Bagging y Random Forests

Alcance:

- Identificar el problema de la falta de generalización de los árboles.
- Conocer la técnica de Bootstrap y su aplicación para Bagging.
- Implementar Bagging con `sklearn`.
- Conocer el problema de árboles correlacionados producto de Bagging.
- Conocer los Random Forests y cómo alivian el problema de los árboles correlacionados.
- Implementar Random Forests con `sklearn`.

Motivación

Resulta que hasta el momento nuestro flujo de trabajo conlleva evaluar múltiples modelos y posteriormente escoger el que tenga un mejor desempeño en los datos. Si bien este enfoque es intuitivo para nosotros, hay un par de contratiempos asociados:

- Cuando implementamos un modelo en los datos, estamos ignorando deliberadamente todos los demás modelos candidatos que no son "lo suficientemente buenos". Estos clasificadores se conocen como **clasificadores débiles** (weak learners, Kearns y Valliant, 1989), que se desempeñan marginalmente mejor que un clasificador aleatorio con chance $1/Ny$ Kearns y Valliant (1989) se preguntaron sobre qué tan válidos son los clasificadores débiles, y qué hacer frente a ellos.
- Cuando elegimos un modelo dentro de una serie de candidatos, inevitablemente favorece a un modelo con mayor sesgo o varianza. Este punto es extremadamente relevante en los árboles de decisión, donde se estima de forma local en los datos de entrenamiento.

Ante estas limitantes, el enfoque es agrupar modelos en lo que se conocen como **ensambles**. La intuición detrás de los ensambles es entrenar múltiples modelos de aprendizaje para resolver el mismo problema. Mediante la agrupación de múltiples clasificadores débiles potenciamos la capacidad de generalización, permitiendo que entre cada modelo exista colaboración y se potencien en su capacidad predictiva (Zhou, 2012). Existen dos grandes modos de ensamblar modelos: **paralelos** y **secuenciales**. En esta lectura abordaremos los ensambles paralelos, específicamente Bagging y Random Forest.

Bagging: Booststrap Aggregation

Consideremos una de las falencias principales de los árboles de decisión: dado que se optimizan de forma local en los datos de entrenamiento, pueden sufrir de alta varianza, disminuyendo su capacidad de generalización a datos previamente no vistos. Ya conocemos algunas estrategias dentro del modelo para aliviar este problema, como definir los criterios de partición, profundidad máxima de los nodos y cantidad de atributos a considerar en el modelo.

Leo Breiman (1984) sugirió otra alternativa: entrenar múltiples árboles de decisión con distintas muestras seleccionadas de forma aleatoria con reemplazo. Esta estrategia de aleatorizar se conoce como bootstraping. Es necesaria una pequeña digresión sobre ésta técnica:

Digresión: Bootstrap

Un problema clásico de la inferencia estadística es el hecho que muchos de sus métodos se apoyan en características asintóticas que guían hacia distribuciones normales y chi cuadrado.

Una alternativa se proporciona mediante el bootstrap, que busca aproximar la distribución de un estadístico mediante simulación Monte Carlo con el muestreo realizado a partir de la distribución empírica o estimada de los datos observados. Este muestreo corresponde a un muestreo con reemplazo.

La idea base del Bootstrap es generar inferencia de una población a partir de una muestra, que se puede obtener mediante el remuestreo de la muestra.

Algorítmo básico de bootstrap

1. Dado datos $x_i \in X$ extraer **N** datos implementando un método definido y guardar los datos en una nueva muestra $x_i^* \in X^*$.

2. Generar un estimado adecuado utilizando esta nueva muestra.

3. Repetir **B** veces los pasos 1 y 2, donde **B** es un número grande. Con esto podremos obtener **B** replicaciones del estimando de interés $\widehat{\theta} \in \widehat{\theta}_B^*$.

4. Finalizada la iteración en B estimar el nuevo estimador con:
    - Estimador puntual de la media:

$$\bar{\hat{\theta}^*} = \frac{1}{B}\sum_{b=1}^{B}\hat{\theta}_b^*$$

    - Estimador puntual de la varianza:

$$s^2_{\hat{\theta},\text{Bootstrap}} = \frac{1}{B-1}\sum_{b=1}^{B}(\hat{\theta}_b^* - \bar{\hat{\theta}_b^*})^2$$

Para ejemplificar su comportamiento, generaremos una simulación donde generamos un remuestreo de los datos disponibles. En la figura `afx.plot_bootstrap` comparamos la distribución teórica de una variable $X_{pop} \sim Normal(0,1)$ con el comportamiento realizado en la cantidad de remuestreos. En la práctica, si tenemos la información poblacional, el bootstrapping no sería necesario.

Al generar distintas instancias sobre esta distribución teórica, somos capaces de imitar su comportamiento, lo que nos permite asimilar características en la medida que re-muestremos todo el espacio de los datos.

```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
%matplotlib inline
import lec8_graphs as afx
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (10, 6)

afx.plot_bootstrap(n_sims=1000)
```

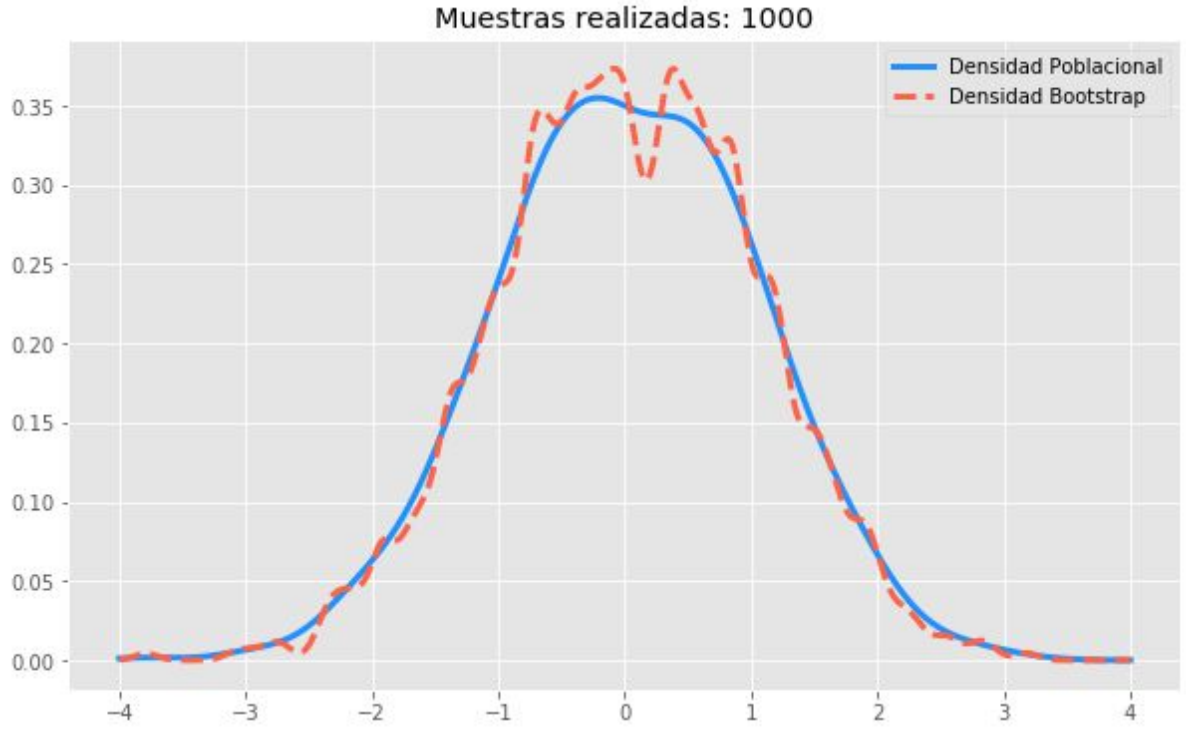

Imagen 1. Densidad Poblacional/ Densidad Bootstrap.

Idea base de Bagging

El principal problema de un árbol es el hecho que puede sufrir de alta varianza en los datos de entrenamiento. Si bien técnicas como pruning puede reducir la varianza al eliminar nodos innecesarios, hay métodos alternativos que se benefician del overfit producto de los árboles. Bagging combina y promedia múltiples modelos. Mediante la ponderación de múltiples árboles se contrarresta la variabilidad de cualquier árbol y reduce el overfit, mejorando su capacidad predictiva.

Pasos

Los pasos para implementar Bagging se puede entender como:

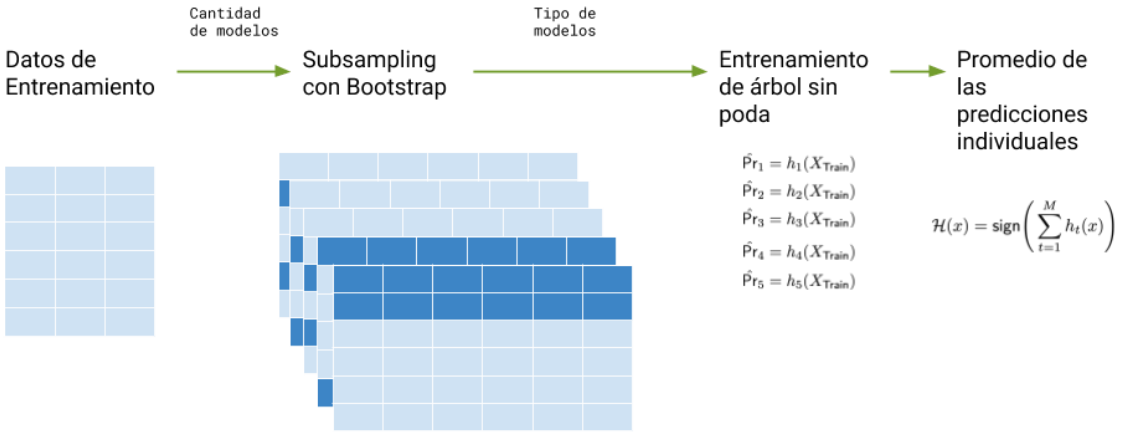

Imagen 2. Implementación de Bagging.

1. Crear **M** muestras mediante bootstrap de la muestra de entrenamiento. Con este proceso, generamos distintos subconjuntos de datos que compartirán una distribución similar a la muestra de entrenamiento.

2. Dentro de cada muestra $m \in M$ se entrena un árbol donde sus nodos crecen al máximo.

3. Para una observación previamente no observada, se promedian las predicciones de *cada árbol* para crear un valor promedio global.

Cabe destacar que este procedimiento es agnóstico a la implementación. Podríamos implementarlo en modelos de regresión y sus variantes, pero resulta que su mejor desempeño es cuando incorporamos datos donde los modelos sufren de alta varianza. Resulta que Bagging es una estrategia generalizable a múltiples modelos, orientada a la reducción de la varianza mediante la repetición del entrenamiento en múltiples muestras. Si bien se puede aplicar en contextos como regresiones, es en los árboles donde tiene un mejor desempeño dado que estabiliza la predicción en múltiples instancias.

Si tomamos un clasificador binario débil que busca asignar clases en el vector objetivo $\gamma = \{-1, 1\}$ y entrenamos un modelo representado con $h_t(x)$ éste tendrá un error de generalización en nuevas observaciones de ε_y.

El error de generalización en el modelo específico será la probabilidad de no ocurrencia en otro modelo $Pr(h_t(x) \neq f(x))$.

Mediante Bagging podemos sumar y promediar cada uno de los clasificadores binarios débiles con:

$$\mathcal{H}(\mathbf{x}) = \text{sign}\Big(\sum_{t=1}^{T} h_t(\mathbf{x})\Big)$$

Donde **T** es la cantidad de modelos entrenados. Para cada una de las observaciones en la muestra de entrenamiento, sumamos la clase predicha en cada modelo. El ensamble H generará un error sólo cuando la mitad o más de los clasificadores débiles $t \in T$ generen un error.

Otra de las ventajas de implementar bootstraping en el proceso de Bagging es atenuar la dependencia entre clasificadores débiles. Dado que todos son generados en base a la misma muestra de entrenamiento, el modelo estará aprendiendo estos datos en una tasa muy rápida. Mediante bootstrap generamos muestras donde ignoraremos ciertas observaciones, induciendo aleatoriedad en el proceso de entrenamiento del modelo.

Uno de los beneficios de bagging es que en promedio una muestra creada con bootstrap contendrá un **.63** de la muestra de entrenamiento. Esto nos deja con un **.33** de los datos que se encontrarán fuera de la muestra. Esta cifra se conoce como el *out-of-bag-sample* para estimar el error **en la muestra de validación**. Este punto confiere una característica deseable de Bagging: En la medida que aumentamos la cantidad de muestras generadas con bootstrap, podemos implementar la out-of-bag-sample para estimar la exactitud del modelo en sus predicciones, conllevando a un proceso similar a la validación cruzada. Re-visitaremos esta idea cuando posteriormente en esta lectura con los Random Forest.

Implementando Bagging para los Precios de California

Para ejemplificar el uso de Bagging, utilizaremos la base de datos sobre los precios de inmuebles en California utilizada en el ejemplo de Árboles de Decisión. Esto nos permitirá realizar comparaciones respecto al comportamiento de ambos modelos. Partamos por implementar el mismo preprocesamiento de datos.

```
import pandas as pd
df = pd.read_csv('cadata.csv', header=1).drop(columns='1')
```

```
df['log_MedianIncome'] = np.log(df['MedianIncome'])
df['log_MedianHouseValue'] = np.log(df['MedianHouseValue'])
df = df.drop(columns=['MedianHouseValue', 'MedianIncome'])
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df.loc[:,
'MedianHouseAge':'log_MedianIncome'],

df['log_MedianHouseValue'],
                                                    test_size=.33,
random_state=11238)
```

Para implementar Bagging con árboles de regresión, debemos incorporar la clase `sklearn.ensemble.BaggingRegressor`. Dado que en esta implementación utilizaremos árboles de regresión que vienen por defecto implementados en la clase, no es necesario especificar sobre el comportamiento de hiper parámetros todavía, con la salvedad de implementar una semilla, para asegurar la replicación de resultados.

Digresión: `sklearn.ensemble.BaggingRegressor/BaggingClassifier`

La API es generalizable, por defecto implementa árboles pero se puede incorporar cualquier modelo que forme parte de `scikit-learn`. Resulta que la implementación se realiza en consideración al efecto reductor de varianza, haciéndolo un procedimiento fuertemente asociado con los árboles de decisión.

Si implementamos nuestro modelo sin modificar hiperparámetros, la clase `BaggingRegressor` generará 10 modelos. Como todo modelo de `scikit-learn`, para entrenarlo necesitamos utilizar el método `fit`.

```
from sklearn.ensemble import BaggingRegressor
bagging_model = BaggingRegressor(random_state=11238).fit(X_train,
y_train)
```

Comencemos por examinar cada uno de los árboles. Éstos se encuentran dentro del modelo en `BaggingRegressor.estimators_`. Vamos a solicitar la información sobre el tipo de modelo, el criterio y la semilla pseudoaleatoria asociada a cada uno de ellos. Observamos que por defecto implementa modelos `sklearn.tree.tree.DecisionTreeRegressor`, con el criterio de optimización `'mse'` y una semilla que varía entre cada uno de ellos.

```
for i in bagging_model.estimators_:
        print("Modelo: {}. Criterio: {}. Semilla: {}".format(type(i),
i.criterion,i.random_state))
```

```
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 134534932
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 1532576145
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 1982095898
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 280845846
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 1572358144
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 198602398
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 120741769
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 860188809
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 2093968571
Modelo: <class 'sklearn.tree.tree.DecisionTreeRegressor'>. Criterio:
mse. Semilla: 1518121755
```

Ahora evaluemos el desempeño del ensamble Bagging con las métricas `mean_squared_error y r2_score`. Para ello tomemos en cuenta que el desempeño del árbol de decisión que implementamos en la lectura anterior era el siguiente:

Métrica	**Valor**
MSE	0.109
MAE	0.157
R^2	0.668

Tabla 1. Tabla de desempeño.

Observamos que simplemente con 10 árboles promediados, nuestro error cuadrático promedio disminuyó en 0.049, la mediana absoluta del error disminuyó en 0.034 y el coeficiente de determinación aumento en .147.

```
from sklearn.metrics import mean_squared_error, median_absolute_error,
r2_score
print("MSE:", mean_squared_error(y_test, bagging_model.predict(X_test)))
print("MAE:", median_absolute_error(y_test,
bagging_model.predict(X_test)))
print("R2:", r2_score(y_test, bagging_model.predict(X_test)))
```

```
MSE: 0.061285050289994675
MAE: 0.12364743782288823
R2: 0.8138666922626886
```

Inspeccionando las predicciones

Ya sabemos cómo se comporta la clase sklearn.ensemble.BaggingRegressor a grandes rasgos. Ahora ejemplificamos cómo se genera el proceso de predicción para una observación específica en el testing set.

La observación que tenemos corresponde a las siguientes características:

```
print(X_test[:1])
```

```
      MedianHouseAge  TotalRooms  TotalBedrooms  Population  Households
\
16934             9.0      1150.0          287.0       377.0       243.0

       Latitude  Longitude  log_MedianIncome
16934     37.56    -122.32          1.343309
```

Su valor verdadero en el vector objetivo es de:

```
y_test[:1]
```

```
16934    12.377923
Name: log_MedianHouseValue, dtype: float64
```

Lo que necesitamos hacer es generar una predicción de esta observación en cada árbol de regresión entrenado en nuestro ensamble de Bagging. Para ello implementaremos la siguiente comprensión de lista:

```
hold_instance = [i.predict(X_test[:1]) for i in
bagging_model.estimators_]
```

Para facilitar el proceso de visualización, generamos un eje X con las etiquetas pertinentes a cada árbol.

```
dec_tree_lab = ["RegressionTree: {}".format(i + 1) for i in
range(len(bagging_model.estimators_))]
```

Cada punto azul en la figura creada corresponde a la predicción realizada por un árbol en específico. La recta roja representa la predicción promediada del ensamble con Bagging. La predicción del modelo toma en consideración cada uno de los puntos específicos.

Retomemos nuestros conocimientos básicos de estadística. Resulta que en la medida que nuestro ensamble de Bagging aumenta la cantidad de estimadores entrenados, estaremos recabando más información sobre el parámetro poblacional a inferir. Esta es una aplicación de la ley de los grandes números.

```
plt.plot(hold_instance, 'o', color='dodgerblue', label='Predicción')
plt.xticks(range(10), dec_tree_lab,rotation =90)
plt.axhline(bagging_model.predict(X_test[:1]), label='Predicción
Bagging')
plt.ylabel(r'$\hat{y} = \sum_{t=1}^{T} h_{t}(\mathbf{x})$')
plt.title('Predicciones específicas en observación X_test[:1]')
plt.legend();
```

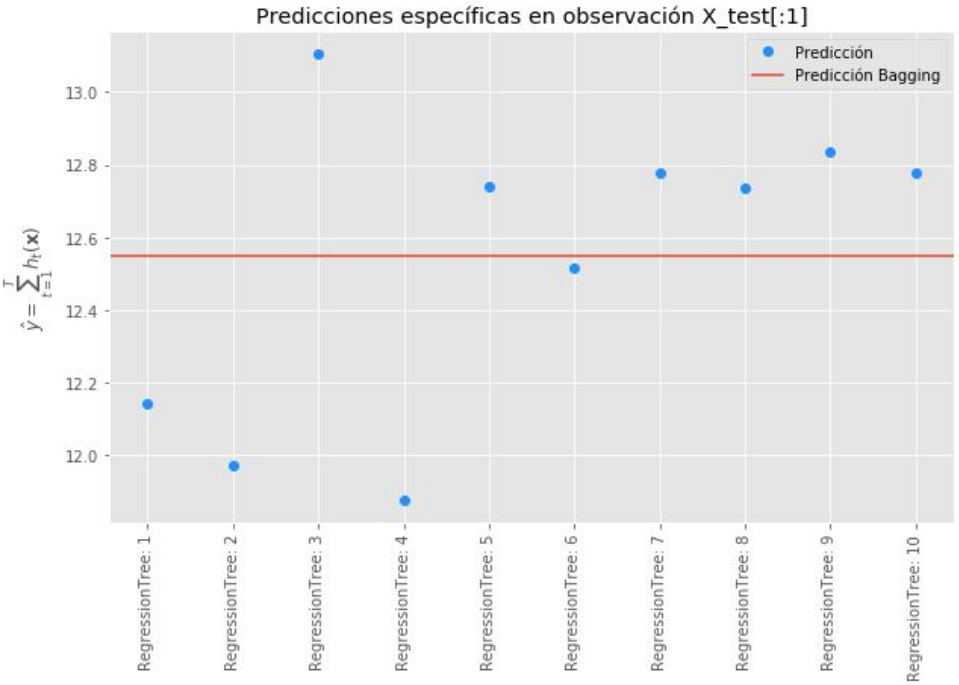

Imagen 3. Predicciones específicas en observación.

Efecto de la cantidad de modelos incorporados en el ensamble

Sigamos expandiendo la idea del comportamiento asintótico de los ensambles. Resulta que este comportamiento también es válido cuando nos referimos al desempeño general del modelo. Consideremos el siguiente experimento: Vamos a entrenar 40 modelos con nuestro regresor Bagging, donde incrementaremos en pasos de 25 la cantidad de estimadores incluídos, hasta llegar a aproximadamente 1000.

```
# generamos 3 listas para guardar resultados
tmp_mse_test, tmp_mae_test, tmp_r2_test = [], [], []

# generamos valores a pasar
n_tree = range(20, 1000, 50)

# Para cada uno de los rangos declarados
for i in n_tree:
    # Entrenamos nuestro modelo
    tmp_bag_model = BaggingRegressor(n_estimators=i,
random_state=11238).fit(X_train,y_train)
    # guardamos el mse
    tmp_mse_test.append(mean_squared_error(y_test,
tmp_bag_model.predict(X_test)))
    # guardamos el mae
    tmp_mae_test.append(median_absolute_error(y_test,
tmp_bag_model.predict(X_test)))
    # guardamos el rcuadrado
    tmp_r2_test.append(r2_score(y_test, tmp_bag_model.predict(X_test)))
```

Graficando los resultados del modelo evaluado en distintos puntos, se aprecia que en los primeros árboles evaluados, la mejora en el desempeño es substancial en sus primeras iteraciones. Los criterios R^2 y el error cuadrático promedio se comportan relativamente similar: a partir de los 500 estimadores, el desempeño se estabilizará.

```
params = [[1, tmp_r2_test,
           r2_score(y_test, tmp_bag_model.predict(X_test)), 'r2'],
          [2, tmp_mse_test,
           mean_squared_error(y_test, tmp_bag_model.predict(X_test)),
'mse'],
          [3, tmp_mae_test,
           median_absolute_error(y_test, tmp_bag_model.predict(X_test)),
'mae']]
plt.figure(figsize=(12, 12))
for i in params:
    plt.subplot(2, 2, i[0])
    afx.plot_bagging_behavior(i[1], i[2], n_tree)
    plt.title(str(i[3]))
    plt.tight_layout()
```

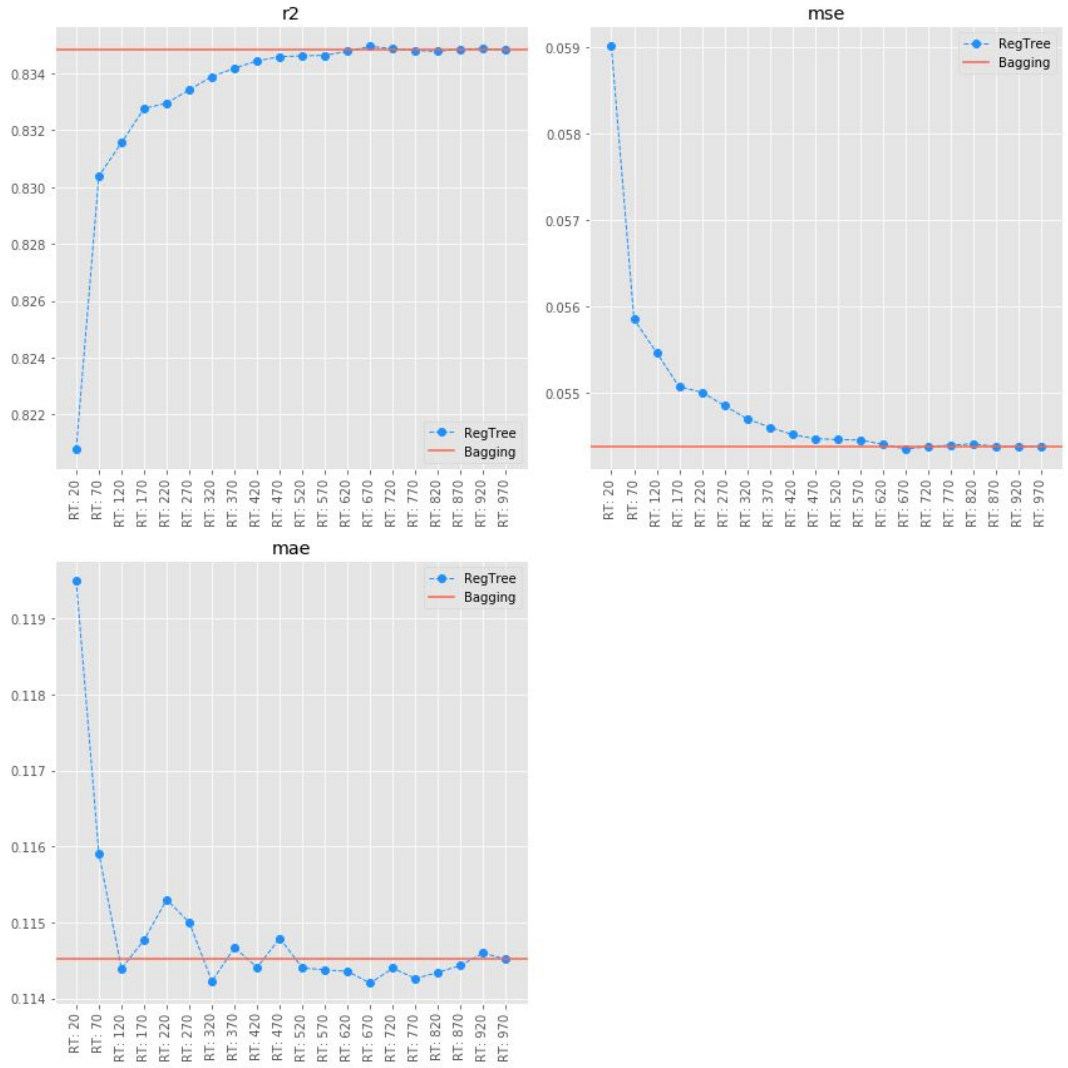

Imagen 4. Gráficas del modelo evaluado.

Probando la correlación entre árboles

Los clasificadores débiles deben ser **diversos**, esto quiere decir que la correlación entre las predicciones de dos árboles deben ser bajas. Implementando nuestro primer modelo con 10 estimadores, las correlaciones bivariadas entre los puntajes predichos de cada árbol son lo suficientemente fuertes como para levantar sospechas. Esto es problemático dado que nuestro modelo de ensamble estará siendo entrenado con información repetida. Para evaluar la correlación entre cada uno de los árboles, generaremos las predicciones de cada uno de los árboles y posteriormente las correlacionaremos.

```
from scipy import stats
import seaborn as sns

store_rho = []

"""
Guardamos los estimadores de nuestro
primer modelo.
"""
bag_est = bagging_model.estimators_

# recorremos dos veces los estimadores
for i in bag_est:
    for j in bag_est:
        # y calculamos las correlaciones entre pares
        store_rho.append(stats.pearsonr(i.predict(X_test),
j.predict(X_test))[0])

store_rho = np.array(store_rho).reshape(len(bag_est), len(bag_est))

sns.heatmap(store_rho, cmap='coolwarm', annot=True,
xticklabels=dec_tree_lab,
            yticklabels=dec_tree_lab);

plt.title("Correlación bivariada entre árboles");
```

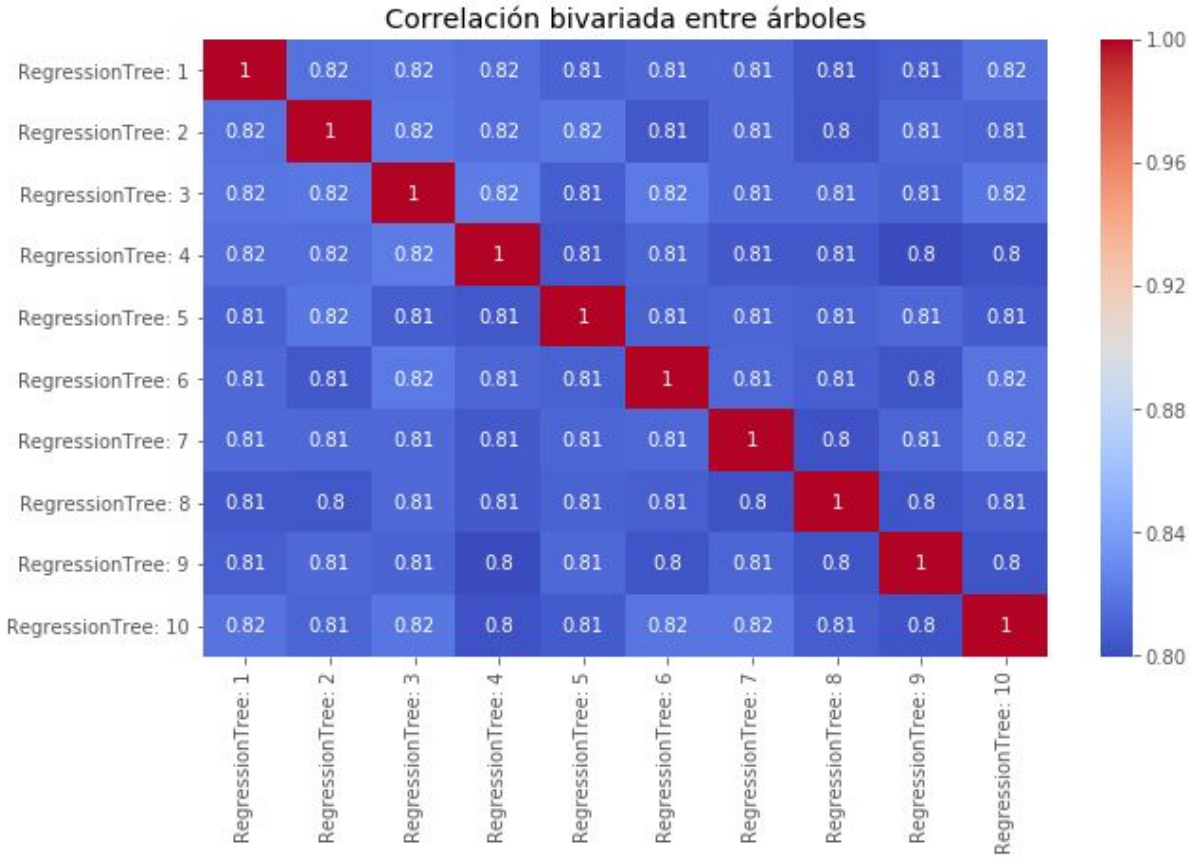

Imagen 5. Correlación bivariada entre árboles.

El heatmap de arriba muestra evidencia a favor de la existencia de árboles correlacionados. Esta es la principal piedra de tope de Bagging, y que da paso para hablar de su siguiente iteración: **Random Forest**.

Random Forests (Bosques aleatorios)

Para solucionar el problema de clasificadores débiles correlacionados entre sí, Breiman (2001) diseñó los Random Forests. Éstos se pueden entender como una extensión de un ensamble Bagging. De manera similar a éstos, se genera un conjunto finito de submuestras con reemplazo mediante Bootstrap. Para resolver el problema de correlación entre clasificadores, se incluye un mecanismo aleatorio de selección de atributos. Durante la construcción de árboles, Random Forests selecciona un subconjunto de atributos de manera aleatoria y prosigue de igual manera con el entrenamiento y selección de particiones. Dado que con bagging **contamos todos los atributos** en cada árbol, Random Forest agrega más aleatoriedad en el proceso de crecimiento de los árboles.

Los Random Forests dependen de un parámetro K que controla la incorporación de aleatoriedad. Cuando $K = \#Atributos$ el árbol construido es idéntico a un proceso determinístico. Cuando **K=1**, el árbol construido tendrá un atributo escogido al azar. Breiman (2001) sugiere que un buen valor es $K = log(\#Atributos)$.

Cabe señalar que la aleatorización se induce en la selección de atributos, no en la selección de divisiones para el conjunto de atributos.

Tienen un peor punto de inicio aleatorio, pero converge a tasas de error bajas en menor cantidad de iteraciones, en comparación a otros modelos. En la etapa de entrenamiento es más eficiente que un ensamble Bagging, dado que implementa árboles aleatorios que necesitan sólo del subconjunto de atributos en oposición a Bagging donde se deben incluir todos.

Pasos para construir un Random Forest

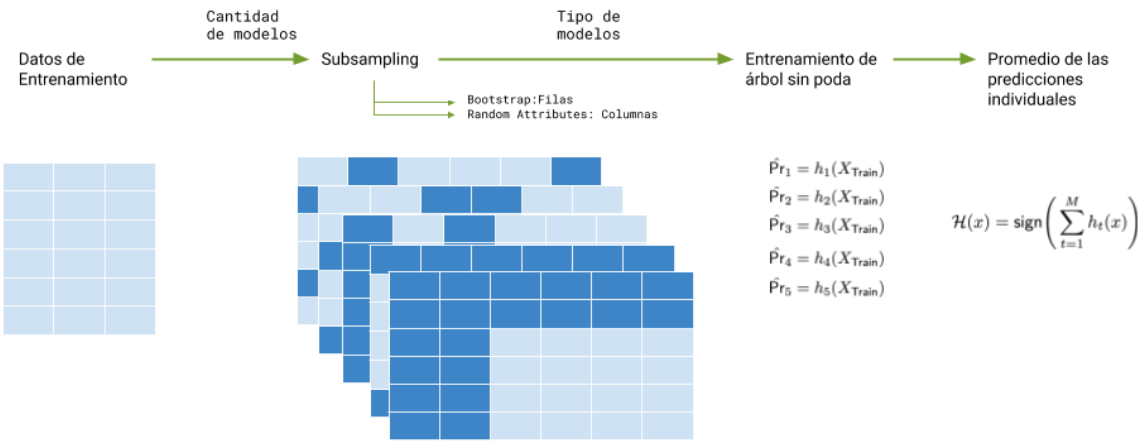

Imagen 6. Pasos para construir un Random Forest.

1. Fijar la cantidad de árboles a construir.

2. Para cada árbol a construir:
   - Generamos una muestra con bootstrap.
   - Hacemos crecer un árbol en ésta.
   - Para cada split existente:
     - Seleccionamos j variables de manera aleatoria de J.
     - Seleccionamos el mejor punto de corte entre las variables j.
     - Particionamos el nodo en dos.

Digresión: La receta para cocinar un buen ensemble

Al igual que hoy en día pasa con los modelos basados en redes neuronales, mucho esfuerzo se puso en algún momento en el desarrollo de métodos de ensamblado ya que aportaron un gran salto en efectividad a partir de los modelos clásicos. Random Forest no es el único método de ensamblado ni mucho menos el "mejor", existen otros modelos que pueden llegar a presentar tanta o mejor efectividad que RF, pero, ¿En qué se diferencian todos estos métodos entre sí?, para responder esta pregunta, veamos en qué se parecen.

Como mencionamos, un método de ensamblado combina una serie de máquinas para conformar una sola máquina: $F = \Sigma_i^N w_i \cdot f_i$.

F es el ensamble, f_i es el **Learner Base**: la máquina básica con la cual se construye el ensamble, en el caso de RF el learner base es un árbol de clasificación. Por otro lado, w_i representa la *"importancia"* que se le dará a una determinada máquina F_i dentro del contexto del ensemble.

Bajo esta definición podemos juntar cualquier conjunto de máquinas que nos gusten y armar un gran monstruo de Frankenstein. Lamentablemente esta aproximación no nos asegura obtener un buen ensemble, la clave de la receta está en **inyectar diversidad**. Podemos inyectar diversidad de diversas formas (pun intended), para esto se puede variar el learner base, manipular directamente los datos (distintos conjuntos de entrenamiento) o incluso variar los hiper parámetros.

En el proceso de manipular los datos, podemos particionar horizontalmente o verticalmente (feature selection). RF implementa ambas, particiona horizontalmente al utilizar distintas muestras bootstrap para entrenar los distintos learner base y al mismo tiempo particiona verticalmente implementando la selección aleatoria de atributos con los que se entrenará a cada learner base.

Out-Of-Bag

Con los modelos anteriores, estábamos acostumbrados a implementar validación cruzada para escoger de mejor manera los hiper parámetros y mejorar nuestra capacidad predictiva. El problema con Bagging y Random Forest es que implementar la validación cruzada puede ser bastante costoso en términos computacionales. Dado que en cada paso de validación cruzada estamos construyendo $K \times n\_estimators$.

Una de las ventajas de los Random Forests es que devuelve un error "fuera de la bolsa".

Dado esta característica de los datos provista mediante bootstrap, la bondad de los clasificadores se puede estimar a partir de estos datos ignorados en cada instancia del modelo. Esto se conoce como la muestra *out-of-bag* (OOB, de aquí en adelante, y nos permite generar una aproximación a la tasa de errores con validación cruzada.

Para cada observación $z_i - (x_i - y_i)$ construir un predictor con random forest mediante el promedio de aquellos árboles con muestras de bootstrap en donde z_i **no aparezca**. Un error estimado *out-of-bag* tendrá un comportamiento idéntico al obtenido con una validación cruzada N fold. Esta es una ventaja de RF por sobre otros algoritmos, dado que se implementa la validación cruzada en el mismo tiempo en que el modelo se ejecuta (Hastie et al., 2009).

Para obtener el estimador OOB, necesitamos registrar los ejemplos de entrenamiento implementados en cada instancia de clasificador débil. La predicción OOB de las observaciones no entrenados con X se obtiene a partir de:

$$\mathcal{H}^{\text{out-of-bag}}(\mathbf{x}) = \underset{y \in \mathcal{Y}}{\text{argmax}} \sum_{t=1}^{T} I(h_t(\mathbf{x}) = y) \cdot \mathbb{I}(\mathbf{x} \notin D_t)$$

A partir de esta predicción se puede obtener el error de generalización asociado a Bagging/Random Forest.

$$\varepsilon^{\text{out-of-bag}} = \frac{1}{|\text{Datos}|} \sum_{\mathbf{x},\mathbf{y} \in \text{Datos}} I(\text{H}^{\text{out-of-bag}}(\mathbf{x} \neq y)$$

Dado que cada árbol se construye a partir de una muestra realizada con bootstrap, para cada árbol existe una porción de los datos no se utiliza con la cual podemos estimar qué tan bueno es nuestra estimación.

Implementación de Random Forest para la clasificación de votos

Para exponer la implementación de un algoritmo Random Forest, utilizaremos los datos en el archivo `voting.csv` de la lectura de Árboles de Clasificación. Implementaremos el mismo proceso de preprocesamiento y selección de atributos que cuando implementamos el modelo `DecisionTreeClassifier`.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
df = pd.read_csv('voting.csv').drop(columns='Unnamed: 0')
# Vamos a binarizar cada variable categórica, ignorando la primera
categoría de referencia
# Región del país
df = pd.concat([df, pd.get_dummies(df['region'], drop_first=True,
prefix='region')], axis=1)
# Nivel educacional del encuestado
df = pd.concat([df, pd.get_dummies(df['education'], drop_first=True,
prefix='ed')], axis=1)
# Sexo del encuestado
df = pd.concat([df, pd.get_dummies(df['sex'], drop_first=True,
prefix='sex')], axis=1)
# Intención de voto
df = pd.concat([df, pd.get_dummies(df['vote'], drop_first=False,
prefix='vote')], axis=1)
# Botamos las variables originales
df = df.drop(columns=['region', 'sex', 'education', 'vote'])
df.head()
# Nos aseguramos de limpiar los datos perdidos
df = df.dropna()
# generamos las muestras definiendo la matriz de atributos
X_train, X_test, y_train, y_test = train_test_split(df.loc[:,
'population':'ed_S'],
# el vector objetivo
df['vote_N'],
# el tamaño de la muestra a dejar como validación
test_size=.33,
# definiendo la semilla pseudoaleatoria
random_state=11238)
```

Para implementar un Random Forest para un problema de clasificación, debemos incorporar la clase `sklearn.ensemble.RandomForestClassifier`.

Digresión: Implementación de Random Forest

Resulta que la implementación original de Random Forest diseñada por Breiman (2001) dista de la implementación en `scikit-learn`. En la implementación de Breiman, cada clasificador **debe** votar por una clase específica. Mientras que la implementación de `scikit-learn` promedia la predicción probabilística de cada clasificador **antes** de asignar la clase.

La implementación similar a Breiman en `scikit-learn` corresponde a `VotingClassifier`.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, roc_auc_score,
classification_report
```

Anteriormente hablamos de las virtudes de la muestra *out-of-bag*. Para solicitarla en nuestro ensamble de Random Forest, necesitamos declarar la opción `oob_score=True`, que permitirá registrar el efecto predictivo de cada modelo dentro del ensamble en la muestra que no fue incluida dentro del bootstrap.

```
voting_rf = RandomForestClassifier(oob_score=True,
random_state=11238).fit(X_train, y_train)
```

```
/Users/veterok/anaconda3/lib/python3.7/site-packages/sklearn/ensemble/fo
rest.py:460: UserWarning: Some inputs do not have OOB scores. This
probably means too few trees were used to compute any reliable oob
estimates.
  warn("Some inputs do not have OOB scores. "
```

Si implementamos el modelo con la cantidad por defecto de estimadores (`n_estimators=10`), `sklearn` advertirá que algunos modelos no tienen puntajes OOB dado que existen muy pocos árboles como para estimar OOB de forma confiable.

Probablemente dado el desbalance entre clases de nuestro vector objetivo, el modelo tiene un peor desempeño en identificar correctamente aquellos casos donde se votó que no.

```
print(classification_report(y_test, voting_rf.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.83      0.89      0.86       572
           1       0.75      0.64      0.69       283

    accuracy                           0.81       855
   macro avg       0.79      0.77      0.78       855
weighted avg       0.81      0.81      0.81       855
```

```
import lec8_graphs as afx
voting_rf.feature_importances_
x_mat_vars = df.loc[:, 'population':'ed_S']
afx.plot_importance(voting_rf, x_mat_vars.columns)
```

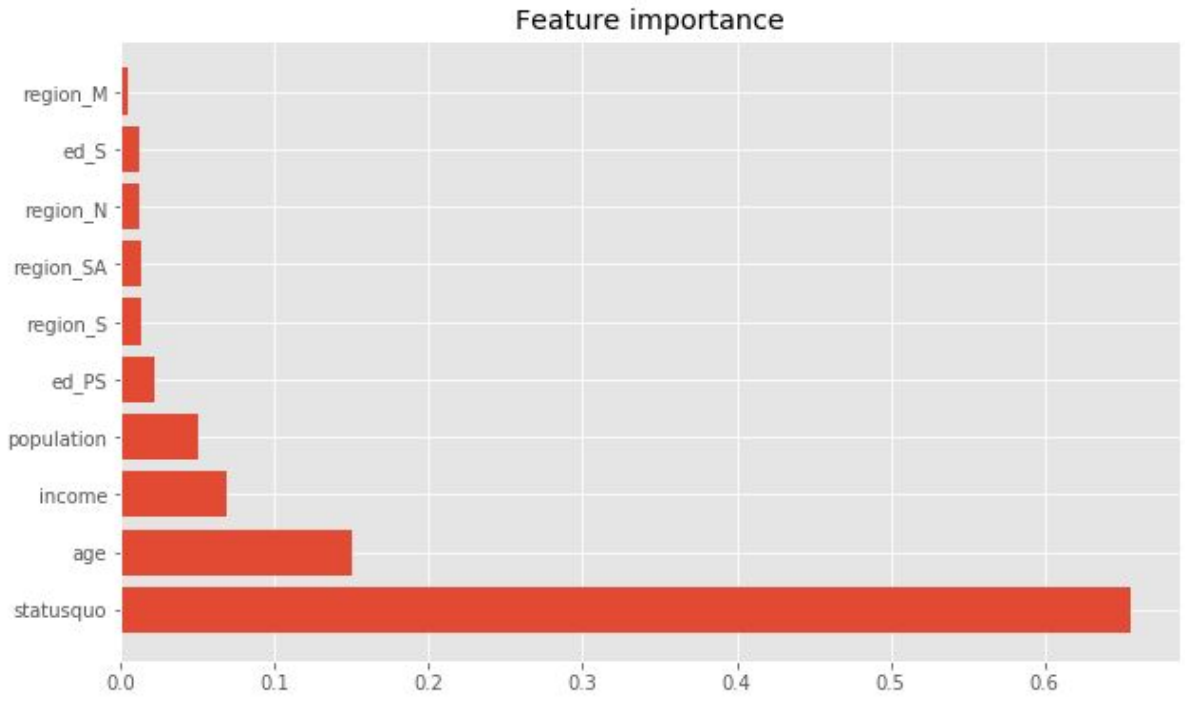

Imagen 7. Feature importance.

Visualizando el efecto de múltiples árboles en la clasificación

Refinemos el modelo mediante la separación de los dos atributos con una mayor importancia relativa en la definición de cortes. Estos son `statusquo` y `age`. Al visualizar las observaciones condicional al vector objetivo, encontramos que aquellos que votaría no en el Plebiscito se posicionan en valores negativos, mientras que el efecto de la edad parece ser constante. Existe una superposición entre observaciones que impide el buen funcionamiento de un clasificador lineal.

```
depured_X = df.loc[:, ['age', 'statusquo', 'vote_N']]
colors = ['dodgerblue', 'tomato']
for i in depured_X['vote_N'].unique():
    plt.scatter(depured_X[depured_X['vote_N'] == i]['age'],
                depured_X[depured_X['vote_N'] == i]['statusquo'],
marker='.',
               label="Vote = {}".format(i), alpha=.5, color=colors[i])
plt.legend();
plt.ylim(-1.3, 1.5);
plt.ylabel('Status Quo');
plt.xlabel('Age');
```

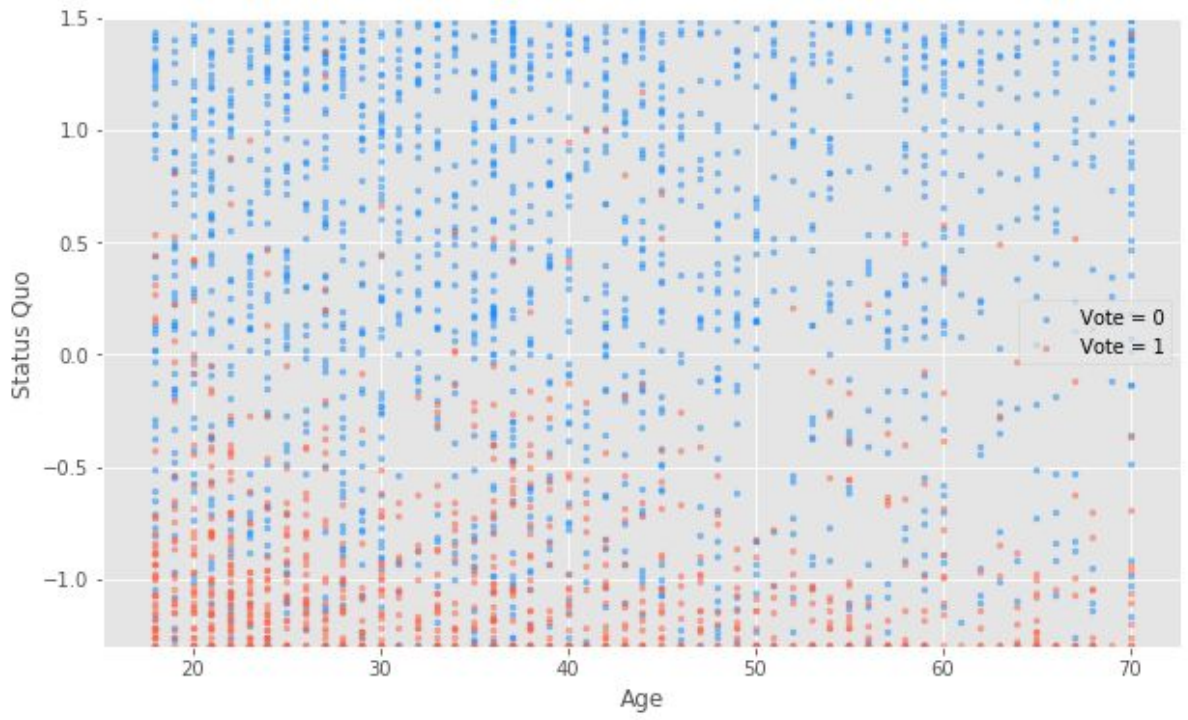

Imagen 8 Status Quo.

Para efectos prácticos, entrenaremos un Random Forest con 500 árboles en la muestra completa, de manera tal de ver cómo se comporta el clasificador.

```
random_forest = RandomForestClassifier(n_estimators=500, n_jobs=-1,
random_state=11238,oob_score=True)
random_forest.fit(depured_X.loc[:, 'age':'statusquo'],
depured_X['vote_N'])
get_x_1 = afx.fetch_lims(depured_X['statusquo'])
get_x_2 = afx.fetch_lims(depured_X['age'])
x_mesh, y_mesh, joint_xy = afx.generate_mesh_grid(depured_X, 'age',
'statusquo')
Z = random_forest.predict_proba(np.c_[x_mesh.ravel(),
y_mesh.ravel()])[:, 1].reshape(x_mesh.shape)
```

```
plt.contourf(x_mesh, y_mesh, Z, cmap='coolwarm')
plt.colorbar();
```

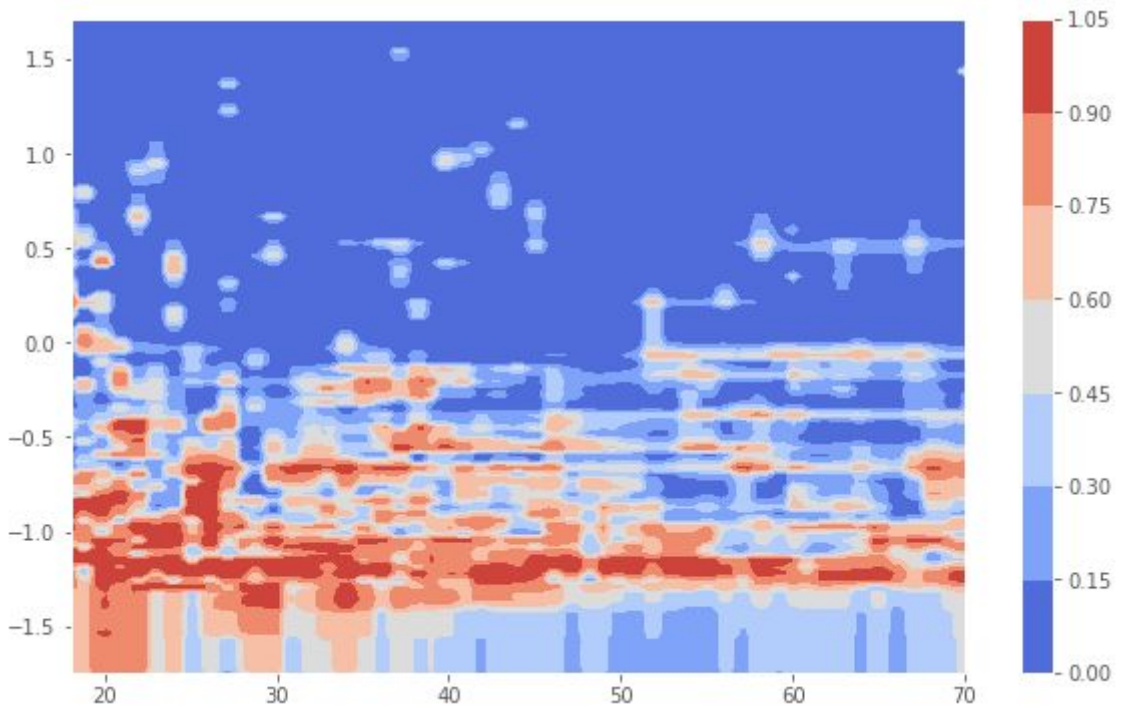

Imagen 9. Random Forets.

La superficie de respuesta inferida en el modelo mapea de forma eficiente la probabilidad de votar No, permitiendo separar aquellas observaciones que se escapan de la norma.

Incorporación de aleatoriedad en los atributos

Anteriormente se mencionó una de las principales características sobre los Random Forest: la capacidad de inducir aleatoriedad en la cantidad de atributos agregados. En la clase `sklearn.ensemble.RandomForestClassifier`, el hiper parámetro se controla con la opción `max_features`. Las opciones disponibles son:

- Un número entero que fija la cantidad de atributos en cada estimador.
- Un número flotante que considera una fracción de los atributos mediante `int(max_features * n_features)`.
- `auto` y `sqrt` conllevan a la misma estrategia, donde la cantidad de atributos se escoge con $\sqrt{n\_atributos}$. Esta es la opción por defecto.
- `log2`, la cantidad de atributos se escoge con $log_2(n\_atributos)$.
- `None`, la cantidad de atributos máximos corresponde a la cantidad de atributos total en la matriz. Esta situación es idéntica a un algoritmo de ensamble Bagging.

Breiman (2001) sugiere que un buen valor por defecto para escoger la cantidad de atributos a aleatorizar es mediante el logaritmo. Para ejemplificar el efecto de la aleatorización de los atributos en los modelos, evaluaremos el desempeño de cada criterio de selección condicional a una serie de estimadores. El código presentado abajo registrará los puntajes OOB y exactitud (medidos como el error, siguiendo las prácticas de Hastie et al. (2009)).

```
# generamos una serie de listas para guardar los valores
tmp_oob_none, tmp_oob_sqrt,tmp_oob_log2 = [], [], []
tmp_test_acc_none, tmp_test_acc_sqrt, tmp_test_acc_log = [], [], []
n_estimators = range(20, 1000, 25)
```

```
# para cada rango de modelos estimados
for i in n_estimators:
    # Implementamos una variante con todos los atributos
    voting_rf_none = RandomForestClassifier(n_estimators=
i,max_features=None,
                                        oob_score=True,
                                        random_state=123).fit(X_train,
y_train)
    # Implementamos una variante donde los atributos se escogen con sqrt
    voting_rf_sqrt = RandomForestClassifier(n_estimators= i,
max_features="sqrt",
                                warm_start=True,
                                oob_score=True,
                                random_state=123).fit(X_train, y_train)
    # Implementamos una variante donde los atributos se escogen con log
    voting_rf_log = RandomForestClassifier(n_estimators= i,
max_features="log2",
                                warm_start=True,
                                oob_score=True,
                                random_state=123).fit(X_train, y_train)
    # Estimamos el error en OOB
    tmp_oob_none.append(1 - voting_rf_none.oob_score_)
    tmp_oob_sqrt.append(1 - voting_rf_sqrt.oob_score_)
    tmp_oob_log2.append(1 - voting_rf_log.oob_score_)
    # Estimamos el error en la exactitud
    tmp_test_acc_none.append(1 - accuracy_score(y_test,
voting_rf_none.predict(X_test)))
    tmp_test_acc_sqrt.append(1 - accuracy_score(y_test,
voting_rf_sqrt.predict(X_test)))
    tmp_test_acc_log.append(1 - accuracy_score(y_test,
voting_rf_log.predict(X_test)))
```

El primer punto a destacar en la figura, es que el no aleatorizar los atributos en cada modelo del ensamble genera un peor desempeño tanto en puntaje OOB y error en la muestra de validación. El segundo punto a considerar es que para este problema el desempeño es idéntico al implementar un criterio de aleatorización de atributos log o sqrt. Un tercer elemento a considerar es el hecho que el comportamiento del puntaje OOB tiende a mejorar en la medida que aumentamos la cantidad de estimadores a incluir en el ensamble.

```
plt.figure(figsize=(12, 4));
fig, axs = plt.subplots(1, 3, sharex=True, sharey=True);
axs[0].plot(tmp_oob_log2, '.-', label='OOB error rate');
axs[0].plot(tmp_test_acc_log, '.-', label='Test error rate');
axs[0].set_title('Log2');
axs[1].plot(tmp_oob_sqrt,'.-',  label='OOB error rate');
axs[1].plot(tmp_test_acc_sqrt,'.-',  label='Test error rate');
axs[1].set_title('Squared Root');
axs[2].plot(tmp_oob_none, '.-', label='OOB error rate');
axs[2].plot(tmp_test_acc_none,'.-',  label='Test error rate');
axs[2].set_title('None');
plt.tight_layout();
plt.legend();
```

```
<Figure size 864x288 with 0 Axes>
```

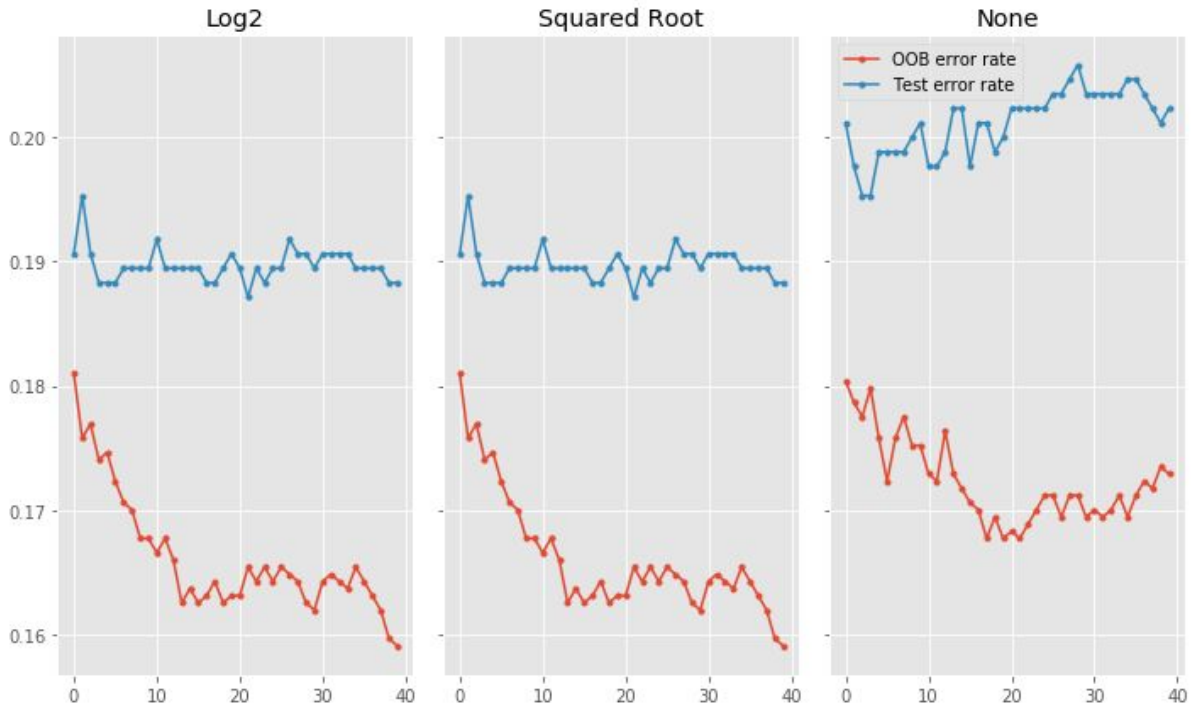

Imagen 10. Gráfico de `Log2`, `Squared Root` y **None**.

Aspectos adicionales a considerar

Pros

- Si bien Bagging se considera un punto procedimental para entender el comportamiento de Random Forests, las clases `sklearn.ensemble.BaggingRegressor` y `sklearn.ensemble.BaggingClassifier` funcionan como wrappers: métodos que nos permiten flexibilizar los modelos a ingresar.

- Por lo general estos modelos tienen un buen desempeño "out-of-the-box", dado que no hay mucho hiper parámetro a modificar que afecte a la forma funcional del modelo (a diferencia de Support Vector MachineS). Todos los hiper parámetros tienen propiedades asintóticas.

- Dado que se itera una serie de veces, tienen un método de validación cruzada sin la necesidad de envolverlo en un `sklearn.model_selection.GridSearchCV`.

- Como se basan en árboles, no es necesario preprocesamiento alguno y son robustos a casos atípicos.

Cons

- Random Forest tiene una baja interpretabilidad, ofuscada por la selección de atributos aleatorizada.

- Tiene una complejidad sustancial dependiendo de la cantidad de datos.

Referencias

- Breiman, Leo. 2001. Random Forests.
- Zhou, 2012. Ensembles Methods. Foundations and Algorithms.
- Hastie, T; Tibshirani, R; Friedman, J. 2009. The Elements of Statistical Learning.
- Kearns y Valliant. 1989. Cryptographic limitations on Learning Boolean Formulae and Finite Automata.