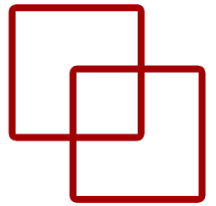
Fase de modelado y optimización

Dr. Manuel Castillo-Cara
www.manuelcastillo.eu
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática
Universidad Nacional de Educación a Distancia (UNED)

Preliminar



-

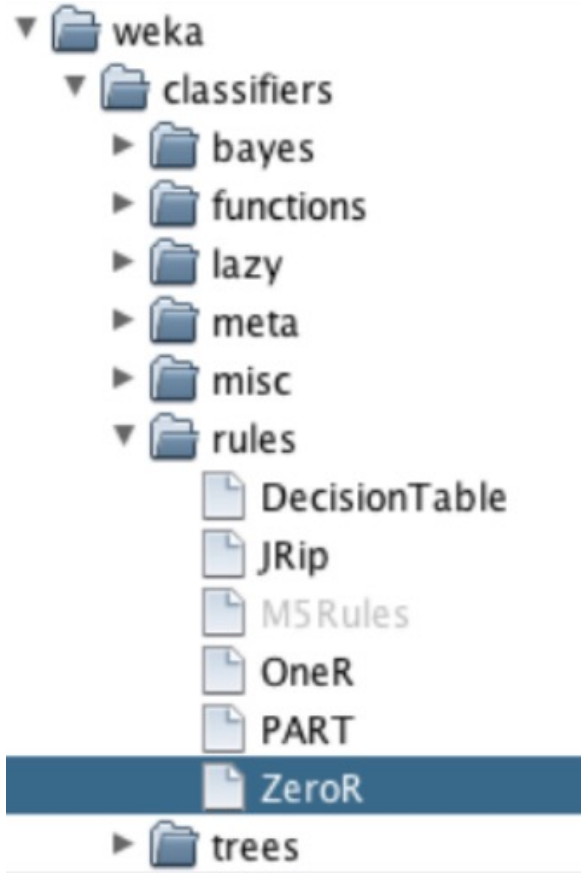
Índice

- Algoritmos de Machine Learning.
  - Algoritmos lineales.
  - Algoritmos no lineales.
- Rendimiento de los algoritmos.
- Algoritmos ensamblados.
  - Bagging.
  - Boosting.
  - Voting.
- Algoritmo Super Lerner

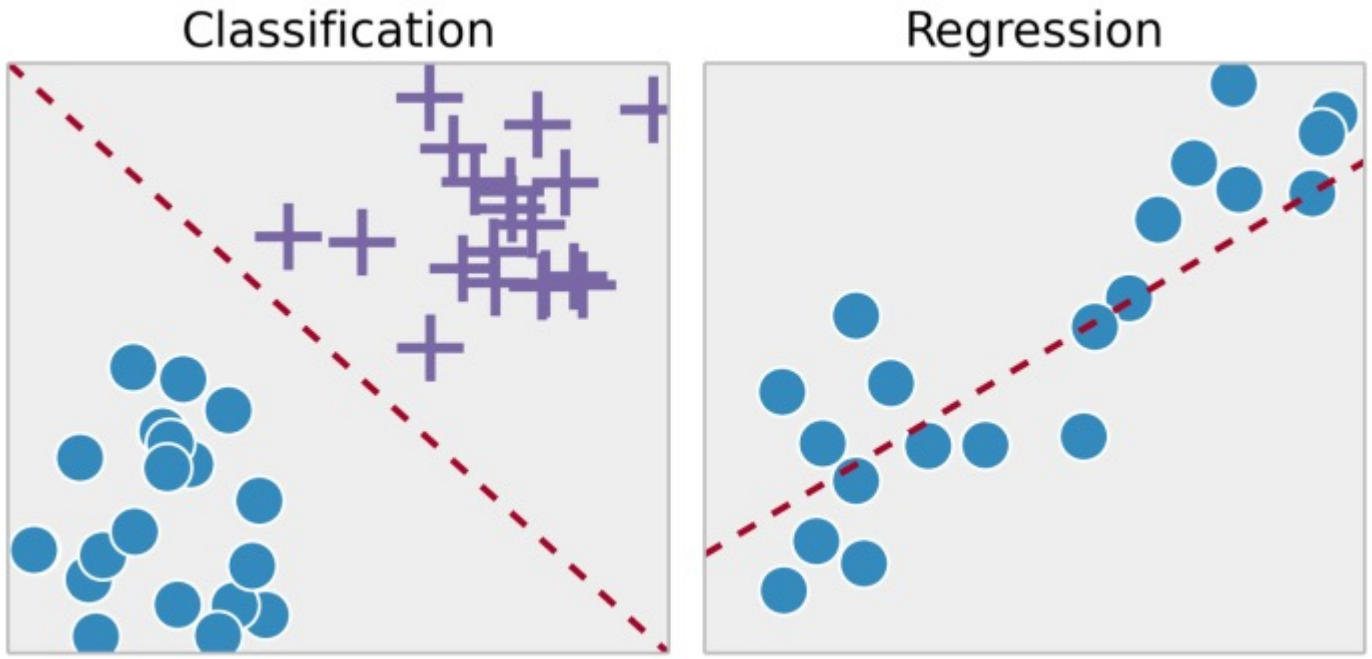
Algoritmos de Machine Learning

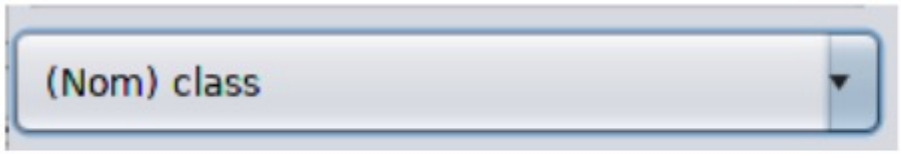
Algoritmos


Machine Learning Algorithms
Deep Learning
Deep Boltzmann Machine (DBM)
Deep Belief Networks (DBN)
Convolutional Neural Network (CNN)
Stacked Auto-Encoders
Ensemble
Random Forest
Gradient Boosting Machines (GBM)
Boosting
Bootstrapped Aggregation (Bagging)
AdaBoost
Stacked Generalization (Blending)
Gradient Boosted Regression Trees (GBRT)
Neural Networks
Radial Basis Function Network (RBFN)
Perceptron
Back-Propagation
Hopfield Network
Regularization
Ridge Regression
Least Absolute Shrinkage and Selection Operator (LASSO)
Elastic Net
Least Angle Regression (LARS)
Rule System
Cubist
One Rule (OneR)
Zero Rule (ZeroR)
Repeated Incremental Pruning to Produce Error Reduction (RIPPER)
Regression
Linear Regression
Ordinary Least Squares Regression (OLSR)
Stepwise Regression
Multivariate Adaptive Regression Splines (MARS)
Locally Estimated Scatterplot Smoothing (LOESS)
Logistic Regression
Bayesian
Naive Bayes
Averaged One-Dependence Estimators (AODE)
Bayesian Belief Network (BBN)
Gaussian Naive Bayes
Multinomial Naive Bayes
Bayesian Network (BN)
Decision Tree
Classification and Regression Tree (CART)
Iterative Dichotomiser 3 (ID3)
C4.5
C5.0
Chi-squared Automatic Interaction Detection (CHAID)
Decision Stump
Conditional Decision Trees
M5
Dimensionality Reduction
Principal Component Analysis (PCA)
Partial Least Squares Regression (PLSR
Sammon Mapping
Multidimensional Scaling (MDS)
Projection Pursuit
Principal Component Regression (PCR)
Partial Least Squares Discriminant Analysis
Mixture Discriminant Analysis (MDA)
Quadratic Discriminant Analysis (QDA)
Regularized Discriminant Analysis (RDA)
Flexible Discriminant Analysis (FDA)
Linear Discriminant Analysis (LDA)
Instance Based
k-Nearest Neighbour (kNN)
Learning Vector Quantization (LVQ)
Self-Organizing Map (SOM)
Locally Weighted Learning (LWL)
Clustering
k-Means
k-Medians
Expectation Maximization
Hierarchical Clustering

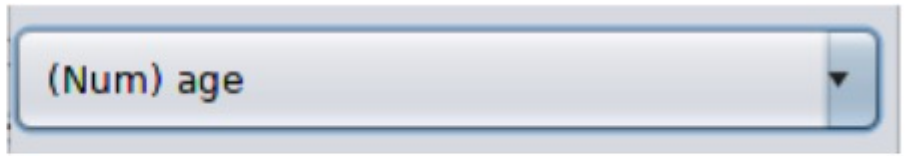

weka
classifiers
bayes
functions
lazy
meta
misc
rules
DecisionTable
JRip
M5Rules
OneR
PART
ZeroR
trees

Clasificación Vs. Regresión





(a) Muestra de un atributo nominal.



(b) Muestra de un atributo numérico.

Taxonomía de los algoritmos

- **Lineales**: el valor objetivo se exprese como una combinación lineal de valores constantes o el producto entre un parámetro y una variable predictiva.
- **No lineales**: no se utilizan funciones como en los lineales.
- **Ensamblados**: combinan las predicciones de múltiples modelos para hacer predicciones más robusta.

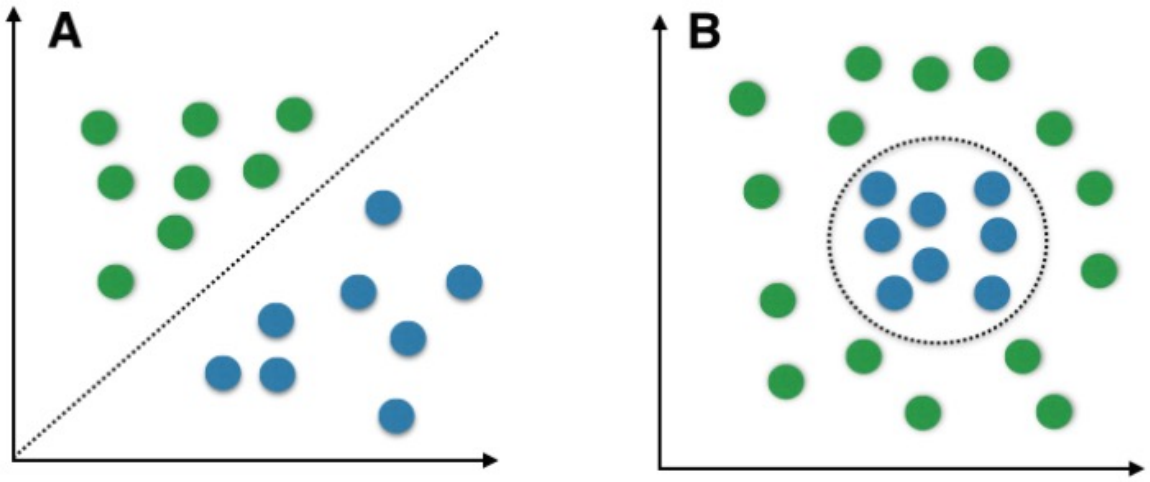
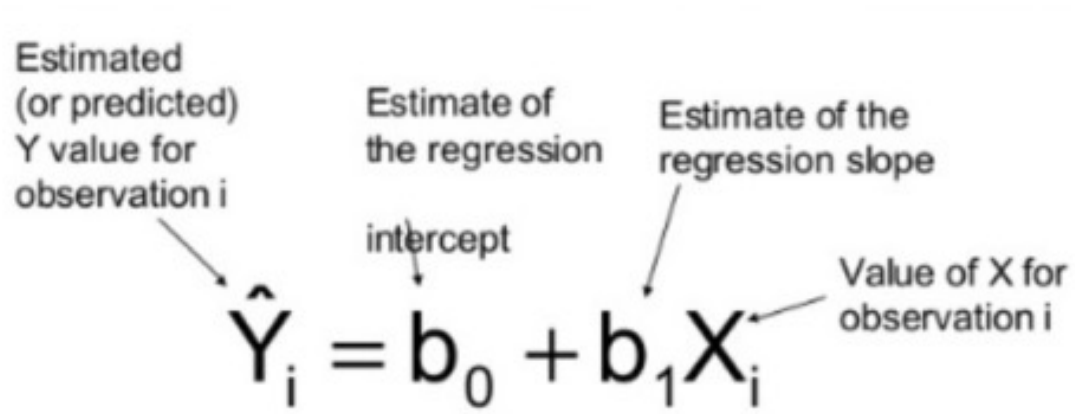


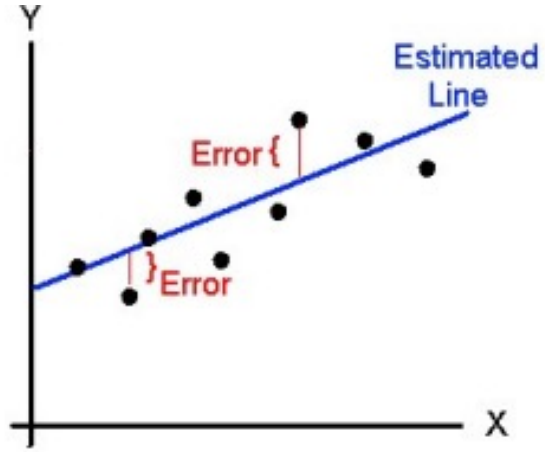
FIGURA 5.19: Algoritmos lineales Vs. No lineales.

Algoritmos lineales

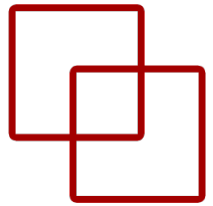
Linear Regression

- Para problemas de regresión.
- Es una aproximación para modelar la relación entre una variable escalar dependiente '*y*' y una o mas variables explicativas nombradas con '*X*'.
- En otras palabras, este modelo lo que realiza es "dibujar una recta" que nos indicará la tendencia de un conjunto de datos continuos.
- Se utiliza la clase *LinearRegression*.





Linear Regression - código

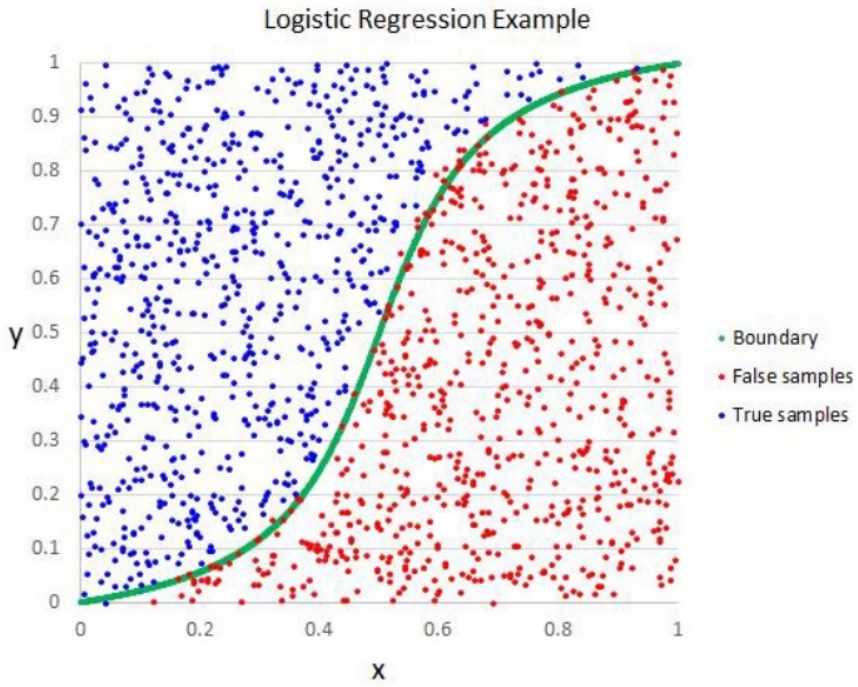


```
# Linear Regression
from sklearn.linear_model import LinearRegression

kfold = KFold(n_splits=10, random_state=7)
model = LinearRegression()
scoring = 'neg_mean_squared_error'
results = cross_val_score(model, X_reg, Y_reg, cv=kfold, scoring=scoring)
print(f"MSE: {results.mean()}")
```

```
MSE: -34.70525594452488
```

Logistic Regression

- Para problemas de clasificación binaria.
- Este modelo ayuda a determinar si la entrada pertenece a un sector específico.
- Utiliza la función sigmoide que tiene un rango de valores de salida entre 0 y 1.
- Se utiliza la clase *LogisticRegression*.

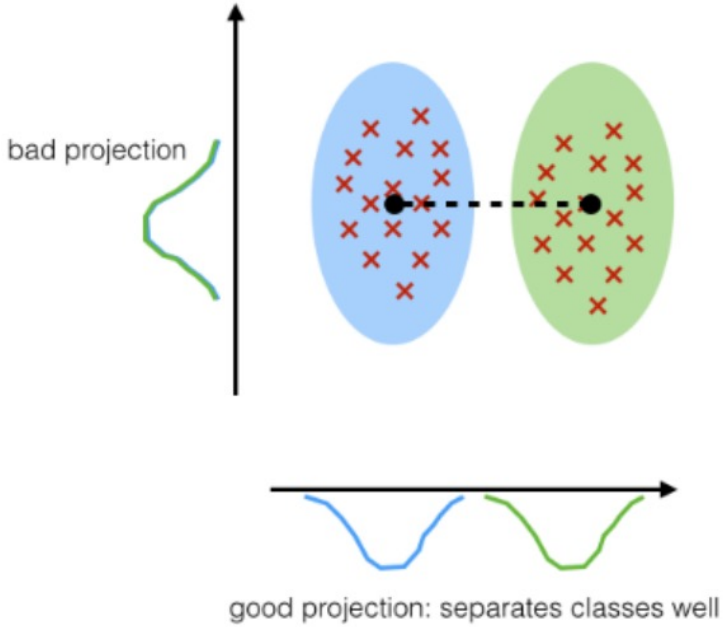



```
# Logistic Regression Classification
from sklearn.linear_model import LogisticRegression

num_folds = 10
kfold = KFold(n_splits=10, random_state=7)
model = LogisticRegression(solver = 'lbfgs', max_iter=1000)
results = cross_val_score(model, X_cla, Y_cla, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 77.60% (5.16%)
```

Linear Discriminant Analysis

- Para problemas de clasificación.
- También supone una distribución gaussiana para las variables de entrada numéricas.
- Se utiliza la clase *LinearDiscriminantAnalysis*



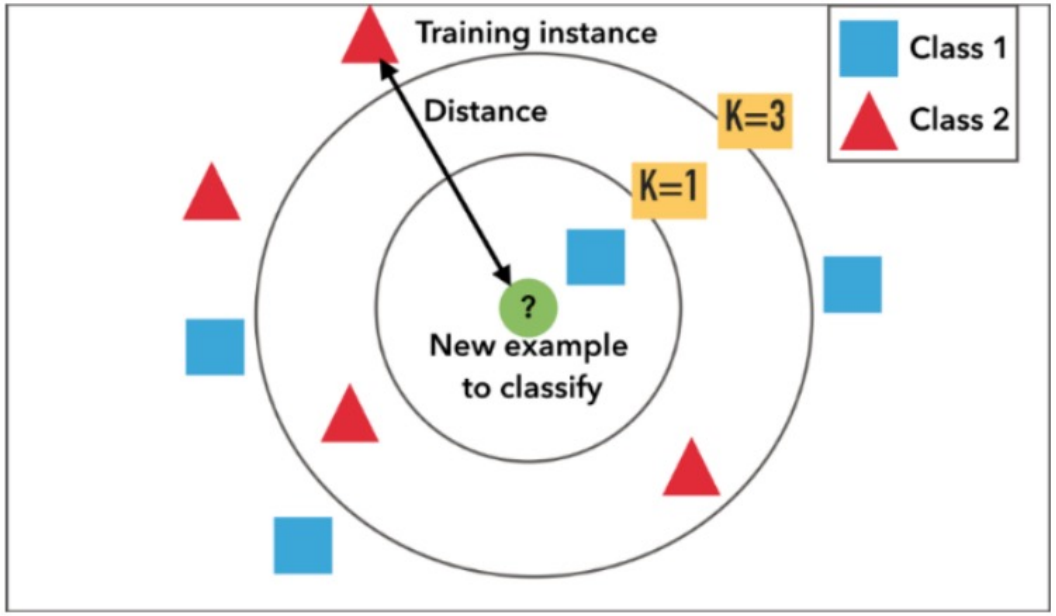

```
# LDA Classification
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

num_folds = 10
kfold = KFold(n_splits=10, random_state=7)
model = LinearDiscriminantAnalysis()
results = cross_val_score(model, X_cla, Y_cla, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 77.35% (5.16%)
```

Algoritmos no lineales

k-Nearest Neighbors

- Clasifica la entrada basándose en una medida de similitud, que a menudo es la distancia en el espacio de los puntos de datos.
- Se hace una predicción eligiendo la clase más frecuente entre los *k* vecinos más cercanos.
- Clasificación: *KNeighborsClassifier*
- Regresión: *KNeighborsRegressor*



k-NN – código

CLASIFICACIÓN

```
# KNN Classification
from sklearn.neighbors import KNeighborsClassifier

num_folds = 10
kfold = KFold(n_splits=10, random_state=7)
model = KNeighborsClassifier()
results = cross_val_score(model, X_cla, Y_cla, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 72.66% (6.18%)
```

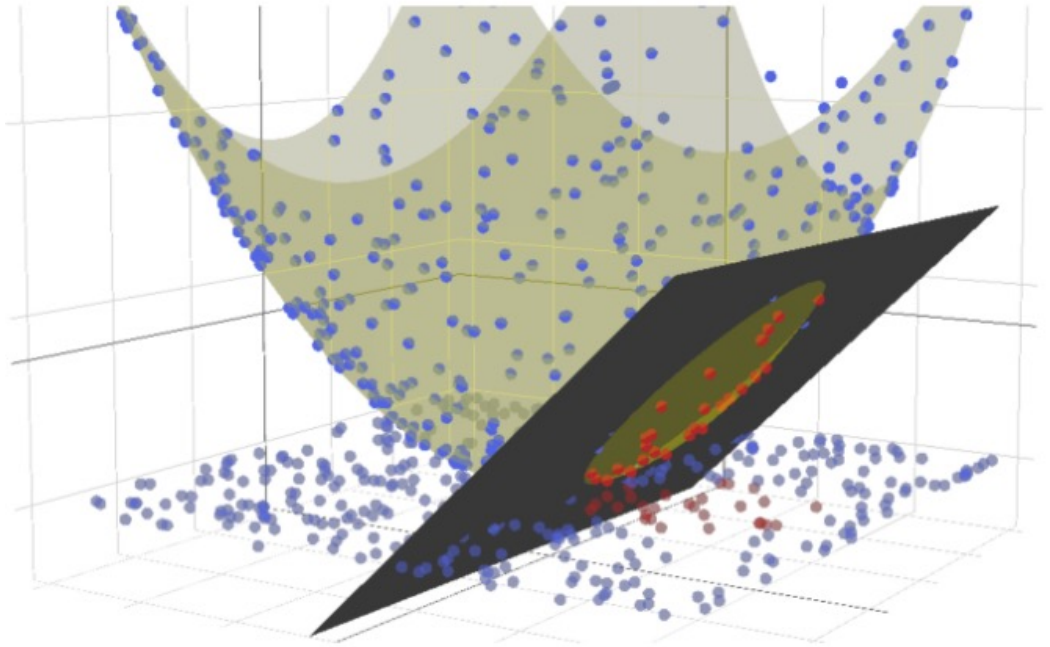
REGRESIÓN

```
# k-NN Regressionore
from sklearn.neighbors import KNeighborsRegressor

kfold = KFold(n_splits=10, random_state=7)
model = KNeighborsRegressor()
scoring = 'neg_mean_squared_error'
results = cross_val_score(model, X_reg, Y_reg, cv=kfold, scoring=scoring)
print(f"MSE: {results.mean()}")
```

```
MSE: -107.28683898039215
```

Support Vector Machine

- Dados los datos en el espacio, SVM construye hiperplanos en un espacio de alta dimensión con una brecha máxima entre ellos.
- Con la ayuda de las funciones del kernel, puede realizar la clasificación de datos de alta dimensión.
- Clasificación: *SVC*
- Regresión: *SVR*



SVM – código

CLASIFICACIÓN

```
# SVM Classification
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

kfold = KFold(n_splits=10, random_state=7)
model = SVC(gamma='scale')
results = cross_val_score(model, X_cla, Y_cla, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 76.04% (5.29%)
```

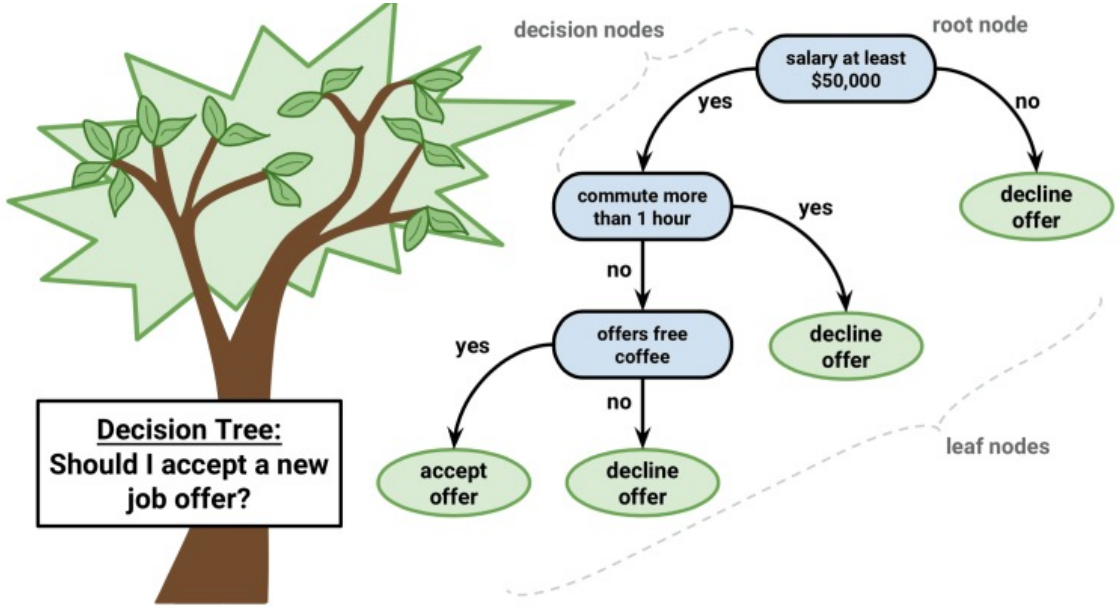
REGRESIÓN

```
# SVM Regression
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVR

num_folds = 10
kfold = KFold(n_splits=10, random_state=7)
model = SVR(gamma='auto')
scoring = 'neg_mean_squared_error'
results = cross_val_score(model, X_reg, Y_reg, cv=kfold, scoring=scoring)
print(f"MSE: {results.mean()}")
```

```
MSE: -91.04782433324428
```

Classification & Regression Trees

- Son modelos predictivos que coloca las observaciones realizadas a partir de los datos en las ramas; estos conducen a las hojas que están etiquetadas con la clasificación correcta.
- Utiliza un conjunto discreto de valores, y las hojas producen el resultado final.
- Tienen mejor comportamiento con atributos discretos (dummy, categóricos) → Es recomendable convertir si están los atributos en numéricos.
- Clasificación: *DecisionTreeClassifier*
- Regresión: *DecisionTreeRegressor*



CART – código

CLASIFICACIÓN

```
# CART Classification
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier

kfold = KFold(n_splits=10, random_state=7)
model = DecisionTreeClassifier()
results = cross_val_score(model, X_cla, Y_cla, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 70.83% (5.90%)
```

REGRESIÓN

```
# CART Regression
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeRegressor

kfold = KFold(n_splits=10, random_state=7)
model = DecisionTreeRegressor()
scoring = 'neg_mean_squared_error'
results = cross_val_score(model, X_reg, Y_reg, cv=kfold, scoring=scoring)
print(f"MSE: {results.mean()}")
```

```
MSE: -38.80579843137255
```

Rendimiento de los algoritmos

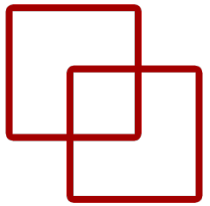
Evaluar el rendimiento

- Cómo comparar la habilidad del modelo usando la tabla resumen.
- Cómo revisar y comparar habilidades de modelos usando diferentes gráficos.
- Cómo comparar la habilidad del modelo usando gráficos entre pares de ellos.
- Cómo verificar si la diferencia en la habilidad del modelo es estadísticamente significativa.

Escoger el mejor modelo

- **Preparar el conjunto de datos**.
  - Proceso de carga de los paquetes y el conjunto de datos para entrenar a los modelos.
- **Resultados el modelo**.
  - Entrenar modelos estándar de machine learning en el conjunto de datos para su evaluación.
- **Comparar los modelos**.
  - Comparar los modelos entrenados usando 8 técnicas diferentes.

Preparar el conjunto de datos



LIBRERÍAS

```
#importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

DATASET

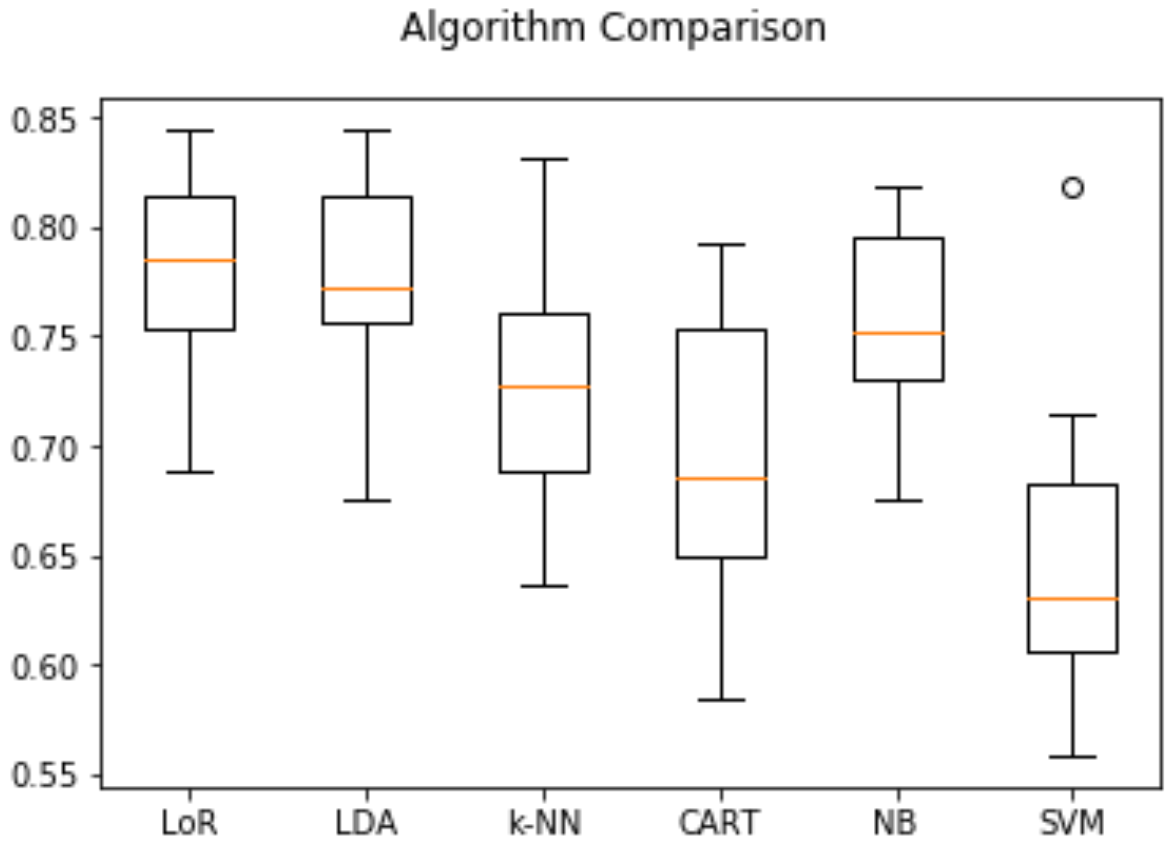
```
# Clasification problem
filename = 'data/pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pd.read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
```

Resultados de los modelos

```
# Compare Algorithms
# prepare models
models = []
models.append(('LoR', LogisticRegression(solver='lbfgs', max_iter=1000)))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('k-NN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = KFold(n_splits=10, random_state=7)
    cv_results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    print(f"{name}: {cv_results.mean()*100.0:,.2f}% ({cv_results.std()*100.0:,.2f}%)")
```

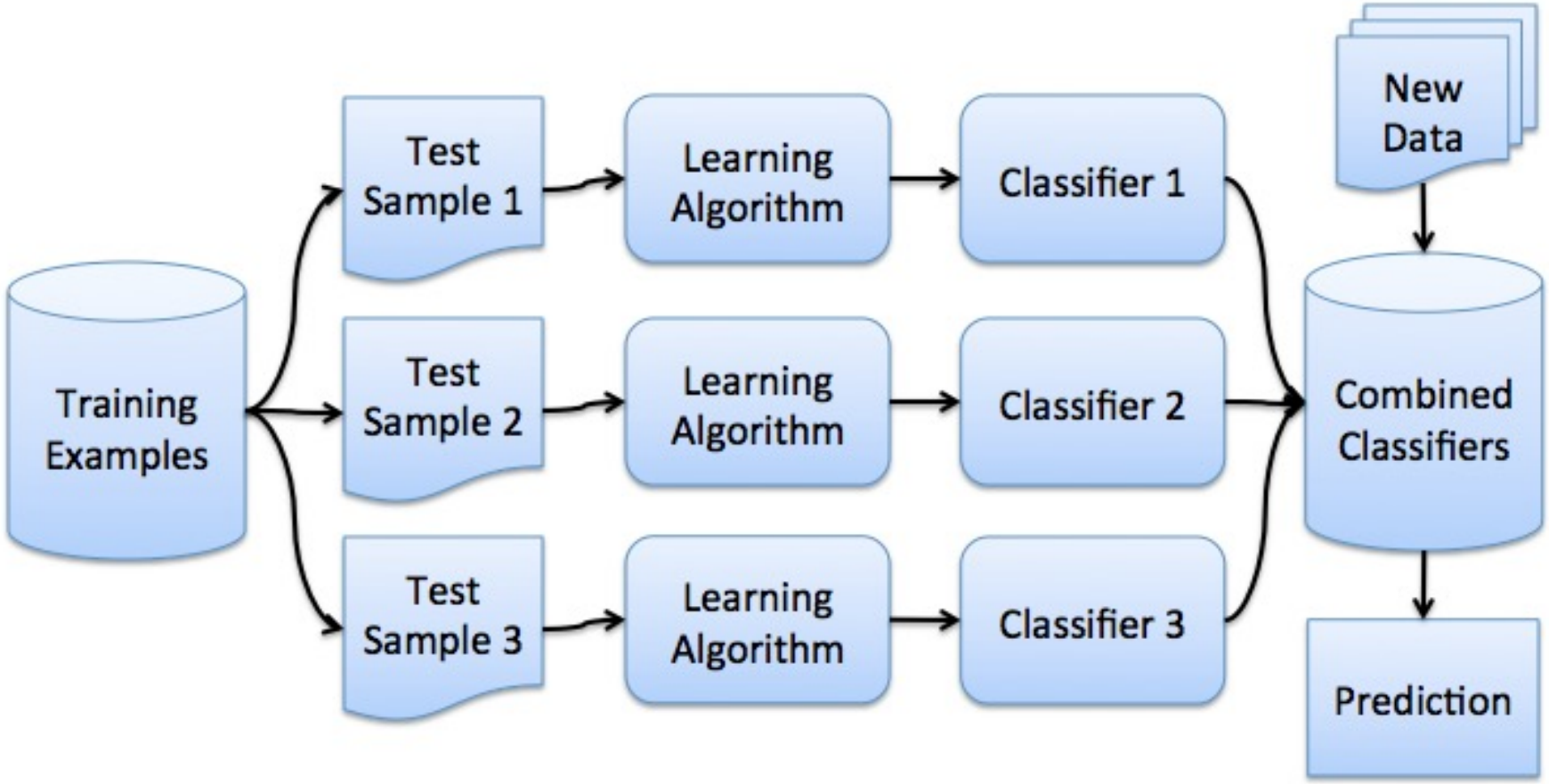
```
LoR: 77.60% (5.16%)
LDA: 77.35% (5.16%)
k-NN: 72.66% (6.18%)
CART: 69.52% (6.47%)
NB: 75.52% (4.28%)
SVM: 65.10% (7.21%)
```

Visualizar resultados



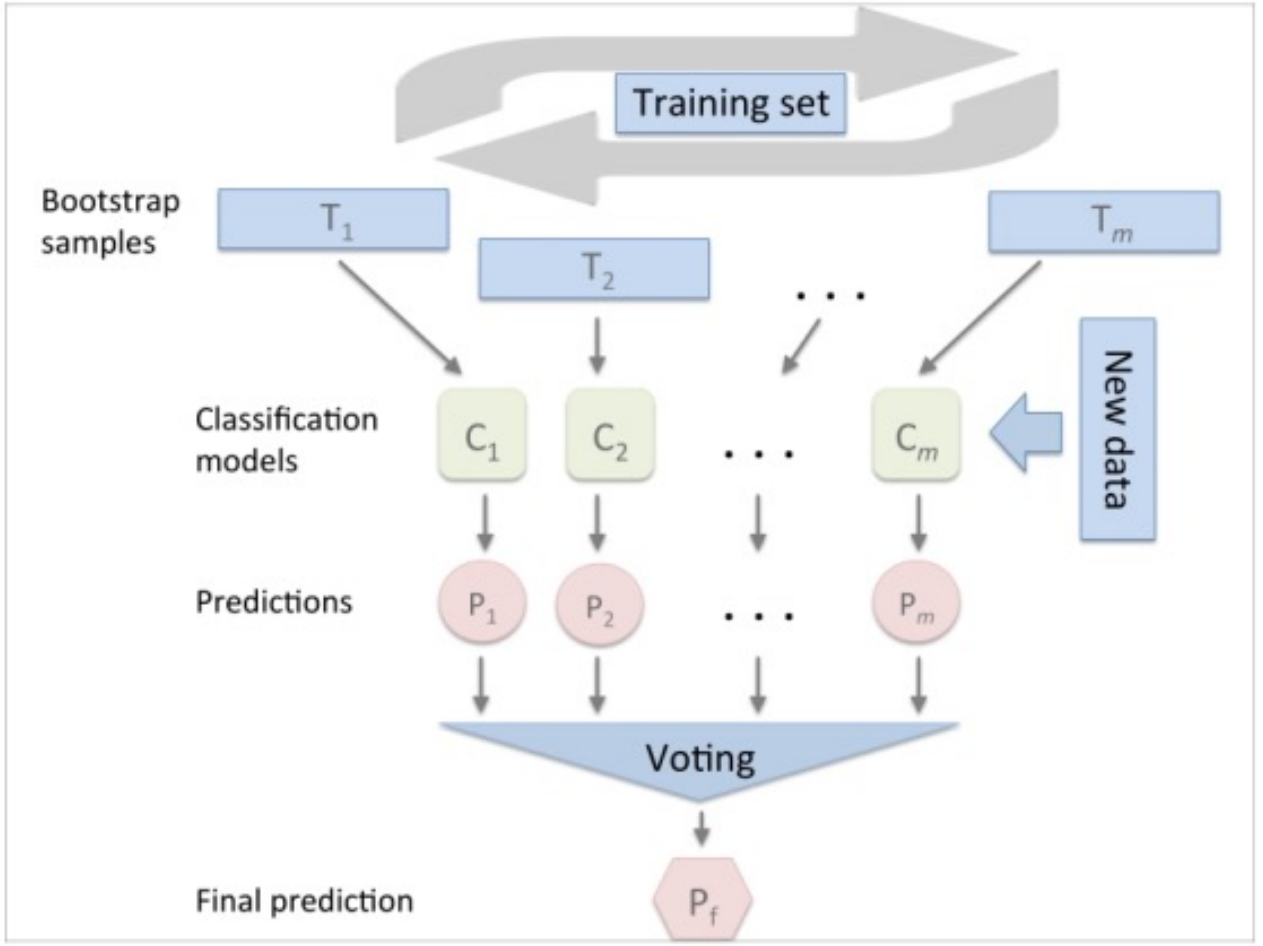
Algoritmos ensamblados

Algoritmos de conjunto

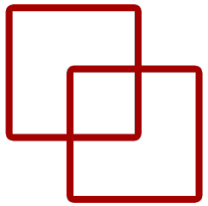


Bagging

- Construye múltiples modelos (típicamente modelos del mismo tipo) a partir de diferentes submuestras del conjunto de datos de entrenamiento. Algoritmos:
  - Bagged Decision Trees
  - Random Forest
  - Extra Trees



Random Forest



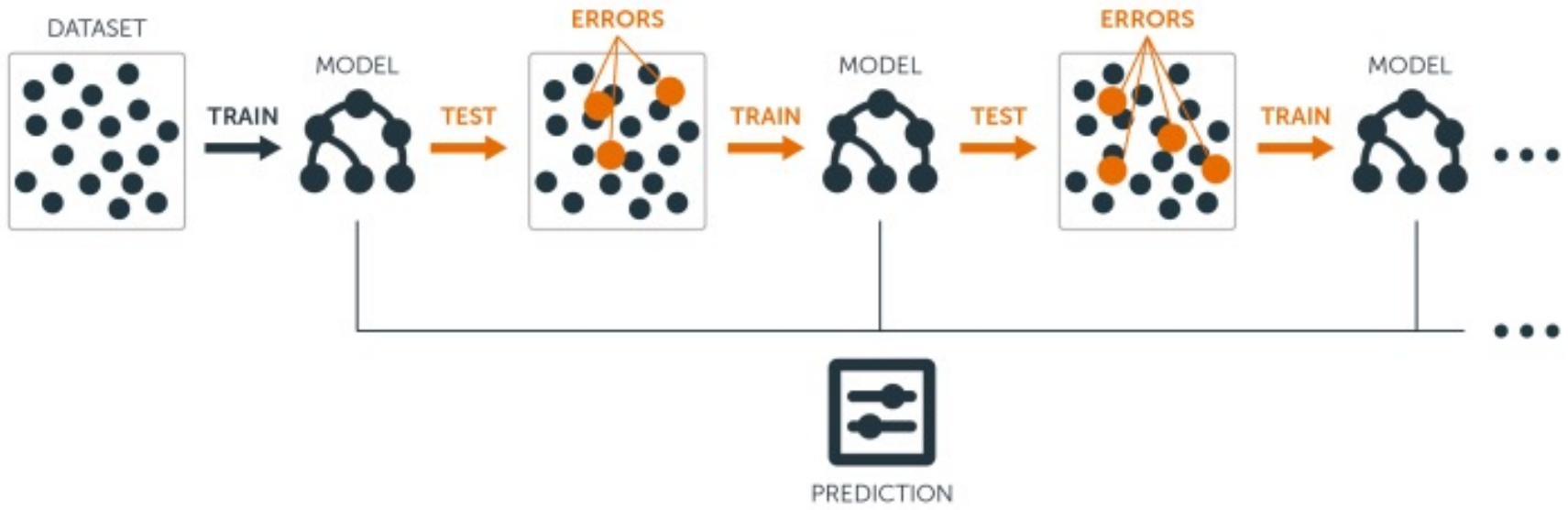
- Las muestras del conjunto de datos de entrenamiento se toman con reemplazo, pero los árboles se construyen de una manera que reduce la correlación entre clasificadores individuales.
  - Específicamente, en lugar de elegir con avidez el mejor punto de división en la construcción de cada árbol, solo se considera un subconjunto aleatorio de características para cada división.
- Puede construir un modelo de Random Forest para la clasificación utilizando la clase *RandomForestClassifier*.
- El siguiente ejemplo se construye 100 árboles y puntos divididos elegidos de una selección aleatoria de 3 características.

```
# Random Forest Classification
from sklearn.ensemble import RandomForestClassifier

num_trees = 100
max_features = 3
kfold = KFold(n_splits=10, random_state=7)
model = RandomForestClassifier(n_estimators=num_trees, max_features=max_features)
results = cross_val_score(model, X, Y, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 77.34% (7.62%)
```

Boosting

- Construye múltiples modelos (típicamente modelos del mismo tipo), cada uno de los cuales aprende a corregir los errores de predicción de un modelo anterior en la cadena.
  - AdaBoost
  - Stochastic Gradient Boosting



AdaBoost

- AdaBoost fue quizás el primer algoritmo de conjunto Boosting exitoso.
- Generalmente funciona ponderando las instancias en el conjunto de datos según lo fácil o difícil que es clasificarlas, lo que permite que el algoritmo les preste más o menos menos atención en la construcción de modelos posteriores.
- Puede construir un modelo AdaBoost para clasificación utilizando la clase *AdaBoostClassifier*.
- El siguiente ejemplo demuestra la construcción de 30 árboles de decisión en secuencia.

```
# AdaBoost for Classification
from sklearn.ensemble import AdaBoostClassifier

num_trees = 30
seed=7
kfold = KFold(n_splits=10, random_state=seed)
model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 76.05% (5.44%)
```

Fase de optimización

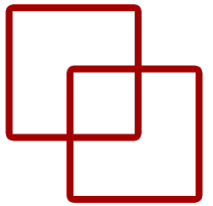
Modelo de linea base

- Antes de buscar los resultados de los mejores hiperparámetros debemos de conocer el resultado que nos dá el modelo que estemos utilizando como línea base.
- La idea de buscar hiperparámetros es mejorar el resultado predictivo que nos dé el modelo.
- En este caso estamos utilizando un algoritmo RiR que no tiene demasiados hiperparámetros por lo que no va a mejorar mucho.
- Sin embargo, algoritmos de taxonomía no lineal mejoran muchísimo conforme configuramos sus hiperparámetros.

```
# RiR Classification
num_folds = 10
kfold = KFold(n_splits=5, random_state=7)
model = Ridge()
results = cross_val_score(model, X, Y, cv=kfold)
print(f"Accuracy: {results.mean()*100.0:,.2f}% ({results.std()*100.0:,.2f}%)")
```

```
Accuracy: 27.61% (1.61%)
```

Grid Search

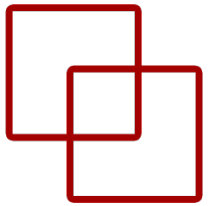


- Es un enfoque para el ajuste de parámetros que construirá y evaluará metódicamente un modelo para cada combinación de parámetros de algoritmo especificados en una cuadrícula (grid),
- Utiliza la clase *GridSearchCV*.
- El siguiente ejemplo evalúa diferentes valores alpha, siendo el óptimo el 1

```
# Grid Search for Algorithm Tuning
from sklearn.model_selection import GridSearchCV

alphas = np.array([1,0.1,0.01,0.001,0.0001,0])
param_grid = dict(alpha=alphas)
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
grid.fit(X, Y)
print(f"Accuracy óptimo: {grid.best_score_.mean()*100.0:,.2f}%")
print(f"Valor de alpha óptimo: {grid.best_estimator_.alpha}")
```

```
Accuracy óptimo: 27.61%
Valor de alpha óptimo: 1.0
```

Random Search



- Realizar una búsqueda aleatoria de parámetros
- Utiliza la clase *RandomizedSearchCV*.
- El siguiente ejemplo evalúa diferentes valores alpha aleatorios entre 0 y 1, siendo el óptimo 0.97.

```
# Random Search for Algorithm Tuning
from scipy.stats import uniform
from sklearn.model_selection import RandomizedSearchCV

param_grid = {'alpha': uniform()}
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_iter=100, random_state=7, cv=5)
rsearch.fit(X, Y)
print(f"Accuracy óptimo: {rsearch.best_score_.mean()*100.0:,.2f}%")
print(f"Valor de alpha óptimo: {rsearch.best_estimator_.alpha}")
```

```
Accuracy óptimo: 27.61%
Valor de alpha óptimo: 0.9779895119966027
```

¡Gracias!

Dr. Manuel Castillo-Cara
www.manuelcastillo.eu
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática
Universidad Nacional de Educación a Distancia (UNED)