Machine Learning I: Introducción a los Métodos de Clasificación Supervisada

CONCEPTOS

PARTE 1: Un enfoque práctico del Machine Learning

1. Acerca del software

   1.1 Integración en el ecosistema de Python

2. ¿Qué es el Machine Learning?

3. Modelando el problema de Machine Learning

4. El problema de la clasificación supervisada. Un ejemplo básico guiado programáticamente

   4.1 Representando el problema en sklearn

   4.2 Aprendiendo y prediciendo

   4.3 Más sobre los datos: The feature space

   4.4 Train y Test

   4.5 Selección de modelo (I)

PARTE 2: Conceptos de aprendizaje y teoría

1. ¿Qué es aprender?

   - Aprendizaje PAC

2. Dentro del modelo de aprendizaje

   - El algoritmo de Machine Learning humano
   - Clase de modelo y espacio de hipótesis
   - Función objetivo
   - Búsqueda/Optimización/Algoritmo de aprendizaje

3. Curvas de aprendizaje y sobreajuste

   - Curvas de aprendizaje
   - Sobreajuste

4. Curas para el sobreajuste

   - Selección de modelo (II)
   - Regularización

- Conjunto

PARTE 3: Primeros modelos

1. Modelos generativos y discriminativos

   9.1 Modelos Bayesianos (Naive Bayes) y algunas aplicaciones

   9.2 Máquinas de Soporte Vectorial (Support Vector Machines)

PARTE 1: Un enfoque práctico del Machine Learning

1. Acerca del software

Scikit-Learn

- Scikit-Learn es una biblioteca de aprendizaje automático escrita en Python.
- Simple y eficiente, tanto para expertos como para no expertos.
- Algoritmos de aprendizaje automático clásicos y bien establecidos.
- Licencia BSD 3.

1.1 Integración en el ecosistema científico de Python

El ecosistema de código abierto de Python proporciona un entorno de trabajo científico versátil y potente, que incluye:

- NumPy (para la manipulación eficiente de arrays multidimensionales);
- SciPy (para estructuras de datos especializadas (por ejemplo, matrices dispersas) y algoritmos científicos de bajo nivel),
- IPython (para exploración interactiva),
- Matplotlib (para visualización)
- Pandas (para la gestión y análisis de datos)
- (y muchos otros...)

Scikit-Learn se basa en NumPy y SciPy y complementa este entorno científico con algoritmos de aprendizaje automático; por diseño, Scikit-Learn es no intrusivo, fácil de usar y fácil de combinar con otras bibliotecas. Usaremos Scikit-Learn como herramienta para entender el aprendizaje automático.

2. ¿Qué es el Machine Learning?

El **Aprendizaje Automático** (ML) se trata de codificar programas que ajustan automáticamente su rendimiento a partir de la exposición a la información codificada en los datos. Este aprendizaje se logra mediante un modelo parametrizado con parámetros ajustables automáticamente según un criterio de rendimiento.

El aprendizaje automático puede considerarse un subcampo de la Inteligencia Artificial (IA).

Hay tres clases principales de ML:

1. Aprendizaje supervisado: Algoritmos que aprenden de un conjunto de entrenamiento de ejemplos etiquetados (ejemplares) para generalizar al conjunto de todas las entradas posibles. Ejemplos de técnicas en aprendizaje supervisado incluyen la regresión y las máquinas de vectores de soporte.

2. Aprendizaje no supervisado: Algoritmos que aprenden de un conjunto de entrenamiento de ejemplos no etiquetados, utilizando las características de las entradas para categorizar las entradas juntas según algún criterio estadístico. Ejemplos de aprendizaje no supervisado incluyen el agrupamiento k-means y la estimación de densidad kernel.

3. Aprendizaje por refuerzo: Algoritmos que aprenden mediante refuerzo de un crítico que proporciona información sobre la calidad de una solución, pero no sobre cómo mejorarla. Las soluciones mejoradas se logran explorando iterativamente el espacio de soluciones. No cubriremos RL en este curso.

3. Modelando el problema del Machine Learning

El primer paso para aplicar la ciencia de datos y el aprendizaje automático es identificar una pregunta interesante para responder. Según el tipo de respuesta que buscamos, estamos apuntando directamente a un conjunto de técnicas.

- Si nuestra pregunta se responde con *SÍ/NO*, estamos frente a un problema de **clasificación**. Los clasificadores también son las técnicas a usar si nuestra pregunta admite solo un conjunto discreto de respuestas, es decir, queremos seleccionar entre un número finito de opciones.
    - Dado un perfil de cliente y actividad pasada, ¿cuáles son los productos financieros que más le interesarían?
    - Dados los resultados de una prueba clínica, ¿este paciente sufre de diabetes?
    - Dada una imagen de resonancia magnética, ¿hay un tumor en ella?
    - Dada la actividad pasada asociada a una tarjeta de crédito, ¿es la operación actual un fraude?
    - Dadas mis habilidades y calificaciones en informática y matemáticas, ¿aprobaré el curso de ciencia de datos?
- Si nuestra pregunta es una predicción de una cantidad (generalmente de valor real), estamos frente a un problema de **regresión**.
    - Dada la descripción de un apartamento, ¿cuál es el valor de mercado esperado del piso? ¿Cuál sería el valor si el apartamento tiene ascensor?

- Dados los registros pasados de actividades de usuario en aplicaciones, ¿cuánto tiempo estará un cliente determinado enganchado a nuestra aplicación?
- Dadas mis habilidades y calificaciones en informática y matemáticas, ¿qué calificación lograré?

Observe que algunos problemas pueden resolverse usando tanto regresión como clasificación. Como veremos más adelante, muchos algoritmos de clasificación son regresores con umbral. Hay cierta habilidad en diseñar la pregunta correcta y esto cambia dramáticamente la solución que obtenemos.

REGLA DE ORO: Nuestro primer principio de diseño a tener en cuenta es que en general si un problema se puede resolver usando una pregunta más simple no use una más compleja. Esto es una instancia del famoso principio KISS (*¡Keep It Simple, Stupid!*).

PREGUNTA: ¿Cuál de las siguientes preguntas corresponde a un problema de classificación?

¿Qué tiempo hará mañana? ¿Es normal este comportamiento? ¿Dónde están mis llaves en esta foto?

4. El problema de la clasificación supervisada: Un ejemplo básico y programático guiado

En un problema de clasificación supervisada, dado un conjunto de ejemplos con sus respectivas etiquetas, nuestro objetivo es predecir a cuál de un conjunto predefinido de clases discretas pertenece una instancia dada.

Formalmente, podemos describir el problema de la siguiente manera: Consideremos un *conjunto de entrenamiento* compuesto por N pares de muestras de datos $\{(x_i, y_i)\}, i=1,\ldots,N$, donde $x_i \in R^d$ está descrito por d características, y y_i es su etiqueta supervisada correspondiente. Por ejemplo, en el caso binario más simple, $y_i \in \{-1,1\}$. Nuestro objetivo es encontrar un modelo $h: R^d \rightarrow R$ tal que, dado un nuevo dato x, prediga correctamente su etiqueta y, es decir, $h(x)=y$.

En aprendizaje automático, generalmente se distinguen dos etapas principales:

- **Entrenamiento**: Dado un conjunto de datos x con sus respectivas etiquetas y, queremos aprender o ajustar un modelo.
- **Prueba o explotación**: Dado un modelo entrenado, queremos aplicarlo a nuevos datos no vistos para predecir sus etiquetas.

Ejemplo de implementación: Reconocimiento de dígitos manuscritos

El problema: Consideremos el reconocimiento de dígitos manuscritos. Dada una imagen de un dígito manuscrito, queremos construir un clasificador que reconozca la etiqueta correcta.

Pasos para la implementación:

1. **Carga de datos**: Comencemos cargando un conjunto de datos de dígitos manuscritos, como el conjunto de datos MNIST.

2. **Preprocesamiento**: Es posible que necesitemos realizar ciertas transformaciones en los datos, como normalización o ajuste de tamaño de las imágenes, para prepararlos adecuadamente para el modelo.

3. **Selección del modelo**:

   Para este problema, podemos considerar diferentes modelos de clasificación:

   - **Support Vector Machine (SVM)**: Es un modelo que busca encontrar el hiperplano óptimo que mejor separa las clases en el espacio de características. Es eficaz cuando hay una clara separación entre las clases.

   - **Red Neuronal Convolucional (CNN)**: Especialmente eficaz en tareas de visión por computadora, como el reconocimiento de imágenes. Las CNNs están diseñadas para procesar datos estructurados en cuadrículas, como las imágenes, y pueden capturar patrones espaciales y de composición en los datos.

   La elección entre SVM y CNN dependerá de varios factores, incluyendo la cantidad de datos disponibles, la complejidad del problema y los recursos computacionales disponibles para el entrenamiento y la evaluación del modelo.

4. **Entrenamiento del modelo**: Utilizando el conjunto de datos etiquetados, entrenamos nuestro modelo seleccionado para que aprenda las características distintivas de cada dígito.

5. **Prueba y evaluación**: Una vez entrenado, evaluamos el modelo utilizando datos que no ha visto antes (conjunto de prueba) para verificar su capacidad para generalizar y predecir correctamente las etiquetas de nuevos dígitos manuscritos.

6. **Persistencia del modelo**: Finalmente, podemos guardar el modelo entrenado en disco para su uso futuro sin necesidad de volver a entrenarlo desde cero cada vez que necesitemos hacer una predicción.

Este ejemplo proporciona una visión más detallada sobre cómo abordar un problema de clasificación supervisada, destacando la importancia de seleccionar el modelo adecuado y los pasos adicionales involucrados en la implementación práctica, como el preprocesamiento de datos y la evaluación del modelo. La elección entre SVM y CNN se presenta como dos enfoques posibles según las características específicas del problema de reconocimiento de dígitos manuscritos.

```
# Cargar conjunto de datos.
from sklearn import datasets
digits = datasets.load_digits()
```

Ahora, verifica los datos que acabas de cargar.

```
# Verificar el formato de los datos.
X, y = digits.data, digits.target

print(X.shape)
print(y.shape)
```

```
(1797, 64)
(1797,)
```

4.1 Representando un problema de Machine Learning en Scikit-Learn

Recuerda la formalización del problema donde el conjunto de datos de entrenamiento consiste en N pares de datos $S=\{(x_i,y_i)\},i=1\ldots N$ donde $x_i \in R^d$ está compuesto por d características/descriptores y $y_i \in \{1,\ldots,K\}$ es una etiqueta de objetivo discreto. En nuestro problema actual, tenemos $N=1797$ ejemplos de datos de números escritos a mano. Cada muestra es una imagen de 8×8. La representación de cada muestra de datos está codificada en un vector. Por esta razón, aplanamos la imagen y la reorganizamos en un vector con $d=64$ correspondientes a los valores de gris/brillo de cada píxel de la imagen. y_i es el valor de la clase objetivo a la que pertenece el número.

Visualicemos el primer dígito utilizando herramientas de visualización disponibles en Scikit-Learn y Matplotlib.

```
%matplotlib inline
# La imagen original del dígito ha sido aplanada, por lo que la
volvemos a dar forma a su forma original
# Verificar la dimensionalidad de los datos, por ejemplo, el primer
elemento en el conjunto de datos X[0]
print(X[0].shape)
```

```
(64,)
```

```
print(X[0])
```

```
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.
3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.
0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10.
12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

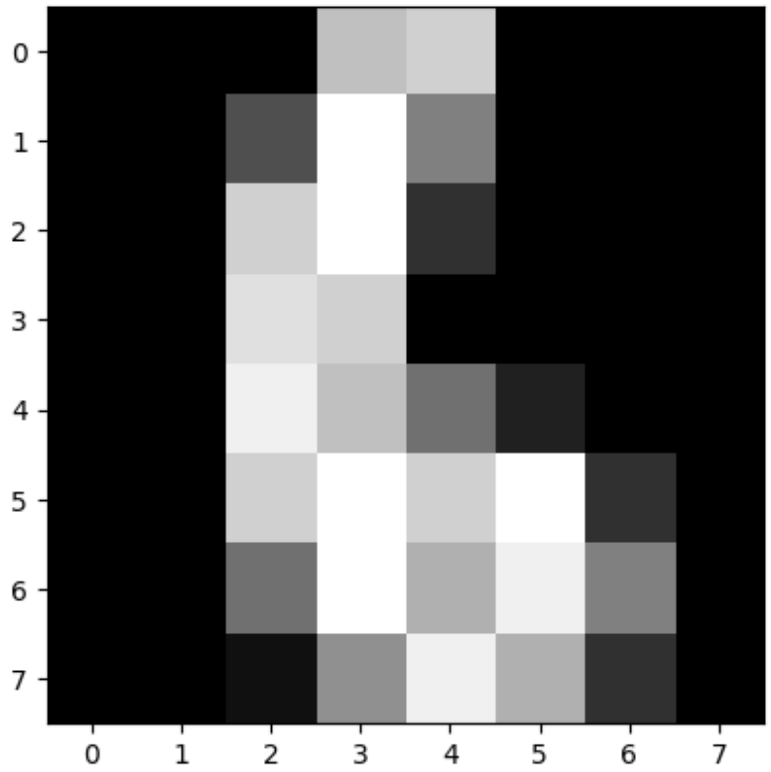
```
# Reajustar a 8x8 para recuperar la imagen original
print(X[0].reshape((8,8)))
```

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
```

```
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

```
# Importar la biblioteca matplotlib para la visualización.
import matplotlib.pyplot as plt

# Mostrar la imagen del séptimo dígito en el conjunto de datos.
# Primero, reajustamos los datos del dígito a su forma original de 8x8 píxeles.
# Utilizamos 'cmap="gray"' para mostrar la imagen en escala de grises.
# El parámetro 'interpolation="nearest"' asegura que la imagen no sea suavizada.
plt.imshow(X[6].reshape((8,8)), cmap="gray", interpolation="nearest")
```

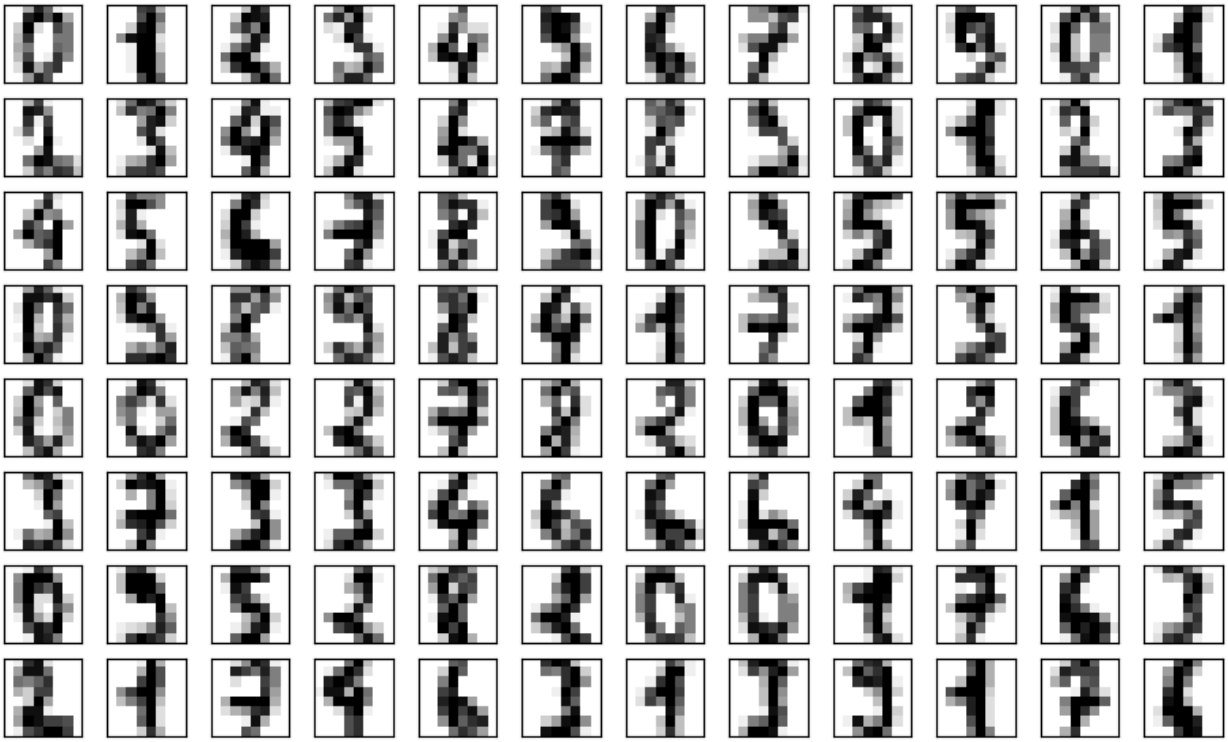
```
<matplotlib.image.AxesImage at 0x7f1a67e5ab00>
```



Veamos algunos de los ejemplos que tenemos en nuestro conjunto de datos.

```
# Importar la biblioteca matplotlib para la visualización.
import matplotlib.pyplot as plt

# Crear una figura con una cuadrícula de subplots de 8 filas por 12 columnas.
fig, ax = plt.subplots(8, 12, subplot_kw={'xticks':[], 'yticks':[]})
```

```
# Iterar sobre cada subplot en la cuadrícula.
for i in range(ax.size):
    # Mostrar la imagen del dígito en la posición 'i' en el conjunto
de datos.
    # Primero, reajustamos los datos del dígito a su forma original de
8x8 píxeles.
    # Utilizamos 'cmap=plt.cm.binary' para mostrar la imagen en una
escala de grises binaria.
    ax.flat[i].imshow(digits.data[i].reshape(8, 8),
cmap=plt.cm.binary)

# Ajustar el tamaño de la figura para que sea de 10 pulgadas de ancho
y 6 pulgadas de alto.
fig.set_size_inches((10, 6))
```



Un problema en Scikit-Learn se modela de la siguiente manera:

- Los datos de entrada están estructurados en arrays de Numpy. Se espera que el tamaño del array sea [n_samples, n_features]:
  - *n_samples*: El número de muestras (N): cada muestra es un elemento a procesar (por ejemplo, clasificar). Una muestra puede ser un documento, una imagen, un sonido, un video, un objeto astronómico, una fila en una base de datos o archivo CSV, o cualquier cosa que se pueda describir con un conjunto fijo de características cuantitativas.

- *n_features*: El número de características (d) o rasgos distintos que pueden usarse para describir cada elemento de manera cuantitativa. Las características generalmente son valores reales, pero pueden ser booleanos, valores discretos o incluso categóricos.

$$ {\rm matriz~de~características:~~~} {\bf X} = \left[\begin{matrix} x_{11} & x_{12} & \cdots & x_{1d}\\ x_{21} & x_{22} & \cdots & x_{2d}\\ x_{31} & x_{32} & \cdots & x_{3d}\\ \vdots & \vdots & \ddots & \vdots\\ \vdots & \vdots & \ddots & \vdots\\ x_{N1} & x_{N2} & \cdots & x_{Nd}\\ \end{matrix} \right] $$

$$ {\rm vector~de~etiquetas:~~~} {\bf y} = [y_1, y_2, y_3, \cdots, y_N] $$

El número de características debe fijarse de antemano. Sin embargo, puede ser de muy alta dimensionalidad (por ejemplo, millones de características) con la mayoría de ellas siendo ceros para una muestra dada.

Ejemplo

Consideremos una representación de documento de texto. Dado un documento de texto, queremos construir una representación para él. En este caso, podríamos usar como descripción del documento un diccionario con todas las palabras posibles en nuestro idioma y crear una descripción del documento como el número de veces que cierta palabra aparece en el documento. Cada documento es una muestra y cada valor que cuenta el número de veces que una palabra aparece en el texto es una característica. Observe que un solo documento usará solo un puñado de palabras. Por lo tanto, hay muchas palabras que no se utilizan y su representación será cero. Este es un caso donde las matrices scipy.sparse pueden ser útiles, ya que son mucho más eficientes en términos de memoria que los arrays de numpy.

Jerga del conjunto de datos

Considerando los datos organizados como en la sección anterior, nos referimos a:

- las **columnas** como características, atributos, dimensiones, regresores, covariables, predictores, variables independientes,
- las **filas** como instancias, ejemplos, muestras,
- el **objetivo** como etiqueta, resultado, respuesta, variable dependiente.

PREGUNTA: Consideremos el siguiente problema: *Nos piden desarrollar un producto similar a Shazzam(tm). Esto es, reconocer el nombre de una canción dada una pequeña muestra de la música.*. Discuta y describa con su compañero un posible vector de características para este problema.

4.2 Aprendizaje y predicción con Scikit-Learn

Todos los objetos en scikit-learn comparten una API uniforme y limitada que consiste en tres interfaces complementarias:

- una interfaz de estimador para construir y ajustar modelos (.fit());
- una interfaz de predictor para hacer predicciones (.predict());
- una interfaz de transformador para convertir datos.

Elijamos un modelo y ajustemos los datos de entrenamiento:

Máquinas de Soporte Vectorial (SVM)

Las **Máquinas de Soporte Vectorial (SVM)** son algoritmos de aprendizaje supervisado utilizados para clasificación y regresión. Son ampliamente reconocidas por su capacidad para manejar problemas de alta dimensionalidad y proporcionar clasificaciones robustas. A continuación, exploramos cómo funcionan las SVM y cómo implementarlas.

Conceptos Básicos

1. **Hiperplano Separador**:
   - Una SVM encuentra un hiperplano en un espacio n-dimensional que separa las clases de datos.
   - En problemas linealmente separables, el hiperplano maximiza el **margen**, que es la distancia más pequeña entre el hiperplano y cualquier punto de datos.
2. **Márgenes y Vectores de Soporte**:
   - **Márgenes**: Son las regiones entre el hiperplano y los puntos de datos más cercanos de cada clase.
   - **Vectores de Soporte**: Son los puntos más cercanos al hiperplano que determinan su posición. Estos puntos son cruciales para el modelo.
3. **Función de Costo**:
   - La SVM intenta minimizar el error mientras maximiza el margen. Esto se representa mediante la siguiente función de optimización:

     $$min \frac{1}{2} \| w \|^2 \text{sujeto a } y_i(w \cdot x_i + b) \geq 1, \forall i$$

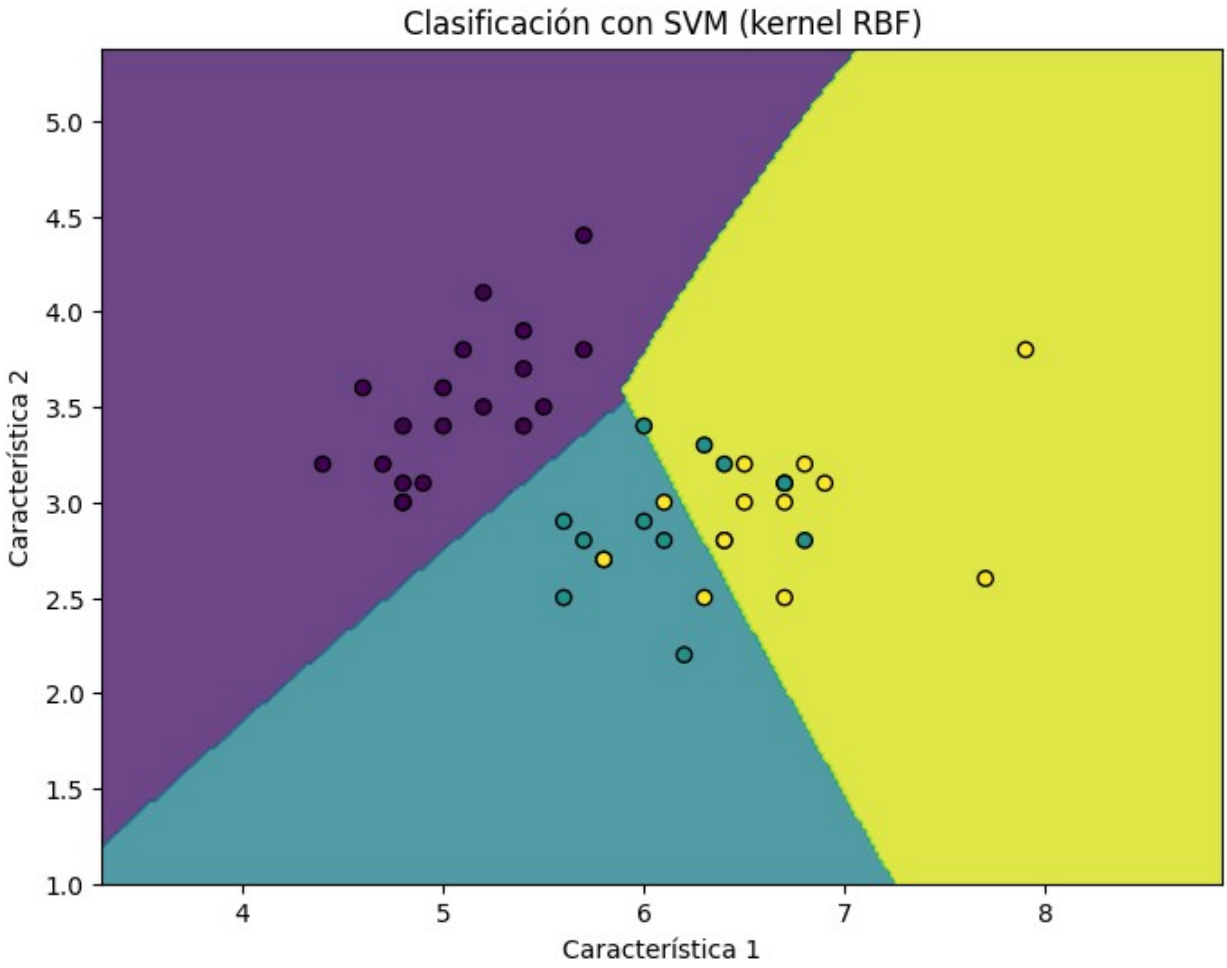
     donde w son los pesos del modelo, b es el sesgo, x_i son los datos, y y_i son las etiquetas de clase.
4. **Kernels**:
   - Para problemas no linealmente separables, se utiliza la técnica del **kernel**. Los kernels transforman los datos a un espacio de mayor dimensionalidad donde puedan separarse linealmente.
   - Algunos tipos comunes de kernels son:
     - **Lineal**: $K(x, x') = x \cdot x'$
     - **RBF (Gaussiano)**: $K(x, x') = \exp(-\gamma \| x - x' \|^2)$
     - **Polinómico**: $K(x, x') = (x \cdot x' + c)^d$

Implementación de SVM

A continuación, implementamos un modelo SVM para clasificar un conjunto de datos bidimensional.

```
# Importar las bibliotecas necesarias
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import matplotlib.pyplot as plt
```

```
import numpy as np

# Cargar el conjunto de datos Iris
iris = load_iris()
X0_svm = iris.data[:, :2]  # Usar solo las dos primeras
características para graficar
y0_svm = iris.target

# Dividir en conjunto de entrenamiento y prueba
X_train_svm, X_test_svm, y_train_svm, y_test_svm =
train_test_split(X0_svm, y0_svm, test_size=0.3, random_state=42)

# Entrenar un modelo SVM con kernel RBF
svm_model = SVC(kernel='rbf', C=1, gamma=0.1, probability=True)
svm_model.fit(X_train_svm, y_train_svm)

# Visualizar el espacio de decisión
h_svm = 0.02
x_min_svm, x_max_svm = X0_svm[:, 0].min() - 1, X0_svm[:, 0].max() + 1
y_min_svm, y_max_svm = X0_svm[:, 1].min() - 1, X0_svm[:, 1].max() + 1
xx_svm, yy_svm = np.meshgrid(np.arange(x_min_svm, x_max_svm, h_svm),
np.arange(y_min_svm, y_max_svm, h_svm))
Z_svm = svm_model.predict(np.c_[xx_svm.ravel(), yy_svm.ravel()])
Z_svm = Z_svm.reshape(xx_svm.shape)

plt.figure(figsize=(8, 6))
plt.contourf(xx_svm, yy_svm, Z_svm, alpha=0.8, cmap='viridis')
plt.scatter(X_test_svm[:, 0], X_test_svm[:, 1], c=y_test_svm,
edgecolor='k', cmap='viridis')
plt.title("Clasificación con SVM (kernel RBF)")
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
plt.show()
```



Evaluación del Modelo

Es importante evaluar el rendimiento de la SVM utilizando métricas estándar. Las métricas comunes incluyen **Precisión**, **Recall**, **F1-Score**, y la **Matriz de Confusión**. Estas métricas se derivan de los siguientes términos fundamentales:

Términos Fundamentales

1. **Verdaderos Positivos (TP)**:
   - Cantidad de ejemplos positivos que el modelo clasifica correctamente como positivos.
2. **Falsos Positivos (FP)**:
   - Cantidad de ejemplos negativos que el modelo clasifica incorrectamente como positivos.
3. **Verdaderos Negativos (TN)**:
   - Cantidad de ejemplos negativos que el modelo clasifica correctamente como negativos.
4. **Falsos Negativos (FN)**:
   - Cantidad de ejemplos positivos que el modelo clasifica incorrectamente como negativos.

Métricas Clave

1. **Precisión (Precision)**:
   - Proporción de predicciones positivas correctas frente al total de predicciones positivas realizadas.

$$\text{Precisión} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

   - Mide la exactitud del modelo para identificar correctamente los positivos.
2. **Sensibilidad (Recall)**:
   - También conocida como **tasa de verdaderos positivos**, mide la proporción de positivos correctamente identificados.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

3. **F1-Score**:
   - Promedio armónico entre la precisión y el recall. Es útil cuando existe un desequilibrio entre las clases.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

4. **Exactitud (Accuracy)**:
   - Proporción de predicciones correctas frente al total de ejemplos.

$$\text{Exactitud} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

```
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import LabelBinarizer

# Predicciones
y_pred_svm = svm_model.predict(X_test_svm)

# Reporte de clasificación
print("Reporte de clasificación:")
print(classification_report(y_test_svm, y_pred_svm))

# Matriz de confusión
cm_svm = confusion_matrix(y_test_svm, y_pred_svm)
print("Matriz de Confusión:")
print(cm_svm)

# AUC para cada clase (utilizando One-vs-Rest)
lb_svm = LabelBinarizer()
y_test_bin_svm = lb_svm.fit_transform(y_test_svm)  # Convertir las
etiquetas de clase a formato binario
y_pred_prob_svm = svm_model.predict_proba(X_test_svm)  # Obtener las
probabilidades de predicción

# Calcular el AUC para cada clase
auc_scores_svm = []
```

```
for i in range(y_test_bin_svm.shape[1]):
    auc_svm = roc_auc_score(y_test_bin_svm[:, i], y_pred_prob_svm[:,
i])
    auc_scores_svm.append(auc_svm)

# Mostrar AUC para cada clase
for i, auc_svm in enumerate(auc_scores_svm):
    print(f"AUC para la clase {i}: {auc_svm}")
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.70      0.54      0.61        13
           2       0.62      0.77      0.69        13

    accuracy                           0.80        45
   macro avg       0.78      0.77      0.77        45
weighted avg       0.81      0.80      0.80        45

Matriz de Confusión:
[[19  0  0]
 [ 0  7  6]
 [ 0  3 10]]
AUC para la clase 0: 1.0
AUC para la clase 1: 0.8834134615384615
AUC para la clase 2: 0.8978365384615384
```

Reporte de Clasificación

El reporte generado para el modelo SVM incluye las siguientes métricas para evaluar el rendimiento por clase:

- **Precision**: Proporción de predicciones correctas dentro de todas las predicciones positivas realizadas.
- **Recall**: Proporción de ejemplos positivos correctamente clasificados.
- **F1-Score**: Promedio armónico de la precisión y el recall.
- **Support**: Número de ejemplos verdaderos de cada clase.

Ejemplo del reporte:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.70      0.54      0.61        13
           2       0.62      0.77      0.69        13

    accuracy                           0.80        45
   macro avg       0.78      0.77      0.77        45
weighted avg       0.81      0.80      0.80        45
```

- **Clase 0**:
  - **Precisión**: El 100% de las predicciones positivas para la clase 0 fueron correctas.
  - **Recall**: El modelo identificó correctamente el 100% de los ejemplos verdaderos de la clase 0.
  - **F1-Score**: El balance entre precisión y recall para la clase 0 es del 100%.
- **Clase 1**:
  - **Precisión**: El 70% de las predicciones positivas para la clase 1 fueron correctas.
  - **Recall**: El modelo identificó correctamente el 54% de los ejemplos verdaderos de la clase 1.
  - **F1-Score**: El balance entre precisión y recall para la clase 1 es del 61%.
- **Clase 2**:
  - **Precisión**: El 62% de las predicciones positivas para la clase 2 fueron correctas.
  - **Recall**: El modelo identificó correctamente el 77% de los ejemplos verdaderos de la clase 2.
  - **F1-Score**: El balance entre precisión y recall para la clase 2 es del 69%.
- **Exactitud General (Accuracy)**:
  - El modelo clasifica correctamente el 80% de todos los ejemplos en el conjunto de datos.
- **Promedios**:
  - **Macro Average**: Promedio no ponderado de precisión, recall y F1-score (78%, 77%, y 77% respectivamente).
  - **Weighted Average**: Promedio ponderado basado en el número de ejemplos en cada clase (81%, 80%, y 80% respectivamente).

Matriz de Confusión

La matriz de confusión muestra cómo el modelo clasifica correctamente (o incorrectamente) los ejemplos de cada clase:

```
[[19  0  0]
 [ 0  7  6]
 [ 0  3 10]]
```

- **Interpretación**:
  - **Clase 0**:
    - **Verdaderos Positivos (TP)**: 19 (correctamente clasificados como 0).
    - **Falsos Positivos (FP)**: 0 (incorrectamente clasificados como 1 o 2).
    - **Falsos Negativos (FN)**: 0 (incorrectamente clasificados como 1 o 2).
    - **Verdaderos Negativos (TN)**: 0 (correctamente clasificados como no 0).
  - **Clase 1**:
    - **Verdaderos Positivos (TP)**: 7 (correctamente clasificados como 1).
    - **Falsos Positivos (FP)**: 0 (incorrectamente clasificados como 0).
    - **Falsos Negativos (FN)**: 6 (incorrectamente clasificados como 2).
    - **Verdaderos Negativos (TN)**: 19 (correctamente clasificados como no 1).
  - **Clase 2**:

- **Verdaderos Positivos (TP)**: 10 (correctamente clasificados como 2).
- **Falsos Positivos (FP)**: 0 (incorrectamente clasificados como 0).
- **Falsos Negativos (FN)**: 3 (incorrectamente clasificados como 1).
- **Verdaderos Negativos (TN)**: 19 (correctamente clasificados como no 2).

Área Bajo la Curva ROC (AUC)

La **Área Bajo la Curva ROC (AUC)** mide la capacidad del modelo para distinguir entre clases positivas y negativas.

- AUC para la clase 0: **1.0** (perfecto, indica que el modelo discrimina perfectamente entre la clase 0 y las demás).
- AUC para la clase 1: **0.8834** (muy bueno, el modelo tiene una excelente capacidad para discriminar entre la clase 1 y las otras clases).
- AUC para la clase 2: **0.8978** (también muy bueno, indicando que el modelo tiene una alta capacidad para distinguir la clase 2 de las otras).

Evaluación Global

Las métricas sugieren que el modelo SVM tiene un rendimiento sólido:

- Alta precisión y recall para la clase 0, lo que indica que el modelo clasifica muy bien la clase 0.
- Aunque la precisión y el recall para las clases 1 y 2 son algo más bajos, el modelo sigue siendo bastante eficiente, con un F1-score decente.
- La matriz de confusión muestra que el modelo comete algunos errores de clasificación, especialmente entre las clases 1 y 2.
- Un valor de AUC cercano a 1 para todas las clases indica que el modelo tiene una excelente capacidad para discriminar entre las clases.

Oportunidades de Mejora:

- Mejorar la clasificación de la clase 1 podría lograrse ajustando los hiperparámetros del modelo o considerando técnicas de balanceo de clases.
- Explorar el uso de otros kernels, como el polinómico o el sigmoide, podría mejorar el rendimiento general en la clasificación de todas las clases.

Optimización de Hiperparámetros

Los hiperparámetros principales que afectan el rendimiento de SVM son:

- **C**: Controla el margen. Valores altos favorecen márgenes más pequeños con menor error de clasificación, mientras que valores bajos permiten márgenes más amplios.
- **Gamma** (para kernels no lineales): Controla la influencia de un punto de datos individual. Valores altos implican mayor influencia local, mientras que valores bajos implican una influencia más global.

Utilizamos una búsqueda de cuadrícula para encontrar los mejores valores:

```python
from sklearn.model_selection import GridSearchCV

# Definir el espacio de búsqueda
param_grid_svm = {
    'C': [0.1, 1, 10, 100],
    'gamma': [1, 0.1, 0.01, 0.001],
    'kernel': ['rbf']
}

# Búsqueda en cuadrícula
grid_svm = GridSearchCV(SVC(), param_grid_svm, refit=True, verbose=2,
cv=5)
grid_svm.fit(X_train_svm, y_train_svm)

print("Mejores parámetros:", grid_svm.best_params_)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[CV] END .........................C=0.1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=0.1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=0.1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=0.1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=0.1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=0.1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=0.1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=0.1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=0.1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=0.1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END .....................C=0.1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .....................C=0.1, gamma=0.001, kernel=rbf; total
```

```
time=   0.0s
[CV] END ......................C=0.1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ......................C=0.1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ............................C=1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ............................C=1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ............................C=1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ............................C=1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ............................C=1, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ..........................C=1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ..........................C=1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ..........................C=1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ..........................C=1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ..........................C=1, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ........................C=1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ........................C=1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ........................C=1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ........................C=1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ........................C=1, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END .......................C=1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .......................C=1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .......................C=1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .......................C=1, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ...........................C=10, gamma=1, kernel=rbf; total
time=   0.0s
```

```
[CV] END .........................C=10, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=10, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=10, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .........................C=10, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=10, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=10, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=10, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=10, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .......................C=10, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ......................C=10, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=10, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=10, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=10, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ......................C=10, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END .....................C=10, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .....................C=10, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .....................C=10, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .....................C=10, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END .....................C=10, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ........................C=100, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ........................C=100, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ........................C=100, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ........................C=100, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ........................C=100, gamma=1, kernel=rbf; total
time=   0.0s
[CV] END ......................C=100, gamma=0.1, kernel=rbf; total
```

```
time=   0.0s
[CV] END .....................C=100, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .....................C=100, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .....................C=100, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END .....................C=100, gamma=0.1, kernel=rbf; total
time=   0.0s
[CV] END ....................C=100, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ....................C=100, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ....................C=100, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ....................C=100, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ....................C=100, gamma=0.01, kernel=rbf; total
time=   0.0s
[CV] END ...................C=100, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ...................C=100, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ...................C=100, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ...................C=100, gamma=0.001, kernel=rbf; total
time=   0.0s
[CV] END ...................C=100, gamma=0.001, kernel=rbf; total
time=   0.0s
Mejores parámetros: {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
```

Explicación del Resultado de GridSearchCV para SVM

Contexto de la Búsqueda

- El objetivo de **GridSearchCV** es encontrar los mejores hiperparámetros para un modelo SVM.
- En este caso, se evaluaron combinaciones de los siguientes parámetros:
  - **C**: [0.1, 1, 10, 100] (regularización).
  - **gamma**: [1, 0.1, 0.01, 0.001] (control de influencia de puntos individuales en el kernel).
  - **kernel**: ['rbf'] (función de núcleo radial).

Detalle del Proceso

- **Número de combinaciones**: Hay 4 valores de **C** × 4 valores de **gamma** × 1 kernel = **16 combinaciones**.
- **Validación cruzada (CV)**: Se ejecutaron **5 particiones (folds)** para cada combinación, resultando en un total de **16 × 5 = 80 ajustes (fits)**.

Salida Proporcionada

- Cada línea representa la evaluación de una combinación de parámetros en uno de los 5 folds de validación cruzada.
- Ejemplo:

```
[CV] END .....................C=0.1, gamma=1, kernel=rbf;
total time=   0.0s
```

  - **[CV]**: Indica que es un fold de validación cruzada.
  - **C=0.1, gamma=1, kernel=rbf**: Parámetros evaluados en esta iteración.
  - **total time=0.0s**: Tiempo tomado para entrenar y evaluar el modelo en este fold (rápido debido al tamaño del conjunto de datos).

Selección de Mejores Parámetros

- Tras evaluar todas las combinaciones, **GridSearchCV** seleccionó los hiperparámetros que maximizaron la métrica de evaluación promedio (por ejemplo, precisión).
- Mejores parámetros encontrados:

```
{'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
```

- Interpretación:
  - **C=10**: Una penalización moderada por errores en el margen, equilibrando la complejidad del modelo.
  - **gamma=0.01**: Un alcance de influencia relativamente amplio para cada punto en el kernel RBF, lo que permite una mayor flexibilidad en la clasificación.
  - **kernel='rbf'**: Uso de un kernel radial para manejar datos no linealmente separables.
- Este proceso asegura que el modelo SVM esté optimizado para los datos proporcionados.
- Los mejores hiperparámetros encontrados pueden ahora usarse para entrenar un modelo final y realizar predicciones en nuevos datos.

Ventajas de SVM

- **Eficiencia en espacios de alta dimensionalidad**.
- **Flexibilidad**: Mediante el uso de kernels, SVM puede manejar tanto problemas lineales como no lineales.
- **Robustez**: Es menos propenso a sobreajustarse en comparación con otros algoritmos.

Limitaciones de SVM

- **Costo computacional**: Puede ser lento para conjuntos de datos grandes.
- **Sensibilidad a hiperparámetros**: La elección de C y γ es crucial para el rendimiento.
- **Interpretabilidad**: Difícil interpretar el modelo en espacios no lineales.

Optimización Visual: Búsqueda de Kernel

Finalmente, comparamos el rendimiento de diferentes kernels en un mismo conjunto de datos.

```
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
for kernel in kernels:
    svm_model = SVC(kernel=kernel, C=1, gamma='scale')
    svm_model.fit(X_train_svm, y_train_svm)
    score = svm_model.score(X_test_svm, y_test_svm)
    print(f"Kernel: {kernel}, Precisión: {score}")
```

```
Kernel: linear, Precisión: 0.8
Kernel: poly, Precisión: 0.7333333333333333
Kernel: rbf, Precisión: 0.8
Kernel: sigmoid, Precisión: 0.28888888888888886
```

Contexto del Experimento

- En este caso, se evaluaron distintos tipos de **kernels** para un modelo SVM.
- Los **kernels** probados fueron:
  - **linear**: Clasificador lineal.
  - **poly**: Kernel polinómico.
  - **rbf**: Kernel radial (función de base radial).
  - **sigmoid**: Kernel sigmoidal.

Configuración de los Parámetros

- **C=1**: Regularización moderada, balanceando margen amplio y errores de clasificación.
- **gamma='scale'**: Escala automática para el parámetro gamma, calculada como (1 / (n_{\ text{features}} \cdot \text{varianza})).

Proceso Realizado

1. Se entrenó un modelo SVM para cada kernel usando el conjunto de entrenamiento (**X_train, y_train**).
2. Cada modelo se evaluó en el conjunto de prueba (**X_test, y_test**).
3. La métrica utilizada fue la **precisión**, definida como el porcentaje de etiquetas correctamente clasificadas.

Resultados Obtenidos

- **Kernel: linear, Precisión: 0.8**
  - El kernel lineal tuvo un buen desempeño, con una precisión alta, lo que sugiere que los datos pueden ser bien separados por un hiperplano.
- **Kernel: poly, Precisión: 0.7333**
  - El kernel polinómico tuvo un desempeño más bajo, lo que podría indicar que la complejidad del modelo no fue adecuada, posiblemente debido a sobreajuste o subajuste.
- **Kernel: rbf, Precisión: 0.8**
  - El kernel RBF mostró un rendimiento similar al kernel lineal, siendo útil para datos no linealmente separables.

- **Kernel: sigmoid, Precisión: 0.2889**
  - El kernel sigmoidal tuvo la peor precisión, lo que sugiere que no fue adecuado para este conjunto de datos.

Interpretación

- La elección del kernel depende de la naturaleza de los datos:
  - **linear** es apropiado si los datos son aproximadamente linealmente separables.
  - **rbf** es útil cuando los datos tienen relaciones no lineales.
  - **poly** y **sigmoid** pueden ser más sensibles a los hiperparámetros y la complejidad de los datos.
- En este experimento, **linear** y **rbf** fueron los más efectivos, mientras que el **sigmoid** no resultó adecuado.
- El kernel lineal fue igual de preciso que el RBF, lo que sugiere que para estos datos, ambos kernels fueron igualmente efectivos.
- **Poly** y **sigmoid** no funcionaron tan bien, por lo que puede ser necesario ajustar más los hiperparámetros o considerar otras opciones de preprocesamiento de datos.
- Es recomendable realizar una optimización de hiperparámetros (e.g., con **GridSearchCV**) para afinar el rendimiento del modelo con kernels más complejos como **poly** y **sigmoid**.

Las **Máquinas de Soporte Vectorial (SVM)** son una herramienta poderosa y versátil para la clasificación, pero requieren una buena selección de parámetros y ajustes para maximizar su rendimiento.

Clasificador K-Nearest Neighbors (KNN) con $k=10$

El algoritmo K-Nearest Neighbors (KNN) es un método de clasificación supervisada ampliamente utilizado debido a su simplicidad y eficacia. Este algoritmo clasifica un nuevo punto de datos basándose en los puntos más cercanos de su vecindad en el conjunto de entrenamiento. A continuación, profundizamos en cómo funciona el KNN con $k=10$.

Conceptos Básicos

1. **Distancia**:
   - El KNN utiliza medidas de distancia para identificar los puntos más cercanos. La **distancia Euclidiana** es la métrica más común y se define como:

     $$d(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

     donde x y y son puntos en un espacio n-dimensional.
   - Otras métricas posibles incluyen la **distancia Manhattan**:

     $$d(x,y)=\sum_{i=1}^{n}|x_i-y_i)$$

     y la **distancia de Minkowski**, que generaliza las anteriores.
2. **Número de Vecinos (k)**:

- El parámetro k controla cuántos puntos cercanos se consideran para la clasificación.
- En nuestro caso, $k = 10$, lo que significa que se analizarán los 10 vecinos más cercanos para tomar una decisión.

3. **Votación Mayoritaria**:
   - El KNN clasifica un punto basándose en una votación entre los k vecinos más cercanos. La clase más frecuente entre estos vecinos es asignada al nuevo punto.

Pasos del Algoritmo KNN

1. **Calcular la Distancia**:
   - Determinar la distancia entre el nuevo punto y todos los puntos en el conjunto de entrenamiento.
2. **Identificar los Vecinos Más Cercanos**:
   - Ordenar los puntos del conjunto de entrenamiento según la distancia calculada.
   - Seleccionar los k puntos más cercanos.
3. **Realizar la Votación**:
   - Contar las clases de los k vecinos más cercanos.
   - Asignar al nuevo punto la clase con mayor frecuencia.
4. **Clasificar el Punto**:
   - Asignar la clase mayoritaria al punto nuevo, completando la predicción.

Ventajas del KNN

- **Simplicidad**: No requiere modelos complejos ni entrenamientos largos.
- **Eficiencia en datos pequeños**: Es efectivo para conjuntos de datos de menor tamaño.
- **No paramétrico**: No hace suposiciones sobre la distribución subyacente de los datos.

Limitaciones del KNN

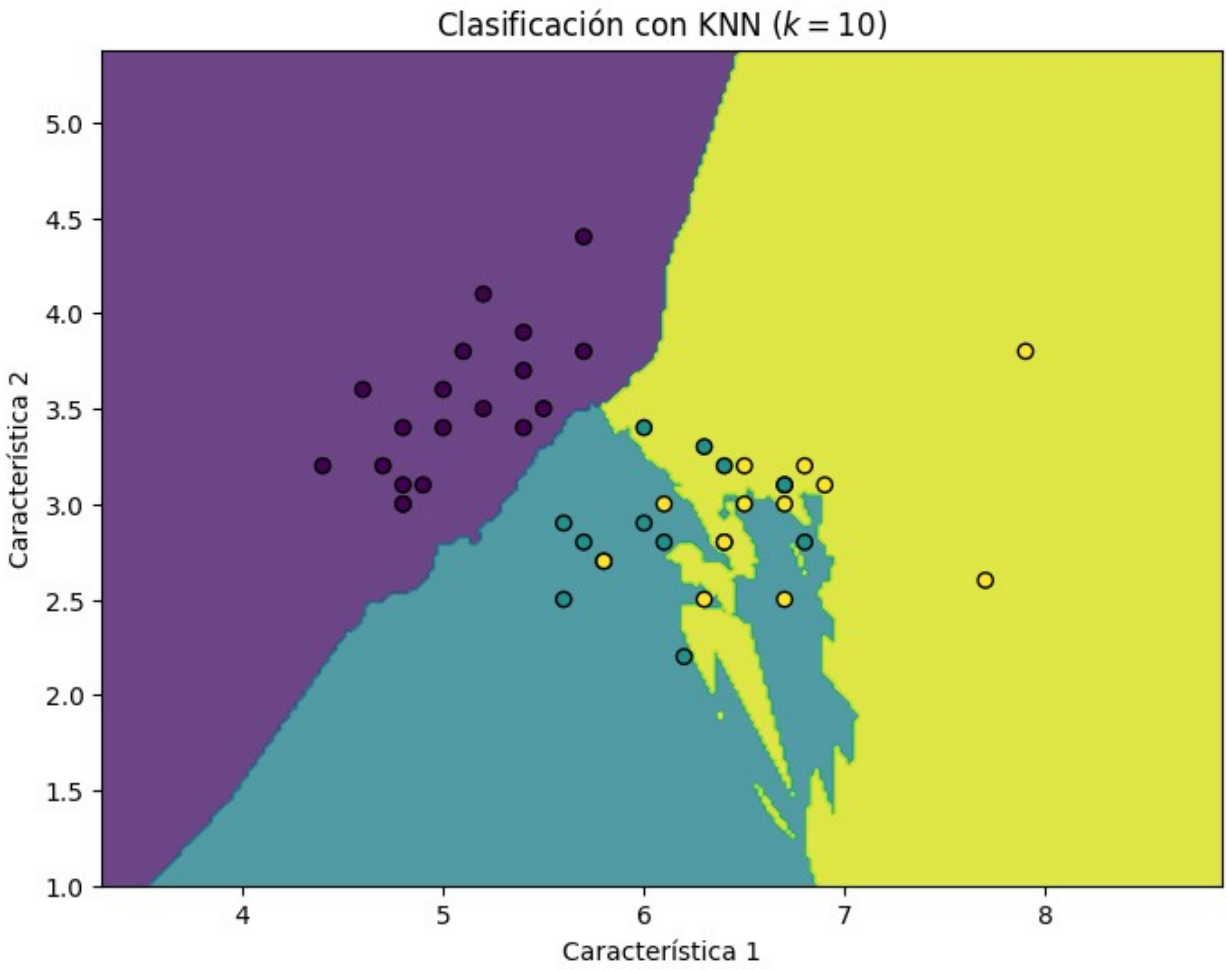
- **Sensibilidad al ruido**: Los puntos atípicos pueden influir en la clasificación.
- **Costo computacional**: Calcular las distancias para todos los puntos del conjunto de entrenamiento puede ser costoso en términos computacionales para conjuntos de datos grandes.
- **Elección de k**: Seleccionar un valor apropiado para k es crucial y puede requerir optimización.

Visualización de KNN

A continuación, implementamos el algoritmo KNN con $k = 10$ y visualizamos cómo clasifica un conjunto de datos bidimensional.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
import matplotlib.pyplot as plt

# Cargar el conjunto de datos Iris
# El conjunto de datos Iris tiene 150 muestras de 3 clases, con 4
```

```
características.
iris_knn = load_iris()
X0_knn = iris_knn.data[:, :2]  # Tomamos solo las dos primeras
características para facilitar la visualización 2D
y0_knn = iris_knn.target

# Dividir en conjunto de entrenamiento y prueba
# test_size: Proporción del conjunto de prueba (30% de las muestras se
utilizan para pruebas).
# random_state: Aleatoriedad para asegurar divisiones reproducibles.
X_train_knn, X_test_knn, y_train_knn, y_test_knn =
train_test_split(X0_knn, y0_knn, test_size=0.3, random_state=42)

# Entrenar el clasificador KNN
# n_neighbors: Número de vecinos a considerar (k=10 en este caso).
knn_model = KNeighborsClassifier(n_neighbors=10)
knn_model.fit(X_train_knn, y_train_knn)

# Visualizar los resultados
# Configurar el espacio de la gráfica
h_knn = 0.02  # Tamaño del paso en la malla.
x_min_knn, x_max_knn = X0_knn[:, 0].min() - 1, X0_knn[:, 0].max() + 1
# Rango de valores para la característica 1.
y_min_knn, y_max_knn = X0_knn[:, 1].min() - 1, X0_knn[:, 1].max() + 1
# Rango de valores para la característica 2.

# Crear una malla de puntos para predecir sus etiquetas
xx_knn, yy_knn = np.meshgrid(np.arange(x_min_knn, x_max_knn, h_knn),
np.arange(y_min_knn, y_max_knn, h_knn))
Z_knn = knn_model.predict(np.c_[xx_knn.ravel(), yy_knn.ravel()])  #
Predecir para cada punto de la malla.
Z_knn = Z_knn.reshape(xx_knn.shape)  # Dar forma a las predicciones
para que coincidan con la malla.

# Graficar el resultado
plt.figure(figsize=(8, 6))
plt.contourf(xx_knn, yy_knn, Z_knn, alpha=0.8, cmap='viridis')  #
Fondo que indica la clasificación predicha.
plt.scatter(X_test_knn[:, 0], X_test_knn[:, 1], c=y_test_knn,
edgecolor='k', cmap='viridis')  # Puntos del conjunto de prueba.
plt.title("Clasificación con KNN ($k=10$)")
plt.xlabel("Característica 1")
plt.ylabel("Característica 2")
plt.show()
```



Evaluación del Modelo

Evaluar el rendimiento del modelo KNN es crucial para determinar su eficacia. Las métricas comunes incluyen **Precisión**, **Recall**, **F1-Score**, y la **Matriz de Confusión**. Estas métricas se derivan de los siguientes términos fundamentales:

Términos Fundamentales

1. **Verdaderos Positivos (TP)**:
   - Cantidad de ejemplos positivos que el modelo clasifica correctamente como positivos.
2. **Falsos Positivos (FP)**:
   - Cantidad de ejemplos negativos que el modelo clasifica incorrectamente como positivos.
3. **Verdaderos Negativos (TN)**:
   - Cantidad de ejemplos negativos que el modelo clasifica correctamente como negativos.
4. **Falsos Negativos (FN)**:
   - Cantidad de ejemplos positivos que el modelo clasifica incorrectamente como negativos.

Métricas Clave

1. **Precisión (Precision)**:
   - Proporción de predicciones positivas correctas frente al total de predicciones positivas realizadas.

$$\text{Precisión} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

   - Mide la exactitud del modelo para identificar correctamente los positivos.
2. **Sensibilidad (Recall)**:
   - También conocida como **tasa de verdaderos positivos**, mide la proporción de positivos correctamente identificados.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

3. **F1-Score**:
   - Promedio armónico entre la precisión y el recall. Es útil cuando existe un desequilibrio entre las clases.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

4. **Exactitud (Accuracy)**:
   - Proporción de predicciones correctas frente al total de ejemplos.

$$\text{Exactitud} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

```
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import LabelBinarizer

# Predicciones
y_pred_knn = knn_model.predict(X_test_knn)

# Reporte de clasificación
print("Reporte de clasificación:")
print(classification_report(y_test_knn, y_pred_knn))

# Matriz de confusión
cm_knn = confusion_matrix(y_test_knn, y_pred_knn)
print("Matriz de Confusión:")
print(cm_knn)

# AUC para cada clase (utilizando One-vs-Rest)
lb_knn = LabelBinarizer()
y_test_bin_knn = lb_knn.fit_transform(y_test_knn)  # Convertir las
etiquetas de clase a formato binario
y_pred_prob_knn = knn_model.predict_proba(X_test_knn)  # Obtener las
probabilidades de predicción

# Calcular el AUC para cada clase
auc_scores_knn = []
```

```
for i in range(y_test_bin_knn.shape[1]):
    auc = roc_auc_score(y_test_bin_knn[:, i], y_pred_prob_knn[:, i])
    auc_scores_knn.append(auc)

# Mostrar AUC para cada clase
for i, auc in enumerate(auc_scores_knn):
    print(f"AUC para la clase {i}: {auc}")
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.62      0.62      0.62        13
           2       0.62      0.62      0.62        13

    accuracy                           0.78        45
   macro avg       0.74      0.74      0.74        45
weighted avg       0.78      0.78      0.78        45

Matriz de Confusión:
[[19  0  0]
 [ 0  8  5]
 [ 0  5  8]]
AUC para la clase 0: 1.0
AUC para la clase 1: 0.8545673076923077
AUC para la clase 2: 0.8569711538461539
```

Reporte de Clasificación

El reporte generado incluye los siguientes términos para cada clase:

- **Precision**: Proporción de predicciones correctas dentro de todas las predicciones positivas realizadas.
- **Recall**: Proporción de ejemplos positivos correctamente clasificados.
- **F1-Score**: Promedio armónico de la precisión y el recall.
- **Support**: Número de ejemplos verdaderos de la clase.

Ejemplo del reporte:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.62      0.62      0.62        13
           2       0.62      0.62      0.62        13

   accuracy                           0.78        45
   macro avg       0.74      0.74      0.74        45
weighted avg       0.78      0.78      0.78        45
```

- **Clase 0**:

  - Precisión: 100% de las predicciones positivas para la clase 0 fueron correctas.
  - Recall: El modelo identificó correctamente el 100% de los ejemplos verdaderos de la clase 0.
  - F1-Score: El balance entre precisión y recall para la clase 0 es del 100%.
- **Clase 1**:
  - Precisión: 62% de las predicciones positivas para la clase 1 fueron correctas.
  - Recall: El modelo identificó correctamente el 62% de los ejemplos verdaderos de la clase 1.
  - F1-Score: El balance entre precisión y recall para la clase 1 es del 62%.
- **Clase 2**:
  - Precisión: 62% de las predicciones positivas para la clase 2 fueron correctas.
  - Recall: El modelo identificó correctamente el 62% de los ejemplos verdaderos de la clase 2.
  - F1-Score: El balance entre precisión y recall para la clase 2 es del 62%.

Matriz de Confusión

La matriz de confusión detalla las predicciones correctas e incorrectas por clase. Para este modelo, la matriz es:

```
[[19  0  0]
 [ 0  8  5]
 [ 0  5  8]]
```

- **Interpretación**:
  - **Clase 0**:
    - Verdaderos Positivos (TP): 19 (correctamente clasificados como clase 0).
    - Falsos Positivos (FP): 0 (no hay ejemplos incorrectamente clasificados como clase 0).
    - Falsos Negativos (FN): 0 (todos los ejemplos de la clase 0 fueron correctamente clasificados).
  - **Clase 1**:
    - Verdaderos Positivos (TP): 8 (correctamente clasificados como clase 1).
    - Falsos Positivos (FP): 5 (ejemplos de clase 2 clasificados incorrectamente como clase 1).
    - Falsos Negativos (FN): 0 (ningún ejemplo de la clase 1 fue clasificado incorrectamente como clase 2).
  - **Clase 2**:
    - Verdaderos Positivos (TP): 8 (correctamente clasificados como clase 2).
    - Falsos Positivos (FP): 5 (ejemplos de clase 1 clasificados incorrectamente como clase 2).
    - Falsos Negativos (FN): 0 (ningún ejemplo de la clase 2 fue clasificado incorrectamente como clase 1).

Área Bajo la Curva ROC (AUC)

El AUC mide la capacidad del modelo para distinguir entre clases positivas y negativas. Un AUC cercano a 1 indica un excelente rendimiento. Para este modelo, el AUC para cada clase es:

```
AUC para la clase 0: 1.0
AUC para la clase 1: 0.8545673076923077
AUC para la clase 2: 0.8569711538461539
```

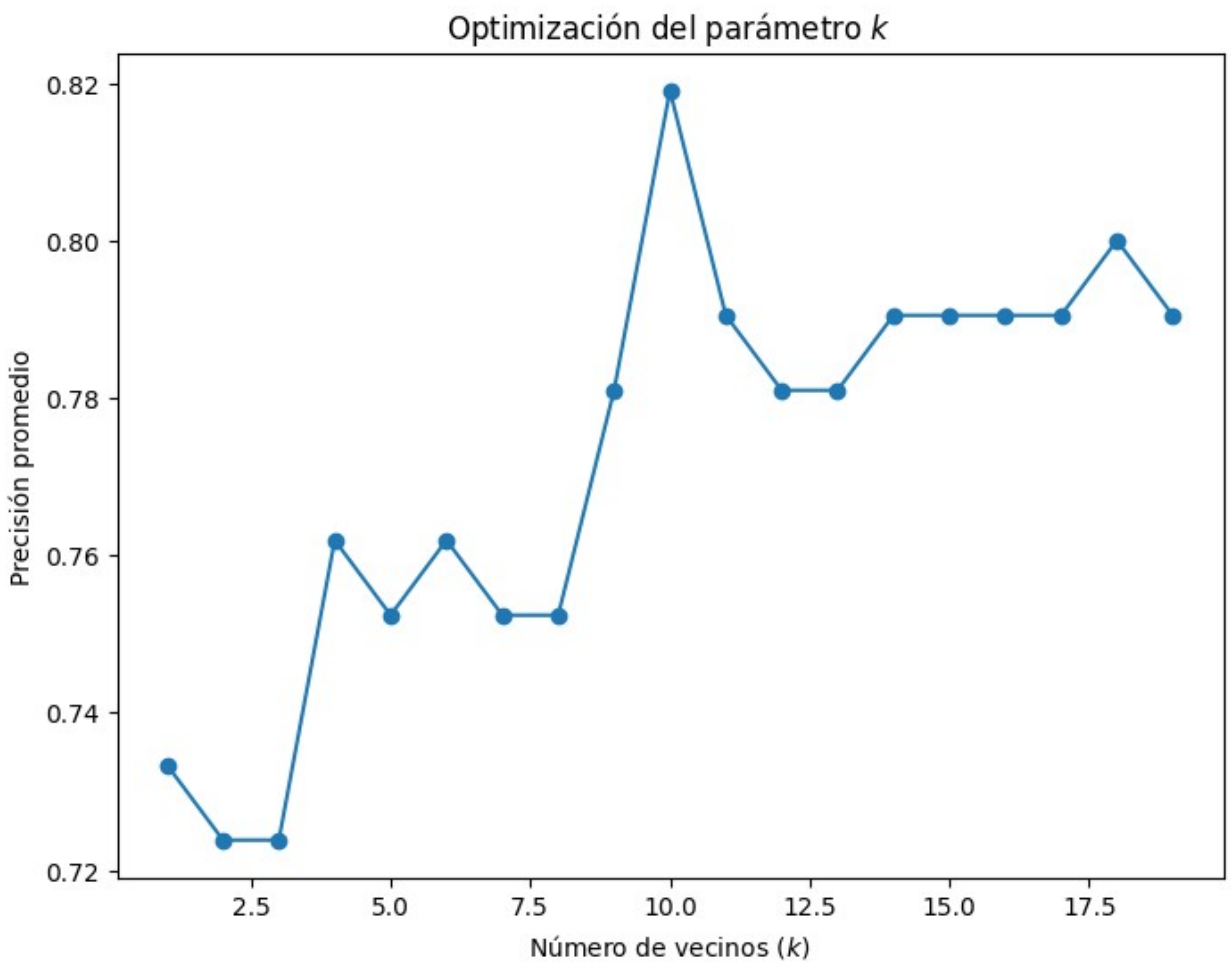
Estas métricas proporcionan una visión integral del rendimiento del modelo. En este caso, la precisión, recall, F1-score y AUC sugieren que el modelo tiene un buen desempeño, aunque existen oportunidades para mejorar, como la reducción de falsos positivos y falsos negativos.

Optimización de k

La elección de k afecta significativamente el rendimiento de KNN. Algunos métodos para seleccionar el mejor valor incluyen:

1. **Validación Cruzada**:
   - Probar diferentes valores de k usando validación cruzada y seleccionar el que maximice la precisión promedio.
2. **Regla General**:
   - Un valor común es $\sqrt{n}$, donde n es el número de muestras en el conjunto de entrenamiento.

```
from sklearn.model_selection import cross_val_score
import numpy as np

# Probar diferentes valores de k
k_values_knn = range(1, 20)
accuracies_knn = []

for k_knn in k_values_knn:
    knn_model = KNeighborsClassifier(n_neighbors=k_knn)
    scores_knn = cross_val_score(knn_model, X_train_knn, y_train_knn,
cv=5)
    accuracies_knn.append(scores_knn.mean())

# Graficar la precisión promedio para diferentes valores de k
plt.figure(figsize=(8, 6))
plt.plot(k_values_knn, accuracies_knn, marker='o')
plt.title("Optimización del parámetro $k$")
plt.xlabel("Número de vecinos ($k$)")
plt.ylabel("Precisión promedio")
plt.show()
```



Con este análisis, el modelo KNN con $k=10$ puede ser ajustado y evaluado de manera efectiva para problemas específicos.

Clasificador K-Nearest Neighbors (KNN) con $k=10$

Conceptos Básicos

1. **Distancia**: KNN utiliza una medida de distancia para encontrar los vecinos más cercanos. La distancia más comúnmente utilizada es la distancia Euclidiana, definida como:

$$d(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

donde x y y son dos puntos en un espacio n-dimensional.

1. **Vecinos más Cercanos**: Para clasificar un nuevo punto, el algoritmo encuentra los k puntos en el conjunto de entrenamiento que están más cerca de ese punto. En nuestro caso, $k=10$.

Pasos del Algoritmo KNN

1. **Calcular la Distancia**: Calcular la distancia entre el nuevo punto y todos los puntos del conjunto de entrenamiento.
2. **Encontrar los Vecinos más Cercanos**: Seleccionar los 10 puntos más cercanos basándose en las distancias calculadas.
3. **Votación Mayoritaria**: Determinar la clase del nuevo punto a través de una votación mayoritaria entre los 10 vecinos más cercanos. La clase que aparece con más frecuencia entre los vecinos es la clase asignada al nuevo punto.

```
# Importar el módulo de vecinos (neighbors) de scikit-learn, que contiene algoritmos para clasificación y regresión basados en vecinos más cercanos.
from sklearn import neighbors

# Crear una instancia del clasificador K-Nearest Neighbors (KNN) con k=10 vecinos.
knn = neighbors.KNeighborsClassifier(n_neighbors=10)

# Entrenar el clasificador KNN utilizando el método fit(), que ajusta el modelo según los datos de entrenamiento X y las etiquetas y.
knn.fit(X, y)
```

```
KNeighborsClassifier(n_neighbors=10)
```

KNN explained

```
# Importar el módulo pickle, que se utiliza para serializar y deserializar objetos de Python.
import pickle

# Abrir un archivo en modo escritura binaria ('wb') para guardar el clasificador.
ofname = open('my_classifier.pkl', 'wb')

# Serializar el clasificador KNN y guardarlo en el archivo.
s = pickle.dump(knn, ofname)

# Cerrar el archivo para asegurarse de que los datos se escriben correctamente.
ofname.close()

# Imprimir el resultado de la operación de dump (debería ser None).
print(s)

# Limpiar el espacio de nombres de la sesión actual.
%reset -f
```

```
None
```

```
# Verificar que no tenemos la variable en el espacio de nombres.
Debería dar un NameError
##print(knn)
```

Ahora vamos a **explotar** el modelo. En este ejemplo, usamos los mismos datos, pero en general, se deben proporcionar nuevos datos no vistos al clasificador entrenado.

```
# Importar los módulos necesarios
from sklearn import neighbors
from sklearn import datasets
import pickle

# Abrir el archivo 'my_classifier.pkl' en modo lectura binaria ('rb')
para cargar el modelo guardado.
ofname = open('my_classifier.pkl', 'rb')

# Cargar el conjunto de datos de dígitos de sklearn para utilizarlo en
la evaluación del modelo.
digits = datasets.load_digits()
X = digits.data  # Cargar las características (imágenes aplanadas de
los dígitos)

# Cargar el modelo KNN previamente guardado en el archivo
'my_classifier.pkl'.
knn = pickle.load(ofname)

# Cerrar el archivo después de cargar el modelo para liberar recursos.
ofname.close()

# Ahora puedes utilizar el modelo cargado (knn) para hacer
predicciones o evaluaciones.

# Calcular la predicción utilizando el modelo KNN cargado.
# Aquí estamos prediciendo el dígito correspondiente a la primera
imagen en el conjunto de datos.
print(knn.predict(X[0,:].reshape(1, -1)))
```

```
[0]
```

```
# Obtener las etiquetas (valores objetivo) del conjunto de datos.
y = digits.target
y
```

```
array([0, 1, 2, ..., 8, 9, 8])
```

```
# Calcular las predicciones para todo el conjunto de datos X
utilizando el modelo KNN cargado.
y_pred = knn.predict(X)
y_pred
```

```
array([0, 1, 2, ..., 8, 9, 8])
```

Para evaluar el rendimiento del clasificador, se puede utilizar la precisión de las predicciones:

$$acc = \frac{\#\text{de predicciones correctas}}{N}$$

Cada estimador tiene un método *.score()* que invoca la métrica de puntuación predeterminada. En el caso de los k Vecinos Más Cercanos, esta es la precisión de clasificación.

```
# Calcular el rendimiento del modelo KNN en el conjunto de entrenamiento.
# ¡SI SABES LO QUE ESTÁS HACIENDO, NUNCA DEBES HACER ESTO DE NUEVO!
knn.score(X, y)
```

```
0.9855314412910406
```

Clasificador de Árbol de Decisión

¿Qué es un clasificador de árbol de decisión?

Un **clasificador de árbol de decisión** es un modelo predictivo de aprendizaje supervisado que aprende reglas de decisión a partir de las características de los datos. Es ampliamente utilizado debido a su simplicidad, facilidad de interpretación y capacidad para manejar tanto datos numéricos como categóricos.

Este clasificador construye un árbol binario donde cada nodo representa una característica del conjunto de datos, y las ramas representan los valores posibles de esa característica. El árbol divide el conjunto de datos en subconjuntos más pequeños hasta que se alcanza un valor final, que es la clase o predicción.

Estructura del Árbol de Decisión

La estructura básica de un árbol de decisión incluye los siguientes elementos:

- **Nodo Raíz**: Es el primer nodo del árbol y representa la característica más importante para realizar la primera división en los datos.
- **Nodos Internos**: Son los nodos que siguen al nodo raíz y representan las características utilizadas para dividir aún más los datos.
- **Hojas (Nodos Terminales)**: Son los nodos al final del árbol y representan las decisiones finales, es decir, las clases o valores predichos.

En resumen, el árbol de decisión sigue un proceso de "división" de los datos en función de características y valores, hasta llegar a una predicción final.

Proceso de Construcción del Árbol de Decisión

El proceso de construcción de un árbol de decisión consta de los siguientes pasos:

1. **Selección de la Mejor Característica**:
   - El primer paso es elegir la característica más significativa que divide el conjunto de datos en los grupos más puros posibles.

- Se utiliza una métrica como **Ganancia de Información** o **Índice de Gini** para evaluar qué característica proporciona la mejor división de las clases.

2. **División del Conjunto de Datos**:
   - Una vez que se ha seleccionado la característica, el conjunto de datos se divide en dos subconjuntos según los valores de esa característica.
3. **Recursividad**:
   - Este proceso de selección y división se repite recursivamente en cada uno de los subconjuntos resultantes.
   - El proceso continúa hasta que se alcanza algún criterio de parada, como una profundidad máxima del árbol, un número mínimo de muestras por hoja, o cuando los datos ya no pueden dividirse más.

Beneficios del Árbol de Decisión

- **Interpretación Fácil**:
  - Los árboles de decisión son extremadamente fáciles de interpretar y visualizar. Esto los hace ideales para problemas donde la transparencia y la explicabilidad son esenciales.
  - Es posible entender cómo llega el modelo a una conclusión siguiendo el camino de divisiones en el árbol.
- **Manejo de Características Mixtas**:
  - Los árboles de decisión no requieren un preprocesamiento extenso, ya que pueden manejar tanto características numéricas como categóricas de manera natural.
  - Esto facilita el trabajo con conjuntos de datos diversos sin la necesidad de transformar las variables.
- **Robustez ante Datos Ruidosos**:
  - Son relativamente robustos a los datos ruidosos y pueden manejar valores faltantes de manera adecuada.
  - Debido a su capacidad para realizar divisiones no lineales, los árboles de decisión pueden captar relaciones complejas entre las variables.

Limitaciones del Árbol de Decisión

- **Sobreajuste (Overfitting)**:
  - Si el árbol crece demasiado (es decir, se profundiza mucho), puede sobreajustarse a los datos de entrenamiento, lo que lleva a un rendimiento deficiente en datos nuevos o no vistos.
  - El árbol aprende detalles irrelevantes y ruido, lo que no generaliza bien a nuevos conjuntos de datos.
- **Bias-Variance Tradeoff**:
  - La profundidad del árbol afecta el **sesgo** y la **varianza** del modelo. Un árbol muy superficial puede tener un sesgo alto (bajo rendimiento), mientras que un árbol profundo puede tener una alta varianza (sobreajuste).
  - Encontrar el equilibrio adecuado en la profundidad del árbol es crucial para obtener un modelo generalizable.

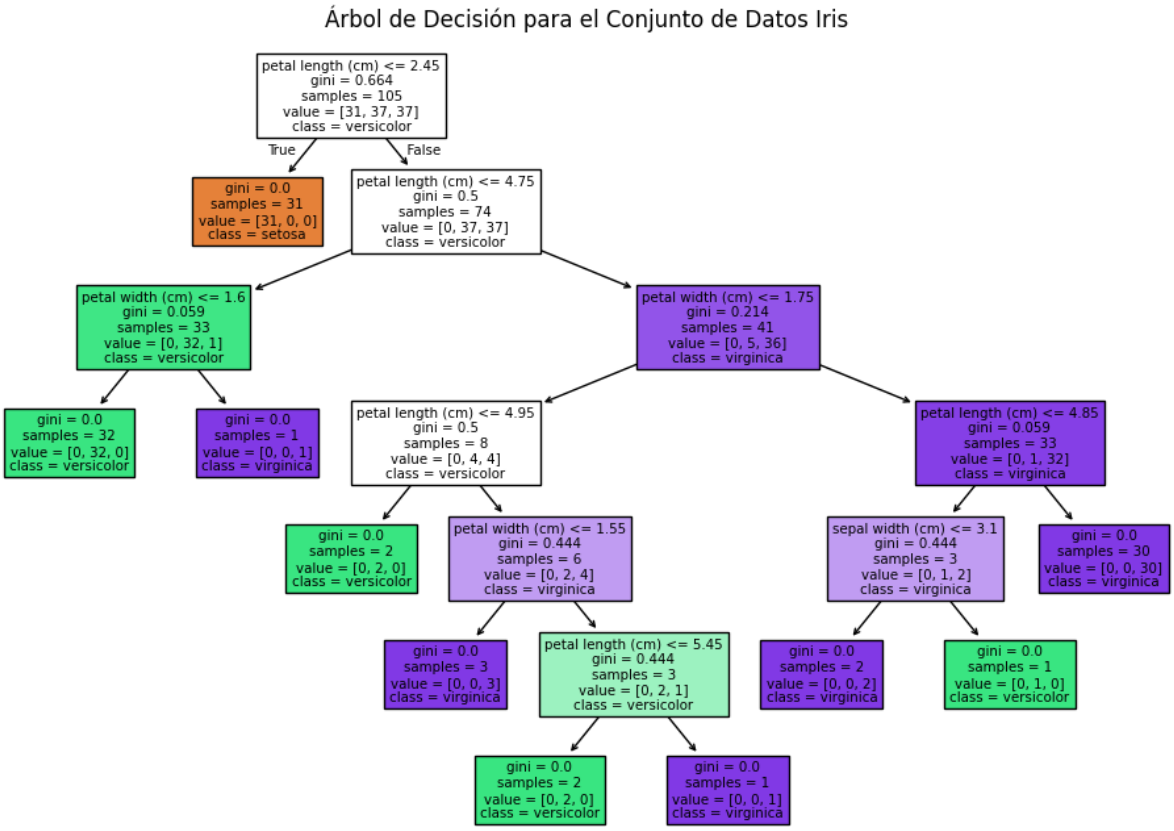
Aplicaciones Comunes

- **Clasificación**:
  - Clasificación de correos electrónicos como spam o no spam, diagnóstico médico, detección de fraudes, predicción de churn en clientes, entre otros.
- **Regresión**:
  - Predicción de precios de viviendas, valores de acciones, pronósticos económicos, entre otros.

Visualización y Construcción del Árbol de Decisión

A continuación, implementamos un clasificador de árbol de decisión utilizando el conjunto de datos Iris y visualizamos el árbol resultante.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

# Cargar el conjunto de datos Iris
iris_dtc = load_iris()
X0_dtc = iris_dtc.data
y0_dtc = iris_dtc.target

# Dividir los datos en conjunto de entrenamiento y prueba
X_train_dtc, X_test_dtc, y_train_dtc, y_test_dtc =
train_test_split(X0_dtc, y0_dtc, test_size=0.3, random_state=42)

# Entrenar el modelo de árbol de decisión
dtc_model = DecisionTreeClassifier(random_state=42)
dtc_model.fit(X_train_dtc, y_train_dtc)

# Visualizar el árbol de decisión
plt.figure(figsize=(12, 8))
plot_tree(dtc_model, filled=True,
feature_names=iris_dtc.feature_names,
class_names=iris_dtc.target_names)
plt.title("Árbol de Decisión para el Conjunto de Datos Iris")
plt.show()
```



Evaluación del Modelo

Evaluar el rendimiento del árbol de decisión es fundamental para entender su efectividad. Algunas métricas comunes incluyen **Precisión**, **Recall**, **F1-Score**, y la **Matriz de Confusión**. Estas métricas se derivan de los siguientes términos fundamentales:

Términos Fundamentales

1. **Verdaderos Positivos (TP)**:
   - Cantidad de ejemplos positivos que el modelo clasifica correctamente como positivos.
2. **Falsos Positivos (FP)**:
   - Cantidad de ejemplos negativos que el modelo clasifica incorrectamente como positivos.
3. **Verdaderos Negativos (TN)**:
   - Cantidad de ejemplos negativos que el modelo clasifica correctamente como negativos.
4. **Falsos Negativos (FN)**:
   - Cantidad de ejemplos positivos que el modelo clasifica incorrectamente como negativos.

Métricas Clave

1. **Precisión (Precision)**:

- Proporción de predicciones positivas correctas frente al total de predicciones positivas realizadas.

$$\text{Precisión} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- Mide la exactitud del modelo para identificar correctamente los positivos.

2. **Sensibilidad (Recall)**:
   - También conocida como **tasa de verdaderos positivos**, mide la proporción de positivos correctamente identificados.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

3. **F1-Score**:
   - Promedio armónico entre la precisión y el recall. Es útil cuando existe un desequilibrio entre las clases.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

4. **Exactitud (Accuracy)**:
   - Proporción de predicciones correctas frente al total de ejemplos.

$$\text{Exactitud} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

```
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import LabelBinarizer

# Predicciones
y_pred_dtc = dtc_model.predict(X_test_dtc)

# Reporte de clasificación
print("Reporte de clasificación:")
print(classification_report(y_test_dtc, y_pred_dtc))

# Matriz de confusión
cm_dtc = confusion_matrix(y_test_dtc, y_pred_dtc)
print("Matriz de Confusión:")
print(cm_dtc)

# AUC para cada clase (utilizando One-vs-Rest)
lb_dtc = LabelBinarizer()
y_test_bin_dtc = lb_dtc.fit_transform(y_test_dtc)  # Convertir las
etiquetas de clase a formato binario
y_pred_prob_dtc = dtc_model.predict_proba(X_test_dtc)  # Obtener las
probabilidades de predicción

# Calcular el AUC para cada clase
auc_scores_dtc = []
for i in range(y_test_bin_dtc.shape[1]):
    auc = roc_auc_score(y_test_bin_dtc[:, i], y_pred_prob_dtc[:, i])
    auc_scores_dtc.append(auc)
```

```python
# Mostrar AUC para cada clase
for i, auc in enumerate(auc_scores_dtc):
    print(f"AUC para la clase {i}: {auc}")
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

Matriz de Confusión:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
AUC para la clase 0: 1.0
AUC para la clase 1: 1.0
AUC para la clase 2: 1.0
```

Reporte de Clasificación

El reporte generado incluye los siguientes términos para cada clase:

- **Precision**: Proporción de predicciones correctas dentro de todas las predicciones positivas realizadas.
- **Recall**: Proporción de ejemplos positivos correctamente clasificados.
- **F1-Score**: Promedio armónico de la precisión y el recall.
- **Support**: Número de ejemplos verdaderos de la clase.

Ejemplo del reporte:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

   accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

- **Clase 0**:
  - Precisión: 100% de las predicciones positivas para la clase 0 fueron correctas.
  - Recall: El modelo identificó correctamente el 100% de los ejemplos verdaderos de la clase 0.

  - F1-Score: El balance entre precisión y recall para la clase 0 es del 100%.
- **Clase 1**:
  - Precisión: 100% de las predicciones positivas para la clase 1 fueron correctas.
  - Recall: El modelo identificó correctamente el 100% de los ejemplos verdaderos de la clase 1.
  - F1-Score: El balance entre precisión y recall para la clase 1 es del 100%.
- **Clase 2**:
  - Precisión: 100% de las predicciones positivas para la clase 2 fueron correctas.
  - Recall: El modelo identificó correctamente el 100% de los ejemplos verdaderos de la clase 2.
  - F1-Score: El balance entre precisión y recall para la clase 2 es del 100%.

Matriz de Confusión

La matriz de confusión detalla las predicciones correctas e incorrectas por clase. Para este modelo, la matriz es:

```
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

- **Interpretación**:
  - **Clase 0**:
    - Verdaderos Positivos (TP): 19 (correctamente clasificados como clase 0).
    - Falsos Positivos (FP): 0 (no hay ejemplos incorrectamente clasificados como clase 0).
    - Falsos Negativos (FN): 0 (todos los ejemplos de la clase 0 fueron correctamente clasificados).
  - **Clase 1**:
    - Verdaderos Positivos (TP): 13 (correctamente clasificados como clase 1).
    - Falsos Positivos (FP): 0 (no hay ejemplos incorrectamente clasificados como clase 1).
    - Falsos Negativos (FN): 0 (todos los ejemplos de la clase 1 fueron correctamente clasificados).
  - **Clase 2**:
    - Verdaderos Positivos (TP): 13 (correctamente clasificados como clase 2).
    - Falsos Positivos (FP): 0 (no hay ejemplos incorrectamente clasificados como clase 2).
    - Falsos Negativos (FN): 0 (todos los ejemplos de la clase 2 fueron correctamente clasificados).

Área Bajo la Curva ROC (AUC)

El AUC mide la capacidad del modelo para distinguir entre clases positivas y negativas. Un AUC cercano a 1 indica un excelente rendimiento. Para este modelo, el AUC para cada clase es:

```
AUC para la clase 0: 1.0
AUC para la clase 1: 1.0
AUC para la clase 2: 1.0
```

Estas métricas proporcionan una visión integral del rendimiento del modelo. En este caso, la precisión, recall, F1-score y AUC sugieren que el modelo tiene un excelente desempeño, con una clasificación perfecta en cada clase.

Optimización del Árbol de Decisión

Para evitar el sobreajuste, es esencial ajustar la complejidad del árbol mediante la **selección de hiperparámetros**. Algunos de los hiperparámetros clave que se pueden optimizar incluyen:

1. **Máxima Profundidad del Árbol**:
   - Limitar la profundidad del árbol para evitar que crezca demasiado y se sobreajuste.
2. **Número Mínimo de Muestras por Hoja**:
   - Asegurar que cada hoja tenga un número mínimo de muestras para evitar divisiones demasiado finas.
3. **Criterio de División (Gini vs. Entropía)**:
   - Probar diferentes criterios para la división de los nodos (Índice de Gini vs. Ganancia de Información) y ver cuál produce mejores resultados.

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import LabelBinarizer
from sklearn.tree import DecisionTreeClassifier

# Definir los hiperparámetros a ajustar
param_grid_dtc = {
    'max_depth': [3, 5, 7, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Realizar búsqueda en cuadrícula con validación cruzada
grid_search_dtc =
GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid_dtc,
cv=5)
grid_search_dtc.fit(X_train_dtc, y_train_dtc)

# Mejor conjunto de hiperparámetros
print(f"Mejores hiperparámetros: {grid_search_dtc.best_params_}")

# Evaluar el modelo con los mejores parámetros
best_clf_dtc = grid_search_dtc.best_estimator_
y_pred_best_dtc = best_clf_dtc.predict(X_test_dtc)
```

```python
# Reporte de clasificación
print("Reporte de clasificación:")
print(classification_report(y_test_dtc, y_pred_best_dtc))

# Matriz de confusión
cm_dtc = confusion_matrix(y_test_dtc, y_pred_best_dtc)
print("Matriz de Confusión:")
print(cm_dtc)

# AUC para cada clase (utilizando One-vs-Rest)
lb_dtc = LabelBinarizer()
y_test_bin_dtc = lb_dtc.fit_transform(y_test_dtc)  # Convertir las etiquetas de clase a formato binario
y_pred_prob_dtc = best_clf_dtc.predict_proba(X_test_dtc)  # Obtener las probabilidades de predicción

# Calcular el AUC para cada clase
auc_scores_dtc = []
for i in range(y_test_bin_dtc.shape[1]):
    auc = roc_auc_score(y_test_bin_dtc[:, i], y_pred_prob_dtc[:, i])
    auc_scores_dtc.append(auc)

# Mostrar AUC para cada clase
for i, auc in enumerate(auc_scores_dtc):
    print(f"AUC para la clase {i}: {auc}")
```

```
Mejores hiperparámetros: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 10}
Reporte de clasificación:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

Matriz de Confusión:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
AUC para la clase 0: 1.0
AUC para la clase 1: 1.0
AUC para la clase 2: 1.0
```

Con estos ajustes y evaluaciones, podemos optimizar el rendimiento del clasificador de árbol de decisión para obtener un modelo preciso y generalizable.

EJERCICIO: Pon todos los pasos en una misma celda y mira de ejecutarla.

```python
# EJERCICIO: Llena esta celda con la solución al ejercicio.

# Importar el conjunto de datos de dígitos desde sklearn
from sklearn import datasets
data = datasets.load_digits()
X, y = data.data, data.target  # Cargar características (X) y etiquetas (y)

# Importar los clasificadores KNN y Árbol de Decisión desde sklearn
from sklearn import neighbors
from sklearn import tree

## LLENA EL RESTO CON EL CÓDIGO DEL EJERCICIO

# Crear un clasificador KNN con k=5 vecinos
knn = neighbors.KNeighborsClassifier(n_neighbors=5)
knn.fit(X, y)  # Entrenar el clasificador KNN con los datos de entrenamiento

# Calcular y mostrar el rendimiento del modelo KNN en el conjunto de entrenamiento
score_knn = knn.score(X, y)
print(f"Rendimiento del modelo KNN en el conjunto de entrenamiento: {score_knn:.4f}")

# Calcular predicciones utilizando el modelo KNN para comparar con los valores originales
y_pred_knn = knn.predict(X)

# Mostrar algunos ejemplos de comparación entre valores originales y predicciones por KNN
print("\nEjemplos de comparación (valor original vs. predicción por KNN):")
for i in range(10):  # Mostrar los primeros 10 ejemplos
    print(f"Valor original: {y[i]}, Predicción KNN: {y_pred_knn[i]}")
```

```
Rendimiento del modelo KNN en el conjunto de entrenamiento: 0.9905

Ejemplos de comparación (valor original vs. predicción por KNN):
Valor original: 0, Predicción KNN: 0
Valor original: 1, Predicción KNN: 1
Valor original: 2, Predicción KNN: 2
Valor original: 3, Predicción KNN: 3
Valor original: 4, Predicción KNN: 4
Valor original: 5, Predicción KNN: 9
Valor original: 6, Predicción KNN: 6
Valor original: 7, Predicción KNN: 7
Valor original: 8, Predicción KNN: 8
Valor original: 9, Predicción KNN: 9
```

```
#import os
#!pip3 install graphviz pydotplus

# Crear un clasificador de árbol de decisión
clf_tree = tree.DecisionTreeClassifier()
clf_tree.fit(X, y)  # Entrenar el clasificador de árbol de decisión
con los datos de entrenamiento

# Calcular y mostrar el rendimiento del modelo de árbol de decisión en
el conjunto de entrenamiento
score_tree = clf_tree.score(X, y)
print(f"Rendimiento del modelo de árbol de decisión en el conjunto de
entrenamiento: {score_tree:.4f}")

# Calcular predicciones utilizando el modelo de árbol de decisión para
comparar con los valores originales
y_pred_tree = clf_tree.predict(X)

# Mostrar algunos ejemplos de comparación entre valores originales y
predicciones por árbol de decisión
print("\nEjemplos de comparación (valor original vs. predicción por
Árbol de Decisión):")
for i in range(10):  # Mostrar los primeros 10 ejemplos
    print(f"Valor original: {y[i]}, Predicción Árbol de Decisión:
{y_pred_tree[i]}")
```

```
Rendimiento del modelo de árbol de decisión en el conjunto de
entrenamiento: 1.0000

Ejemplos de comparación (valor original vs. predicción por Árbol de
Decisión):
Valor original: 0, Predicción Árbol de Decisión: 0
Valor original: 1, Predicción Árbol de Decisión: 1
Valor original: 2, Predicción Árbol de Decisión: 2
Valor original: 3, Predicción Árbol de Decisión: 3
Valor original: 4, Predicción Árbol de Decisión: 4
Valor original: 5, Predicción Árbol de Decisión: 5
Valor original: 6, Predicción Árbol de Decisión: 6
Valor original: 7, Predicción Árbol de Decisión: 7
Valor original: 8, Predicción Árbol de Decisión: 8
Valor original: 9, Predicción Árbol de Decisión: 9
```

4.3 Más sobre los datos: The feature space

Cuando trabajamos con datos, especialmente en el aprendizaje automático, a menudo comenzamos con valores brutos. En nuestro caso, estamos tratando con imágenes de dígitos, y los valores brutos son los niveles de gris de cada píxel en estas imágenes. Para dar sentido a estos datos y potencialmente mejorar cómo distinguimos entre diferentes dígitos, podemos extraer o derivar nuevas características de los datos brutos. Estas características son atributos

que calculamos a partir de los datos, basados en nuestra comprensión de lo que podría ser importante para identificar las diferencias entre clases (en este caso, diferentes tipos de dígitos).

Características Derivadas: Simetría y Área

Para el conjunto de datos de dígitos, consideramos tres características específicas que podrían ayudarnos a diferenciar entre dígitos:

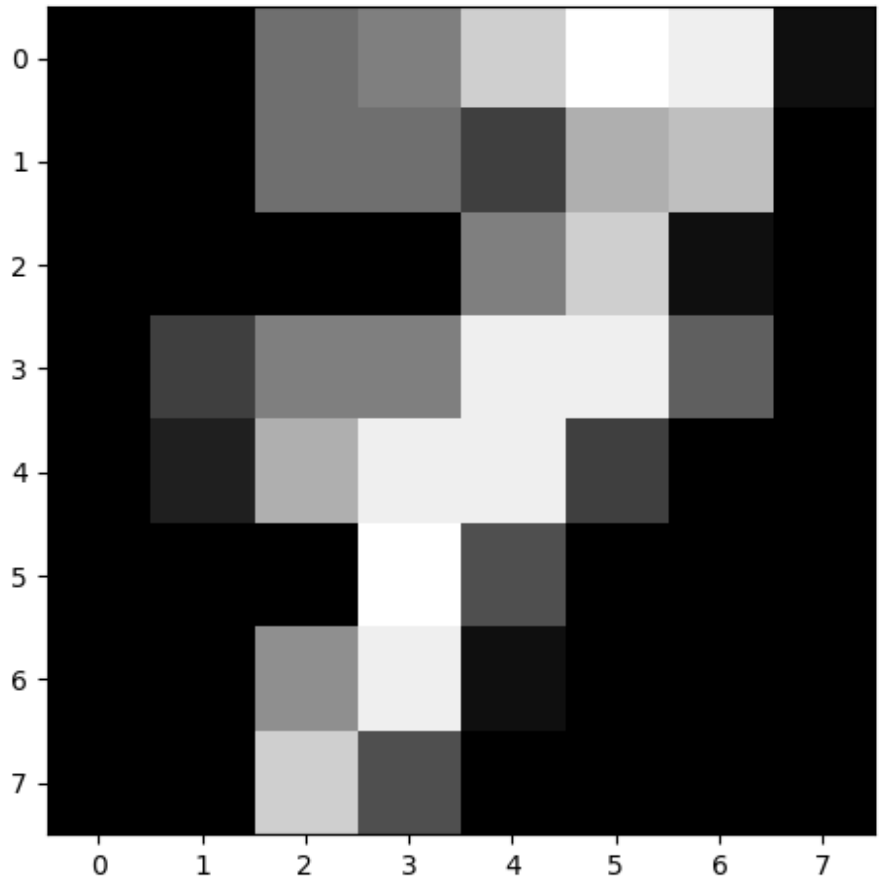
- **Simetría Horizontal**: ¿Qué tan similar es la mitad izquierda de la imagen a la mitad derecha?
- **Simetría Vertical**: ¿Qué tan similar es la mitad superior de la imagen a la mitad inferior?
- **Área**: La suma de todos los valores de píxeles, que puede considerarse como el "peso" del dígito.

El Código Explicado

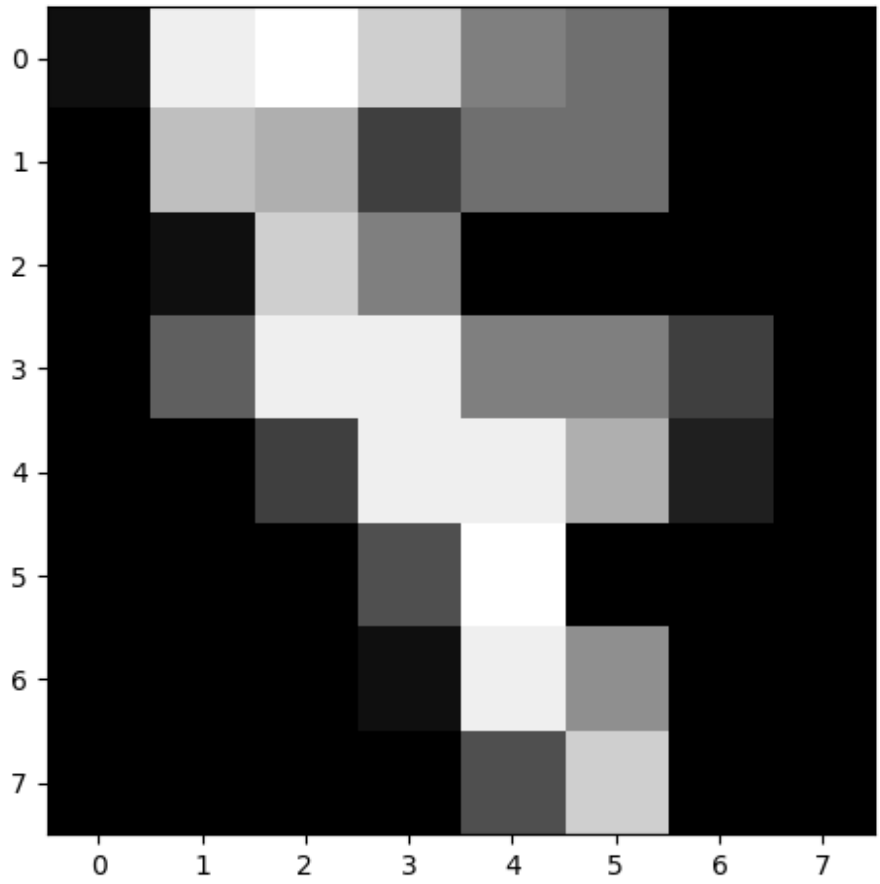
Visualizando la Simetría

Primero, vamos a observar un dígito y explorar su simetría:

```
from skimage import io
import numpy as np

# Seleccionar el octavo dígito en el conjunto de datos y remodelarlo a
su forma original 8x8
tmp = X[7].reshape((8, 8))

# Visualizar el dígito original
print("Dígito Original:")
# Normalizar la imagen y convertir a uint8 para evitar la advertencia
normalized_image = ((tmp - tmp.min()) / (tmp.max() - tmp.min()) *
255).astype(np.uint8)
io.imshow(normalized_image)  # Ahora la imagen está en el rango
correcto
io.show()

# Visualizar el espejo horizontal del dígito original
print("Espejo Horizontal del Dígito:")
# Normalizar el espejo horizontal y convertir a uint8
normalized_image_mirror = (((tmp[:, ::-1]) - tmp.min()) / (tmp.max() -
tmp.min()) * 255).astype(np.uint8)
io.imshow(normalized_image_mirror)  # Imagen normalizada y convertida
a uint8
io.show()

Dígito Original:
```




```
Espejo Horizontal del Dígito:
```



Esta parte del código simplemente nos muestra la imagen original de un dígito y su versión volteada horizontalmente para darnos una intuición visual sobre la simetría.

Cálculo de Nuevas Características:

A continuación, calculamos las tres características (simetría horizontal, simetría vertical y área) para cada dígito en el conjunto de datos:

```
import numpy as np

# Inicializar un arreglo para contener las nuevas características para
cada imagen.
Xnew = np.zeros((y.shape[0], 3))

for i in range(y.shape[0]):
    area = sum(X[i])  # Calcular la característica de área.
    tmp = X[i].reshape((8, 8))

    # Calcular simetría horizontal multiplicando la imagen por su
espejo horizontal.
    symH = tmp * tmp[:, ::-1]

    # Calcular simetría vertical multiplicando la imagen por su espejo
```

```
vertical.
    symV = tmp * tmp[::-1, :]

    # Almacenar las características calculadas en Xnew.
    Xnew[i, :] = [sum(symH.flatten()), area, sum(symV.flatten())]

# Imprimir las nuevas características y su forma.
print("Nuevas características calculadas:")
print(Xnew)
print("Forma de Xnew:", Xnew.shape)
```

```
Nuevas características calculadas:
[[2808.  294. 2886.]
 [3670.  313. 3888.]
 [2810.  344. 3512.]
 ...
 [4776.  374. 4734.]
 [3656.  344. 3248.]
 [4422.  392. 4220.]]
Forma de Xnew: (1797, 3)
```

```
# Guardar este conjunto de datos para uso posterior
import pickle

# Abrir un archivo en modo binario para escritura
ofname = open('my_digits_data.pkl', 'wb')

# Utilizar pickle para serializar y guardar los datos Xnew y y en el
archivo
s = pickle.dump([Xnew, y], ofname)

# Cerrar el archivo después de guardar
ofname.close()

# Imprimir un mensaje indicando que el guardado ha terminado
print('DONE')
```

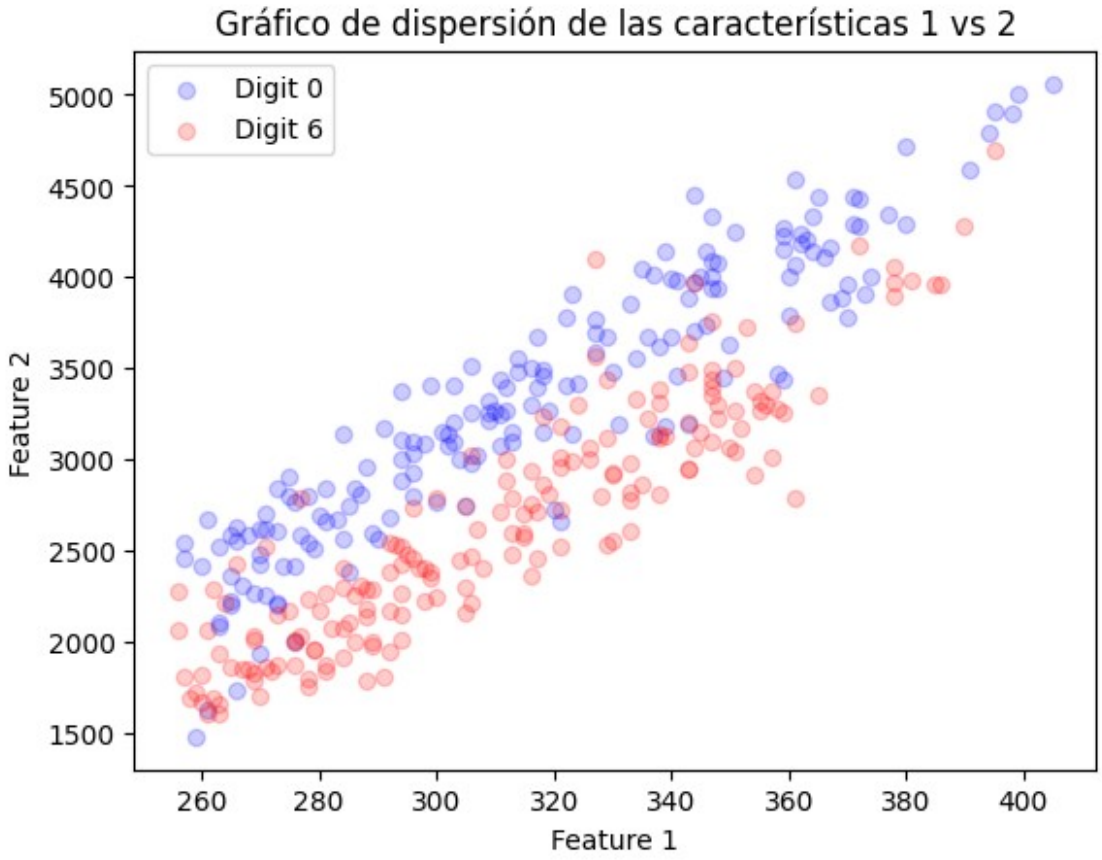
```
DONE
```

Este script transforma cada imagen para calcular y almacenar nuestras tres características elegidas: las sumas de los productos de la imagen y sus espejos horizontales y verticales (para la simetría) y la suma de los valores de los píxeles (para el área).

Visualización del Espacio de Características Finalmente, visualizamos cómo dos dígitos, 0 y 6, difieren en este nuevo espacio de características:

```
import matplotlib.pyplot as plt

# Encontrar los índices de los dígitos 0 y 6 en las etiquetas y
idxA = y == 0
```

```
idxB = y == 6

# Elegir qué características trazar (seleccionar las columnas
correspondientes en Xnew)
feature1 = 1
feature2 = 2

# Graficar las características para los dígitos 0 y 6
plt.figure()
plt.scatter(Xnew[idxA, feature1], Xnew[idxA, feature2], c='blue',
alpha=0.2, label='Digit 0')
plt.scatter(Xnew[idxB, feature1], Xnew[idxB, feature2], c='red',
alpha=0.2, label='Digit 6')
plt.xlabel(f'Feature {feature1}')
plt.ylabel(f'Feature {feature2}')
plt.legend()
plt.title(f'Gráfico de dispersión de las características {feature1} vs
{feature2}')
plt.show()
```



Esta sección crea un gráfico de dispersión para comparar visualmente las características seleccionadas para los dígitos 0 y 6. El gráfico nos ayuda a ver si y cómo estos dígitos pueden distinguirse basándose en las nuevas características que hemos creado.

Conclusión Al derivar nuevas características de los datos brutos, a menudo podemos descubrir patrones que nos ayudan a diferenciar entre clases de manera más efectiva.

El proceso de usar información del dominio del conocimiento para crear características discriminantes se llama extracción de características.

Datos Brutos vs. Extracción de Características

Cuando trabajamos con modelos de aprendizaje automático, a menudo encontramos dos enfoques principales para preparar nuestros datos: usar datos brutos directamente o extraer características de estos datos. Ambos enfoques tienen sus ventajas y desventajas, que son cruciales para entender para un desarrollo de modelo efectivo.

Datos Brutos

Ventajas:

- **Accesibilidad:** Los datos brutos se pueden usar directamente sin la necesidad de procesamiento o transformación adicionales. Este enfoque no requiere conocimiento específico del dominio, lo que lo hace sencillo y accesible para los practicantes en todos los niveles.

Desventajas:

- **Redundancia y Dimensionalidad:** Los datos brutos a menudo son altamente redundantes, conteniendo información que puede no ser necesaria para hacer predicciones o clasificaciones precisas. Esta redundancia típicamente resulta en espacios dimensionales muy grandes, lo que puede complicar el entrenamiento del modelo y llevar a tiempos de computación más largos.
- **Discriminabilidad Desconocida:** Con los datos brutos, no siempre está claro qué características son importantes para distinguir entre clases. Esta falta de claridad puede obstaculizar la capacidad de aprendizaje del modelo, ya que puede centrarse en características irrelevantes.

Extracción de Características

Ventajas:

- **Información Discriminante:** La extracción de características tiene como objetivo identificar y capturar la información más relevante en los datos para la tarea en cuestión. Al centrarse en la información discriminante, los modelos pueden lograr un mejor rendimiento con menos datos.
- **Dimensionalidad y Complejidad Reducidas:** A través de la extracción de características, los datos se transforman en un espacio de menor dimensión que refleja los aspectos más importantes de los datos originales. Esta reducción en la dimensionalidad y la complejidad puede llevar a tiempos de entrenamiento más rápidos y modelos más eficientes.

Desventajas:

- **Conocimiento del Dominio Requerido:** A diferencia del uso de datos brutos, la extracción de características a menudo requiere conocimiento específico del dominio

para identificar qué características probablemente sean informativas. Este requisito puede hacer que la extracción de características sea menos accesible para aquellos sin experiencia en el dominio.

Medición de rendimiento

Existen diferentes criterios para medir el rendimiento de un clasificador y la métrica más adecuada suele depender del problema. Cuando no se dispone de información previa sobre el problema, generalmente usamos la precisión de clasificación. Cuando estamos frente a un problema de clasificación múltiple (hay muchas clases para elegir) podemos usar la matriz de confusión. Los elementos de la matriz de confusión M se definen de la siguiente manera,

$$M(i,j) = \#\text{de muestras de la clase j predichas como clase i}$$

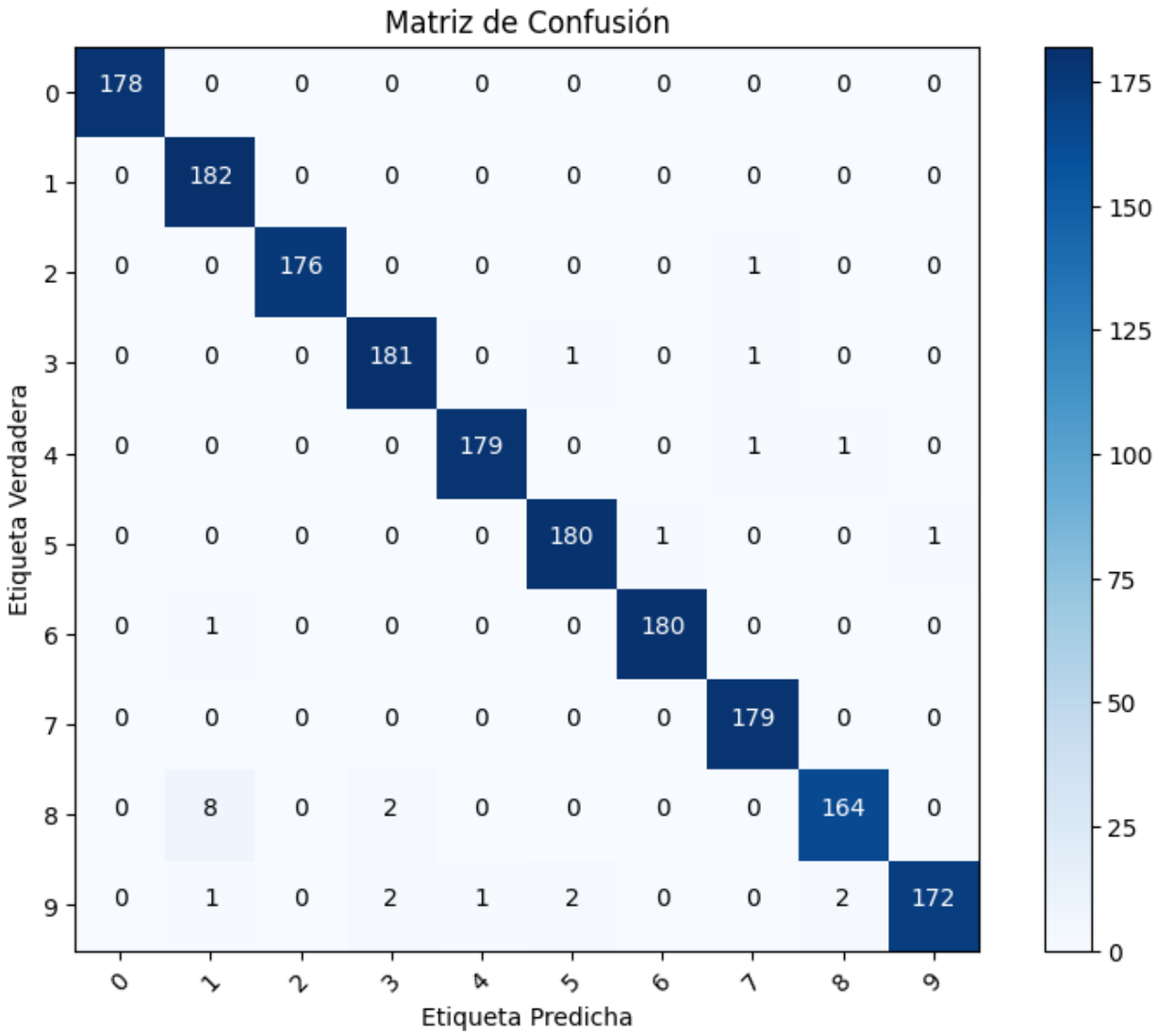
matriz de confusión

Verifiquemos estos valores:

```
import matplotlib.pyplot as plt
from sklearn import metrics
import numpy as np

def plot_confusion_matrix(y, y_pred):
    # Generar la matriz de confusión
    cm = metrics.confusion_matrix(y, y_pred)

    # Configurar la figura y el tamaño
    plt.figure(figsize=(8, 6))

    # Mostrar la matriz de confusión como una imagen de colores
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Matriz de Confusión')
    plt.colorbar()  # Mostrar la barra de colores que representa los
valores

    # Configurar etiquetas de los ejes x e y
    classes = np.unique(y)
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    # Agregar números como anotaciones de texto en cada celda
    threshold = cm.max() / 2.  # Umbral para decidir el color del
texto
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], 'd'),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > threshold else "black")

    # Ajustar el diseño para mejorar la legibilidad
```

```
    plt.tight_layout()

    # Etiquetas de los ejes y título
    plt.ylabel('Etiqueta Verdadera')
    plt.xlabel('Etiqueta Predicha')

# Ejemplo de uso (suponiendo que y e y_pred están definidos en otra
parte del código)
# y e y_pred son las etiquetas verdaderas y predichas, respectivamente
# Métrica de precisión de clasificación
print("Precisión de clasificación:", metrics.accuracy_score(y,
y_pred))

# Llamar a la función para graficar la matriz de confusión
plot_confusion_matrix(y, y_pred)

# Mostrar el gráfico
plt.show()
```

```
Precisión de clasificación: 0.9855314412910406
```



PREGUNTA: ¿Cuales son las clases con más confusión?

4.4 Train y Test

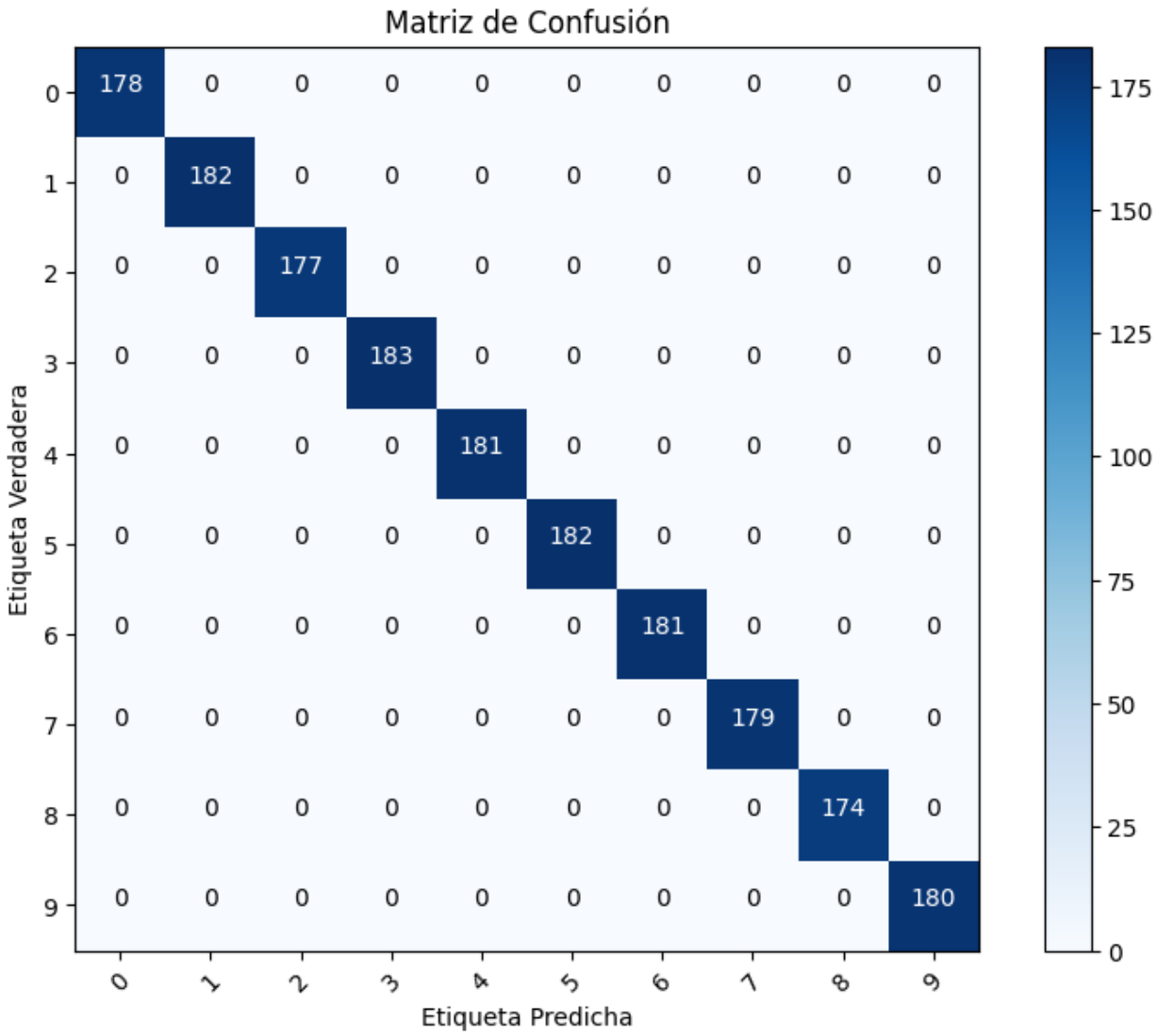
Más intuición detrás del proceso de aprendizaje

Comprender el proceso de aprendizaje en el aprendizaje automático es crucial para aplicar efectivamente los modelos para resolver problemas. El fragmento de código a continuación demuestra cómo entrenar un clasificador de K Vecinos Más Cercanos (KNN) utilizando un conjunto de datos con características, hacer predicciones y evaluar el rendimiento del modelo. Este proceso es fundamental para el aprendizaje automático y proporciona información sobre cómo los modelos aprenden de los datos para hacer predicciones.

```
from sklearn import neighbors
from sklearn import metrics
```

```
# Crear una instancia del clasificador KNeighborsClassifier con 1 vecino.
knn = neighbors.KNeighborsClassifier(n_neighbors=1)

# Entrenar el clasificador en el conjunto de datos.
knn.fit(Xnew, y)

# Realizar predicciones en el conjunto de datos utilizando el modelo entrenado.
yhat = knn.predict(Xnew)

# Imprimir la precisión de clasificación del modelo.
print("Precisión de clasificación:", metrics.accuracy_score(yhat, y))

# Graficar la matriz de confusión para las etiquetas verdaderas y las etiquetas predichas.
plot_confusion_matrix(y, yhat)

# Nota: El modelo con características (Xnew) se utiliza aquí en lugar del conjunto de datos original,
# lo que indica que se han aplicado pasos de preprocesamiento o técnicas de extracción de características al conjunto de datos original.
```

```
Precisión de clasificación: 1.0
```



PREGUNTA:

¿Cuál es el accuracy del clasificador en set de train? ¿Necesitaos mejorar en nuevos datos?

Entendiendo el Rendimiento del Clasificador con Divisiones de Train/Test

Hasta ahora, hemos evaluado el rendimiento de nuestro clasificador utilizando el mismo conjunto de datos con el que fue entrenado. Aunque esto puede proporcionar algunas ideas sobre cuán bien el modelo ha aprendido los datos de entrenamiento, no representa con precisión cómo el modelo se desempeñará en nuevos datos no vistos. En aplicaciones del mundo real, esperamos que nuestro modelo haga predicciones sobre datos que nunca ha encontrado durante la fase de entrenamiento.

La Importancia de las Divisiones de Entrenamiento/Prueba

Para simular un escenario más realista y entender mejor la capacidad de generalización del modelo, dividimos nuestro conjunto de datos en dos conjuntos separados:

- **Conjunto de entrenamiento:** Esta porción de los datos se utiliza para entrenar el modelo. Aprende de estos datos, ajustando sus parámetros para adaptar las características de entrada dadas a los valores objetivo correspondientes.
- **Conjunto de prueba:** Este conjunto se mantiene separado del proceso de entrenamiento. Después de que el modelo ha sido entrenado, usamos estos datos para evaluar su rendimiento. El conjunto de prueba actúa como un sustituto de nuevos datos no vistos.

¿Por Qué Dividir?

Dividir los datos nos ayuda a:

1. **Evaluar la generalización:** Al evaluar el modelo en datos que no ha visto antes, podemos medir qué tan bien se generaliza a nuevos ejemplos. Esto es un mejor indicador de su rendimiento en el mundo real.
2. **Detectar el sobreajuste:** Si un modelo se desempeña excepcionalmente bien en los datos de entrenamiento pero pobremente en los datos de prueba, es probable que esté sobreajustado. El sobreajuste ocurre cuando un modelo aprende el ruido en los datos de entrenamiento en lugar del patrón subyacente, lo que hace que se desempeñe mal en cualquier dato fuera del conjunto de entrenamiento.
3. **Ajustar parámetros:** La división de entrenamiento/prueba nos permite ajustar los parámetros del modelo y seleccionar la mejor configuración del modelo. Comparando el rendimiento del modelo en los conjuntos de entrenamiento y prueba, podemos tomar decisiones informadas sobre la configuración de parámetros y las elecciones de modelos.

Implementando Divisiones de Entrenamiento/Prueba

En la práctica, el conjunto de datos se divide aleatoriamente en conjuntos de entrenamiento y prueba, a menudo con proporciones como 70%/30%, 80%/20%, o similares, dependiendo del tamaño del conjunto de datos y del problema específico. Herramientas como `train_test_split` de `sklearn.model_selection` pueden automatizar este proceso, asegurando que los datos se dividan aleatoria y apropiadamente.

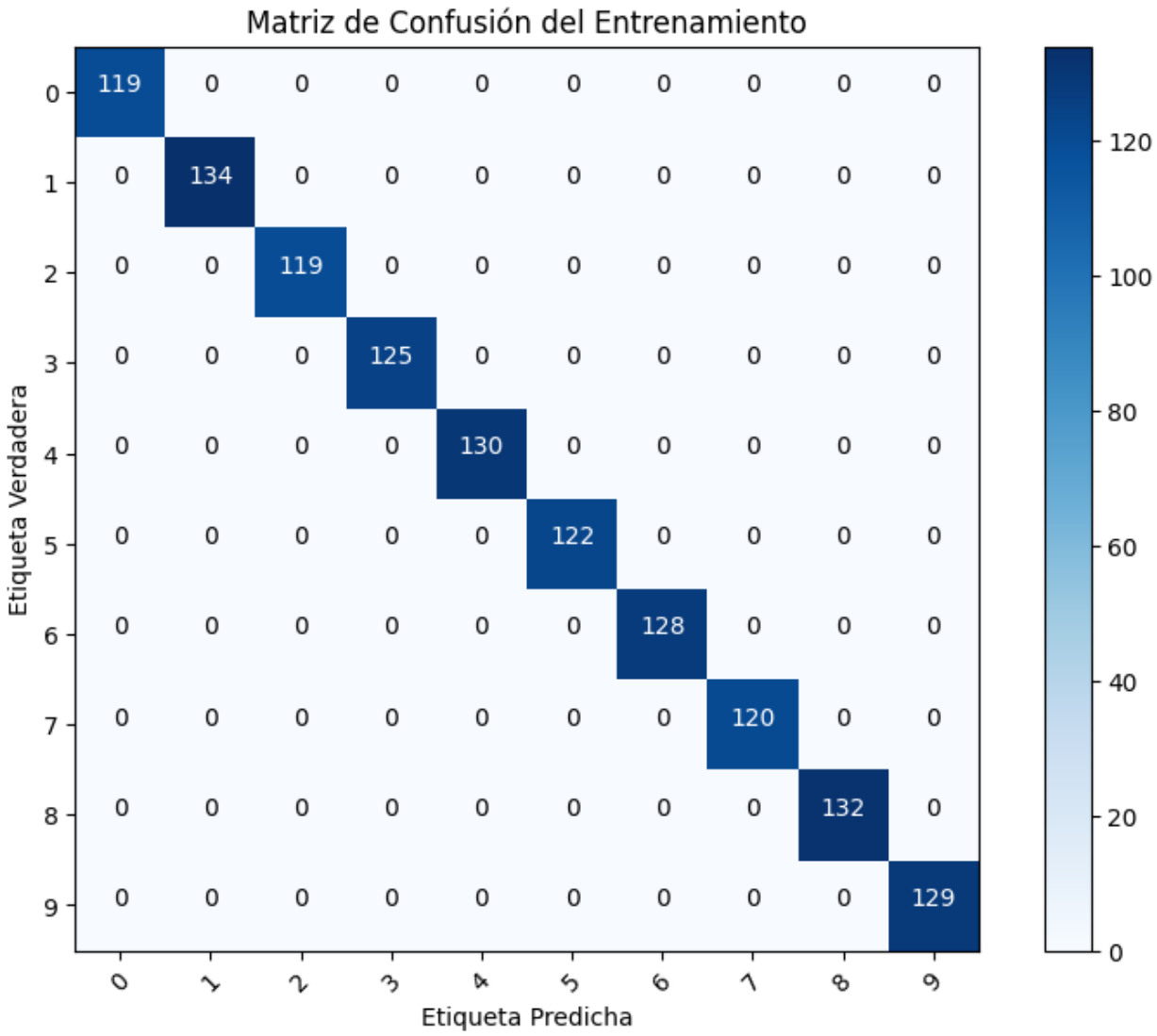
Al adoptar este enfoque, aseguramos que nuestra evaluación del modelo sea más robusta e indicativa de cómo el modelo se desempeñará en aplicaciones prácticas, aumentando así nuestra confianza en sus predicciones sobre datos no vistos.

```
# Resetear el espacio de trabajo para asegurar un entorno limpio
%reset -f

# Cargar el conjunto de datos de dígitos desde un archivo pickle
import pickle

# Abrir el archivo que contiene el conjunto de datos
with open('my_digits_data.pkl', 'rb') as ofname:
```

```
    # Cargar el conjunto de datos desde el archivo
    data = pickle.load(ofname)
    # Asignar características a X y etiquetas de destino a y
    X, y = data[0], data[1]

# Preparar el conjunto de datos para una simulación realista:
aleatorizarlo y dividirlo en subconjuntos de entrenamiento y prueba
import numpy as np

# Permutar aleatoriamente una secuencia de índices basada en el tamaño
de y
perm = np.random.permutation(y.size)

# Definir la proporción del conjunto de datos para asignar al
entrenamiento
PRC = 0.7
# Calcular el punto de división para dividir el conjunto de datos
split_point = int(np.ceil(y.shape[0] * PRC))

# Dividir el conjunto de datos en conjuntos de entrenamiento y prueba
basados en el punto de división calculado
X_train = X[perm[:split_point]]
y_train = y[perm[:split_point]]

X_test = X[perm[split_point:]]
y_test = y[perm[split_point:]]

# Imprimir las formas de los conjuntos de entrenamiento y prueba para
verificar
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)

# Entrenar un clasificador de Vecinos Más Cercanos (K-Nearest
Neighbors) en los datos de entrenamiento
from sklearn import neighbors

# Inicializar el clasificador con 1 vecino para simplicidad
knn = neighbors.KNeighborsClassifier(n_neighbors=1)
# Ajustar el clasificador a los datos de entrenamiento
knn.fit(X_train, y_train)

# Predecir las etiquetas para el conjunto de entrenamiento y evaluar
el rendimiento
yhat = knn.predict(X_train)

# Importar las bibliotecas necesarias para la evaluación del
rendimiento
from sklearn import metrics
import matplotlib.pyplot as plt

# Imprimir estadísticas de entrenamiento
```

```
print("\nESTADÍSTICAS DE ENTRENAMIENTO:")
print("Precisión de clasificación:", metrics.accuracy_score(yhat,
y_train))

# Visualizar la matriz de confusión
def plot_confusion_matrix(cm, classes, title='Matriz de Confusión',
cmap=plt.cm.Blues):
    # Configurar la figura y el tamaño
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('Etiqueta Verdadera')
    plt.xlabel('Etiqueta Predicha')
    plt.show()

# Generar la matriz de confusión a partir de las predicciones de los
datos de entrenamiento
cm = metrics.confusion_matrix(y_train, yhat)
# Llamar a la función para graficar la matriz de confusión mejorada
plot_confusion_matrix(cm, classes=np.unique(y_train), title="Matriz de
Confusión del Entrenamiento")
```

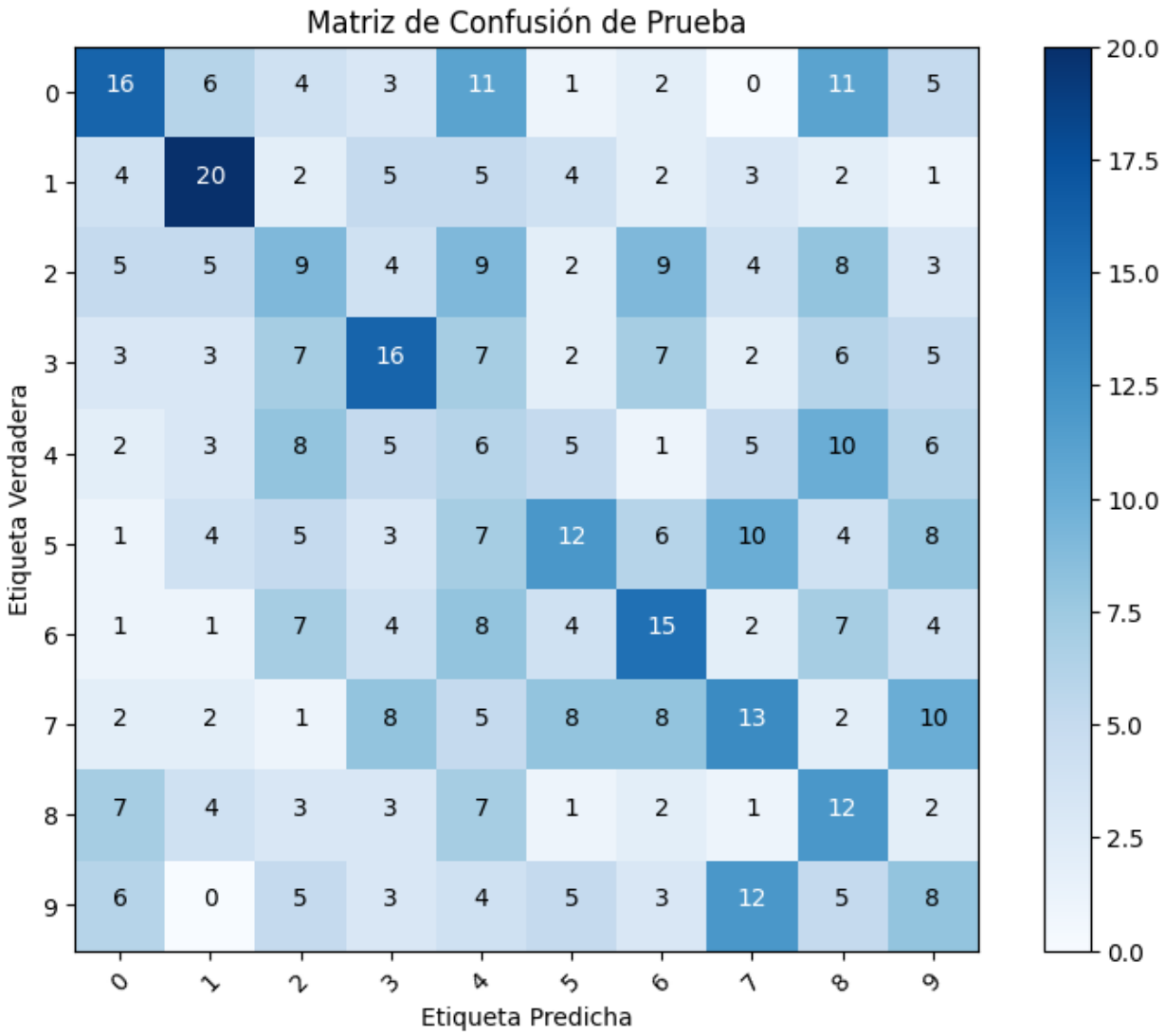
```
(1258, 3) (539, 3) (1258,) (539,)

ESTADÍSTICAS DE ENTRENAMIENTO:
Precisión de clasificación: 1.0
```




```
# Predecir las etiquetas para el conjunto de prueba para evaluar el
rendimiento
yhat = knn.predict(X_test)

# Importar las bibliotecas necesarias para la evaluación del
rendimiento
from sklearn import metrics
import matplotlib.pyplot as plt
import numpy as np

# Imprimir estadísticas de prueba
print("ESTADÍSTICAS DE PRUEBA:")
print("Precisión de clasificación:", metrics.accuracy_score(yhat,
y_test))

# Función para graficar la matriz de confusión con números para mayor
```

```
claridad
def plot_confusion_matrix_with_numbers(cm, title='Matriz de
Confusión', cmap=plt.cm.Blues):
    """
    Esta función grafica una matriz de confusión con los conteos
reales mostrados en la matriz para una mejor claridad.
    """
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(np.unique(y_test)))
    plt.xticks(tick_marks, np.unique(y_test), rotation=45)
    plt.yticks(tick_marks, np.unique(y_test))

    # Iterar sobre las dimensiones de los datos y crear anotaciones de
texto.
    fmt = 'd'  # Formato como entero decimal
    thresh = cm.max() / 2.  # Umbral para el color del texto basado en
el fondo
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('Etiqueta Verdadera')
    plt.xlabel('Etiqueta Predicha')

# Generar la matriz de confusión a partir de las predicciones de los
datos de prueba
cm_test = metrics.confusion_matrix(y_test, yhat)

# Graficar la matriz de confusión mejorada con números para los datos
de prueba
plot_confusion_matrix_with_numbers(cm_test, "Matriz de Confusión de
Prueba")

plt.show()
```

```
ESTADÍSTICAS DE PRUEBA:
Precisión de clasificación: 0.23562152133580705
```



Entendiendo la Variabilidad en el Rendimiento del Modelo

Al evaluar el rendimiento de un modelo, especialmente en el aprendizaje automático, es importante reconocer que los resultados pueden variar cada vez que se ejecuta el proceso. Esta variación se debe a varios factores, incluyendo la aleatoriedad en la división del conjunto de datos en conjuntos de entrenamiento y prueba, y la naturaleza estocástica inherente de muchos algoritmos de aprendizaje.

¿Por Qué Varía el Rendimiento?

- **División de Datos:** Cada vez que dividimos aleatoriamente el conjunto de datos en conjuntos de entrenamiento y prueba, el modelo está expuesto a datos ligeramente diferentes durante el entrenamiento. Esto puede llevar a variaciones en cómo el modelo aprende y generaliza.
- **Condiciones Iniciales del Modelo:** Para algoritmos que involucran inicialización aleatoria (por ejemplo, pesos iniciales en redes neuronales), diferentes condiciones iniciales pueden llevar a diferentes caminos y resultados de aprendizaje.

- **Varianza de Muestreo:** El subconjunto de datos elegido para el entrenamiento y la prueba podría no representar completamente la distribución general del conjunto de datos, lo que lleva a variaciones en las métricas de rendimiento.

Simulando Condiciones del Mundo Real

Para aproximar más precisamente el error de prueba y tener en cuenta estas variaciones, podemos emplear una técnica conocida como *validación cruzada*. Más sencillamente, sin embargo, podemos repetir el proceso de dividir, entrenar y probar el modelo varias veces, cada vez con una división aleatoria diferente. Al promediar las métricas de rendimiento a través de estas iteraciones, podemos obtener una estimación más estable y confiable del rendimiento del modelo.

Implementando Múltiples Iteraciones

Mejoremos nuestra simulación ejecutando el proceso de división de entrenamiento-prueba, entrenamiento y evaluación varias veces. Después de cada iteración, registraremos las métricas de rendimiento del modelo. Una vez que todas las iteraciones estén completas, calcularemos el rendimiento promedio. Este enfoque nos da una comprensión más matizada de cómo es probable que nuestro modelo se desempeñe en datos no vistos, haciendo nuestra evaluación más robusta y realista.

Este proceso no solo ayuda a mitigar los efectos de la aleatoriedad y la varianza, sino que también proporciona información sobre la consistencia y fiabilidad del modelo bajo diferentes condiciones. Al entender y aplicar estos conceptos, podemos tomar decisiones más informadas sobre la selección, ajuste y despliegue del modelo.

```
from sklearn import model_selection, neighbors, metrics
import numpy as np

# Definir el tamaño del conjunto de prueba (proporción)
PRC = 0.3

# Crear un arreglo para almacenar las precisiones de clasificación
obtenidas en cada repetición
acc = np.zeros((10,))

# Realizar 10 repeticiones del proceso de división y evaluación
for i in range(10):
    # Dividir el conjunto de datos en conjuntos de entrenamiento y
prueba de manera aleatoria
    X_train, X_test, y_train, y_test =
model_selection.train_test_split(X, y, test_size=PRC, random_state=42)

    # Inicializar el clasificador KNN con 1 vecino
    knn = neighbors.KNeighborsClassifier(n_neighbors=1)

    # Entrenar el clasificador en el conjunto de entrenamiento
    knn.fit(X_train, y_train)
```

```
    # Predecir las etiquetas para el conjunto de prueba
    yhat = knn.predict(X_test)

    # Calcular la precisión de clasificación y almacenarla
    acc[i] = metrics.accuracy_score(yhat, y_test)

# Reorganizar el arreglo de precisión para mostrar los resultados de
manera adecuada
acc.shape = (1, 10)

# Imprimir el error esperado promedio (1 - precisión promedio)
print("Error esperado promedio: " + str(1 - np.mean(acc[0])))
```

```
Error esperado promedio: 0.7851851851851852
```

```
acc
```

```
array([[0.21481481, 0.21481481, 0.21481481, 0.21481481, 0.21481481,
        0.21481481, 0.21481481, 0.21481481, 0.21481481, 0.21481481]])
```

Aclarando las Métricas de Error en Machine Learning

En el ámbito del aprendizaje automático y la teoría del aprendizaje estadístico, es esencial cuantificar el rendimiento de nuestros modelos. Para hacer esto, introducimos una nomenclatura específica para las métricas de error que calculamos durante el entrenamiento y evaluación del modelo.

Error Dentro de la Muestra (E_{in})

- **Definición:** El error dentro de la muestra, también conocido como error de entrenamiento, mide el error en todas las muestras de datos observadas dentro del conjunto de entrenamiento. Refleja qué tan bien el modelo se ajusta a los datos en los que fue entrenado.

- **Fórmula:** La representación matemática se da por:

$$E_{\text{in}} = \frac{1}{N} \sum_{i=1}^{N} e\left(x_i, y_i\right)$$

  donde N es el número de muestras en el conjunto de entrenamiento, y $e\left(x_i, y_i\right)$ denota el error de la predicción en la i-ésima muestra.

Error Fuera de la Muestra (E_{out})

- **Definición:** El error fuera de la muestra, o error de generalización, mide el error esperado en datos no vistos. Esta métrica es crucial ya que indica la capacidad del modelo para generalizar más allá de los datos de entrenamiento.

- **Aproximación:** Aproximamos E_{out} reservando una parte del conjunto de entrenamiento para pruebas, de modo que el modelo no esté expuesto a estos datos durante el entrenamiento.

$$E_{\text{out}} = E_{x,y}\left(e(x,y)\right)$$

Aquí, la expectativa $E_{x,y}$ refleja el error promedio sobre todas las muestras posibles no vistas.

Error Instantáneo ($e(x_i, y_i)$)

- **Definición:** Para medir el error en predicciones individuales, definimos el error instantáneo. Esta métrica evalúa el error en un solo punto de datos.
- **Ejemplo:** En el contexto de la clasificación, podríamos usar la función indicadora para evaluar si una muestra está clasificada correctamente:

$$e(x_i, y_i) = I\left[h(x_i) = y_i\right] = \begin{cases} 1 & \text{si } h(x_i) = y_i \\ 0 & \text{de lo contrario} \end{cases}$$

donde $h(x_i)$ es la etiqueta predicha para la muestra x_i, y y_i es la etiqueta verdadera.

Entendiendo la Relación Entre E_{in} y E_{out}

Es un principio fundamental que el error fuera de la muestra se espera que sea mayor o igual que el error dentro de la muestra:

$$E_{\text{out}} \geq E_{\text{in}}$$

Esta desigualdad subraya el desafío del sobreajuste, donde un modelo podría funcionar excepcionalmente bien en los datos de entrenamiento (E_{in} es bajo) pero mal en nuevos datos no vistos (E_{out} es alto). El objetivo principal en el entrenamiento del modelo es minimizar E_{out}, asegurando que nuestro modelo generalice bien a nuevos datos.

4.5 Selección de Modelo (I)

En el contexto del aprendizaje automático, **la selección de modelo** es un proceso crítico donde determinamos el clasificador o modelo más adecuado para una aplicación específica. Esta decisión se basa en el rendimiento del modelo, típicamente evaluado usando el error esperado en un conjunto de prueba.

Por Qué la Selección de Modelo es Importante

El objetivo principal de la selección de modelo es identificar el modelo que mejor generaliza a datos no vistos. Dado que diferentes modelos tienen diversas fortalezas y debilidades dependiendo de la naturaleza de los datos y la tarea en cuestión, seleccionar el "mejor" modelo es crucial para lograr un alto rendimiento.

Escenario Simplista de Selección de Modelo

Considera un escenario donde tenemos una colección de diferentes clasificadores a nuestra disposición. El objetivo es sencillo: seleccionar el clasificador que tenga el mejor rendimiento según una métrica preestablecida, usualmente el que tenga la menor tasa de error en el conjunto de prueba.

Pasos para la Selección de Modelo

1. **Entrenar Múltiples Clasificadores:** Comienza entrenando cada modelo candidato en el mismo conjunto de datos de entrenamiento. Esto asegura que cada modelo aprenda de la misma información.
2. **Evaluación en el Conjunto de Pruebas:** A continuación, evalúa el rendimiento de cada modelo en un conjunto de pruebas separado. Este conjunto de pruebas no debe haber sido visto por los modelos durante el entrenamiento, asegurando que nuestra evaluación refleje la capacidad de cada modelo para generalizar.
3. **Comparar Tasas de Error:** Calcula la tasa de error (u otra métrica de rendimiento relevante) para cada modelo en el conjunto de pruebas. La tasa de error nos da una medida cuantitativa de cuán a menudo el modelo realiza predicciones incorrectas.
4. **Seleccionar el Mejor Modelo:** Finalmente, selecciona el modelo con la menor tasa de error en el conjunto de pruebas. Este modelo es considerado el "mejor" entre los candidatos para nuestra aplicación específica.

Consideraciones

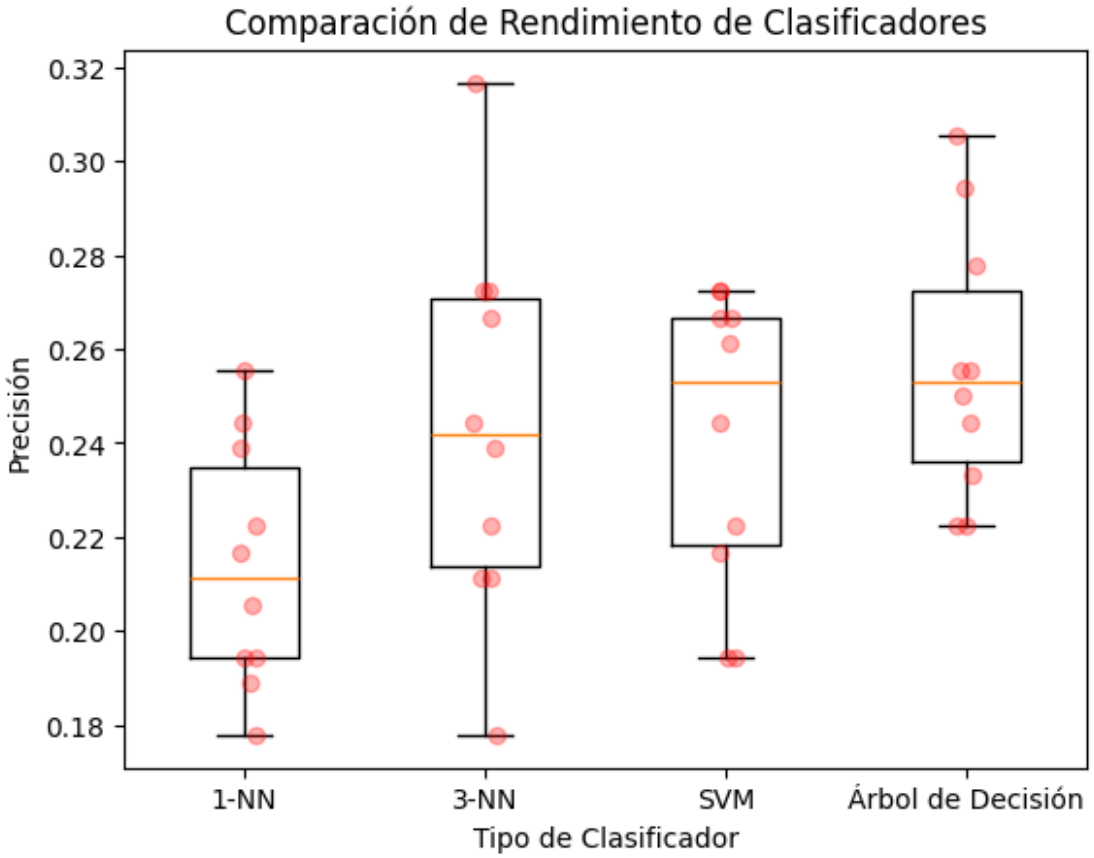
- **Tasa de Error como Métrica:** Aunque usar la tasa de error es común, otras métricas como precisión, recall, puntuación F1 o AUC podrían ser más apropiadas dependiendo de los requisitos específicos de la aplicación.
- **Conjuntos de Validación:** Además de un conjunto de pruebas, un conjunto de validación también puede ser utilizado durante la selección de modelo. Esto permite el ajuste de hiperparámetros del modelo sin usar el conjunto de pruebas, que debería reservarse para la evaluación final.
- **Complejidad y Rendimiento:** Es esencial equilibrar la complejidad del modelo con el rendimiento. Un modelo más complejo podría producir una tasa de error ligeramente menor pero a costa de la interpretabilidad, eficiencia computacional o el riesgo de sobreajuste.

Al seguir cuidadosamente el proceso de selección de modelo, podemos elegir con confianza un clasificador que ofrezca el mejor equilibrio entre precisión, complejidad y generalización para nuestra aplicación.

```
# Importar las bibliotecas necesarias de sklearn para selección de
modelos, clasificadores y métricas
from sklearn import model_selection
from sklearn import neighbors
from sklearn import tree
from sklearn import svm
from sklearn import metrics
import matplotlib.pyplot as plt
import numpy as np  # Asegurarse de que numpy esté importado para
operaciones con matrices

# Definir la proporción del conjunto de datos a utilizar para pruebas
PRC = 0.1

# Inicializar una matriz para almacenar los resultados de precisión de
cada clasificador a través de las iteraciones
```

```
acc_r = np.zeros((10, 4))  # 10 iteraciones, 4 clasificadores

# Repetir el experimento 10 veces para obtener una distribución de
métricas de rendimiento
for i in range(10):
    # Dividir el conjunto de datos en conjuntos de entrenamiento y
prueba utilizando validación cruzada estratificada
    X_train, X_test, y_train, y_test =
model_selection.train_test_split(X, y, test_size=PRC, stratify=y)

    # Inicializar clasificadores con configuraciones específicas
    nn1 = neighbors.KNeighborsClassifier(n_neighbors=1)  #
Clasificador 1-Vecino Más Cercano
    nn3 = neighbors.KNeighborsClassifier(n_neighbors=3)  #
Clasificador 3-Vecinos Más Cercanos
    svc = svm.SVC()  # Clasificador SVM (Support Vector Machine)
    dt = tree.DecisionTreeClassifier()  # Clasificador Árbol de
Decisión

    # Entrenar cada clasificador con el conjunto de entrenamiento
    nn1.fit(X_train, y_train)
    nn3.fit(X_train, y_train)
    svc.fit(X_train, y_train)
    dt.fit(X_train, y_train)

    # Predecir las etiquetas para el conjunto de prueba usando cada
clasificador entrenado
    yhat_nn1 = nn1.predict(X_test)
    yhat_nn3 = nn3.predict(X_test)
    yhat_svc = svc.predict(X_test)
    yhat_dt = dt.predict(X_test)

    # Calcular y almacenar la precisión para cada clasificador
    acc_r[i][0] = metrics.accuracy_score(yhat_nn1, y_test)
    acc_r[i][1] = metrics.accuracy_score(yhat_nn3, y_test)
    acc_r[i][2] = metrics.accuracy_score(yhat_svc, y_test)
    acc_r[i][3] = metrics.accuracy_score(yhat_dt, y_test)

# Visualizar los resultados de precisión para cada clasificador
utilizando un gráfico de caja
plt.boxplot(acc_r);

# Superponer puntos rojos para mostrar los resultados individuales de
precisión para comparación visual
for i in range(4):
    xderiv = (i+1)*np.ones(acc_r[:, i].shape) + (np.random.rand(10,)-
0.5) * 0.1
    plt.plot(xderiv, acc_r[:, i], 'ro', alpha=0.3)

# Personalizar el gráfico con etiquetas apropiadas
```

```
ax = plt.gca()
ax.set_xticklabels(['1-NN', '3-NN', 'SVM', 'Árbol de Decisión'])

plt.xlabel('Tipo de Clasificador')
plt.ylabel('Precisión')
plt.title('Comparación de Rendimiento de Clasificadores')

# Mostrar el gráfico
plt.show()
```



Entendiendo las Técnicas de Validación Cruzada

La validación cruzada es un método fundamental en el aprendizaje automático para evaluar cómo los resultados de un análisis estadístico se generalizarán a un conjunto de datos independiente. Es especialmente útil en escenarios donde el objetivo es predecir, y se desea estimar cuán precisamente un modelo predictivo funcionará en la práctica. El proceso que hemos discutido es una forma de validación cruzada, que abarca varios métodos diferentes, incluyendo la **validación cruzada leave-one-out** y la **validación cruzada K-fold**.

Validación Cruzada Leave-One-Out

- **Cómo Funciona:** En la validación cruzada leave-one-out (LOOCV), para un conjunto de datos con N muestras, el modelo se entrena usando $N - 1$ muestras y se prueba en la

muestra única restante. Este proceso se repite N veces, con cada una de las N muestras utilizadas exactamente una vez como conjunto de prueba.

- **Ventajas:** LOOCV utiliza casi todos los datos para el entrenamiento, lo que lo convierte en una excelente opción para conjuntos de datos pequeños.
- **Desventajas:** Puede ser computacionalmente costoso para conjuntos de datos más grandes, ya que el modelo necesita ser entrenado desde cero N veces.

Validación Cruzada K-Fold

- **Cómo Funciona:** En la validación cruzada K-fold, el conjunto de entrenamiento se divide aleatoriamente en K subconjuntos de igual tamaño. De los K subconjuntos, un único subconjunto se retiene como datos de validación para probar el modelo, y los $K-1$ subconjuntos restantes se utilizan como datos de entrenamiento. El proceso de validación cruzada se repite K veces (los pliegues), con cada uno de los K subconjuntos utilizados exactamente una vez como datos de validación.
- **Ventajas:** Este método es menos costoso computacionalmente que LOOCV, especialmente para conjuntos de datos grandes. También permite una mezcla más completa de los datos, ya que cada pliegue se utiliza tanto para entrenamiento como para validación.
- **Estimación de Confianza:** La validación cruzada K-fold no solo proporciona una estimación del rendimiento del modelo, sino que también permite calcular un intervalo de confianza alrededor del rendimiento estimado usando la variación en el rendimiento a través de los pliegues.

Elección Entre Métodos de Validación Cruzada

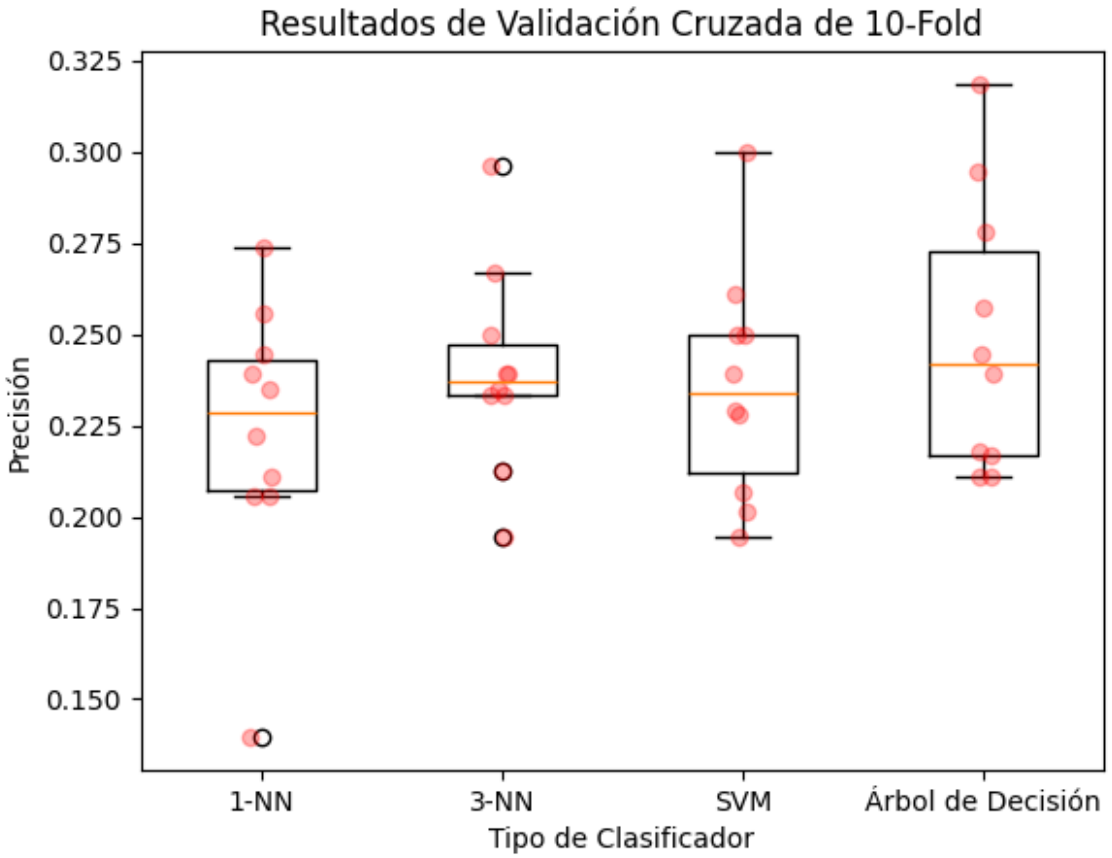
- **Tamaño del Conjunto de Datos:** Para conjuntos de datos más pequeños, LOOCV podría ser preferible debido a su uso intensivo de datos para el entrenamiento. Para conjuntos de datos más grandes, la validación cruzada K-fold a menudo es más práctica debido a su menor carga computacional.
- **Varianza y Sesgo:** La validación cruzada K-fold tiende a tener una varianza más baja como estimador del error de prueba, ya que se utilizan y promedian múltiples conjuntos de prueba. LOOCV, al probar solo un ejemplo a la vez, puede tener una varianza más alta.
- **Intervalos de Confianza:** La capacidad de estimar intervalos de confianza en la validación cruzada K-fold ayuda a comprender la fiabilidad del proceso de evaluación del modelo.

En resumen, la elección de la técnica de validación cruzada puede impactar significativamente la eficiencia y efectividad de los procesos de selección y evaluación de modelos. La clave es equilibrar el costo computacional con los beneficios del enfoque de cada método para manejar la varianza y aprovechar los datos disponibles.

```
from sklearn import model_selection, neighbors, tree, svm, metrics
import numpy as np
import matplotlib.pyplot as plt

# Inicializar una matriz para almacenar los puntajes de precisión para
cada pliegue y modelo
acc = np.zeros((10, 4))  # 10 pliegues, 4 modelos

# Crear un objeto KFold para validación cruzada de 10-fold
```

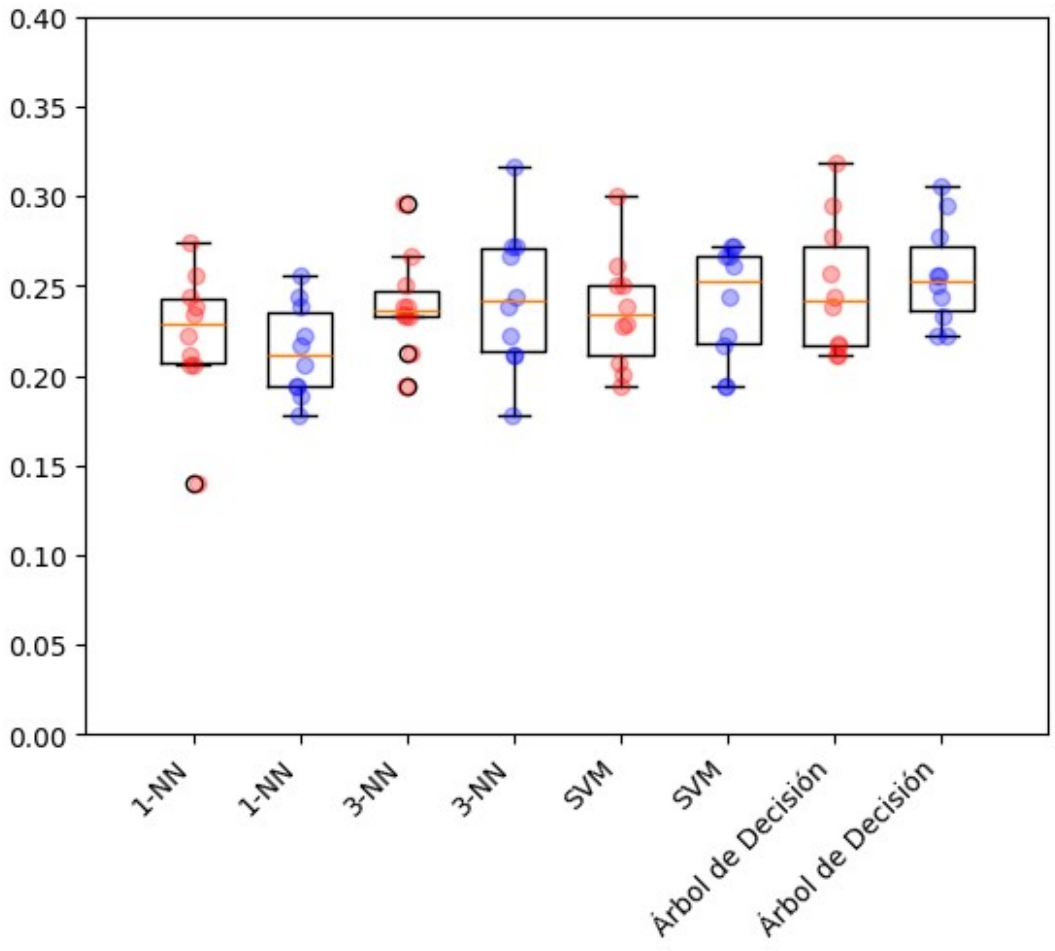
```
kf = model_selection.KFold(n_splits=10, shuffle=True)

# Contador de bucle
i = 0

# Iterar sobre cada pliegue definido por KFold
for train_index, test_index in kf.split(X):
    # Dividir los datos en conjuntos de entrenamiento y prueba basados
en el pliegue actual
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Inicializar clasificadores
    nn1 = neighbors.KNeighborsClassifier(n_neighbors=1)  # 1-vecino
más cercano
    nn3 = neighbors.KNeighborsClassifier(n_neighbors=3)  # 3-vecinos
más cercanos
    svc = svm.SVC()  # Máquina de vectores de soporte (SVM)
    dt = tree.DecisionTreeClassifier()  # Árbol de decisión

    # Entrenar cada clasificador en el conjunto de entrenamiento
    nn1.fit(X_train, y_train)
    nn3.fit(X_train, y_train)
    svc.fit(X_train, y_train)
    dt.fit(X_train, y_train)

    # Realizar predicciones en el conjunto de prueba
    yhat_nn1 = nn1.predict(X_test)
    yhat_nn3 = nn3.predict(X_test)
    yhat_svc = svc.predict(X_test)
    yhat_dt = dt.predict(X_test)

    # Calcular y almacenar la precisión para cada clasificador
    acc[i][0] = metrics.accuracy_score(yhat_nn1, y_test)
    acc[i][1] = metrics.accuracy_score(yhat_nn3, y_test)
    acc[i][2] = metrics.accuracy_score(yhat_svc, y_test)
    acc[i][3] = metrics.accuracy_score(yhat_dt, y_test)

    # Incrementar el contador del bucle
    i += 1

# Visualizar los puntajes de precisión como un gráfico de caja para
cada clasificador
plt.boxplot(acc)

# Superponer los puntajes individuales de precisión como puntos rojos
para una mejor visualización
for i in range(4):
    xderiv = (i+1) * np.ones(acc[:, i].shape) + (np.random.rand(10,) -
0.5) * 0.1
```

```
    plt.plot(xderiv, acc[:, i], 'ro', alpha=0.3)

# Configurar las etiquetas para cada clasificador en el eje x
ax = plt.gca()
ax.set_xticklabels(['1-NN', '3-NN', 'SVM', 'Árbol de Decisión'])

plt.xlabel('Tipo de Clasificador')
plt.ylabel('Precisión')
plt.title('Resultados de Validación Cruzada de 10-Fold')

plt.show()
```




```
# Solo por diversión, vamos a juntar ambos gráficos
fig = plt.figure()
ax = plt.gca()

# Iterar sobre cada clasificador
for i in range(4):
    # Graficar el gráfico de caja para los resultados de validación
cruzada y repetición
    plt.boxplot([acc[:,i], acc_r[:,i]], positions=[2*i+1, 2*i+2],
widths=0.6)
```

```
    # Graficar los puntos individuales de precisión para validación
cruzada (rojo) y repetición (azul)
    xderiv = (2*i+1) * np.ones(acc[:,i].shape) + (np.random.rand(10,)
- 0.5) * 0.1
    plt.plot(xderiv, acc[:,i], 'ro', alpha=0.3)
    xderiv = (2*i+2) * np.ones(acc[:,i].shape) + (np.random.rand(10,)
- 0.5) * 0.1
    plt.plot(xderiv, acc_r[:,i], 'bo', alpha=0.3)

# Establecer límites y etiquetas de los ejes
plt.xlim(0, 9)
plt.ylim(0, 0.4)
ax.set_xticklabels(['1-NN', '1-NN', '3-NN', '3-NN', 'SVM', 'SVM',
'Árbol de Decisión', 'Árbol de Decisión'], rotation=45, ha="right")

# Mostrar el gráfico
plt.show()
```



Resumen: Interfaz de estimador de Scikit-learn

Scikit-learn se esfuerza por tener una interfaz uniforme en todos los métodos, y veremos ejemplos de estos a continuación. Dado un objeto *estimador* de scikit-learn llamado `model`, los siguientes métodos están disponibles:

- Disponible en **todos los Estimadores**
  - `model.fit()` : ajusta los datos de entrenamiento. Para aplicaciones de aprendizaje supervisado, esto acepta dos argumentos: los datos `X` y las etiquetas `y` (por ejemplo, `model.fit(X, y)`). Para aplicaciones de aprendizaje no supervisado, esto acepta solo un argumento, los datos `X` (por ejemplo, `model.fit(X)`).
- Disponible en **estimadores supervisados**
  - `model.predict()` : dado un modelo entrenado, predice la etiqueta de un nuevo conjunto de datos. Este método acepta un argumento, los nuevos datos `X_new` (por ejemplo, `model.predict(X_new)`), y devuelve la etiqueta aprendida para cada objeto en el array.
  - `model.predict_proba()` : Para problemas de clasificación, algunos estimadores también proporcionan este método, que devuelve la probabilidad de que una nueva observación tenga cada etiqueta categórica. En este caso, la etiqueta con la probabilidad más alta es devuelta por `model.predict()`.
  - `model.score()` : para problemas de clasificación o regresión, la mayoría (¿todos?) de los estimadores implementan un método de puntuación. Las puntuaciones están entre 0 y 1, con una puntuación más alta indicando un mejor ajuste.
- Disponible en **estimadores no supervisados**
  - `model.transform()` : dado un modelo no supervisado, transforma nuevos datos en la nueva base. Esto también acepta un argumento `X_new`, y devuelve la nueva representación de los datos basada en el modelo no supervisado.
  - `model.fit_transform()` : algunos estimadores implementan este método, que realiza de manera más eficiente un ajuste y una transformación en los mismos datos de entrada.