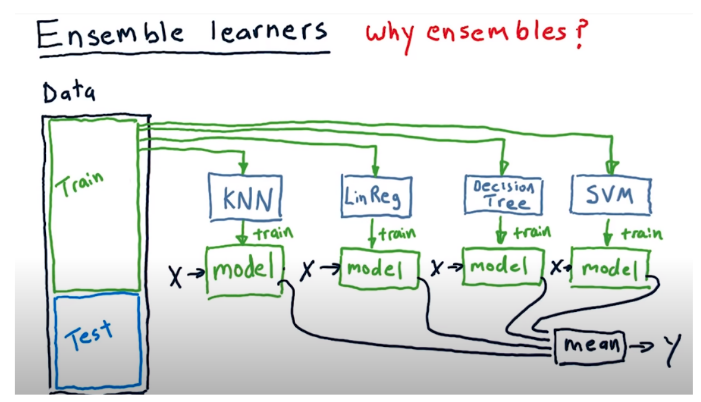
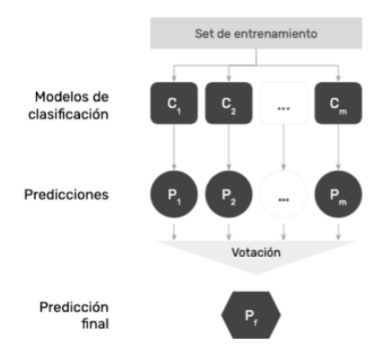
**Modelos de Ensamble**

Consiste en, para un conjunto de datos de entrenamiento, aplicarle diferentes algoritmos de predicción y obtener un resultado que surja de la acción conjunta de los mismos. El resultado generalmente es más robusto que el resultado que se obtendría a partir de cada modelo por separado, aún estando lo más optimizado posible.



Se entrena a muchos modelos y a partir de sus resultados se vota el resultado final, por mayoría:



Conviene que los modelos sean distintos, para que agreguen información nueva a la votación. Por lo tanto, conviene trabajar con modelos poco correlacionados. La diferencia de estos modelos se puede dar ya sea por una diferencia en la **población de datos**, o bien por una **técnica de modelado** utilizada diferente. Incluso puede ser también el mismo modelo, pero aplicándole hiperparámetros distintos.

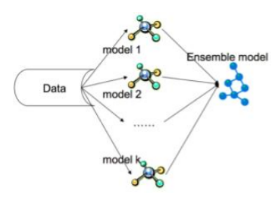
**Ensambles:** Los **modelos de ensamble** en **aprendizaje automático** **combinan las decisiones** de **varios modelos** para **mejorar el rendimiento general**.

Con los modelos de ensamble se **minimizan las principales causas de error** en los modelos de aprendizaje: **ruido, sesgo** y **varianza**.

Están diseñados para **mejorar la estabilidad y precisión** de los algoritmos de aprendizaje automático. La **combinación de sus resultados** suele ser **mejor en términos de precisión** que el uso de **un** **solo modelo aislado**.

**Estrategia de “Divide and Conquer”** para **mejorar la Performance**.

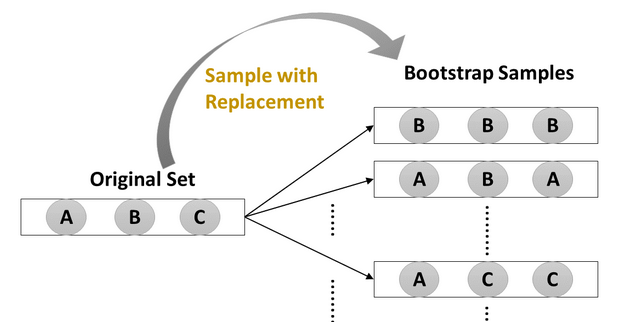
La combinación de **varios modelos base** busca **ampliar el espacio de hipótesis** posibles para representar los datos. De esta manera, podemos **mejorar la precisión predictiva** del modelo combinado resultante:



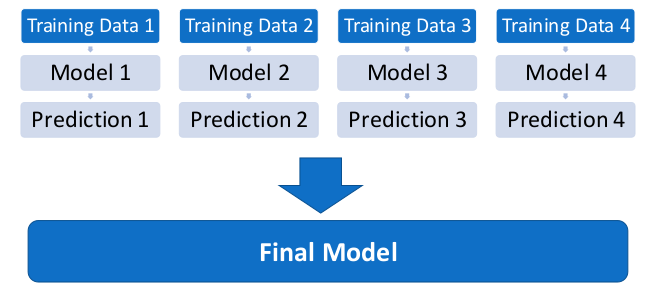
Hay  **dos familias de Métodos de Ensamble:**

1. **Métodos de Averaging** (basado en promedios): Se construyen varios estimadores de forma independiente y luego se promedian sus predicciones (sobre todo en modelos de Regresión; en modelos de clasificación se vota por mayoría). IE: Métodos de **Bagging** y su implementación particular, **Random Forest**.

**Bagging = Bootstrap Aggregation:** Genera a partir de un dataset original, N nuevos datasets con la misma cantidad de variables independientes, pero tomando **muestras con repetición**. De forma tal que podemos tener observaciones repetidas en estos datasets:

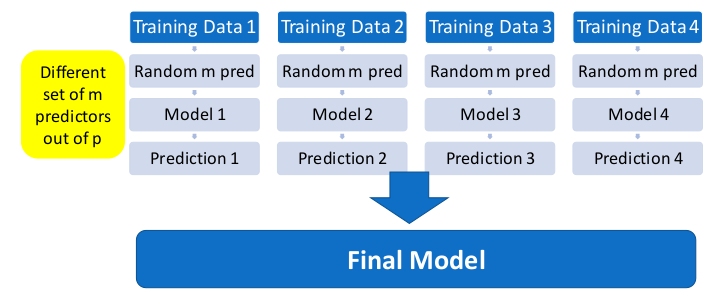


De esta manera, podemos entrenar N modelos distintos a partir de estos N datasets generados con bootstrapping:



Si se usan **árboles de decisión, estos no se podan**. De esta forma no aumentamos el sesgo, y esperamos que el modelo de ensamble reduzca la varianza de S2 a S2/N.

**Random Forest:** Como lo que más **nos conviene** es que los **modelos** usados en el modelo de ensamble sirvan para escenarios bien distintos, es decir, estén lo más **descorrelacionados** posible, una forma de lograr esto se puede usar la misma **técnica de bagging**, pero **aplicando muestreos sin repetición** con **algunas variables independientes en vez de** usar **todas**. De esta manera, se van a generar N datasets con un muestreo de M variables independientes de un total de P variables independientes; que servirán para entrenar N modelos.



De esta manera, Bagging en realidad termina siendo un caso particular de Random Forest en el cual el tamaño de las muestras de los predictores M coincide con el tamaño de la población de los predictores P.

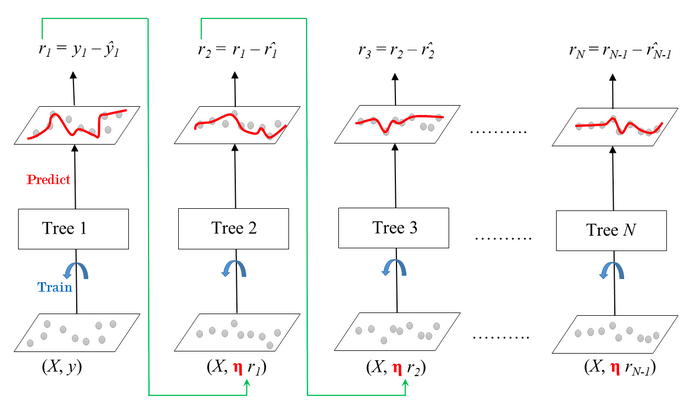
Empíricamente, se descubrió que con estos valores se suelen generar modelos descorrelacionados:

* Para **Regresiones:** M = P/3
* Para **Clasificaciones:** M = sqrt(P)

1. **Métodos de Boosting**: Los estimadores base se construyen secuencialmente. Se busca reducir el sesgo del estimador combinado, centrándose en los casos con peor performance. A partir de varios modelos débiles se produce un ensamble potente. IE: **AdaBoost, Gradient Tree Boosting**.

A diferencia de con Bagging o Random Forest, en Boosting los modelos se entrenan de forma secuencial de forma tal que cada modelo vaya aprendiendo a partir de los errores de su predecesor:

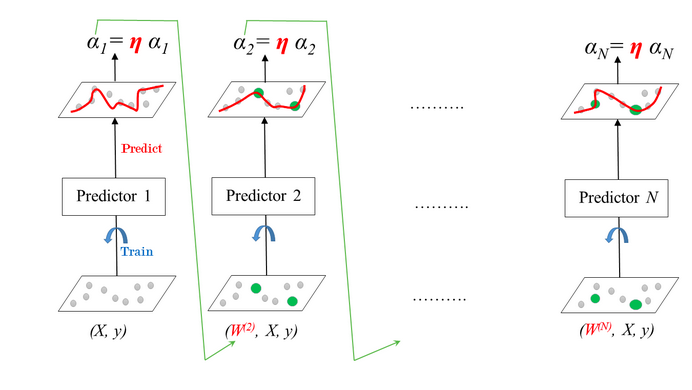
1. **Gradient Boost:** Se trata de un método de aprendizaje lento donde los sucesivos modelos de árboles de decisión se entrenan para predecir los residuales del árbol antecesor. Tiene un hiperparámetro de Learning Rate η que es un escalar mayor que 0 y menor que 1 que multiplica los residuales para asegurar la convergencia. Si reducimos η, es conveniente que aumentemos el número de estimadores N.



El meta modelo final será: 

**En Python:** sklearn usa GradientBoostingRegressor y GradientBoostingClassifier

1. **ADA Boost:** Adaptative Boosting. Adapta la importancia de los predictores. Le asigna más peso a aquellos con más error. A diferencia de con Random Forest, acá a los árboles se los poda a tal punto de usar un nodo raíz y dos nodos hojas (árboles stump). Los votos de estos árboles pueden tener más peso que el resto. Cada stump aprende del anterior. El primer Stump aprende dándole igual importancia a cada una de las k observaciones con que se lo entrena. Se elige el feature que genera la menor entropía/Gini y se arma un stump a partir de dicho feature. Teniendo en cuenta cuántas observaciones clasifica mal, se le va a asignar un peso entre 0 y 1 a este stump. Si es 1, quiere decir que no logró clasificar nada correctamente, entonces se va a penalizar su voto en la predicción final. Si es 0 quiere decir que clasificó todo perfecto entonces su voto va a valer más en la predicción final. Se ponderan las observaciones dándole mayor importancia a aquellas que no fueron bien clasificadas y quitándole peso a las que fueron correctamente clasificadas. El segundo Stump va a usar el weighted Gini index para elegir la mejor partición y se repite el proceso sucesivamente:

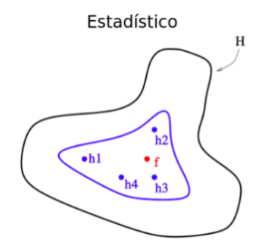
****

1. **XG Boost**: Es un caso específico de gradient boosting en el que se aplica Regularización en la Función de Costo para evitar Overfitting

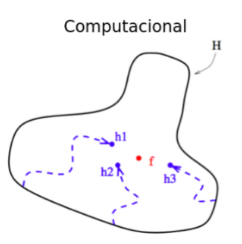
Buscamos en un cierto **espacio de Hipótesis H** la **función f más apropiada** para describir la **relación entre nuestras *features* y el objetivo**.

Las **limitaciones** con las que se puede encontrar un **clasificador base** para no poder aproximar la función de clasificación real mejor que un modelo de ensamble son tres:

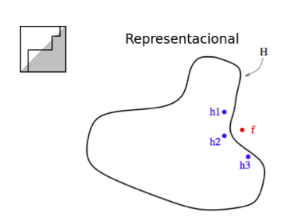
1. **Problemas Estadísticos:** Contar con una cantidad de datos de entrenamiento pequeña. Al promediar las predicciones de varios clasificadores base distintos podemos aproximar mejor la realidad:



1. **Problemas Computacionales:** Aunque tengamos suficientes datos de entrenamiento, sorteando así el problema estadístico, puede ser computacionalmente difícil encontrar el mejor estimador h. IE: En árboles de decisión, se opta por un modelo voraz porque sería muy complejo que el algoritmo haga una búsqueda exhaustiva sobre el espacio de hipótesis de todos los posibles clasificadores. En cambio, si tenemos varios clasificadores base con diferentes puntos de partida, podemos aproximar f mejor que cualquier clasificador individualmente.:

****

1. **Problemas de Representación:** Por limitaciones de cálculo algunos algoritmos no pueden representar f: IE un árbol de decisión aproxima por rectas, si la función f es curva, es difícil que el árbol de decisión por sí solo pueda aproximarla bien, pero logra un mejor resultado promediando con otros árboles:



Para que un clasificador de ensamble mejore los resultados de cualquiera de sus clasificadores base, es necesario que los mismos sean precisos y diversos:

* **Capacidad Predictiva:** Deben hacer **mejores predicciones** que la **totalmente aleatoria (AUC > 0.5).**
* **Diversidad**: Deben **cometer distintos errores ante los mismos casos**. Si esto ocurre, los errores de los predictores estarán poco correlacionados y cuando uno de error, los otros probablemente sean correctos, haciendo que el voto mayoritario sea correcto. Para lograr esto, necesitamos que el error rate individual de cada predictor sea mayor a un 50%.

**En Python:** Usamos el dataset Hitters para entrenar un modelo de ensamble que prediga el valor de log(salary). Vamos a combinar un modelo de regresión lineal simple, con un modelo de regresión Lasso y un árbol de regresión, promediando las predicciones de los 3 modelos .

from sklearn.linear\_model import LinearRegression, Lasso

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error

from sklear import tree

data\_raw = pd.read\_csv(‘../Data/Hitters.csv’)

print(data\_raw.shape)

data\_complete = data\_raw.dropna()

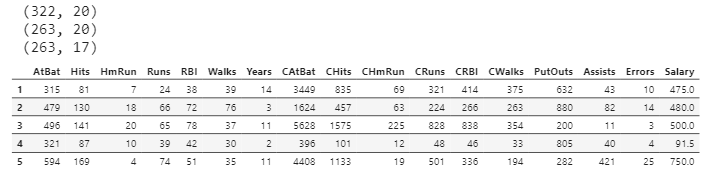
print(data\_complete.shape)

data\_columns = [‘AtBat’, ‘Hits’, ‘HmRun’, ‘Runs’, ‘RBI’, ‘Walks’, ‘Years’, ‘CAtBat’, ‘CHits’, ‘CHmRun’, ‘CRuns’, ‘CRBI’, ‘CWalks’, ‘PutOuts’, ‘Assists’, ‘Errors’, ‘Salary’]

data = data\_complete.loc[:,data\_columns]

print(data.shape())

data.head()



X = data.drop(‘Salary’, axis = 1)

print(X.shape)

y = np.log(data.Salary)

print(y.shape)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state = 127)



scaler = StandardScaler()

X\_train\_scl = scaler.fit\_transform(X\_train)

X\_test\_scl = scaler.transform(X\_test)

# Entrenamos el modelo de regresión lineal:

model\_1 = LinearRegression()

fit\_1 = model\_1.fit(X\_train\_scl, y\_train)

predict\_1 = model\_1.predict(X\_test\_scl)

performance\_1 = mean\_squared\_error(y\_test, predict\_1)



# Entrenamos el modelo Lasso:

model\_2 = Lasso(alpha = 0.05)

fit\_2 = model\_2.fit(X\_train\_scl, y\_train)

predict\_2 = model\_2.predict(X\_test\_scl)

performance\_2 = mean\_squared\_error(y\_test, predict\_2)



# Ahora entrenamos el árbol de regresión:

model\_3 = tree.DecisionTreeRegressor(random\_state = 127)

model\_3.fit(X\_train\_scl, y\_train)

predict\_3 = model\_3.predict(X\_test\_scl)

performance\_3 = mean\_squared\_error(y\_test, predict\_3)



# Ahora construimos el ensamble de los 3 modelos:

def predict\_ensamble(X, model\_1, model\_2, model\_3):

y\_pred1 = model\_1.predict(X)

y\_pred2 = model\_2.predict(X)

y\_pred3 = model\_3.predict(X)

result = (y\_pred\_1 + y\_pred\_2 + y\_pred\_3)/3

return result

y\_pred\_ensemble = predict\_ensamble(X\_test\_scl, model\_1, model\_2, model\_3)

performance\_ensemble = mean\_squared\_error(y\_test, y\_pred\_ensemble)



El **error cuadrático medio de ensamble es el menor obtenido.**

**Conclusiones:**

1. Los modelos de ensamble generalmente tienen menos overfitting y performean mejor.
2. Los métodos de ensamble mejoran el rendimiento de los modelos de base individuales. Esto es porque pueden aproximar mejor la función real en un problema de aprendizaje supervisado, al usar un poco de cada modelo.
3. Estos métodos trabajan mejor en escenarios más complejos, aunque pueden ser complicados y difíciles de interpretar.