Combinación de modelos de regresión:

- **Carga y ajuste de la base de datos MI**: Cargar los datos. Realizar los procedimientos estadísticos necesarios que considere necesarios para entener la base de datos.
- **Combinación paralela de clasificadores**: Estudio de diferentes métodos de combinación de clasificadores en paralelo: árboles de decisión, Bagging, and Boosting.
- **Combinación secuencial de clasificadores**: Estudio de diferentes métodos de combinación de clasificadores en secuencia: KNN, SVM, Stacking, y Cascading.
- **Conclusión de los métodos de clasificación aplicados a los problema**: Analizar cuál de las herramientas de clasificación utilizadas es la mejor para el problema del planteado.

Infarto de Miocardio (IM):

El Infarto de Miocardio (IM) es uno de los problemas más desafiantes de la medicina moderna. El infarto agudo de miocardio se asocia con una alta mortalidad en el primer año posterior al mismo. La incidencia de IM sigue siendo alta en todos los países. Esto es especialmente cierto para la población urbana de países altamente desarrollados, que está expuesta a factores de estrés crónico, nutrición irregular y no siempre equilibrada. El curso de la enfermedad en pacientes con infarto de miocardio es diferente. El IM puede ocurrir sin complicaciones o con complicaciones que no empeoran el pronóstico a largo plazo. Al mismo tiempo, aproximadamente la mitad de los pacientes en los períodos agudo y subagudo tienen complicaciones que conducen al empeoramiento de la enfermedad e incluso a la muerte. Incluso un especialista experimentado no siempre puede prever el desarrollo de estas complicaciones. En este sentido, la predicción de las complicaciones del infarto de miocardio para llevar a cabo oportunamente las medidas preventivas necesarias es una tarea importante.

In [1]:
```
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
import scipy.stats
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn import ensemble
from sklearn import svm

%matplotlib inline
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
import os
os.chdir('/content/drive/My Drive/')
```

Combinación de clasificadores:

El ensemble learning es una estrategia en la que se utiliza un grupo de modelos para resolver un problema mediante la combinación estratégica de diversos modelos de aprendizaje automático en un sólo modelo predictivo.

En general, los métodos de ensemble se utilizan principalmente para mejorar la precisión del rendimiento general de un modelo y combinar varios modelos diferentes, también conocidos como aprendices básicos, para predecir los resultados, en lugar de utilizar un sólo modelo.

¿Por qué entrenamos tantos clasificadores diferentes en lugar de uno solo? Bueno, el uso de varios modelos para predecir el resultado final en realidad reduce la probabilidad de sopesar las decisiones tomadas por modelos deficientes (sobreentrenados, no debidamente ajustados...).

Cuanto más diversos sean estos aprendices básicos, más poderoso será el modelo final.

Una vez llegados a este punto, podemos dividir los ensembles en cuatro categorías:

1. **Bagging**: El bagging se utiliza principalmente para reducir la variación en un modelo. Un ejemplo simple de bagging es el algoritmo Random Forest.
2. **Boosting**: El boosting se utiliza principalmente para reducir el sesgo en un modelo. Ejemplos de algoritmos de impulso son Ada-Boost, XGBoost, árboles de decisión mejorados por gradiente, etc.
3. **Stacking**: el stacking se utiliza principalmente para aumentar la precisión de predicción de un modelo.
4. **Cascading**: esta clase de modelos son muy precisos. La conexión en cascada se usa principalmente en escenarios en los que no puede permitirse cometer un error. Por ejemplo, una técnica en cascada se usa principalmente para detectar transacciones fraudulentas con tarjetas de crédito.

Datos:

La base de datos original consta de 1700 pacientes con 124 variables registradas. Existen valores faltantes y outlayers. En esta base de datos original, las columnas 2-112 son los descriptores o variables para predecir posibles complicaciones o respuestas que están en las columnas 113-124. La descripción detallada y los datos están disponibles en el siguiente enlace.

Es una base de datos lista para trabajar. Debeís cargar una base de datos llamada *MI_database_final.csv*. Los valores faltantes ya están imputados y los *outliers* han sido eliminados e imputados. Igualmente las variables han sido reducidas a las más relevantes teniendo en cuenta el outcome que será **ZSN_A**. Es una variable categórica Significa la presencia de Insuficiencia Cardíaca (IC) crónica en la anamnesis.

In [ ]:
```
datos = pd.read_csv('MI_database_final-1.csv')
```

In [ ]:
```
datos.head()
```

Out[ ]:

	ID	L_BLOOD	AGE	ALT_BLOOD	K_BLOOD	ROE	S_AD_ORIT	AST_BLOOD	Na_BLOOD	TIME_B_S	...	GIPER_Na	ritm_ecg_p_04	fibr_ter_03	GT_POST	fibr_ter_06	np09	zab_leg_04	n_r_ecg_p_02	n_p_ecg_p_05	ZSN_A
0	1	8.0	77.0	0.38	4.7	16.0	180.0	0.22	138.0	4.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	2	7.8	55.0	0.38	3.5	3.0	120.0	0.18	132.0	2.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	3	10.8	52.0	0.30	4.0	10.0	180.0	0.11	132.0	3.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	4	8.0	68.0	0.75	3.9	10.0	120.0	0.37	146.0	2.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
4	5	8.3	60.0	0.45	3.5	10.0	160.0	0.22	132.0	9.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 104 columns

Para poder probar varios modelos, primero vamos a dividir el dataset entre train y test. El outcome es **ZSN_A**.

In [ ]:
```
# Fijamos una semilla
seed = 13

# Dividimos nuestro dataframe en X e y para separar la variable objetivo del resto
X = datos.drop('ZSN_A', axis=1)
y= datos[['ZSN_A']]

# Realizamos la división en subconjuntos de prueba y test con random_state=13
# (la seed) asignando el 40% al subconjunto de prueba (test) dejando así el 60%
# restante para el conjunto de entrenamiento (train)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
                                                    random_state=seed)

# Comprobamos el resultado
print('Dimensiones de X_train: {}'.format(X_train.shape))
print('Dimensiones de y_train: {}'.format(y_train.shape))
print('Dimensiones de X_test: {}'.format(X_test.shape))
print('Dimensiones de y_test: {}'.format(y_test.shape))
```

```
Dimensiones de X_train: (1020, 103)
Dimensiones de y_train: (1020, 1)
Dimensiones de X_test: (680, 103)
Dimensiones de y_test: (680, 1)
```

Combinación paralela de clasificadores:

Árboles de decisión:

Para poder comparar el aumento de rendimiento obtenido a medida que vamos aplicando técnicas nuevas, utilizaremos como baseline un simple árbol de decisión.

In [ ]:
```
# Definimos el árbol de decisión con profundidad máxima de 3 niveles
arbol = DecisionTreeClassifier(max_depth=3, random_state=seed)

# Evaluamos el árbol de decisión con validación cruzada en el
# subconjunto de entrenamiento aplicando 5 conjuntos (parámetro cv)
validacion_cruzada = cross_val_score(arbol, X_train, y_train, cv=5,
                                     scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada: \n{}".format(validacion_cruzada))
print("\nPrecisión promedio: {}".format(validacion_cruzada.mean()))

# Entrenamos el modelo (árbol de decisión) en el subconjunto de
# entrenamiento completo
arbol.fit(X_train, y_train)

# Hacemos predicciones en el subconjunto de prueba
predicciones_arbol = arbol.predict(X_test)

# Evaluar el rendimiento en el conjunto de prueba
precision_arbol = accuracy_score(y_test, predicciones_arbol)
print("\nPrecisión en el Conjunto de Prueba: {}".format(precision_arbol))
```

```
Precisiones con validación cruzada:
[0.91176471 0.90196078 0.90196078 0.89705882 0.89705882]

Precisión promedio: 0.9019607843137255

Precisión en el Conjunto de Prueba: 0.8735294117647059
```

Hemos obtenido buenas precisiones en la validación curzada con el subconjunto de entrenamiento, en cada iteración por separado el resultado ha oscilado en torno al 90% (que ha sido la precisión promedio obtenida en este caso), entre la mejor iteración y la peor, sólo ha habido una diferencia de un 1.3% para la precisión aproximadamente.

En cuanto al subconjunto de prueba, la precisión obtenida en la validación final ha sido del 87%, lo cual solo difiere en un 3% respecto al resultado obtenido para el subconjunto de entrenamiento.

En general hemos obtenido resultados bastante buenos.

Bagging:

La idea central del bagging es usar réplicas del conjunto de datos original y usarlas para entrenar diferentes clasificadores.

Crearemos subconjuntos muestreando aleatoriamente un montón de puntos del conjunto de datos de entrenamiento con reemplazo.

Ahora entrenaremos clasificadores individuales en cada uno de estos subconjuntos bootstrap.

Cada uno de estos clasificadores base predecirá la etiqueta de clase para un problema dado. Aquí es donde combinamos las predicciones de todos los modelos base. Esta parte se llama etapa de agregación. Es por eso que encontraréis los ensembles bagging por el nombre de ensembles de agregación.

In [ ]:
```
# Importamos la función RandomForestClassifier de sklearn
from sklearn.ensemble import RandomForestClassifier


# Primero debemos convertir las variables outcome de los subconjuntos de
# entrenamiento y prueba en arrays de una dimensión antes de poder entrenar
# el modelo Random Forest, de lo contrario obtendremos un warning indeseado
y_train_array = y_train['ZSN_A'].to_numpy()
y_test_array = y_test['ZSN_A'].to_numpy()

# Definimos el Random Forest con 20 árboles y profundidad máxima de 3 niveles
# manteniendo la seed de apartados anteriores
bosque = RandomForestClassifier(n_estimators=20, max_depth=3, random_state=seed)

# Evaluamos el Random Forest con validación cruzada en el
# subconjunto de entrenamiento
validacion_cruzada_bosque = cross_val_score(bosque, X_train, y_train_array,
                                            cv=5, scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada para Random Forest: \n{}"
      .format(validacion_cruzada_bosque))
print("\nPrecisión promedio: {}".format(validacion_cruzada_bosque.mean()))

# Entrenamos el modelo Random Forest en el subconjunto de entrenamiento completo
bosque.fit(X_train, y_train_array)

# Hacemos las predicciones sobre el conjunto de prueba
bosque_predicciones = bosque.predict(X_test)

# Evaluamos la precisión de nuestro modelo en el subconjunto de prueba
bosque_precision = accuracy_score(y_test_array, bosque_predicciones)
print("\nPrecisión en el subconjunto de prueba para Random Forest: {}"
      .format(bosque_precision))
```

```
Precisiones con validación cruzada para Random Forest:
[0.90686275 0.90686275 0.90686275 0.90196078 0.90196078]

Precisión promedio: 0.9049019607843137

Precisión en el subconjunto de prueba para Random Forest: 0.8808823529411764
```

Vemos que el método Random Forest, al promediar los resultados entre varios modelos de árbol simple, obtiene resultados más consistentes entre cada iteración. Esto lo vemos ya que en cada iteración la precisión es siempre 90% con una variación muy pequeña (por debajo del orden de los decimales).

En cuanto al resultado obtenido para el subconjunto de prueba, tenemos una precisión del 88% (lo cual sólo difiere en un 2% de la precisión obtenida para el subconjunto de entrenamiento).

Hemos obtenido resultados bastante buenos como era de esperar, ya que en el modelo de árbol de decisión simple tuvimos un resultado similar, en esta ocasión, mejorando la precisión en un 1%.

Boosting:

El boosting se utiliza para convertir a los clasificadores de base débil en fuertes. Los clasificadores débiles generalmente tienen una correlación muy débil con las etiquetas de clase verdaderas y los clasificadores fuertes tienen una correlación muy alta entre el modelo y las etiquetas de clase verdaderas.

El boosting capacita a los clasificadores débiles de manera iterativa, cada uno tratando de corregir el error cometido por el modelo anterior. Esto se logra entrenando un modelo débil en todos los datos de entrenamiento, luego construyendo un segundo modelo que tiene como objetivo corregir los errores cometidos por el primer modelo. Luego construimos un tercer modelo que intenta corregir los errores cometidos por el segundo modelo y así sucesivamente. Los modelos se agregan de forma iterativa hasta que el modelo final ha corregido todos los errores cometidos por todos los modelos anteriores.

Cuando se agregan los modelos en cada etapa, se asignan algunos pesos al modelo que está relacionado con la precisión del modelo anterior. Después de agregar un clasificador débil, los pesos se vuelven a ajustar. Los puntos clasificados incorrectamente reciben pesos más altos y los puntos clasificados correctamente reciben pesos más bajos. Este enfoque hará que el siguiente clasificador se centre en los errores cometidos por el modelo anterior.

El boosting reduce el error de generalización tomando un modelo de alto bias y baja varianza y reduciendo el bias en un nivel significativo. Recuerde, el bagging reduce la varianza. Al igual que el bagging, el boosting también nos permite trabajar con modelos de clasificación y regresión.

In [ ]:
```
# Importamos la función GradientBoostingClassifier de sklearn
from sklearn.ensemble import GradientBoostingClassifier


# Volveremos a usar las variables y_train_array e y_test_array nuevamente
# como ya hicimos en el caso anterior para Bagging con Random Forest

# Definimos el  modelo Gradient Boosting Classifier con 20 árboles similares
# a los del caso anterior y la misma seed
boosting = GradientBoostingClassifier(n_estimators=20,
                                      max_depth=3, random_state=seed)

# Evaluamos el Gradient Boosting Classifier con validación cruzada en el
# subconjunto de entrenamiento
validacion_cruzada_boosting = cross_val_score(boosting, X_train, y_train_array,
                                              cv=5, scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada para Gradient Boosting:\n{}"
      .format(validacion_cruzada_boosting))
print("\nPrecisión promedio: {}".format(validacion_cruzada_boosting.mean()))

# Entrenamos el Gradient Boosting Classifier en el subconjunto
# de entrenamiento completo
boosting.fit(X_train, y_train_array)

# Hacemos las predicciones sobre el subconjunto de prueba
predicciones_boosting = boosting.predict(X_test)

# Evaluamos la precisión de nuestro modelo en el subconjunto de prueba
precision_boosting = accuracy_score(y_test_array, predicciones_boosting)
print("\nPrecisión en el subconjunto de prueba para Gradient Boosting: {}"
      .format(precision_boosting))
```

```
Precisiones con validación cruzada para Gradient Boosting:
[0.90196078 0.8872549  0.90196078 0.8872549  0.8872549 ]

Precisión promedio: 0.8931372549019606

Precisión en el subconjunto de prueba para Gradient Boosting: 0.8705882352941177
```

En cuanto a las precisiones en las diferentes iteraciones del modelo, vemos que son resultados buenos pero no tan estables como el caso anterior. La precisión promedio en este caso es del 89% (inferior que en el caso anterior que ya comentamos para el Bagging).

En cuanto a la precisión del Grading Boosting sobre el subconjunto de prueba, hemos obtenido una precisión del 87% (lo cual difiere sólo en un 2% respecto al subconjunto de entrenamiento). Si comparamos el resultado obtenido en este caso con el anterior para el Bagging, hemos obtenido una precisión peor en un 1%.

Por último, hemos notado que el tiempo de respuesta ha sido ligeramente mayor con respecto al Bagging con Random Forest que realizamos antes. En base a esto, para este conjunto de datos, el método Gradient Boosting aunque obtiene buenos resultados no es óptimo ya que existe al menos un método mejor (el Bagging con Random Forest).

Combinación secuencial de clasificadores base diferentes:

Para poder hacer combinación secuencial de modelos, necesitamos tener varios modelos diferentes entrenados. En nuestro caso, ya tenemos un árbol de decisión. Vamos a entrenar algunos de modelos más.

KNN (K-nearest neighbors):

In [ ]:
```
# Definimos el modelo k-neighbors con 2 vecinos
k_vecinos = KNeighborsClassifier(n_neighbors=2)

# Volveremos a usar las variables y_train_array e y_test_array nuevamente

# Evaluamos el k-neighbors Classifier con validación cruzada en el
# subconjunto de entrenamiento
validacion_cruzada_k = cross_val_score(k_vecinos, X_train, y_train_array,
                                       cv=5, scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada para k-neighbors:\n{}"
      .format(validacion_cruzada_k))
print("\nPrecisión promedio: {}".format(validacion_cruzada_k.mean()))

# Entrenamos el k-neighbors con el subconjunto de entrenamiento completo
k_vecinos.fit(X_train, y_train_array)

# Hacemos las predicciones en el subconjunto de prueba
predicciones_k = k_vecinos.predict(X_test)

# Evaluamos la precisión de nuestro modelo en el subconjunto de prueba
precision_k = accuracy_score(y_test_array, predicciones_k)
print("\nPrecisión en el subonjunto de prueba para k-neighbors: {}"
      .format(precision_k))
```

```
Precisiones con validación cruzada para k-neighbors:
[0.89705882 0.89705882 0.89215686 0.8872549  0.88235294]

Precisión promedio: 0.8911764705882353

Precisión en el subonjunto de prueba para k-neighbors: 0.8779411764705882
```

SVM (Support Vector Machines):

In [ ]:
```
# Definimos el SVM con gamma = 0.07
modelo_svm = svm.SVC(gamma=0.07)

# Evaluamos el SVM con validación cruzada sobre el subconjunto de entrenamiento
validacion_cruzada_svm = cross_val_score(modelo_svm, X_train, y_train_array,
                                         cv=5, scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada para SVM:\n{}"
      .format(validacion_cruzada_svm))
print("\nPrecisión promedio: {}".format(validacion_cruzada_svm.mean()))

# Entrenamos el modelo SVM con el subconjunto de entrenamiento completo
modelo_svm.fit(X_train, y_train_array)

# Hacemos las predicciones en el subconjunto de prueba
predicciones_svm = modelo_svm.predict(X_test)

# Evaluamos la precisión de nuestro modelo en el subconjunto de prueba
precision_svm = accuracy_score(y_test_array, predicciones_svm)
print("\nPrecisión en el subconjunto de prueba para SVM: {}"
      .format(precision_svm))
```

```
Precisiones con validación cruzada para SVM:
[0.90686275 0.90686275 0.90686275 0.90196078 0.90196078]

Precisión promedio: 0.9049019607843137

Precisión en el subconjunto de prueba para SVM: 0.8808823529411764
```

Stacking:

Todos los modelos individuales se entrenan por separado en el conjunto completo de datos de entrenamiento y se ajustan para lograr una mayor precisión. La compensación de bias y varianza se tiene en cuenta para cada modelo. El modelo final, también conocido como metaclasificador, se alimenta de las etiquetas de clase predichas por los modelos base o de las probabilidades predichas para cada etiqueta de clase. Luego, el metaclasificador se entrena en función de los resultados dados por los modelos base.

En el stacking, se entrena un nuevo modelo en función de las predicciones realizadas por los modelos anteriores. Este proceso se lleva a cabo de forma secuencial. Esto significa que varios modelos se entrenan en la etapa 1 y se ajustan con precisión. Las probabilidades pronosticadas de cada modelo de la etapa 1 se alimentan como entrada a todos los modelos en la etapa 2. Los modelos en la etapa 2 luego se ajustan con precisión y las salidas correspondientes se alimentan a los modelos en la etapa 3 y así sucesivamente. Este proceso se produce varias veces en función de la cantidad de capas de apilamiento que desee utilizar.

In [ ]:
```
# Tenemos ya definidos los clasificadores base, recordemos:
# arbol, k_vecinos y modelo_svm

# Tenemos las predicciones con cada uno de ellos, recordemos:
# predicciones_arbol, predicciones_k y predicciones_svm

# Juntamos todas las predicciones usando column_stack de numpy
predicciones_stacked = np.column_stack((predicciones_arbol,
                                      predicciones_k, predicciones_svm))

# Definimos el clasificador Gradient Boosting Classifier
modelo_stacked = GradientBoostingClassifier(n_estimators=20, max_depth=3,
                                            random_state=seed)

# Evaluamos el modelo con validación cruzada sobre el subconjunto de prueba
validacion_cruzada_stacked = cross_val_score(modelo_stacked,
                                             predicciones_stacked, y_test_array,
                                             cv=5, scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones de la validación cruzada con el modelo stacked:\n{}"
      .format(validacion_cruzada_stacked))
print("\nPrecisión promedio: {}".format(validacion_cruzada_stacked.mean()))
```

```
Precisiones de la validación cruzada con el modelo stacked:
[0.88235294 0.88235294 0.875      0.88235294 0.86764706]

Precisión promedio: 0.8779411764705882
```

En este caso la precisión ha sido similar a emplear cada uno de los modelos de clasificadores por separado, ya sea con Bagging o Boosting. Al menos para este conjunto de datos, no merece la pena el aumento en el coste computacional si no obtenemos una mejoría sustancial en la precisión del modelo ya que, por ejemplo, haciendo Bagging con Random Forest ya obtuvimos una precisión similar (88%).

Cascading:

El caso de cascading es parecido al de stacking pero utilizando no solamente las predicciones parciales de los clasificadores base, sino también los datos originales.

In [ ]:
```
# Tenemos ya definidos los clasificadores base, recordemos:
# arbol, k_vecinos y modelo_svm

# Tenemos las predicciones con cada uno de ellos, recordemos:
# predicciones_arbol, predicciones_k y predicciones_svm

# Juntamos todas las predicciones juntadas usando column_stack de numpy, pero
# teniendo en cuenta también el subconjunto X_test
predicciones_cascading = np.column_stack((X_test, predicciones_arbol,
                                          predicciones_k, predicciones_svm))


# Definimos el clasificador (Gradient Boosting Classifier)
modelo_cascading = GradientBoostingClassifier(n_estimators=20, max_depth=3,
                                              random_state=seed)

# Evaluamos el modelo cascading con validación curzada sobre el
# subconjunto de prueba
validacion_cruzada_cascading = cross_val_score(modelo_cascading,
                                               predicciones_cascading,
                                               y_test_array, cv=5,
                                               scoring='accuracy')

# Mostramos los resultados de la validación cruzada
print("Precisiones con validación cruzada con el modelo cascading):\n{}"
      .format(validacion_cruzada_cascading))
print("\nPrecisión promedio: {}".format(validacion_cruzada_cascading.mean()))
```

```
Precisiones con validación cruzada con el modelo cascading):
[0.86764706 0.88970588 0.875      0.85294118 0.85294118]

Precisión promedio: 0.8676470588235293
```

Obtenemos en este caso un resultado incluso peor (aunque sólo por un 1%) respecto al stacking.

Conclusiones:

Tenemos las siguientes precisiones promedio para el subconjunto de prueba en cada caso:

- Árbol de decisión simple: 87.35%
- Bagging (Random Forest): 90.49%
- Boosting (Random Forest): 87.05%
- KNN: 89.11%
- SVM: 88.08%
- Stacking (árbol + knn + svm): 87.79%
- Cascading (árbol + knn + svm): 86.76%

Vamos a representar gráficamente estos valores:

In [46]:
```
# Importamos el módulo matplotlib.pyplot para crear el gráfico
import matplotlib.pyplot as plt

# Listamos en un array las precisiones obtenidas en cada caso para el
# subconjunto de prueba
precisiones_metodos = np.array([0.8735, 0.9049, 0.8705, 0.8911, 0.8808, 0.8779,
                                0.8676])
# Listamos los nombres de cada método respetando el orden en el que hemos
# listado sus respectivas precisiones anteriormente
nombres_metodos = ['Árbol', 'Bagging (RF)', 'Boosting (RF)', 'KNN', 'SVM',
                   'Stacking', 'Cascading']

# Creamos un gráfico de barras
plt.bar(nombres_metodos, precisiones_metodos, color='lightgreen')

# Agregamos etiquetas a los ejes y título al gráfico
plt.xlabel('Método')
plt.xticks(rotation=45)
plt.ylabel('Precisión')
plt.ylim(0, 1)
plt.title('Precisiones de cada método de clasificación')
# Agregamos una línea que  ayude a destacar la precisión máxima de entre todas
# las obtenidas con los distintos métodos
plt.axhline(y=np.max(precisiones_metodos), color='red', linestyle='--')
# Resaltamos con un texto lo mismo
plt.text(s='Precisión máxima', x=precisiones_metodos[2],
         y=np.max(precisiones_metodos)+0.01, color='red')

# Mostramos el gráfico
plt.show()
```

Como podemos ver, para este conjunto de datos, el mejor resultado lo hemos obtenido mediante una combinación paralela de clasificadores siguiendo una estrategia Bagging usando un modelo de Random Forest (concretamente obtuvimos una precisión del 90.49%).

Además de obtener el mejor resultado, consideramos que es el método más simple e intuitivo de combinación de clasificadores y también el que conlleva un gasto computacional menor, sobre todo si lo comparamos con Boosting (donde el tiempo de ejecución de la celda con el script de dicho método fue ligeramente mayor al menos en nuestro equipo) o con métodos de combinación secuencial como lo son Stacking y Cascading que precisan de diferentes clasificadores base entrenados previamente.