Taller de Aprendizaje Automático

Ensambles de predictores

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

1. ¿Cómo y por qué combinar predictores?
2. Predictores base. Árboles de Decisión
3. Estrategias de combinación de predictores
4. Bagging
5. Random Forests
6. Boosting. Adaboost y Gradient Boosting

Aprendizaje de ensambles: motivación

- Toma de decisiones difíciles: es común que tengamos en cuenta la opinión de varios expertos para mejorar la decisión.
- Cada modelo se puede ver como un experto y es esperable que si los combinamos obtendremos predicciones más confiables.
- Métodos de ensambles: permiten entrenar y combinar conjuntos de predictores.
- *No free lunch theorem:* no existe un predictor universalmente mejor que otro para cualquier conjunto de datos.
- Como a priori no hay un mejor predictor para un problema dado $\Rightarrow$ se prueban varios.
- Es más robusto combinar los predictores individuales siempre y cuando:
  1. Tengan un buen desempeño: basta que acierten más del 50%.
  2. Sean diversos (independientes): cometan diferentes errores.

Aprendizaje de ensambles: cómo funciona

La combinación de predictores consta de dos tareas:

1. Entrenar un conjunto de predictores base a partir de los datos.
2. Aplicar una estrategia para combinarlos en un único predictor.

Ejemplo 1:

- Entrenar varios predictores.
- Nueva muestra: cada modelo produce su predicción.
- Se clasifica según el voto de la mayoría.

Ejemplo 2:

- Supongamos que cada clasificador puede cuantificar la confianza en su predicción. (e.g: si en 10-NN, 9 vecinos de $\mathbf{x}$ son de la clase $\mathcal{C}_k$, confianza alta).
- Confianza del clasificador y_m en clasificar $\mathbf{x}$: $C(y_m|\mathbf{x}) = \max_{j=1,\ldots,K} \hat{P}(\mathcal{C}_k|\mathbf{x}, y_m)$.
- Se clasifica según mayoría ponderada por $C(y_m|\mathbf{x})$.

Características deseables del conjunto de predictores base

- **Diversidad:**
  - predictores especializados en distintas características
  - predictores entrenados sobre distintos datos: subconjuntos al azar
  - distintos tipos de predictor: lineal, *k-NN*, redes, SVM, ...
  - distintos parámetros: distintos k en *k-NN*, ...
- **Independencia:**
  - máxima reducción de error
  - máxima reducción de varianza
- **Complementariedad:**
  - variedad de especialización

Un predictor bien adaptado para ensambles: Árboles de decisión

- Conjunto de entrenamiento:

  $\{(\mathbf{x}_n, t_n), \quad n = 1, \ldots, N\}$, con:
  - $\mathbf{x}_n = (x_n^1, x_n^2, \ldots, x_n^D) \in \mathcal{R}$ (espacio de características de D atributos),
  - t_n la etiqueta de clase (clasificación) o el valor correspondiente (regresión).

- Dado el conjunto de entrenamiento, entrenar un árbol de decisión consiste en:
  - Encontrar una partición del espacio de características en hiper-rectángulos $\{R_1, R_2, \ldots, R_L\}$, $R_i \cap R_j = \emptyset$, $\cup_{l=1}^{L} R_l = \mathcal{R}$.
  - Esta partición será óptima en algún sentido a definir.

Árboles de decisión

Una vez entrenado el árbol, para cada hiper-rectángulo R_l se consideran las muestras de entrenamiento que contiene. Luego:

- En Clasificación: a cada R_l se le asigna la etiqueta mayoritaria entre las muestras de entrenamiento que contiene.
- En regresión: a cada R_l se le asigna el promedio de los valores de las muestras de entrenamiento que contiene.

Inferencia: a la muestra de test $\mathbf{x}$ se le asigna la etiqueta (clasificación) o valor (regresión) del hiper-rectángulo en el que cae.

Ejemplo: árbol de clasificación mediante decisiones binarias recursivas

- Entrenamos un árbol de decisión sobre la base Iris, de profundidad máxima `max_depth = 2`, usando la clase `DecisionTreeClassfier`,
- `DecisionTreeClassfier` implementa el algoritmo CART*†, que produce particiones binarias.
- Es posible visualizar el árbol resultante mediante el método `export_graphviz()`

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

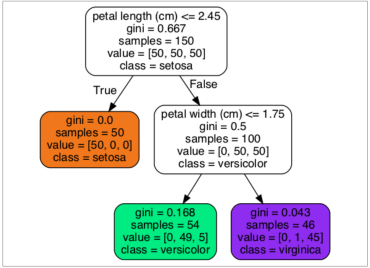
```python
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

*L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984

†Otro algoritmo de árbol muy popular es ID3, con sus implementaciones C4.5 y C5.0.

Ejemplo: árbol de clasificación mediante decisiones binarias recursivas

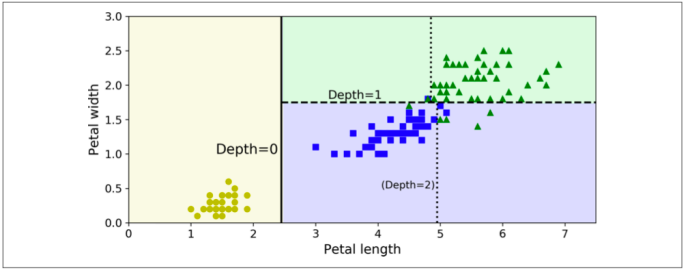
Inferencia: dada una muestra $\mathbf{x} = (x^1, x^2, \ldots, x^D)$, se recorre el árbol desde la raíz, testeando en cada nodo la condición sobre el atributo x^d correspondiente. Al llegar a la hoja o nodo terminal, queda determinada la predicción para $\mathbf{x}$ (cada hoja es un hiper-rectángulo R_l).



Campos de los nodos:

- `samples`. Cantidad de muestras de entrenamiento en las hojas debajo del nodo.
- `value`. Distribución de esas muestras según la clase: [setosa, versicolor, virginica].
- `class`. Clase asignada al nodo (la de la muestra de entrenamiento mayoritaria).
- `Gini`: índice de Gini (heterogeneidad del nodo).

Ejemplo: árbol de clasificación mediante decisiones binarias recursivas



Medidas de impureza de nodo: índice de Gini y entropía

Índice de Gini

- Medida de impureza o heterogeneidad de un nodo. Para el nodo i, se define como

$$G_i = \sum_k^K p_{i,k}(1 - p_{i,k}) = 1 - \sum_k {p_{i,k}}^2$$

  donde $p_{i,k}$ es la proporción de muestras de la clase k en el nodo i.
- Si el nodo i es "puro"(todas las muestras son de la misma clase), $G_i = 0$.
- G_i es máximo cuando la distribución de las muestras en el nodo es uniforme en las clases.

Entropía

Alternativamente se puede usar la entropía $H_i = -\sum_{k=1,\, p_{i,k}\neq 0}^K p_{i,k} \log_2 p_{i,k}$ con la opción `criterion=entropy`. Suele conducir a resultados similares.

Entrenamiento: construcción del árbol de clasificación con CART

- CART particiona recursivamente los nodos, buscando el par (d, s_d) (d el atributo y s_d un umbral) que minimiza el costo

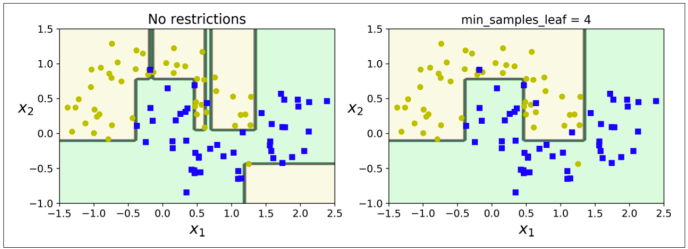
$$J(d, s_d) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}},$$

  con m la cantidad de muestras en el nodo, $m_{\text{left/right}}$ la cantidad de muestras en el nodo hijo izquierdo/derecho, y $G_{\text{left/right}}$ el índice de Gini del nodo hijo izquierdo/derecho.

- Deja de dividir cada nodo cuando la impureza no se reduce o se alcanza `max_depth`.

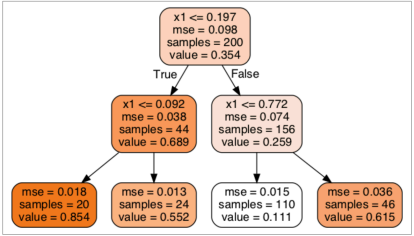
Hiper-parámetros de regularización

- Los árboles son *no paramétricos:* no asumen ningún modelo paramétrico de distribución.
- Tienden a sobre ajustar a los datos, por lo que se requiere aplicar regularización.
- Hiper-parámetros de regularización en `DecisionTreeClassifier`: máxima profundidad (`max_depth`), cantidad mínima de muestras en un nodo para autorizar dividirlo (`min_samples_split`), cantidad mínima de muestras en una hoja (`min_samples_leaf`), cantidad máxima de hojas (`max_leaf_nodes`).



Árboles de regresión

- `DecisionTreeRegressor` implementa árboles de decisión para regresión con CART.
- Ejemplo: ajuste de parábola con ruido, árbol de regresión de profundidad `max_depth=2`.

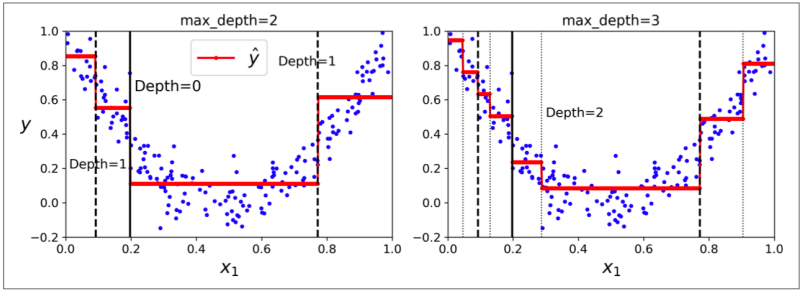


Campos de los nodos:

- `samples`. Cantidad de muestras de entrenamiento en las hojas debajo del nodo.
- `value`. Valor promedio de los valores de las `samples` muestras de entrenamiento en el nodo.
- `mse`. Error cuadrático medio de las `samples` muestras de entrenamiento en el nodo (medida típica de dispersión usada para regresión).

Árboles de regresión

Resultado del ajuste de la parábola ruidosa para `max_depth=2` y `max_depth=3`



Entrenamiento: construcción del árbol de regresión

Igual que para árboles de clasificación, utilizando el MSE en lugar de índice de Gini o entropía.

- Para cada nodo, con m_{nodo} muestras de entrenamiento, se calcula su MSE:

$$\text{MSE}_{\text{nodo}} = \sum_{i \in \text{nodo}} \|\mathbf{x}_i - \bar{\mathbf{x}}_{\text{nodo}}\|^2, \quad \text{con} \quad \bar{\mathbf{x}}_{\text{nodo}} = \frac{1}{m_{\text{nodo}}} \sum_{i \in \text{nodo}} \mathbf{x}_i.$$

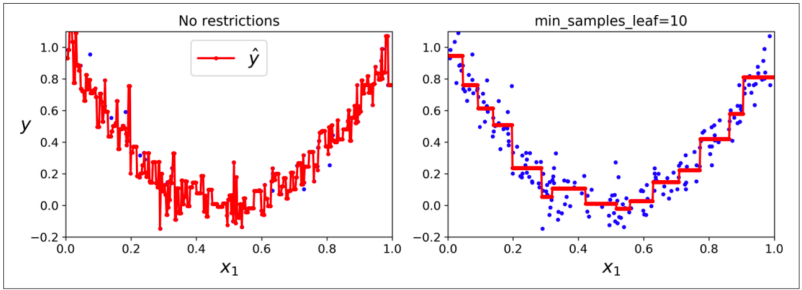
- Se busca la característica d y el umbral t_d que minimiza el costo

$$J(d, s_d) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}},$$

  con m la cantidad de muestras en el nodo, $m_{\text{left/right}}$ la cantidad de muestras en el nodo hijo izquierdo/derecho, y $\text{MSE}_{\text{left/right}}$ el MSE del nodo hijo izquierdo/derecho.

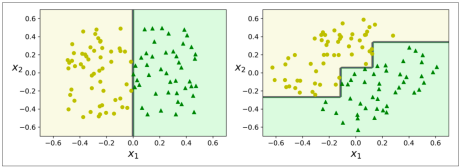
Sobre-ajuste y regularización

- Sin restricciones se sobre ajusta a los datos de entrenamiento.
- Hiper-parámetros de regularización similares a los de clasificación.

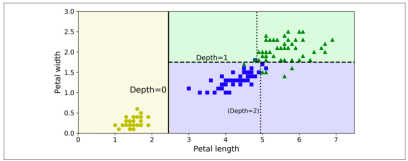


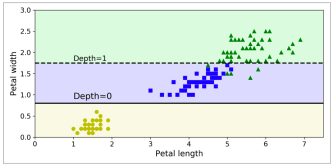
Árboles de decisión: consideraciones finales

- Invarianza frente a transformaciones en los datos
  - No son sensibles al escalado de los datos, pero si a las rotaciones.



- Inestabilidad de la solución resultante
  - Son muy sensibles a pequeñas variaciones en los datos.
    Ejemplo: si se quita la muestra de mayor ancho de *Iris versicolor*, se obtiene el árbol de la derecha.





Estrategias de combinación de predictores

- Entrenamiento en paralelo, sobre un mismo conjunto de entrenamiento
  - Mezcla de expertos:
    distintos tipos de predictor (SVM, k-NN, etc.) entrenados sobre los mismos datos
- Entrenamiento en paralelo, con distintos conjuntos de entrenamiento
  - Bagging (Boostrap Aggregating):
    distintos modelos del mismo tipo entrenados sobre distintos muestreos de los datos
  - Random forest:
    combina bagging sobre árboles de decisión con selección aleatoria de atributos
- Entrenamiento secuencial o en cascada
  - Boosting: al pasar por la cascada de predictores, a las muestras difíciles de clasificar se les va aumentando el peso en el costo total, para que los predictores cuesta arriba se focalicen en su predicción. La decisión es un voto ponderado por el desempeño de cada predictor.

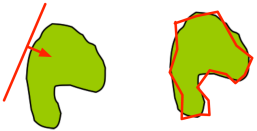
Compromiso sesgo-varianza

Error cuadrático medio $= (\text{Sesgo})^2 + \text{Varianza} + \text{Ruido}$

- Ruido: independiente del predictor; error irreducible, intrínseco al problema.
- Varianza: sensibilidad del predictor ante fluctuaciones en el conjunto de entrenamiento.
- Sesgo: error medio de predicción (sobre todos los conjuntos de datos) del predictor.

Reducción de varianza: si los conjuntos de entrenamiento son independientes, la combinación reduce la varianza sin afectar el sesgo (e.g. bagging).

Reducción del sesgo: la combinación o el promedio de modelos simples tiene mayor capacidad que cada modelo por separado (e.g. boosting, mezcla de expertos).



Sesgo: un clasificador lineal vs. mezcla de clasificadores lineales

¿Por qué funciona?

M clasificadores independientes, todos con probabilidad de error p:

$$P(m \text{ errores}) = \binom{M}{m} p^m (1-p)^{M-m}.$$

Votación por mayoría:

$$P(\text{clasificación incorrecta}) = \sum_{m=\lceil M/2 \rceil}^{M} \binom{M}{m} p^m (1-p)^{M-m}.$$

	$p = 0.3$
M	P(clasificación incorrecta)
11	0.078225
21	0.026390
121	0.000002

	$p = 0.49$
M	P(clasificación incorrecta)
11	0.472948
121	0.412750
10001	0.022731

¿Por qué funciona?

Efecto de la especialización

Supongamos:

- Muestras etiquetadas $(\mathbf{x}_n, t_n)$, $n = 1, \ldots, N$.
- $y_m(\mathbf{x})$ clasificadores entrenados, $m = 1, \ldots, M$.
- $\{R_1, R_2, \ldots, R_L\}$ partición del espacio de características $\mathcal{R}$.
- $y^*(\mathbf{x})$ clasificador con menor tasa de error global.
- En R_j elijo $y_{i(j)}(\mathbf{x})$ tal que $\forall\, m = 1, \ldots, M$,

$$P(y_{i(j)}(\mathbf{x}_n) = t_n | \mathbf{x}_n \in R_j) \geq P(y^*(\mathbf{x}_n) = t_n | \mathbf{x}_n \in R_j).$$

Entonces,

$$P(\text{clasificación correcta}) = \sum_{j=1}^{L} P(\mathbf{x}_n \in R_j) P(y_{i(j)}(\mathbf{x}_n) = t_n | \mathbf{x}_n \in R_j) \geq P(y^*(\mathbf{x}_n) = t_n, \mathbf{x}_n \in \mathcal{R}).$$

Predictores Bagging

Bagging: Bootstrap Aggregating*

- M subconjuntos de entrenamiento: muestreo con reposición (bootstrap)† de los datos.
- M predictores: cada uno entrenado con uno de los subconjuntos de entrenamiento.
- Las predicciones se combinan por mayoría (eventualmente de forma ponderada).

Idealmente: subconjuntos independientes, muestreados de los datos.

En la práctica: un solo conjunto de entrenamiento

$$\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \to \text{bootstrap} \to \mathcal{S}_m = \{\mathbf{x}_1^{(m)}, \ldots, \mathbf{x}_{N_m}^{(m)}\}.$$

Predictor base inestable (sesgo pequeño, varianza grande), e.g. árboles de decisión.

*L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, p. 123–140, Aug. 1996

†B. Efron, "Bootstrap methods: Another look at the jackknife," *The Annals of Statistics*, vol. 7, pp. 1–26, Jan. 1979

Bagging en Scikit-Learn

Entrenar 500 árboles de decisión sobre 100 instancias tomadas al azar

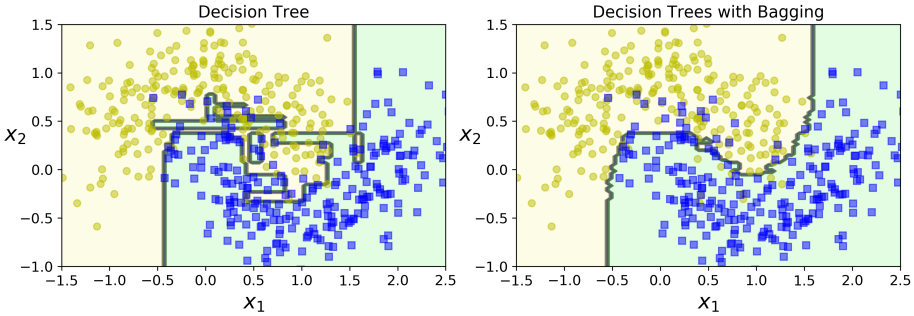
- con reposición (`bootstrap=True`): Bagging
- sin reposición (`bootstrap=False`): Pasting

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Por defecto, `BaggingClassifier` hace soft voting (voto ponderado) cuando el clasificador base puede estimar la probabilidad de clase.

Bagging en Scikit-Learn

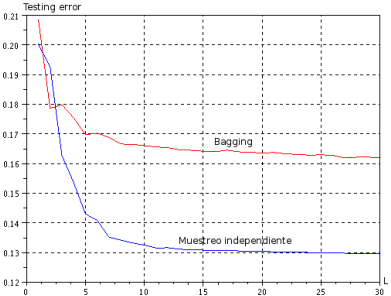
Árbol de decisión vs. ensamble bagging con 500 árboles



Predictores Bagging

¿Por qué funciona?

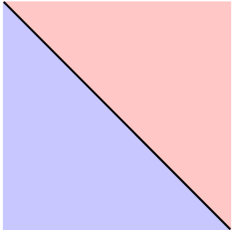
- Con M predictores independientes con probabilidad de error p, el voto por mayoría mejora.
- Bagging busca generar predictores independientes usando bootstrap en el conjunto $\mathcal{D}$.
- En realidad son pseudo-independientes pues se toman muestras del mismo conjunto $\mathcal{D}$.
- Implica un deterioro en el desempeño, pero el muestreo independiente limitaría el M.



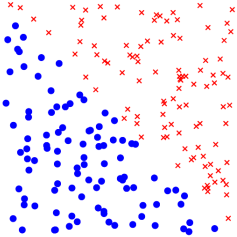
Evolución del error con M.

Predictores Bagging

Ejemplo: dos clases, los y_m árboles de decisión



Frontera verdadera

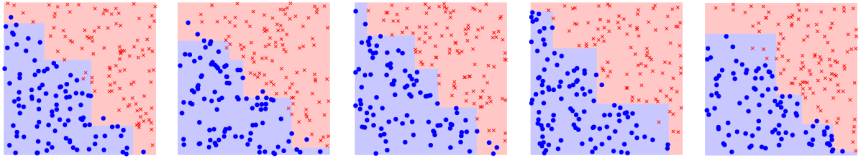


Un muestreo $\mathcal{S}_m$

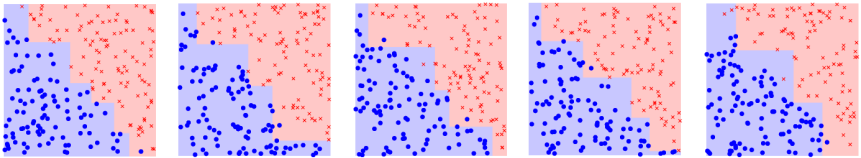
Árboles de decisión:

- Sesgo bajo: en promedio, frontera de decisión cercana a las verdadera.
- Varianza grande: frontera de decisión muy sensible a la muestra específica.

Predictores Bagging



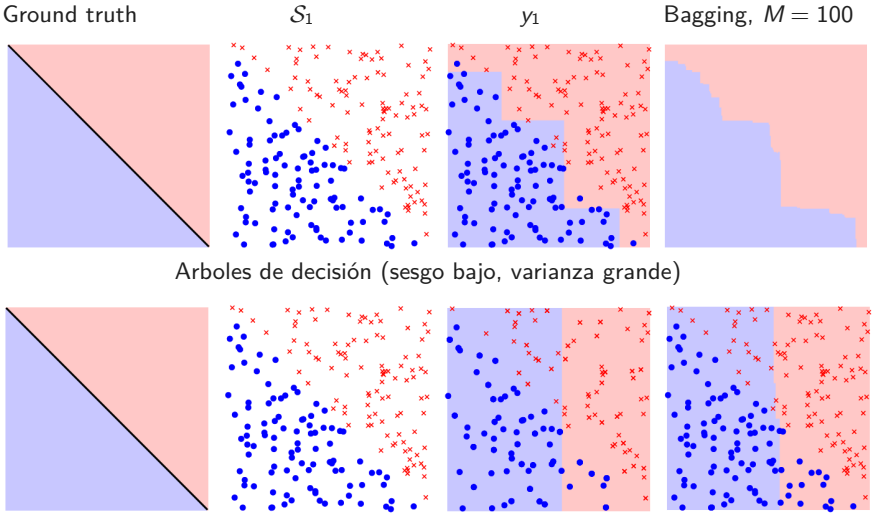
$\mathcal{S}_1$ $\mathcal{S}_2$ $\mathcal{S}_3$ $\mathcal{S}_4$ $\mathcal{S}_5$



$\mathcal{S}_6$ $\mathcal{S}_7$ $\mathcal{S}_8$ $\mathcal{S}_9$ $\mathcal{S}_{10}$

Predictores Bagging

Predictores Bagging



Random Forests

Sobre los predictores base, vimos que:

- Conviene combinar predictores débiles (sesgo bajo, varianza alta).
- Mayor independencia entre predictores base, menor error de predicción en la combinación.

Random Forests*

- Bagging con árboles de decisión, modificado para reducir la correlación entre los árboles.
- Construyendo cada árbol sobre un subconjunto de características seleccionadas al azar.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

*L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, p. 5–32, Oct. 2001

Random Forests: ¿por qué funciona?

Con boostrap genero árboles de decisión idénticamente distribuidos (sesgo b y varianza σ^2).

- El sesgo del promedio también vale $b_{RF} = b$;
- La varianza del promedio de B árboles vale

$$\sigma^2_{RF} = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

  con ρ el coeficiente de correlación de Pearson de un par de árboles.
- Al aumentar B el segundo término se reduce, pero no el primero.

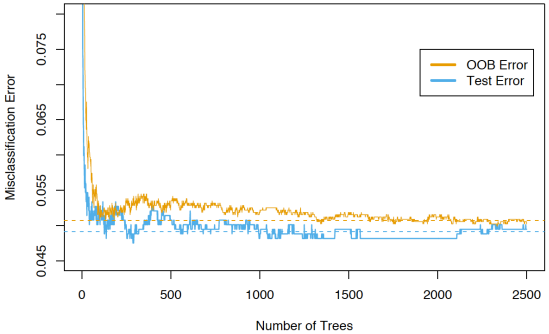
Para reducir la varianza de la combinación, Random Forests propone:

- Diseñar el conjunto de árboles para reducir ρ, cuidando no aumentar mucho σ^2;
- Construir cada árbol con un subconjunto aleatorio de d características entre el total de D.

Estimación Out-of-bag (OOB)

El error de predicción se estima promediando los errores de predicción de las muestras OOB.

- Para cada muestra de entrenamiento $(\mathbf{x}_n, t_n)$, se construye su predictor RF promediando sólo los árboles de los muestreos bootstrap que no la contienen.




```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

Boosting

- Diferencia principal con métodos como bagging: los predictores base se entrenan secuencialmente o en cascada.
- Cada predictor se entrena condicionalmente a la performance de los predictores ya entrenados: fuerza a focalizarse en las muestras difíciles.

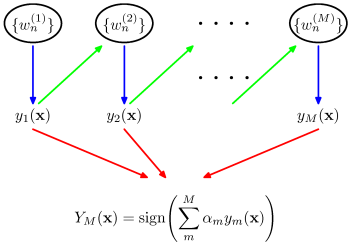
El algoritmo de boosting más popular es Adaboost.*

- Cada predictor base se entrena usando una forma ponderada de las muestras: los pesos dependen de la performance de los predictores previos en clasificar las muestras.
- A la muestra mal predicha por un predictor base se le asignan un peso mayor a la hora de entrenar el predictor base siguiente.
- Una vez entrenados todos los predictores de la cascada, sus predicciones se combinan mediante mayoría ponderada.

*Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997

Adaboost

- Cada $y_m(\mathbf{x})$ se entrena con una versión ponderada de los datos, $\left\{w_n^{(m)}, n = 1, \ldots, N\right\}$.
- Los pesos $w_n^{(m)}$ dependen de la performance de $y_{m-1}(\mathbf{x})$ (y de los anteriores) al clasificar a $\mathbf{x}_n$, crecen para los datos mal clasificados y permanecen fijos para los bien clasificados.
- El meta-predictor es el promedio ponderado de los y_m: $Y_M(\mathbf{x}) = \mathrm{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right)$.
- Los α_m son el logaritmo del cociente: tasa de acierto / tasa de error.




```python
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

Adaboost: ejemplo

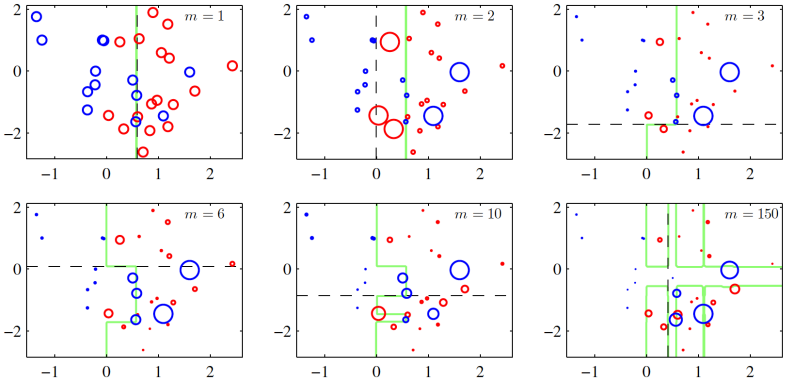


Figure 14.2 Illustration of boosting in which the base learners consist of simple thresholds applied to one or other of the axes. Each figure shows the number m of base learners trained so far, along with the decision boundary of the most recent base learner (dashed black line) and the combined decision boundary of the ensemble (solid green line). Each data point is depicted by a circle whose radius indicates the weight assigned to that data point when training the most recently added base learner. Thus, for instance, we see that points that are misclassified by the $m = 1$ base learner are given greater weight when training the $m = 2$ base learner.

Gradient Boosting

Gradient boosting* nace de la siguiente observación (más intuitiva en regresión):

- Consideremos un predictor $y = f(\mathbf{x})$, y la función de costo cuadrática,

$$L(f) = \sum_{n=1}^{N} L(y_n, f(\mathbf{x}_n)) = \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n))^2 .$$

- Supongamos que queremos optimizar f iterativamente, y llamemos $f^{(m)}$ el modelo en la iteración m. Para esto se busca un nuevo estimador h que refine el predictor:

$$f^{(m+1)}(\mathbf{x}) = f^{(m)}(\mathbf{x}) + h(\mathbf{x}).$$

- Se busca que h se ajuste lo mejor posible al residuo $y - f^{(m)}(\mathbf{x})$.
- Esto se puede implementar como una cascada de predictores, donde el predictor f_m ajusta el error de aproximación obtenido por el predictor f_{m-1}.

*J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001

Gradient Boosting en Scikit-Learn

1. Entrenamos un primer árbol de decisión:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

2. Entrenamos el segundo árbol sobre los residuos del primero:

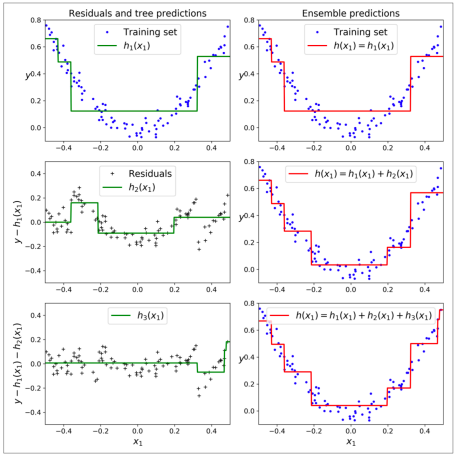
```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

3. Entrenamos el tercer árbol sobre los residuos del segundo:

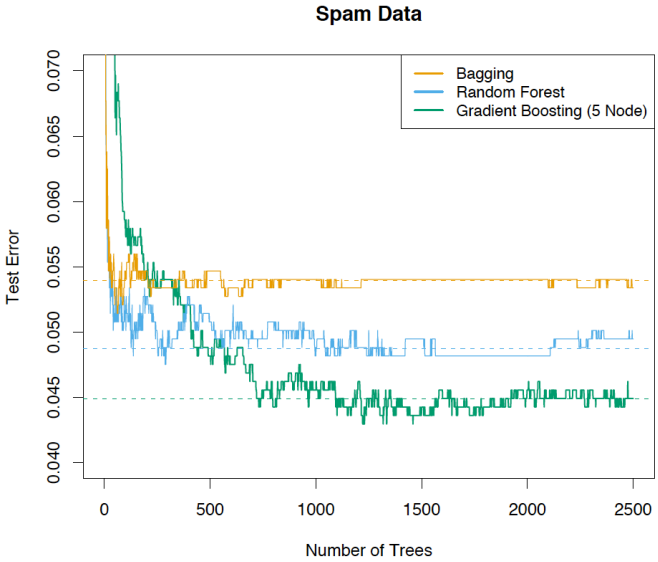
```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Inferencia sobre el ensamble:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```



Comparación: Bagging vs. RF vs. GB



Extreme Gradient Boosting (XGBoost)*

Librería Python de referencia†, que implementa una versión optimizada de Gradient Boosting

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

Ofrece varias funcionalidades útiles, como una implementación automática de early stopping:

```
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

† https://xgboost.readthedocs.io

* T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016

Referencias

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, p. 123–140, Aug. 1996.

B. Efron, "Bootstrap methods: Another look at the jackknife," *The Annals of Statistics*, vol. 7, pp. 1–26, Jan. 1979.

L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, p. 5–32, Oct. 2001.

Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*. O'Reilly Media, Inc., 2022.

J. H. Friedman, "Stochastic gradient boosting," *Comput. Stat. Data Anal.*, vol. 38, p. 367–378, Feb. 2002.

T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.