SESIÓN ALGORITMOS DE CLASIFICACIÓN

CONTENIDOS:

- Regresión logística: en qué consiste la regresión logística.
    - La función sigmoídea.
    - Ventajas y desventajas de utilizar el algoritmo y su implementación en python.
- K-Nearest Neighbors: en qué consiste el algoritmo K-NN.
    - El concepto de escalamiento de datos.
    - El concepto de distancia euclideana.
    - Selección del valor de k y la exactitud.
- Árbol de decisión: qué es un árbol de decisión.
    - Principales hiperparámetros de un árbol de decisión.
    - Medidas de impureza de los nodos.
    - Ventajas y desventajas y su implementación en python.
- Bosques aleatorios: el concepto de “bagging”.
- Support vector machine: qué es una máquina de soporte de vectores.
    - Cómo funciona el clasificador svm.
    - En qué tipo de problema puede utilizarse.
    - Tipos de kernel y sus características.

REGRESIÓN LOGÍSTICA

La regresión logística es un algoritmo de aprendizaje supervisado utilizado principalmente para problemas de **clasificación binaria**, es decir, cuando queremos predecir una de dos clases posibles (por ejemplo, sí/no, 1/0, verdadero/falso). Aunque su nombre incluye "regresión", no se usa para predecir valores continuos, sino para estimar la probabilidad de que una observación pertenezca a una clase específica.

¿En qué consiste?

1. **Objetivo**: Predecir la probabilidad de que una observación pertenezca a una clase (por ejemplo, la probabilidad de que un correo sea spam).
2. **Salida**: La salida de la regresión logística es un valor entre 0 y 1, que representa la probabilidad de que la observación pertenezca a la clase positiva (por ejemplo, 1).
3. **Función de decisión**: Si la probabilidad es mayor que un umbral (generalmente 0.5), se clasifica como clase positiva; de lo contrario, como clase negativa.

La función sigmoidea

La **función sigmoidea** es una función matemática que transforma cualquier valor real en un rango entre 0 y 1. Es clave en la regresión logística porque convierte la salida de un modelo lineal en una probabilidad.

Fórmula de la función sigmoidea:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Ilustración 1 Fórmula función sigmoidea.

Donde:

- z es la salida de un modelo lineal (por ejemplo, $z = b_0+b_1x_1+b_2x_2+\cdots+b_nx_n$).
- e es la base del logaritmo natural (aproximadamente 2.718).

Características:

- Cuando z es grande y positivo, $\sigma(z)$ se acerca a 1.
- Cuando z es grande y negativo, $\sigma(z)$ se acerca a 0.
- Cuando $z = 0$, $\sigma(z)=0.5$.

VENTAJAS Y DESVENTAJAS DE LA REGRESIÓN LOGÍSTICA

Ventajas:

1. **Simplicidad**: Es fácil de implementar y entender.
2. **Eficiencia**: Requiere menos recursos computacionales comparado con algoritmos más complejos.
3. **Interpretabilidad**: Los coeficientes del modelo pueden interpretarse en términos de su impacto en la probabilidad de la clase positiva.
4. **Funciona bien con datos linealmente separables**: Es efectivo cuando la relación entre las variables independientes y la variable dependiente es aproximadamente lineal.
5. **Probabilidades**: Proporciona directamente la probabilidad de pertenecer a una clase, lo que es útil en muchos contextos.

Desventajas:

1. **Limitación a relaciones lineales**: No funciona bien si la relación entre las variables no es lineal.
2. **Sensibilidad a outliers**: Los valores atípicos pueden afectar significativamente el modelo.
3. **Overfitting**: Puede ocurrir si hay demasiadas variables independientes o si el modelo es demasiado complejo.
4. **No es adecuado para clasificación multiclase**: Aunque se puede extender (por ejemplo, con One-vs-Rest o Softmax), no es su fortaleza.

Implementación en Python

En Python, la regresión logística se puede implementar fácilmente utilizando bibliotecas como **scikit-learn**. Aquí podemos ver un ejemplo básico:

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Datos de ejemplo
X = [[1, 2], [2, 3], [3, 4], [4, 5]]  # Variables independientes
y = [0, 0, 1, 1]  # Variable dependiente (clases)

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

# Crear el modelo de regresión logística
modelo = LogisticRegression()

# Entrenar el modelo
modelo.fit(X_train, y_train)

# Predecir con el modelo
y_pred = modelo.predict(X_test)

# Evaluar el modelo
print("Precisión:", accuracy_score(y_test, y_pred))
print("Matriz de confusión:", confusion_matrix(y_test, y_pred))
```

Salida:

```
Precisión: 0.0
Matriz de confusión: [[0 1]
 [0 0]]
```

Ilustración 2 Salida regresión logística.

K-NEAREST NEIGHBORS: EN QUÉ CONSISTE EL ALGORITMO K-NN

El algoritmo K-Nearest Neighbors **(K-NN)** es un método de aprendizaje supervisado utilizado tanto para problemas de **clasificación** como de **regresión**. Es un algoritmo basado en instancias, lo que significa que no aprende un modelo explícito a partir de los datos de entrenamiento, sino que utiliza los datos directamente para hacer predicciones.

¿En qué consiste?

1. **Objetivo**: Dado un nuevo dato (punto), el algoritmo encuentra los k*k* puntos más cercanos en el conjunto de entrenamiento y utiliza sus etiquetas para predecir la etiqueta del nuevo dato.
2. **Clasificación**: Para clasificación, la etiqueta más común entre los k*k* vecinos se asigna al nuevo dato.
3. **Regresión**: Para regresión, el valor predicho es el promedio de los valores de los k*k* vecinos.
4. **No paramétrico**: No hace suposiciones sobre la distribución de los datos.

El concepto de escalamiento de datos

El **escalamiento de datos** es un paso importante en la preparación de datos para el algoritmo K-NN, ya que este algoritmo es sensible a la escala de las variables.

¿Por qué es necesario?

- K-NN se basa en medidas de distancia (como la distancia euclidiana) para encontrar los vecinos más cercanos.
- Si las variables tienen escalas diferentes (por ejemplo, una variable está en metros y otra en kilómetros), la variable con valores más grandes dominará la distancia, lo que puede sesgar los resultados.

Métodos comunes de escalamiento:

1. **Normalización (Min-Max Scaling)**:
    - Transforma los valores al rango [0, 1].

Fórmula:

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Ilustración 3 Fórmula Min-Max Scaling.

Donde μ es la media y σ es la desviación estándar.

2. **Estandarización (Z-score Scaling)**:
    - Transforma los datos para que tengan media 0 y desviación estándar 1.

Fórmula:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

Ilustración 4 Fórmula Estandarización.

El concepto de distancia euclidiana

La distancia euclidiana es una medida de distancia entre dos puntos en un espacio multidimensional. Es la distancia "en línea recta" entre dos puntos.

Fórmula:

Para dos puntos $P=(p1,p2,\ldots,p_n)$ y $Q=(q1,q2,\ldots,q_n)$, la distancia euclidiana d se calcula como:

$$d(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

Ilustración 5 Fórmula distancia Euclidiana.

Selección del valor de K y la exactitud

El valor de *k* (número de vecinos) es un hiperparámetro crítico en K-NN. Su elección afecta directamente la precisión y el rendimiento del modelo.

Consideraciones:

1. ***k* pequeño (por ejemplo, *k*=1)**:
    - El modelo es muy sensible al ruido y a los outliers.
    - Puede causar **overfitting** (el modelo se ajusta demasiado a los datos de entrenamiento).
    - La frontera de decisión es muy irregular.
2. ***k* grande**:
    - El modelo es más suave y menos sensible al ruido.
    - Puede causar **underfitting** (el modelo es demasiado simple y no captura la complejidad de los datos).
    - La frontera de decisión es más suave.
3. **Elección óptima de *k***:
    - Se suele elegir mediante validación cruzada (cross-validation).
    - Un valor común es *k*=*n*, donde *n* es el número de muestras en el conjunto de entrenamiento.

Exactitud (*Accuracy*)

La **exactitud** es una métrica común para evaluar el rendimiento de un modelo de clasificación. Se define como la proporción de predicciones correctas sobre el total de predicciones.

Fórmula:

$$\text{Exactitud} = \frac{\text{Número de predicciones correctas}}{\text{Número total de predicciones}}$$

Ilustración 6 Fórmula Accuracy.

Relación con *k*:

- La elección de *k* afecta la exactitud del modelo.
- Un *k* mal elegido puede reducir la exactitud debido a overfitting o underfitting.

Implementación en Python

Aquí podemos ver un ejemplo básico de cómo implementar K-NN en Python usando **scikit-learn**:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Datos de ejemplo
X = [[1, 2], [2, 3], [3, 4], [4, 5]]  # Variables independientes
y = [0, 0, 1, 1]  # Variable dependiente (clases)

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

# Escalar los datos
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Crear el modelo K-NN (por ejemplo, k=3)
modelo = KNeighborsClassifier(n_neighbors=2)

# Entrenar el modelo
modelo.fit(X_train, y_train)

# Predecir con el modelo
y_pred = modelo.predict(X_test)

# Ver los datos escalados
print("X_train escalado:", X_train)
print("X_test escalado:", X_test)

# Ver las etiquetas reales y predichas
print("Etiqueta real (y_test):", y_test)
print("Etiqueta predicha (y_pred):", y_pred)

# Evaluar el modelo
print("Exactitud:", accuracy_score(y_test, y_pred))
```

Explicación del código:

1. **StandardScaler**: Escala los datos para que tengan media 0 y desviación estándar 1.
2. **KNeighborsClassifier**: Crea el modelo K-NN con k=3*k*=3.
3. **fit**: Entrena el modelo con los datos de entrenamiento.
4. **predict**: Realiza predicciones con el modelo entrenado.
5. **accuracy_score**: Evalúa la exactitud del modelo.

Salida esperada:

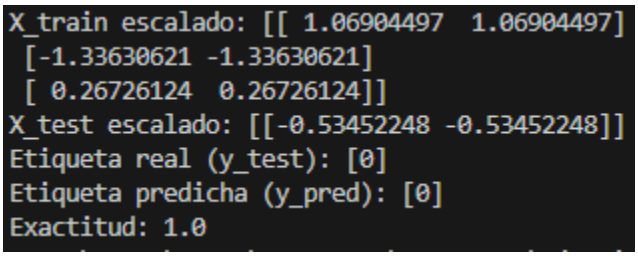

Ilustración 7 Salida esperada K-NN.

ÁRBOL DE DECISIÓN

Un árbol de decisión es un algoritmo de aprendizaje supervisado utilizado tanto para problemas de clasificación como de regresión. Es un modelo basado en reglas que divide recursivamente el conjunto de datos en subconjuntos más pequeños y homogéneos, utilizando condiciones sobre las características (variables independientes) para tomar decisiones.

¿Cómo funciona?

1. **Nodos**:
    - **Nodo raíz**: Representa el conjunto de datos completo.
    - **Nodos internos**: Representan decisiones basadas en una característica.
    - **Nodos hoja**: Representan el resultado final (clase o valor predicho).

2. **Divisiones**:
    - El árbol divide los datos en función de las características que maximizan la homogeneidad (pureza) de los subconjuntos resultantes.
    - Cada división se basa en una condición (por ejemplo, $x1 \leq 5$).
3. **Criterio de parada**:
    - El árbol deja de dividirse cuando se alcanza un criterio predefinido, como una profundidad máxima o un número mínimo de muestras en un nodo.

Principales Hiperparámetros de un Árbol de Decisión

Los hiperparámetros son configuraciones que controlan el comportamiento del modelo. Algunos de los más importantes en un árbol de decisión son:

1. **max_depth**:
    - Profundidad máxima del árbol.
    - Un valor más alto permite un árbol más complejo, pero puede causar overfitting.
    - Un valor más bajo limita la complejidad, pero puede causar underfitting.
2. **min_samples_split**:
    - Número mínimo de muestras requeridas para dividir un nodo.
    - Un valor más alto evita divisiones en nodos con muy pocas muestras.
3. **min_samples_leaf**:
    - Número mínimo de muestras requeridas en un nodo hoja.
    - Ayuda a evitar hojas con muy pocas muestras.
4. **criterion**:
    - Medida de impureza utilizada para evaluar las divisiones.
    - Para clasificación: "gini" o "entropy".
    - Para regresión: "mse" (error cuadrático medio).

5. **max_features**:
    - Número máximo de características consideradas para dividir un nodo.
    - Ayuda a controlar la complejidad y evitar overfitting.

Medidas de Impureza de los Nodos

Las medidas de impureza evalúan cuán mezcladas están las clases en un nodo. Un nodo es "puro" si todas las muestras pertenecen a la misma clase.

1. **Índice de Gini** (para clasificación):
    - Mide la probabilidad de clasificar incorrectamente una muestra aleatoria.
    - Fórmula:

$$\text{Gini} = 1 - \sum_{i=1}^{n} (p_i)^2$$

Ilustración 8 Fórmula Índice de Gini.

Donde p_i es la proporción de muestras de la clase *i* en el nodo.

    - Un valor de 0 indica pureza máxima.

2. **Entropía** (para clasificación):
    - Mide el desorden o incertidumbre en el nodo.
    - Fórmula:

$$\text{Entropía} = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Ilustración 9 Fórmula Entropía

    - Un valor de 0 indica pureza máxima.

3. **Error Cuadrático Medio (MSE)** (para regresión):
    - Mide la varianza de los valores predichos en el nodo.
    - Fórmula:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$$

Ilustración 10 Fórmula MSE.

Donde $\bar{y}$ es el valor promedio en el nodo.

Ventajas y Desventajas

Ventajas:

1. **Interpretabilidad**: Los árboles de decisión son fáciles de entender y visualizar.
2. **No requiere escalamiento**: No es necesario escalar las características.
3. **Manejo de datos mixtos**: Puede manejar tanto características numéricas como categóricas.
4. **No paramétrico**: No hace suposiciones sobre la distribución de los datos.

Desventajas:

1. **Propenso a overfitting**: Puede crear árboles muy complejos que se ajustan demasiado a los datos de entrenamiento.
2. **Inestabilidad**: Pequeños cambios en los datos pueden generar árboles muy diferentes.
3. **Sesgo hacia clases dominantes**: En problemas de clasificación desbalanceados, puede favorecer a la clase mayoritaria.
4. **Limitado para relaciones no lineales**: No es ideal para capturar relaciones complejas entre características.

En Python, puedes implementar un árbol de decisión utilizando la biblioteca **scikit-learn**. Aquí podemos ver un ejemplo básico:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score


# Cargar el conjunto de datos Iris
data = load_iris()
X = data.data  # Características
y = data.target  # Etiquetas

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

# Crear el modelo de árbol de decisión
modelo = DecisionTreeClassifier(max_depth=3, random_state=42)

# Entrenar el modelo
modelo.fit(X_train, y_train)

# Predecir con el modelo
y_pred = modelo.predict(X_test)

# Evaluar el modelo
print("Exactitud:", accuracy_score(y_test, y_pred))
```

Explicación del código:

1. **load_iris**: Carga el conjunto de datos Iris, que es un clásico conjunto de datos para clasificación.
2. **train_test_split**: Divide los datos en conjuntos de entrenamiento y prueba.
3. **DecisionTreeClassifier**: Crea el modelo de árbol de decisión con una profundidad máxima de 3.
4. **fit**: Entrena el modelo con los datos de entrenamiento.
5. **predict**: Realiza predicciones con el modelo entrenado.

6. **accuracy_score**: Evalúa la exactitud del modelo.

Salida esperada:

```
Exactitud: 1.0
```

Ilustración 11 Salida esperada Árbol de decisión.

BOSQUES ALEATORIOS

Los bosques aleatorios son un algoritmo de aprendizaje supervisado que se utiliza tanto para problemas de clasificación como de regresión. Es una técnica de ensamblado (*ensemble*) que combina múltiples árboles de decisión para mejorar la precisión y reducir el overfitting.

El Concepto de "Bagging"

El término bagging (*Bootstrap Aggregating*) es una técnica en la que se entrena un conjunto de modelos (en este caso, árboles de decisión) en diferentes subconjuntos de los datos de entrenamiento, y luego se combinan sus predicciones para obtener un resultado final más robusto.

1. **Bootstrap**:
    - Se crean múltiples subconjuntos de los datos de entrenamiento mediante muestreo con reemplazo. Esto significa que algunas muestras pueden aparecer varias veces en un subconjunto, mientras que otras pueden no aparecer.
2. **Entrenamiento**:
    - Cada árbol de decisión se entrena en uno de estos subconjuntos. Además, en cada división de un árbol, solo se considera un subconjunto aleatorio de características (variables independientes). Esto añade diversidad a los árboles.
3. **Agregación**:
    - Para clasificación, la predicción final es la moda (la clase más frecuente) de las predicciones de todos los árboles.

- Para regresión, la predicción final es el promedio de las predicciones de todos los árboles.

Ventajas de Bosques Aleatorios:

- **Reducción del overfitting**: Al combinar múltiples árboles, se suavizan las predicciones y se reduce la varianza.
- **Robustez**: Es menos sensible al ruido y a los outliers.
- **Importancia de características**: Proporciona una medida de la importancia de cada característica.

Desventajas de Bosques Aleatorios:

- **Menos interpretable**: Aunque los árboles individuales son interpretables, el bosque completo no lo es.
- **Mayor costo computacional**: Entrenar múltiples árboles puede ser más lento que entrenar un solo árbol.

Implementación en Python

Veamos un ejemplo sencillo de implementación “Bagging”:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Crear un conjunto de datos de ejemplo
X, y = make_classification(
    n_samples=500, n_features=5, n_informative=3, random_state=42
)

# Dividir el conjunto de datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

```python
# Crear un clasificador base (por ejemplo, un árbol de decisión)
base_clf = DecisionTreeClassifier()

# Implementar Bagging con el clasificador base
bagging_clf = BaggingClassifier(estimator=base_clf, n_estimators=10,
random_state=42)

# Entrenar el modelo de Bagging
bagging_clf.fit(X_train, y_train)

# Realizar predicciones
y_pred = bagging_clf.predict(X_test)

# Calcular la precisión
accuracy = accuracy_score(y_test, y_pred)
print(f"Precisión del modelo Bagging: {accuracy:.2f}")
```

Explicación código:

- **make_classification**: Crea datos de prueba sintéticos para simular un problema de clasificación.
- **BaggingClassifier**: Combina múltiples modelos base (árboles de decisión) para mejorar la precisión.

Salida esperada:

```
Precisión del modelo Bagging: 0.94
```

Ilustración 12 Salida esperada Bagging.

SUPPORT VECTOR MACHINE (SVM)

Una Máquina de Soporte de Vectores **(SVM)** es un algoritmo de aprendizaje supervisado utilizado principalmente para problemas de clasificación, aunque también puede usarse para regresión (en cuyo caso se llama **SVR**, *Support Vector Regression*).

¿Qué es una SVM?

- **Objetivo**: Encontrar un hiperplano (una frontera de decisión) que separe las clases de la manera más óptima posible.
- **Hiperplano**: En un espacio de n dimensiones, un hiperplano es un subespacio de $n-1$ dimensiones que divide el espacio en dos regiones.

¿Cómo funciona el clasificador SVM?

1. **Separación lineal**:
    - En un problema de clasificación binaria, SVM busca el hiperplano que maximiza el margen entre las dos clases. El margen es la distancia entre el hiperplano y los puntos más cercanos de cada clase (llamados **vectores de soporte**).
2. **Datos no linealmente separables**:
    - Si los datos no son linealmente separables, SVM utiliza un **kernel** para transformar los datos a un espacio de mayor dimensión donde sí sean linealmente separables.
3. **Vectores de soporte**:
    - Son los puntos de datos más cercanos al hiperplano. Estos puntos son críticos para definir la posición y orientación del hiperplano.
4. **Optimización**:
    - SVM resuelve un problema de optimización para encontrar el hiperplano óptimo. Este problema se formula como una minimización de una función objetivo sujeta a restricciones.

¿En qué tipo de problemas puede utilizarse?

- **Clasificación binaria**: SVM es muy efectivo para problemas de dos clases.
- **Clasificación multiclase**: Aunque SVM es inherentemente binario, se puede extender a problemas multiclase usando técnicas como **One-vs-One** o **One-vs-Rest**.
- **Regresión**: Con SVR, SVM también puede usarse para predecir valores continuos.

Tipos de Kernel y sus Características

En problemas no lineales, los datos deben transformarse a un espacio de mayor dimensionalidad donde sean linealmente separables, y los kernels son funciones que permiten a las **SVM** llevar a cabo esta transformación sin calcular explícitamente las nuevas dimensiones, utilizando un enfoque conocido como trucos del kernel. Veamos los tipos de Kernel más comunes:

1. **Kernel Lineal**:

No realiza ninguna transformación. Asume que los datos son linealmente separables en el espacio original.

Fórmula:

$$K(x_i, x_j) = x_i^T x_j$$

Ilustración 13 Fórmula Kernel Lineal.

Uso: Cuando los datos ya son linealmente separables.

2. **Kernel Polinómico**:

Transforma los datos a un espacio de mayor dimensión usando una función polinómica.

Fórmula:

$$K(x_i, x_j) = (x_i^T x_j + c)^d$$

Ilustración 14 Fórmula Kernel Polinómico.

Donde *c* es una constante y *d* es el grado del polinomio.

Uso: Para datos con relaciones no lineales, pero no demasiado complejas.

3. **Kernel Radial (RBF, Radial Basis Function)**:

Transforma los datos usando una función gaussiana. Es el kernel más utilizado.

Fórmula:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

Ilustración 15 Fórmula Kernel Radial.

Donde *γ* controla el alcance de la influencia de cada punto.

Uso: Para datos con relaciones no lineales complejas.

4. **Kernel Sigmoide**:

Similar a la función de activación de una red neuronal.

Fórmula:

$$K(x_i, x_j) = \tanh(\alpha x_i^T x_j + c)$$

Ilustración 16 Fórmula Kernel Sigmoide

Uso: Menos común, pero puede ser útil en algunos casos.

Ventajas y Desventajas de SVM

Ventajas:

- **Efectivo en espacios de alta dimensión**: Funciona bien incluso cuando el número de características es mayor que el número de muestras.
- **Versatilidad**: Puede manejar problemas lineales y no lineales mediante el uso de kernels.
- **Robustez**: Es menos propenso al overfitting, especialmente en espacios de alta dimensión.

Desventajas:

- **Costo computacional**: Puede ser lento en conjuntos de datos muy grandes.
- **Dificultad para elegir el kernel**: La elección del kernel y sus parámetros puede ser complicada.

- **Poca interpretabilidad**: A diferencia de los árboles de decisión, los modelos SVM son menos interpretables.

Implementación en Python

Aquí podemos ver un ejemplo básico de cómo implementar SVM en Python:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Cargar el conjunto de datos Iris
data = load_iris()
X = data.data  # Características
y = data.target  # Etiquetas

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

# Crear el modelo SVM con kernel RBF
modelo = SVC(kernel="rbf", gamma="scale", random_state=42)

# Entrenar el modelo
modelo.fit(X_train, y_train)

# Predecir con el modelo
y_pred = modelo.predict(X_test)

# Evaluar el modelo
print("Exactitud:", accuracy_score(y_test, y_pred))
```

Explicación del código:

1. **SVC**: Crea el modelo SVM con un kernel RBF.
2. **gamma='scale'**: Define cómo de lejos llega la influencia de un solo punto de entrenamiento.

```
Exactitud: 1.0
```

Ilustración 17 Salida esperada SVM Kernel RBF.

ACTIVIDAD PRÁCTICA GUIADA

Paso 1: Cargar y Explorar los Datos

Comenzamos cargando el conjunto de datos Iris y explorando sus características.

```python
from sklearn.datasets import load_iris
import pandas as pd

# Cargar el conjunto de datos Iris
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Características
y = pd.Series(data.target)  # Etiquetas

# Mostrar las primeras filas de los datos
print("Características (X):")
print(X.head())

print("\nEtiquetas (y):")
print(y.head())

# Información sobre el conjunto de datos
# print("\nInformación del conjunto de datos:")
# print(data.DESCR)
```

- **X**: Contiene las características (longitud y ancho del sépalo y pétalo).
- **y**: Contiene las etiquetas (0: setosa, 1: versicolor, 2: virginica).
- **DESCR**: Proporciona una descripción detallada del conjunto de datos.

Salida:

```
Características (X):
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Etiquetas (y):
0    0
1    0
2    0
3    0
4    0
dtype: int64
```

Ilustración 18 Salida paso 1.

Paso 2: Dividir los Datos en Entrenamiento y Prueba

Dividimos los datos en un conjunto de entrenamiento (75%) y un conjunto de prueba (25%).

```python
from sklearn.model_selection import train_test_split

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

print(f"\nTamaño del conjunto de entrenamiento: {X_train.shape}")
print(f"Tamaño del conjunto de prueba: {X_test.shape}")
```

- **train_test_split**: Divide los datos de manera aleatoria pero reproducible (gracias a random_state=42).
- **test_size=0.25**: El 25% de los datos se usará para prueba.

Salida:

```
Tamaño del conjunto de entrenamiento: (112, 4)
Tamaño del conjunto de prueba: (38, 4)
```

Ilustración 19 Salida paso 2.

Paso 3: Entrenar y Evaluar Modelos

Ahora entrenaremos y evaluaremos los cuatro modelos: **Regresión Logística**, **Árbol de Decisión**, **Bosque Aleatorio**, y **SVM**.

3.1 Regresión Logística

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Crear y entrenar el modelo de regresión logística
modelo_lr = LogisticRegression(max_iter=200, random_state=42)
modelo_lr.fit(X_train, y_train)

# Predecir y evaluar
y_pred_lr = modelo_lr.predict(X_test)
print("\nExactitud de Regresión Logística:", accuracy_score(y_test,
y_pred_lr))
```

Salida:

```
Exactitud de Regresión Logística: 1.0
```

Ilustración 20 Salida paso 3.1.

3.2 Árbol de Decisión

```
from sklearn.tree import DecisionTreeClassifier

# Crear y entrenar el modelo de árbol de decisión
modelo_dt = DecisionTreeClassifier(max_depth=3, random_state=42)
modelo_dt.fit(X_train, y_train)

# Predecir y evaluar
```

```
y_pred_dt = modelo_dt.predict(X_test)
print("Exactitud de Árbol de Decisión:", accuracy_score(y_test, y_pred_dt))
```

Salida:

```
Exactitud de Árbol de Decisión: 1.0
```

Ilustración 21 Salida paso 3.2.

3.3 Bosque Aleatorio

```
from sklearn.ensemble import RandomForestClassifier

# Crear y entrenar el modelo de bosque aleatorio
modelo_rf = RandomForestClassifier(n_estimators=100, random_state=42)
modelo_rf.fit(X_train, y_train)

# Predecir y evaluar
y_pred_rf = modelo_rf.predict(X_test)
print("Exactitud de Bosque Aleatorio:", accuracy_score(y_test, y_pred_rf))
```

Salida:

```
Exactitud de Bosque Aleatorio: 1.0
```

Ilustración 22 Salida paso 3.3.

3.4 Support Vector Machine (SVM)

```
from sklearn.svm import SVC

# Crear y entrenar el modelo SVM
modelo_svm = SVC(kernel="rbf", gamma="scale", random_state=42)
modelo_svm.fit(X_train, y_train)

# Predecir y evaluar
y_pred_svm = modelo_svm.predict(X_test)
print("Exactitud de SVM:", accuracy_score(y_test, y_pred_svm))
```

Salida:

```
Exactitud de SVM: 1.0
```

Ilustración 23 Salida paso 3.4.

Paso 4: Comparar los Resultados

Finalmente, comparamos la exactitud de los cuatro modelos.

```
print("\nComparación de Exactitud:")
print(f"- Regresión Logística: {accuracy_score(y_test, y_pred_lr):.4f}")
print(f"- Árbol de Decisión: {accuracy_score(y_test, y_pred_dt):.4f}")
print(f"- Bosque Aleatorio: {accuracy_score(y_test, y_pred_rf):.4f}")
print(f"- SVM: {accuracy_score(y_test, y_pred_svm):.4f}")
```

Salida:

```
Comparación de Exactitud:
- Regresión Logística: 1.0000
- Árbol de Decisión: 1.0000
- Bosque Aleatorio: 1.0000
- SVM: 1.0000
```

Ilustracion 24 Salida paso 4.

Ejercicio Completo:

```
from sklearn.datasets import load_iris
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Cargar el conjunto de datos Iris
data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Características
y = pd.Series(data.target)  # Etiquetas
```

```python
# Mostrar las primeras filas de los datos
print("Características (X):")
print(X.head())

print("\nEtiquetas (y):")
print(y.head())

# Información sobre el conjunto de datos
# print("\nInformación del conjunto de datos:")
# print(data.DESCR)

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)

print(f"\nTamaño del conjunto de entrenamiento: {X_train.shape}")
print(f"Tamaño del conjunto de prueba: {X_test.shape}")


# Crear y entrenar el modelo de regresión logística
modelo_lr = LogisticRegression(max_iter=200, random_state=42)
modelo_lr.fit(X_train, y_train)

# Predecir y evaluar
y_pred_lr = modelo_lr.predict(X_test)
print("\nExactitud de Regresión Logística:", accuracy_score(y_test,
y_pred_lr))


# Crear y entrenar el modelo de árbol de decisión
modelo_dt = DecisionTreeClassifier(max_depth=3, random_state=42)
modelo_dt.fit(X_train, y_train)

# Predecir y evaluar
y_pred_dt = modelo_dt.predict(X_test)
print("Exactitud de Árbol de Decisión:", accuracy_score(y_test, y_pred_dt))

# Crear y entrenar el modelo de bosque aleatorio
modelo_rf = RandomForestClassifier(n_estimators=100, random_state=42)
modelo_rf.fit(X_train, y_train)

# Predecir y evaluar
y_pred_rf = modelo_rf.predict(X_test)
```

```
print("Exactitud de Bosque Aleatorio:", accuracy_score(y_test, y_pred_rf))


# Crear y entrenar el modelo SVM
modelo_svm = SVC(kernel="rbf", gamma="scale", random_state=42)
modelo_svm.fit(X_train, y_train)

# Predecir y evaluar
y_pred_svm = modelo_svm.predict(X_test)
print("Exactitud de SVM:", accuracy_score(y_test, y_pred_svm))

print("\nComparación de Exactitud:")
print(f"- Regresión Logística: {accuracy_score(y_test, y_pred_lr):.4f}")
print(f"- Árbol de Decisión: {accuracy_score(y_test, y_pred_dt):.4f}")
print(f"- Bosque Aleatorio: {accuracy_score(y_test, y_pred_rf):.4f}")
print(f"- SVM: {accuracy_score(y_test, y_pred_svm):.4f}")
```