disminuyendo en cada paso el número de nodos terminales y donde T_m es simplemente el nodo raíz. Para aquello notemos que pese a que el número de subárboles de T_0 es potencialmente grande, siempre es un número finito. Con esto, si $T(\alpha)$ es el árbol que minimiza el costo R_α, entonces al aumentar α, aquel árbol seguirá siendo el óptimo hasta llegar a un punto de salto α', en el cual un nuevo árbol $T(\alpha')$ se convierte en el mínimo y así sucesivamente.

Este punto se encuentra guardando los costos y tamaños de subárboles dados por podar en algún nodo. Cuando podemos esto consistirá en deshacer la separación hecha en el nodo en cuestión, con lo cual nos quedará la estimación que teníamos para el subconjunto original sin separar partes derecha e izquierda. La idea es encontrar iterativamente el nodo más débil que podamos podar.

A continuación el algoritmo de poda (Ripley, 2008), luego del cual podemos enunciar los resultados que lo justifican. Precisemos que T_t se refiere al subárbol cuya raíz (nodo del cual salen todas las ramas) es el nodo t.

Algoritmo 2 Poda de costo-complejidad

1: **function** SUCESIONARBOLES(T)
2: **set** $k = 0$, $T_0 = T$, $\alpha = \infty$
3: **for** t **in** nodos no terminales desde abajo hacia arriba **do**:
4: calcular $R(T_t)$ y $|T_t|$ sumando sobre los descendientes e incluyendo contribuciones en t
5: **set** $g(t) = \frac{R(t)-R(T_t)}{|T_t|-|t|}$
6: **set** $\alpha = \min(\alpha, g(t))$
7: **for** t **in** nodos no terminales de arriba hacia abajo **do**:
8: **if** $g(t) = \alpha$ **then**
9: Reemplazar T_t por t (podar)
10: **set** $k = k + 1$, $\alpha_k = \alpha$ y $T_k = T$
11: **if** T es un árbol de un sólo nodo **then**
12: **return** $\alpha_1, \ldots, \alpha_m$, $T_1, \ldots, T_m$
13: **else**
14: Ir a paso (3)

Notemos que la función g nace de querer despejar aquel α que le de suficiente peso al tamaño del árbol de modo que $R_\alpha(T_t) = R_\alpha(t)$, esto es

$$R(T_t) + \alpha|T_t| = R(t) + \alpha|t| \iff \alpha = \frac{R(t) - R(T_t)}{|T_t| - |t|}\,. \tag{6.10}$$

Proposición 6.0.1 (Consistencia de poda costo-complejidad)**.** *Sea $g(t,T) = \frac{R(t)-R(T_t)}{|T_t|-|t|}$ para un nodo t y un subárbol T_t con raíz en t.*

1. *El resultado de podar en un nodo t si $R_\alpha(t) \leq R_\alpha(T_t)$ al visitar los nodos de abajo hacia arriba, el árbol resultante es*
$$T(\alpha) = \arg\min_{\tilde{T} \leq T} R_\alpha(\tilde{T}) \tag{6.11}$$

2. *Sea $\tilde{\alpha} = \min\{g(t,T) : t$ es nodo no terminal de $T\}$, podar en todos aquellos nodos que cumplan $g(t,T) = \tilde{\alpha}$ resulta en $T(\tilde{\alpha})$. Además, $g(t, T(\tilde{\alpha})) > \tilde{\alpha}$ para todo nodo t no terminal en $T(\tilde{\alpha})$.*

3. *Para $\beta > \alpha$, $T(\beta)$ es subárbol de $T(\alpha)$ y es el resultado de β-podar $T(\alpha)$.*

Hasta ahora lo único que hemos hecho es definir una secuencia de árboles conveniente, sin embargo esto nos deja la responsabilidad de elegir un buen árbol para nuestro estimador final. Lo que tenemos hasta ahora son:

$$T = T_0 < T_1, \ldots, T_{m-1} < T_m\,, \tag{6.12}$$

donde $<$ denota la relación "ser subárbol". Además tenemos una sucesión $\alpha_0 < \alpha_1 < \ldots < \alpha_m$ (donde en este caso $<$ es la relación "menor a" usual en los números reales). Es de esperar que el error de entrenamiento aumente a medida que aumentamos α . Este no es necesariamente el caso en un conjunto

de testeo, pues es probable que los árboles con muchas hojas sean resultado de un sobreajuste. Una manera razonable de escoger un árbol final es tomar aquel que tenga menos error en un conjunto de entrenamiento o validación. También podemos hacer uso de técnicas como validación cruzada.

6.2.5. Interpretabilidad

Nos hemos restringido al ajuste de un árbol, sin embargo es útil pensar en las ventajas que puede tener este tipo de modelos respecto a otros vistos en el curso. Por su naturaleza, los árboles tienen una buena capacidad de interpretabilidad.

Usando las variables que se usaron para cortar cada nodo y los valores para el corte óptimo, podemos explicar la diferencia en estimaciones obtenidas. A esto lo llamamos capacidad de interpretación, y es importante a la hora de usar aprendizaje de máquinas para decisiones con repercusión en el mundo real.

En este caso se ha ajustado un árbol para clasificar imágenes de dígitos escritos a mano (dataset *mnist*). Acá, cada variable corresponde a un pixel en particular, que en La Figura 33 están destacados en rojo. Cuando se llegan a hojas del árbol se logra, en muchos casos, distinguir dígitos. Por otro lado, nodos intermedios denotan conjuntos impuros (con varios dígitos posibles), de los cuales es necesario realizar un corte. Se vislumbran entonces que pixeles son más importantes a la hora de distinguir dígitos y que desencadenan en la decisión tomada por el árbol de clasificación.

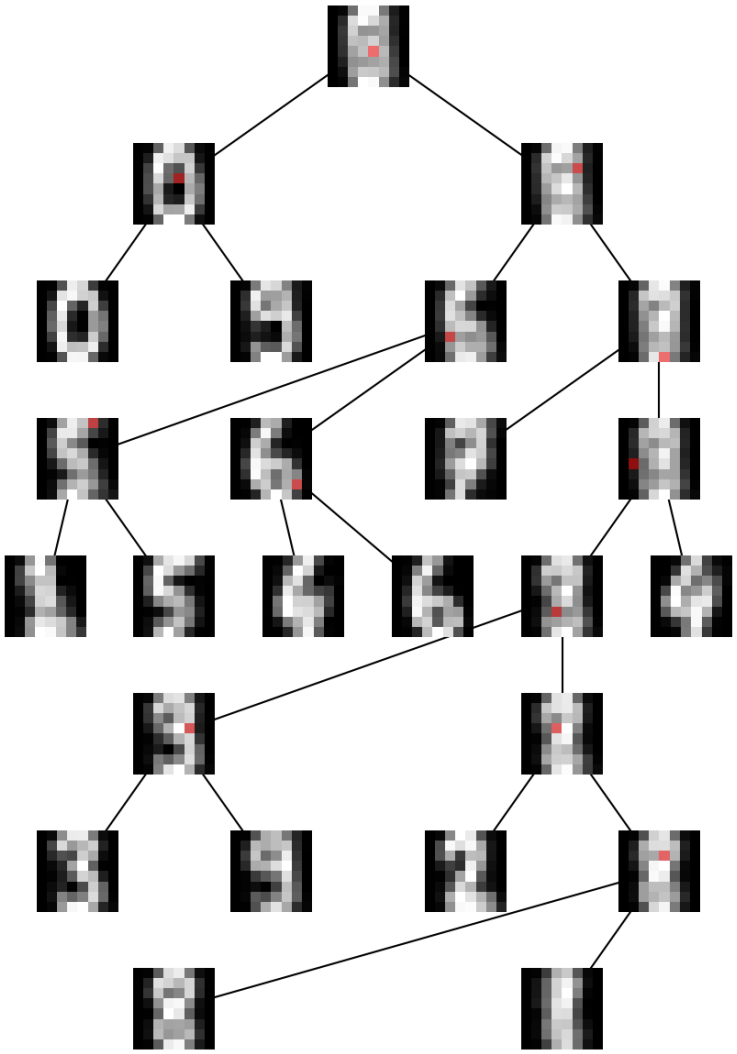

Fig. 33. Visualización de un árbol de clasificación para el dataset *mnist*.

6.3. Bagging

Esta sección tiene como objetivo mostrar un método para ensamblar modelos generales (Breiman, 1996). Antes de discutirlo, necesitamos la noción del concepto *bootstrapping*.

6.3.1. Método Bootstrapping

En palabras simples, *bootstrapping* es un procedimiento que consiste en escoger aleatoriamente puntos de datos con repetición, desde el conjunto de datos original. La repetición de esto nos permite acceder a varias distribuciones que intentan aproximar la original.

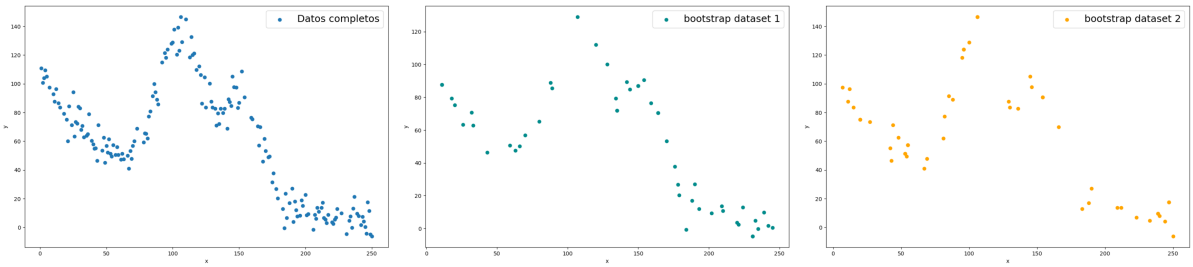

Fig. 34. Ejemplo de conjuntos de datos muestreados con *bootstrapping*.

Con estas distribuciones, que serán típicamente de menor tamaño, se espera capturar la variabilidad de los datos. Un ejemplo de uso es ajustar modelos a las diferentes distribuciones de *bootstrap*. Una idealización de lo anterior se muestra en La Figura 35, donde se usan dos re-muestreos (con repetición) para el ajuste.

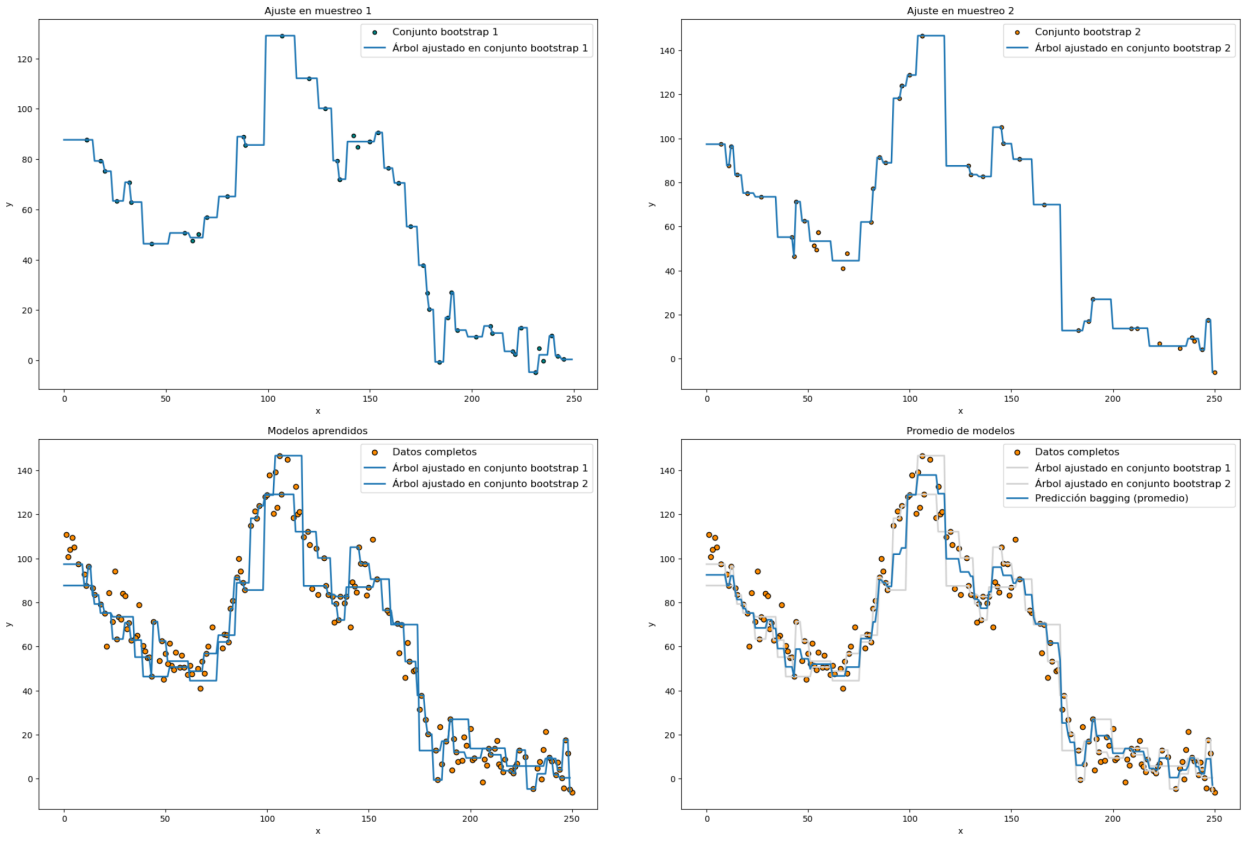

Fig. 35. Ejemplo de predicción *bagging* (promedio) usando dos modelos de árboles.

Podemos luego ver la diferencia de los estimadores versus los datos originales. Hacer esto varias veces nos puede dar una noción punto por punto de la varianza en ciertos puntos del input. Es interesante notar que si en vez de samplear desde los datos originales considerásemos un modelo con ruido Gaussiano y muestreamos de esa distribución obtendríamos el equivalente modelo ajustado con mínimos cuadrados a medida que consideramos más distribuciones.

El caso general es consistente con encontrar un modelo que maximice la verosimilitud. Usaremos este principio para definir *Bagging*, que será usar la información de varios modelos aprendidos en distintas distribuciones de *bootstrap*.

6.3.2. Bagging: agregación de modelos

Sea $\{\mathcal{D}^{(b)}\}_{b=1}^{B}$ una colección finita de conjuntos de datos tomados con el método *bootstrap*, es decir, muestreando repetidamente del conjunto de datos original $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}$ con repetición. Supongamos que en cada conjunto ajustamos un modelo, con lo cual tenemos B modelos: $\{\varphi(x, \mathcal{D}^{(b)})\}_{b=1}^{B}$.

El estimador resultante de usar el método *bagging* en cada caso estará dado por:

$$\varphi_{bagging}(x) = \frac{1}{B}\sum_{b=1}^{B} \varphi(x, \mathcal{D}^{(b)})\,. \tag{6.13}$$

El nombre ***bagging*** proviene de la combinación de ***b****ootstrap* y ***agg****regaga****ting***, lo último correspondiendo a agregación de modelos, lo cual viene de usar el promedio. En el caso de clasificación, lo usual es votar, i.e., tomar la clase que sea preferida por la mayor cantidad de modelos.

Es natural preguntarse en qué casos será conveniente usar *bagging* por sobre un modelo ajustado en el conjunto de datos original. La respuesta a esto dependerá de la sensibilidad de los modelos de base. En efecto denotemos

$$\varphi_{bagging}(x) = \mathbb{E}_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right]\,, \tag{6.14}$$

donde $\tilde{D} \sim D$ denota que el conjunto de datos usado para ajustar el modelo fue tomado usando muestreos que distribuyen de acuerdo a la distribución de probabilidad del modelo original. Usando esto, podemos acotar el error esperado usando que:

$$\mathbb{E}_{\tilde{D}\sim D}\left[y - \varphi(x, \tilde{D})\right]^2 = y^2 - 2y\mathbb{E}_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right] + \mathbb{E}_{\tilde{D}\sim D}\left[\varphi^2(x, \tilde{D})\right]\,. \tag{6.15}$$

Además, gracias a la desigualdad de Cauchy-Schwartz, podemos acotar el tercer término:

$$\mathbb{E}_{\tilde{D}\sim D}\left[\varphi^2(x, \tilde{D})\right] \geq \mathbb{E}_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right]^2\,. \tag{6.16}$$

Con esto queda:

$$\mathbb{E}_{\tilde{D}\sim D}\left[y - \varphi(x, \tilde{D})\right]^2 \geq (y - \mathbb{E}_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right])^2 \tag{6.17}$$

$$= (y - \varphi_{bagging}(x))^2\,. \tag{6.18}$$

Luego considerando todos los datos obtenemos que el error cuadrático promedio de los modelos será mayor al error cuadrático del estimador *bagging*. El margen de mejora depende solamente de cuán fuerte es la desigualdad de la inecuación 6.16, que podemos re-escribir como

$$Var_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right] = \mathbb{E}_{\tilde{D}\sim D}\left[\varphi^2(x, \tilde{D})\right] - \mathbb{E}_{\tilde{D}\sim D}\left[\varphi(x, \tilde{D})\right]^2 \geq 0\,. \tag{6.19}$$

Dicho de otro modo, la varianza del modelo para las distribuciones de *bootstrap* serán esenciales para que existan ganancias significativas. El método *bagging* es entonces recomendable para modelos

inestables, i.e., que varíen más bruscamente con cambios en los datos. Para modelos que no varíen demasiado con las distintas distribuciones *bagging* no introduce mejora (y en la práctica puede empeorar la predicción).

6.3.3. Bosques aleatorios y variaciones

En principio es razonable pensar que la técnica *bagging* funcionará con árboles, pues estos son relativamente sensibles a los datos utilizados (sobretodo si no podamos o si no usamos criterios de parada exigentes). Justamente cuando tenemos árboles suficientemente profundos, podemos asumir que tendrán un sesgo muy pequeño. La técnica *bagging* no afecta el sesgo de los estimadores, mejora sus resultados más bien reduciendo la varianza.

Supongamos que aprendemos B árboles, cada uno en un conjunto de datos generados con *bootstrapping*. Teóricamente cada estimador $\varphi(x, \mathcal{D}^{(b)})$ tendrá la misma varianza, que podemos denotar σ^2, pues los estimadores son independientes e idénticamente distribuidos. En la práctica los estimadores no son necesariamente independientes. Sea ρ la correlación dos-a-dos de los estimadores. En este caso, la varianza del estimador *bagging* (promedio) está dada por:

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2\,,$$

a diferencia del caso i.i.d., para el cual la varianza final es

$$\frac{1}{B}\sigma^2\,.$$

En ambos casos la varianza se reduce al aumentar B, sin embargo el término $\rho\sigma^2$ nos restringe la reducción de varianza. La idea de los bosques aleatorios (conocidos como *random forest* en inglés) será reducir la correlación de los estimadores para reducir la varianza sin afectar fuertemente el sesgo, y de esta forma reducir el error. Haremos esto seleccionando aleatoriamente variables con las cuales aprender cada árbol, como se señala en Algoritmo 3 (Breiman, 2001).

Algoritmo 3 Bosques Aleatorios

```
1: function BosqueAleatorio(D, B)
2:     for b ∈ {1, ..., B} do
3:         Obtener D^(b) de D con el método bootstrap.
4:         Ajustar un árbol T_b en D^(b) del modo siguiente:
5:         for nodo in T_b do
6:             seleccionar m variables de las p posibles
7:             seleccionar la mejor variable y corte
8:             partir del nodo según el corte
       return {T_b}_{b=1}^B
```

Para realizar las predicciones finales usamos:

$$\varphi_{RF}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)\,,$$

o bien la clase con más votos en el caso de clasificación.

Lo aleatorio de los bosques aleatorios está en la selección de m variables. El número m es un parámetro en sí mismo, pero suele usarse $p/3$ para regresión y $\sqrt{p}$ para clasificación, donde p es el número de parámetros.

Tanto en el caso de bosques aleatorios como en *bagging* general, se dice que los estimadores no sobreajustan. Si bien esto es cierto en teoría, en la práctica es usual que muchas variables no sean útiles para la predicción. El efecto de la selección de variables hace que muchas veces se ignoren las pocas variables que si pueden explicar los datos. Es posible que muchos de los árboles tengan una expresividad innecesaria, al usar variables inútiles, y por ende resulten en una mala generalización.

6.3.4. Árboles extremadamente aleatorios

Una alternativa a los bosques aleatorios es usar un promedio de modelos pero aprendidos en el mismo conjunto de datos (i.e., sin el método bootstrap). La "extrema" aleatoriedad proviene de modificar el ajuste de arboles del siguiente modo:

- Usar un subconjunto aleatorio de variables (como en bosques aleatorios).
- Para cada variable del subconjunto generar cortes de modo aleatorio, en vez de seleccionar el mejor de todos. De aquellos cortes generados, se escoge el mejor de acuerdo al criterio empleado.

A este método se le denomina árboles extremadamente aleatorios (*ExtraTrees* en inglés). Aparte de la decorrelación de los estimadores, *ExtraTrees* tiene una mejor eficiencia computacional que los árboles clásicos (Geurts, Ernst, y Wehenkel, 2006).

6.4. Boosting

En las sección *bagging* aprendimos como mejorar conjuntamente una colección de estimadores, ajustados cada uno por separado. En tal configuración asumimos que los modelos son insesgados y nos enfocamos en reducir varianza. En esta sub-sección veremos el método *boosting*, que ajusta un conjunto de modelos sesgados de manera adaptativa para reducir el sesgo del estimador final.

6.4.1. Motivación: algoritmos fuertes versus débiles

Antes de adentrar en detalles, recordemos las preguntas teóricas cuyas respuestas decantaron en el método final. En el análisis matemático de modelos predictivos es usual trabajar en el marco conceptual PAC, que significa **p**robablemente **a**proximadamente **c**orrecto. En tal contexto, solemos seleccionar estimadores que con alta probabilidad tengan un bajo error de generalización (que sean aproximadamente correctos). En términos matemáticos, solemos tomar $\epsilon > 0$, $0 < \delta < 1$ de modo que nuestro algoritmo se dirá PAC-aprendible si logra tener un error de a lo más ϵ con probabilidad $1 - \delta$ (Schapire y Freund, 2012).

Por otro lado, podemos considerar un paradigma alternativo en el cual dado $\gamma > 0$ exijamos un modelo que tenga un error $\frac{1}{2} - \gamma$ con probabilidad menor a δ. Dicho de otro modo, en vez de pedir una alta probabilidad de precisión arbitraria, pedimos que la probabilidad de que nuestro algoritmo sea peor que 50 % sea baja. A un algoritmo que cumpla esto se le llamará débilmente PAC-aprendible (en contraste con PAC-aprendible, que también se le denomina fuertemente PAC-aprendible en este contexto).

La intuición nos dice que como la débil PAC-aprendibilidad es menos "exigente" que la fuerte, habrán más algoritmos débiles que algoritmos fuertes. Esto resulta ser falso. *boosting* en su sentido original es un algoritmo que al tomar un algoritmo débil es capaz de convertirlo en uno fuerte (en el sentido de PAC aprendibilidad). La existencia de algoritmos de *boosting* implica la equivalencia de la débil y fuerte PAC-aprendibilidad.

6.4.2. Algoritmo *AdaBoost*

Consideremos el caso en el que tenemos acceso a clasificadores de base que son débiles, en el sentido de ser ligeramente mejores que una elección por azar. Esta ligera mejora respecto de un modelo trivial es

la única exigencia que se le da a los clasificadores de base. Estos serán tratados como caja negra y serán llamados como sub-rutina para mejorar la predicción final.

La clave del éxito para los métodos boosting es escoger los conjuntos de entrenamientos para los estimadores de base de tal modo que estos sean "forzados" a inferir nueva información de los datos que no estaba presente anteriormente. Al escoger puntos de datos en los cuales el rendimiento de los algoritmos de base son incluso peores que su rendimiento débil usual, podemos ajustar modelos débiles en ellos para acercarnos al rendimiento fuerte deseado. Esto se ilustra en La Figura 36.

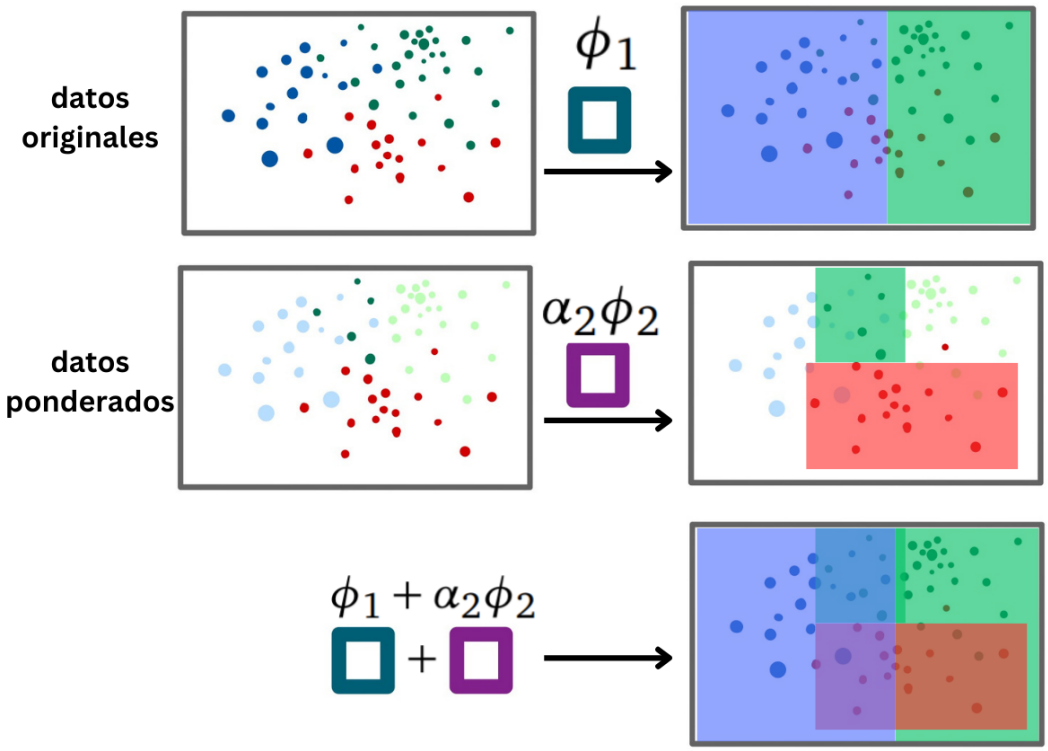

Fig. 36. Intuición del método *boosting*.

Supongamos que tenemos un conjunto de datos $D = \{(x_i, y_i)\}_{i=1}^N$, donde N será el tamaño del dataset. Consideraremos únicamente el caso de clasificación binaria donde las etiquetas están dadas por el conjunto -1,1. Supongamos que tenemos un modelo ϕ_m que consideramos débil. Usando la heurística anterior, queremos ajustar un nuevo modelo débil ϕ_{m+1}.

Usaremos la pérdida exponencial: $L(\phi) = e^{-y_i\phi(x_i)}$. Buscaremos minimizar esa pérdida, usando el estimador:

$$\phi(x) = \phi_m(x) + \alpha_{m+1}\phi_{m+1}(x)\,,$$

con lo cual, debemos resolver

$$\min_{\alpha_{m+1}} L(\phi(x)) = \min_{\alpha_{m+1}} \sum_{i=1}^{N} e^{-y_i(\phi_m(x_i)+\alpha_{m+1}\phi_{m+1}(x))} \tag{6.20}$$

$$= \min_{\alpha_{m+1}} \sum_{i=1}^{N} w_i^{(m)} e^{\alpha_{m+1}\phi_{m+1}(x)}\,, \tag{6.21}$$

donde definimos $w_i^{(m)} = e^{-y_i\phi_m(x_i)}$ para $i = 1, \ldots, N$, que lo podemos interpretar como el dataset original ponderado por el error (exponencial) del modelo ϕ_m.

Proposición 6.0.2. *El α_{m+1} que minimiza la función anterior está dado por*

$$\alpha_{m+1} = \frac{1}{2}\log(\frac{1-\epsilon_m}{\epsilon_m}), \tag{6.22}$$

donde $\epsilon_m = \dfrac{\sum_{y_i \neq \phi_m(x_i)} w_i}{\sum_{i=1}^N w_i}$.

El desarrollo anterior resulta en el algoritmo 4 (Freund y Schapire, 1997; Hastie, Tibshirani, y Friedman, 2001). Se demostró que puede convertir clasificadores débiles en un clasificador débil (para el caso binario).

Algoritmo 4 AdaBoost

function AdaBoost(D, M)
  Set $w_i = \frac{1}{N}\ \forall i = 1, \ldots, N$ (con N tamaño del dataset)
  for $m = 1, \ldots, M$ **do**
    Entrenar un modelo débil ϕ_m minimizando $\displaystyle\sum_{y_i \neq \phi_m(x_i)} w_i$
    Set $\epsilon_m = \dfrac{\sum_{y_i \neq \phi_m(x_i)} w_i}{\sum_{i=1}^N w_i}$
    Set $\alpha_m = \frac{1}{2} log(\frac{1-\epsilon_m}{\epsilon_m})$
    Set $w_i = w_i \cdot \exp(-y_i \phi_m(x_i)\alpha_m)$
    Actualizar $\{w_i\}_{i=1}^N$ de modo que $\sum_{i=1}^N w_i = 1$
  return $\phi(x) = signo\left(\sum_{m=1}^M \alpha_m \phi_m(x)\right)$

Recordando las nociones de débil y fuerte aprendibilidad, podemos enunciar el siguiente resultado:

Teorema 6.1 (Boosting)**.** *Una clase objetivo $\mathcal{C}$ es (eficientemente) debilmente aprendible, si y solo si es (eficientemente) fuertemente aprendible*

Es evidente que un modelo fuerte es en particular débil. Por el contrario, la conversa (débil PAC-aprendibilidad implica fuerte PAC-aprendibilidad) es resultado de aplicar el algoritmo AdaBoost a modelos de base débiles como caja negra. Esta equivalencia fue presentada por Yoav Freund y Robert Schapire, quienes ganaron el prestigioso premio Gödel gracias a esta contribución.

Si bien la demostración del teorema tiene elementos que escapan el enfoque de este curso, enunciamos a continuación un resultado que nos da una idea de la capacidad de AdaBoost para reducir el error (en el conjunto de entrenamiento) a medida que aumentamos la cantidad de estimadores.

Teorema 6.2. *Sea $\gamma_m = \frac{1}{2} - \epsilon_m$ y sea D_1 una distribución inicial arbitraria sobre el dataset de entrenamiento. El error de entrenamiento (ponderado por los pesos) del clasificador combinado con respecto a D_1 está acotado por:*

$$\mathbb{P}_{i \sim \mathcal{D}_1}(\phi(x_i) \neq y_i) \leq \prod_{m=1}^{M} \sqrt{1 - 4\gamma_m^2} \leq e^{-2\sum_{m=1}^M \gamma_m^2}. \tag{6.23}$$

Notemos que esta cota depende de γ_m. Recordemos que ϵ_m es la tasa de error. Mientras mas grande sea este, más pequeño será γ_m y por ende más cercano a 1 será $\sqrt{1-4\gamma_m^2}$ (pero de cualquier modo tenemos un número menor a uno). El teorema muestra un decrecimiento exponencial del error en el número de estimadores M. Esto garantiza bajo error incluso cuando el margen entre una elección azarosa (error $\frac{1}{2}$) y el error de los estimadores es pequeño.

6.4.3. Modelamiento aditivo por etapas

La justificación teórica que nos lleva al algoritmo de AdaBoost es en realidad un caso particular del método de modelamiento aditivo por etapas hacia adelante (*forward stagewise additive modelling* en inglés) (Hastie y cols., 2001). En efecto, este método toma un modelo de base (cuyos parámetros son a determinar) y genera iterativamente un modelo sumando el modelo de base ponderado por algún factor. Optimizamos tanto los parámetros del modelo como el factor en cada iteración. Esto es, partiendo de $f_0(x) = 0$, repetimos:

- Para $m = 1, \ldots, M$ resolver

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i, \gamma)) \,,$$

  donde b es nuestro modelo de base parametrizado por γ.

- Actualizar la sucesión de funciones

$$f_m(x) = f_{m-1}(x) + \beta b(x; \gamma_m) \,.$$

Está configuración nos permite usar una variedad de funciones de perdida distintas, así como también variadas familias de modelos. Una restricción obvia es que la combinación de estas sea fácilmente optimizable, pues la minimización a pasos puede ser difícil de computar.

Notemos que al usar el error cuadrático estamos minimizando (en cada paso m):

$$\sum_{i=1}^{N} (y_i - f_{m-1}(x_i) + \beta b(x_i, \gamma))^2 = \sum_{i=1}^{N} (r_{mi} - \beta b(x_i, \gamma))^2 \,.$$

donde hemos usado la notación $r_{mi} = y_i - f_{m-1}(x_i)$, que denota el residuo de la función f_{m-1} respecto al valor real y_i. Notemos como nuestro problema se reduce al ajuste de un modelo a los residuos del modelo en el paso anterior. Esto coincide con la noción de *boosting* inicial (aprender donde nos equivocamos).

6.4.4. *GradientBoosting*: boosting como descenso de gradiente funcional

Volviendo al caso general, nos podemos plantear la situación en la cual no busquemos minimizar con respecto a ponderadores y parámetros, si no más bien busquemos una función que minimice la perdida que estemos usando.

$$\hat{f} = \arg\min_{f} L(f) \,.$$

En esta versión funcional, podemos construir un algoritmo que refleje el caso general. Esta vez podemos apoyarnos en el uso del algoritmo de descenso de gradiente para acercarnos al minimizado del costo. Informalmente podemos escribir:

$$f_{n+1} = f_n - \rho_n \nabla L(f_n) \,.$$

para alguna sucesión de pasos $\{\rho_n\}_{n \geq 0}$ y algún f_0 conveniente. Notemos como tenemos una suma sucesiva de funcionales. Podemos conectar esta noción con el modelo aditivo paso por paso. Pero primero consideremos una aproximación numérica del problema. Podemos tomar el conjunto de evaluaciones de la función en el conjunto de datos como representación de este:

$$\tilde{f} = (f(x_0), \ldots, f(x_N)) \,.$$

Esto juega el rol de parámetros de nuestro modelo. La ventaja es que el gradiente también se vuelve un vector de N dimensiones:

$$\nabla L(\tilde{f})_i = \left(\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right) .$$

Es importante que nuestra función de costo sea diferenciable para aplicar este método. Cuando tenemos una forma cerrada para la derivada podemos simplemente evaluar en el punto de dato en cuestión.

Tenemos entonces una manera de actualizar las evaluaciones de nuestra función, sin embargo esto no nos da una expresión que podamos actualizar a un nuevo input arbitrario. Es por esto que usaremos estos N puntos para ajustar nuestro modelo débil. Para la función inicial, basta con ajustar un modelo débil con los datos originales como lo haríamos usualmente. El método se resume en el algoritmo 5 (Murphy, 2022).

Algoritmo 5 GradientBoosting

function GradientBoosting(D, M)
  Ajustar un modelo débil en los datos $\mathcal{D}$, i.e., fijar $\phi_0(x) = \arg\min_\phi \sum_{i=1}^N L(y_i, \phi(x_i))$
  for $m = 1, \ldots, M$ **do**
    Calcular

$$r_{im} = - \left[\frac{\partial L(y_i, \phi(x_i))}{\partial \phi(x_i)} \right]_{\phi(x_i)=\phi_{m-1}(x_i)}$$

    Entrenar un modelo débil ϕ_m minimizando $\sum_{i=1}^N (r_{im} - \phi(x_i))^2$
    Calcular $\rho_m = \arg\min_\rho \sum_{i=1}^n L(y_i, \phi_{m-1}(x_i) + \rho\phi_m(x_i))$
    Actualizar $\phi_m(x) = \phi_{m-1}(x) + \rho\phi_m(x_i)$
  return $\phi(x) = \phi_M(x)$

Si nuestra elección de pérdida es el error cuadrático, entonces basta notar que

$$r_i = \frac{\partial L(y_i, \phi(x_i))}{\partial \phi(x_i)} = y_i - \phi(x_i)$$

para obtener nuevamente el ajuste de nuestro modelo al residuo del modelo anterior. Esto es consistente con usar el modelamiento aditivo por etapas y se le denomina L2boosting, correspondiente a una de las muchas variantes de *boosting* (Hastie y cols., 2001).

Algoritmo 6 GradientTreeBoosting

function GradientTreeBoosting(D, M)
  Ajustar un árbol ϕ_0 en los datos $\mathcal{D}$.
  for $m = 1, \ldots, M$ **do**
    Calcular $r_{im} = - \left[\frac{\partial L(y_i, \phi(x_i))}{\partial \phi(x_i)} \right]_{\phi(x_i)=\phi_{m-1}(x_i)}$.
    Ajustar un árbol ϕ_m en $\{r_{im}\}_{i=1}^N$.
    Fijar R_{jm} como los conjuntos correspondientes a los nodos, enumerados de 1 hasta J_m.
    for $j \in 1, \ldots, J_m$ **do**
      $\gamma_{jm} = \arg\min_\gamma \sum_{x_j \in R_{jm}} L(y_i, \phi_{m-1}(x_i) + \gamma)$
    Calcular $\rho_m = \arg\min_\rho \sum_{i=1}^n L(y_i, \phi_{m-1}(x_i) + \rho\phi_m(x_i))$
    Actualizar $\phi_m(x) = \phi_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} \mathbf{1}_{R_{jm}}(x)$
  return $\phi(x) = \phi_M(x)$

Como aplicación particular de la idea de *Gradient boosting* observemos el algoritmo 6, llamado *Gradient tree boosting*. La principal diferencia con usar *Gradient boosting* con árboles como modelo de base es que ajustamos el factor multiplicativo en cada hoja del árbol y lo usamos para reemplazar derechamente la estimación para ese subconjunto en cuestión. Esto hace que se ponderen más el aporte de aquellos nodos importantes para mejorar la predicción del ensamblaje. Este modelo fue considerado por Leo Breinman como el mejor modelo *off-the-shelf* en el caso de clasificación, i.e., un modelo confiable para variadas tareas de clasificación, usualmente logrando muy buenas métricas.

6.4.5. Aspectos prácticos

Boosting es un método ampliamente utilizado y adaptable. Su distintas variantes han tenido variadas aplicaciones. Un ejemplo clásico es el algoritmo de detección de rostros de Viola-Davis (2001), que usa una variante de AdaBoost. A diferencia de métodos usuales que usan redes neuronales, la implementación de Viola-Davis tiene muchos menos parámetros y con rápida inferencia (muchas cámaras portátiles incorporan el algoritmo).

Un punto en contra de los modelos tipo *boosting* es el hecho de necesitar ajustar un estimador para realizar el paso siguiente. Esta naturaleza secuencial del aprendizaje limita su escalabilidad para problemas grandes. Gran parte del éxito de las redes neuronales hoy en día se debe a que buena parte de su entrenamiento se puede realizar de manera paralelizada en GPUs. Esto no es posible para modelos aditivos donde el aprendizaje se hace por etapas, pero han existido variantes de *boosting* que si paralelizan una parte del entrenamiento (como *XGBoost*).

6.4.6. Interpretabilidad de modelos basados en árboles

Otra desventaja del método es limitada capacidad de interpretación, sobretodo respecto al uso de árboles. Podemos recurrir a métodos para estimar la importancia relativa de las distintas variables. En el caso de un árbol podemos usar

$$I^2 = \frac{1}{M} \sum_{m=1}^{M} I_l^2(T_m) \tag{6.24}$$

para estimar la importancia de la variable l. Acá T_m es el árbol m-ésimo de la suma (por ejemplo usando *boosting* o *bagging*). A I_l^2 se le denomina relevancia cuadrática y está dada por:

$$I_l^2(T) = \sum_{t=1}^{J-1} \hat{i}_t^2 \, \mathbf{1}_{\{v(t)=l\}}^{(t)} \, . \tag{6.25}$$

Acá la suma se hace sobre los nodos internos de un árbol. En cada una de ellas, una de las variables $X_{v(t)}$ es usada para cortar el conjunto en dos regiones. Por otro lado, $\hat{i}_t^2$ denota la mejora estimada en términos del error cuadrático con respecto a no haber realizado el corte.

6.4.7. Más pérdidas y variaciones

Como vimos en secciones anteriores, es posible usar otras funciones de pérdida L tanto para modelamiento aditivo por etapas como para *Gradient boosting*. Por ejemplo, usar:

$$L(y, \phi) = |y - \phi(x)|$$

hace que debamos usar el signo de $y - \phi(x)$ como derivada en *Gradient boosting*. Esto significa que nos movemos en la dirección del valor real y.

También podemos considerar la pérdida *Logloss* o entropía cruzada para el caso clasificación binaria:

$$L(y, \phi) = \log(1 - e^{-y\phi(x)}) .$$

Esta pérdida tiene el mismo minimizador de población que la pérdida exponencial, con lo cual usar ambas es equivalente en el caso límite de tener un dataset infinito. Pese a esto, usar la *Logloss* tiene la ventaja de tener una interpretación probabilística usando:

$$p(y = 1|x) = \frac{1}{1 + e^{-2\phi(x)}} .$$

Además, los errores son penalizados severamente con la pérdida exponencial. Por el contrario la *Logloss* castiga linealmente los errores. Usarla deriva en el algoritmo *LogitBoost* (Hastie y cols., 2001).

Más variaciones pueden surgir del uso de estrategias para evitar el sobreajuste, entre las cuales encontramos:

- Detención temprana: una alternativa a fijar un número de estimadores M pequeño es tener un conjunto de validación que permita evaluar cuando es apropiado parar de agregar modelos.
- *Schrinkage*: ponderar cada actualización por algún factor pequeño.
- Podar estimadores, esto es, evaluar los elementos de la suma y eliminarlos si su error es alto.
- *Stochastic gradient boosting*: escoger de manera aleatoria un mini-batch con el cual entrenar cada modelo débil. Esto guarda similitud con el método de *bagging*.

Se han desarrollado además numerosas variantes que intentan corregir algunas desventajas de los métodos de *boosting*. Los ejemplos más usados son *CatBoot* (capaz de manejar variables categóricas), *Histogram-based Gradient Boosting* (simplifica las variables agrupándolas como histogramas para simplificar los cortes) y *Extreme Gradient Boosting* (construcción de árboles de forma paralela, aproximación de segundo orden del gradiente para acelerar cálculos y regularización), también conocido como *XGBoost*.

Referencias

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Phil. Trans. R. Soc.*, *53*, 370–418. Descargado de `https://doi.org/10.1214/13-STS438` doi: 10.1098/rstl.1763.0053

Bellhouse, D. R. (2004). The reverend thomas bayes, frs: A biography to celebrate the tercentenary of his birth. *Statistical Science*, *19*(1), 3–32.

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and Trends® in Machine Learning*, *2*(1), 1-127.

Bengio, Y. (2016). *What's yoshua bengio's opinion on max welling's position paper äre ml and statistics complementary?* Descargado de `https://www.quora.com/Whats-Yoshua-Bengios-opinion-on-Max-Wellings-position-paper-Are-ML-and-Statistics-Complementary`

Ben-Israel, A., y Greville, T. (2006). *Generalized inverses: Theory and applications*. Springer New York.

Boser, B. E., Guyon, I. M., y Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. En *Proceedings of the fifth annual workshop on computational learning theory* (pp. 144–).

Bostrom, N. (2014). *Superintelligence: Paths, dangers, strategies*. Oxford University Press.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*, 123–140.

Breiman, L. (2001). Random forests. *Machine learning*, *45*, 5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., y Stone, C. J. (1984). *Classification and regression trees*. Wadsworth & Brooks.

Davenport, T. H., y Patil, D. (2012). *Data scientist: The sexiest job of the 21st century.* Descargado de `https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century`

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., y Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. En *Cvpr09.*

Farley, B., y Clark, W. (1954). Simulation of self-organizing systems by digital computer. *IRE Transactions on Information Theory*, *4*(4), 76-84.

Fayyad, U., Piatetsky-Shapiro, G., y Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine*, *17*(3).

Ferguson, T. S. (1973). A bayesian analysis of some nonparametric problems. *Annals of Statistics*, *1*(2), 209– 230.

Freund, Y., y Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119–139.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybern.*, *36*, 193–202.

Gal, Y. (2015). *The science of deep learning.* Descargado de `YarinGals'swebsite:http://mlg.eng.cam.ac.uk/yarin/blog_5058.html`

Geurts, P., Ernst, D., y Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, *63*, 3–42.

Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, *521*, 452–459.

Harari, Y. N. (2015). *Sapiens: A brief history of humankind*. Harper.

Hastie, T., Tibshirani, R., y Friedman, J. (2001). *The elements of statistical learning*. Springer.

Hjort, N., Holmes, C., Müller, P., y Walker, S. (2010). *Bayesian nonparametrics*. Cambridge University Press.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the USA*, *79*, 2554–2558.

James, G., Witten, D., Hastie, T., y Tibshirani, R. (2014). *An introduction to statistical learning: With applications in r*. Springer.

Jordan, M. I., Ghahramani, Z., Jaakkola, T., y Saul, L. (1999). An introduction to variational methods for graphical models. *Machine Learning*, *37*, 183-233.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., y Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541-551.

Lighthill, J. (1973). Artificial intelligence: A general survey. *Artificial Intelligence: a paper symposium.*

Minsky, M. (1952). *A neural-analogue calculator based upon a probability model of reinforcement* (Inf. Téc.). Boston, MA: Harvard University Psychological Laboratories.

Minsky, M., y Papert, S. (1969). *Perceptrons: an introduction to computational geometry*. MIT.

Murphy, K. P. (2022). *Probabilistic machine learning: An introduction*. MIT Press. Descargado de `probml.ai`

Neal, R. M. (1993). *Probabilistic inference using markov chain monte carlo methods* (Inf. Téc.). Toronto, Canada: University of Toronto, Department of Computer Science.

Pierce, G. (1949). *The song of insects*. Harvard College Press.

Rasmussen, C. E., and Williams, C. K. (2006). *Gaussian processes for machine learning*. The MIT Press.

Ripley, B. D. (2008). Pattern recognition and neural networks. En (cap. 7: Tree-structured Classifiers).

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *23*(6008), 533-536.

Russell, S. J., y Norvig, P. (2009). *Artificial intelligence: A modern approach* (3.ª ed.). Pearson Education.

Salakhutdinov, G. E. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*, 504-507.

Schapire, R. E., y Freund, Y. (2012). *Boosting: Foundations and algorithms. adaptive computation and machine learning*. Mit Press London.

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine*, *41*(324).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–503. Descargado de `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`

Stigler, S. M. (2013, 08). The true title of bayes's essay. *Statist. Sci.*, *28*(3), 283–288. Descargado de `https://doi.org/10.1214/13-STS438` doi: 10.1214/13-STS438

Tibshirani, R. (1996). Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, *58*(1), 267-288.

Tikhonov, A. N., y Arsenin, V. Y. (1977). *Solution of ill-posed problems*. Washington: Winston & Sons.

Turing, A. (1950). Computing intelligence and machinery. *Mind*, *59*(236), 433-460.

Vapnik, V., and Chervonenkis, A. . (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, *16*, 264-280.

Welling, M. (2015). Are ml and statistics complementary? *Roundtable discussion at the 6th IMSISBA meeting on " Data Science in the next 50 years"*.

Werbos, P. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences* (Tesis Doctoral no publicada). Harvard University.