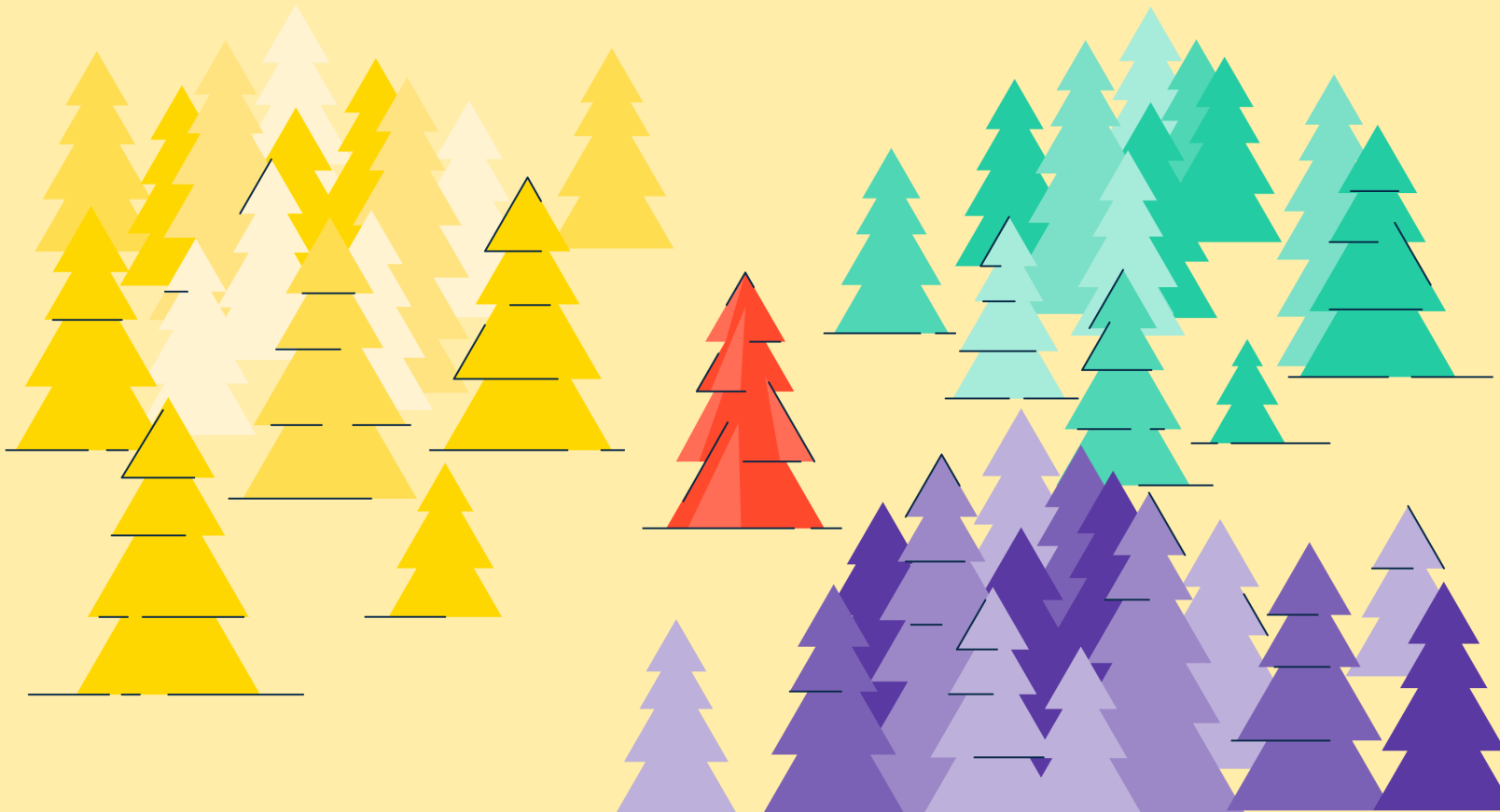
# Guía completa de K-Nearest Neighbors (Con ejemplos de código en Python)

Por Antonio Richaud



# 1. Introducción a K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) es un algoritmo de aprendizaje supervisado bastante útil y fácil de entender, que se puede ocupar tanto en tareas de clasificación como de regresión. A pesar de su simplicidad, KNN es bastante poderoso y se utiliza en una amplia variedad de aplicaciones, desde la recomendación de productos hasta la detección de anomalías.

## ¿Cómo funciona KNN?

El funcionamiento de KNN se basa en una premisa simple pero efectiva: objetos similares están cerca entre sí. Este algoritmo clasifica un nuevo punto de datos basado en la categoría mayoritaria de sus K vecinos más cercanos, o predice un valor promedio de esos vecinos si se trata de una tarea de regresión.

**El proceso de KNN puede dividirse en los siguientes pasos:**

### 1. Representación de los datos

Cada punto de datos en el conjunto de entrenamiento se representa en un espacio n-dimensional, donde n es el número de características o variables. Por ejemplo, si estamos clasificando flores según el largo y ancho de sus pétalos, cada flor es un punto en un espacio bidimensional.

### 2. Cálculo de la distancia

Para un nuevo punto de datos, KNN calcula la distancia entre este punto y todos los demás puntos del conjunto de entrenamiento. Las distancias más comunes son la Euclidiana, Manhattan o Minkowski. Estas distancias

determinan qué tan similares son los puntos. Por ejemplo, la distancia Euclidiana se calcula con la siguiente fórmula:

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

donde $x_i$ y $y_i$ son las coordenadas de los puntos en el espacio. No se espanten con ese monstruo de fórmula, vamos a ir explicándola poquito a poquito para que sea entendible. ;)

### 3. Selección de los vecinos más cercanos

Después de calcular la distancia entre el nuevo punto de datos y todos los otros puntos en el conjunto de entrenamiento, KNN selecciona los K puntos más cercanos, que son los vecinos más similares al nuevo punto.

### 4. Clasificación o predicción

En el caso de la clasificación, KNN toma algo así como una “votación” entre los vecinos más cercanos. El nuevo punto de datos se clasifica en la categoría que tiene la mayoría de los votos entre sus K vecinos.

Para la regresión, en lugar de votar, KNN toma el promedio de los valores de los K vecinos para hacer la predicción del nuevo punto de datos.

### 5. Decisión

Y finalmente, el algoritmo asigna la clase (en clasificación) o predice un valor (en regresión) para el nuevo punto de datos, basado en la mayoría o el promedio de los valores de los K vecinos seleccionados. Sencillo ¿no?

Este algoritmo no requiere un proceso de entrenamiento complejo, lo que convierte a KNN en un método de “aprendizaje perezoso”. Esto significa que toda la “inteligencia” ocurre durante la fase de predicción, cuando se hace la comparación con el conjunto de datos de entrenamiento. Este enfoque tiene ventajas y desventajas: si bien KNN es fácil de implementar y puede funcionar bien con conjuntos de datos pequeños, se vuelve computacionalmente costoso a medida que el tamaño del conjunto de datos crece, ya que implica calcular la distancia con cada punto en el conjunto de entrenamiento para cada predicción.

### Ejemplo sencillo de implementación en python

Veamos nuestro primer ejemplo práctico de cómo implementar KNN en Python utilizando la biblioteca scikit-learn:

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Hacer predicciones
predicciones = knn.predict(X_test)

# Mostrar las predicciones
print("Predicciones:", predicciones)
```

**Explicación del código**:

**1. Carga de datos**: Se carga el conjunto de datos Iris, un conjunto de datos clásico para la clasificación de flores en tres categorías diferentes basadas en el tamaño y la forma de sus pétalos y sépalos.

**2. División de datos**: Los datos se dividen en conjuntos de entrenamiento y prueba. El 30% de los datos se utiliza para probar la eficacia del modelo.

**3. Creación del modelo**: Se crea un modelo KNN con 3 vecinos (n_neighbors = 3). Esto significa que, para clasificar un nuevo punto de datos, el modelo considerará las 3 instancias más cercanas en el conjunto de entrenamiento.

**4. Entrenamiento**: El modelo se entrena ajustando el clasificador KNN a los datos de entrenamiento.

**5. Predicción**: Se utilizan los datos de prueba para hacer predicciones y ver cómo de bien el modelo KNN puede clasificar nuevas instancias.

Este ejemplo súper básico nos da una idea clara de cómo funciona KNN. A lo largo de todo este documento, vamos a platicar sobre cómo optimizar este modelo ajustando hiperparámetros como el valor de K, el tipo de métrica de distancia utilizada, y cómo manejar datos categóricos y escalar características para mejorar la precisión y eficiencia del modelo. :)

# 2. Elección del valor de K en KNN

El valor de K es un hiperparámetro bien importante en el algoritmo KNN, ya que define cuántos vecinos se considerarán al clasificar o predecir el valor de un nuevo punto de datos. Elegir un valor de K adecuado es esencial para el rendimiento del modelo, ya que afecta directamente la capacidad del algoritmo para generalizar a datos no vistos.

**¿Cómo elegir el valor correcto de K?** 🤔

**K pequeño (p. ej., K = 1):** Si K es demasiado pequeño, el modelo puede volverse susceptible al ruido en los datos. Esto significa que un solo punto de datos atípico puede influir demasiado en la clasificación, lo que puede llevar a un sobreajuste. Un K muy pequeño suele resultar en un modelo muy flexible que sigue demasiado de cerca los datos de entrenamiento.

**K grande (p. ej., K = un número muy alto):** Si K es demasiado grande, el modelo puede volverse demasiado general. Esto significa que estará promediando sobre un gran número de vecinos, lo que puede llevar a un subajuste. Un K grande reduce la variabilidad, pero puede también diluir la influencia de puntos de datos más relevantes.

**Elección del K óptimo:** El valor óptimo de K a menudo se encuentra mediante prueba y error, utilizando técnicas como la validación cruzada. Este método implica probar varios valores de K y elegir el que minimice el error en un conjunto de validación. Generalmente, se recomienda probar valores impares de K para evitar empates en la votación durante la clasificación.

## Selección de K con validación cruzada

Aquí te dejo un ejemplo sencillo en python, donde se realiza una búsqueda del valor óptimo de K mediante validación cruzada con el mismo conjunto de datos del ejemplo anterior:

```python
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

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

# Lista para almacenar las puntuaciones de validación cruzada
k_values = range(1, 31)
cv_scores = []

# Probar diferentes valores de K
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())

# Determinar el valor óptimo de K
optimal_k = k_values[cv_scores.index(max(cv_scores))]
print(f"El valor óptimo de K es: {optimal_k}")

# Graficar la relación entre K y la precisión
plt.plot(k_values, cv_scores)
plt.xlabel('Valor de K')
plt.ylabel('Precisión de Validación Cruzada')
plt.title('Selección del Valor de K')
plt.show()
```

## Explicación del código

**1. Cálculo de precisión con validación cruzada**: Para cada valor de K en el rango de 1 a 30, se crea un modelo KNN y se evalúa su precisión

utilizando validación cruzada con 5 particiones (cv = 5). Se calcula la precisión promedio para cada valor de K y se almacena en cv_scores.

**2. Selección del K óptimo**: Se identifica el valor de K que maximiza la precisión promedio obtenida en la validación cruzada. Este valor se considera como el óptimo para el modelo dado.

**3. Visualización**: Se grafica la relación entre el valor de K y la precisión de validación cruzada para ver cómo cambia el rendimiento del modelo a medida que varía K. Esto permite identificar visualmente el punto donde el modelo logra su mejor rendimiento.

Este pequeño fragmento de código proporciona una manera sistemática de seleccionar el valor de K que maximiza la capacidad predictiva del modelo, minimizando el riesgo de sobreajuste o subajuste.

# 3. Métricas de distancia en KNN

En el algoritmo K-Nearest Neighbors (KNN), la elección de la métrica de distancia es fundamental, ya que determina cómo se calcula la “cercanía” entre los puntos de datos. Esta cercanía es lo que guía la clasificación o regresión en KNN, ya que el algoritmo selecciona los K puntos más cercanos para realizar predicciones. Existen varias métricas de distancia comunes que se pueden utilizar, y la elección de una u otra puede tener un impacto significativo en el rendimiento del modelo, ojito con eso.

## Distancia Euclidiana

La distancia euclidiana es la métrica más común y natural para medir la distancia entre dos puntos en un espacio n-dimensional. Es la distancia “recta” entre dos puntos, y se calcula utilizando el teorema de Pitágoras. La fórmula para la distancia euclidiana entre dos puntos $x$ y $y$ en un espacio de $n$ dimensiones es:

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

**Características:**

• Es la métrica por defecto en muchas implementaciones de KNN.

• Es intuitiva y fácil de interpretar.

• Funciona bien cuando las características tienen una escala similar, pero puede ser problemático si las características tienen rangos muy diferentes (en ese caso, es recomendable escalar los datos).

**Veamos un código de ejemplo usando distancia Euclidiana 👀**

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN utilizando la distancia euclidiana
knn = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con distancia Euclidiana: {precision:.4f}")
```

## Distancia Manhattan

La **distancia Manhattan** (también conocida como distancia L1 o de bloque) mide la distancia entre dos puntos a lo largo de los ejes del espacio. En lugar de calcular la línea recta, como en la distancia euclidiana, la distancia Manhattan suma las diferencias absolutas de sus coordenadas. La fórmula es:

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

**Características:**

• Es útil cuando las características están en diferentes escalas o cuando las diferencias entre las dimensiones son importantes.

• Funciona bien en espacios de alta dimensión donde las características no están correlacionadas.

**Ahora un ejemplo usando distancia Manhattan 👀**

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN utilizando la distancia manhattan
knn = KNeighborsClassifier(n_neighbors=3, metric='manhattan')
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con distancia Manhattan: {precision:.4f}")
```

## Distancia de Minkowski

La distancia de Minkowski es una generalización de la distancia euclidiana y la distancia Manhattan. Está definida por un parámetro p que determina el tipo de distancia que se calcula:

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

- Para p = 1, la distancia de Minkowski es equivalente a la distancia Manhattan.

- Para p = 2, es equivalente a la distancia Euclidiana.

- Para valores de p mayores, la distancia Minkowski se enfoca más en las mayores diferencias entre las coordenadas.

**Características:**

- Ofrece flexibilidad para adaptar la métrica de distancia al problema específico.

- Se puede ajustar el valor de p para obtener el mejor rendimiento.

**En la siguiente pagina viene el código pero seguro que ya te imaginas como es :b**

**Aquí esta el ejemplo usando distancia Minkowski 👀**

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN utilizando la distancia minkowski
knn = KNeighborsClassifier(n_neighbors=3, metric='minkowski', p=3)
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con distancia de Minkowski (p=3): {precision:.4f}")
```

## Distancia coseno

La **distancia coseno** mide la similitud entre dos vectores calculando el coseno del ángulo entre ellos. Esta métrica es especialmente útil en aplicaciones donde la magnitud de los vectores no es tan importante como su dirección, como en el procesamiento de texto o datos de alta dimensión.

La fórmula para la similitud coseno es:

$$\text{sim}(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

La distancia coseno, que es 1 menos la similitud coseno, por lo que se define como:

$$d(x, y) = 1 - \frac{x \cdot y}{\|x\| \|y\|}$$

**Características:**

- Ideal para problemas de clasificación de texto y otras tareas en las que los vectores son de alta dimensión.
- Ignora la magnitud, enfocándose en la dirección de los vectores.

**Como no podía ser de otra manera, aquí esta el ejemplo en código**

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN utilizando la distancia coseno
knn = KNeighborsClassifier(n_neighbors=3, metric='cosine')
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con distancia Coseno: {precision:.4f}")
```

## Comparación y selección de la métrica de distancia 🤔

La elección de la métrica de distancia depende en gran medida de la naturaleza del conjunto de datos y del problema que se está resolviendo. Algunas consideraciones importantes incluyen:

- **Escalado de características**: La distancia euclidiana y Minkowski pueden ser sensibles a las diferencias en la escala de las características, por lo que es recomendable normalizar o estandarizar los datos antes de aplicar estas métricas.

- **Dimensionalidad**: En problemas de alta dimensionalidad, las métricas como la distancia Manhattan pueden ofrecer un mejor rendimiento que la distancia euclidiana.

- **Dominio del problema**: La distancia coseno es especialmente útil en dominios donde la dirección de los vectores es más importante que su magnitud, como en el análisis de texto.

En la práctica, es recomendable experimentar con diferentes métricas de distancia y usar técnicas como la validación cruzada para determinar cuál proporciona los mejores resultados para tu conjunto de datos específico. :)

# 4. K-Nearest Neighbors ponderado (Weighted KNN)

En el enfoque tradicional de KNN, todos los K vecinos más cercanos contribuyen por igual a la decisión final, ya sea en la clasificación o en la regresión. Sin embargo, esto no siempre es ideal. En algunos casos, puede ser más beneficioso asignar más peso a los vecinos más cercanos, ya que es probable que sean más representativos de la clase o el valor que se desea predecir. Este enfoque se conoce como **KNN Ponderado (Weighted KNN)**.

**¿Cómo funciona KNN ponderado?**

En lugar de considerar que cada uno de los K vecinos tiene la misma influencia, KNN ponderado asigna un peso a cada vecino basado en su distancia al punto de datos que se está clasificando o para el que se está prediciendo un valor. Los vecinos más cercanos reciben un mayor peso, lo que significa que tienen una mayor influencia en la predicción final.

Existen diferentes maneras de definir estos pesos:

**1. Pesos uniformes**: Todos los vecinos tienen el mismo peso. Este es el enfoque tradicional de KNN.

**2. Pesos basados en la distancia**: Los vecinos más cercanos reciben un peso más alto. Una forma común de calcular el peso es utilizando la inversa de la distancia:

$$\text{peso}(x_i) = \frac{1}{\text{distancia}(x_i, x)}$$

Aquí, $x_i$ es uno de los K vecinos y $x$ es el punto de datos para el cual se está haciendo la predicción. De esta forma, los vecinos más cercanos tendrán un peso mayor y, por tanto, una mayor influencia en< la predicción final.

### Ejemplo de código para KNN ponderado

Aquí te dejo un código para que sepas implementar KNN Ponderado en Python utilizando scikit-learn y en la siguiente pagina lo explicamos detalladamente para que no queden dudas:

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# KNN con pesos uniformes (todos los vecinos tienen el mismo peso)
knn_uniform = KNeighborsClassifier(n_neighbors=5, weights='uniform')
knn_uniform.fit(X_train, y_train)
y_pred_uniform = knn_uniform.predict(X_test)
accuracy_uniform = accuracy_score(y_test, y_pred_uniform)
print(f"Precisión con pesos uniformes: {accuracy_uniform:.4f}")

# KNN con pesos basados en la distancia (los vecinos más cercanos tienen
más peso)
knn_distance = KNeighborsClassifier(n_neighbors=5, weights='distance')
knn_distance.fit(X_train, y_train)
y_pred_distance = knn_distance.predict(X_test)
accuracy_distance = accuracy_score(y_test, y_pred_distance)
print(f"Precisión con pesos basados en la distancia:
{accuracy_distance:.4f}")
```

**Explicación detallada del código:**

**1. KNN con pesos uniformes**: Se crea un modelo KNN donde todos los vecinos tienen el mismo peso (weights = ‘uniform' ). Este es el enfoque tradicional donde la distancia entre los puntos no afecta su influencia en la predicción.

**2. KNN con pesos basados en la distancia**: En este modelo, los vecinos más cercanos tienen más influencia en la predicción (weights = 'distance'). La precisión del modelo se calcula y se compara con la de KNN con pesos uniformes.

**3. Comparación de resultados**: Al final, se imprime la precisión de ambos modelos para observar cuál da mejores resultados en este caso específico. En muchos casos, KNN ponderado (basado en la distancia) puede mejorar la precisión del modelo, especialmente cuando hay variabilidad significativa en las distancias de los vecinos más cercanos.

**¿Cuándo usar KNN ponderado?** 🤔

El uso de KNN ponderado es recomendable en los siguientes escenarios:

- **Heterogeneidad en las distancias**: Si los vecinos cercanos están a distancias significativamente diferentes, es razonable dar más importancia a los más cercanos.

- **Datos ruidosos**: KNN ponderado puede ayudar a reducir el impacto de los puntos de datos ruidosos que se encuentran más lejos del punto en cuestión.

- **Desequilibrio de clases**: En problemas de clasificación con desequilibrio de clases, ponderar por distancia puede ayudar a evitar

que las clases menos representadas sean eclipsadas por las más representadas.

KNN ponderado agrega una capa adicional de flexibilidad y puede mejorar la precisión del modelo, pero también puede aumentar la complejidad computacional. Por lo tanto, es importante evaluar su rendimiento en el contexto específico del problema que estás resolviendo.

# 5. K-Nearest Neighbors para regresión

Mientras que KNN se utiliza comúnmente para tareas de clasificación, donde el objetivo es asignar una clase a un nuevo punto de datos, también es un algoritmo poderoso para tareas de **regresión**, donde el objetivo es predecir un valor numérico continuo.

### ¿Cómo funciona KNN en la regresión? 🤔

En lugar de votar por la clase mayoritaria entre los K vecinos más cercanos, el algoritmo toma el **promedio** de los valores asociados a esos K vecinos para hacer la predicción del valor continuo del nuevo punto de datos. Este proceso se puede resumir en los siguientes 3 sencillos pasos:

**1. Cálculo de distancias**: Igual que en la clasificación, se calcula la distancia entre el nuevo punto de datos y todos los puntos en el conjunto de entrenamiento utilizando una métrica de distancia como la euclidiana, Manhattan, Minkowski, etc.

**2. Selección de los K vecinos más cercanos**: Después de calcular las distancias, se seleccionan los K puntos de datos más cercanos al nuevo punto de datos.

**3. Promedio de los valores**: Se toman los valores (en nuestro caso, numéricos) asociados a estos K vecinos y se calcula su promedio. Este promedio es el valor predicho para el nuevo punto de datos.

$$\hat{y} = \frac{1}{K} \sum_{i=1}^{K} y_i$$

Aquí, $y_i$ representa los valores conocidos de los K vecinos más cercanos.

## Aqui te dejo un código para KNN en regresión

```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Generar un conjunto de datos de regresión
X, y = make_regression(n_samples=100, n_features=1, noise=0.1,
random_state=42)

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

# Crear y entrenar el modelo KNN para regresión
knn_regressor = KNeighborsRegressor(n_neighbors=5)
knn_regressor.fit(X_train, y_train)

# Hacer predicciones
y_pred = knn_regressor.predict(X_test)

# Calcular el error cuadrático medio
mse = mean_squared_error(y_test, y_pred)
print(f"Error Cuadrático Medio: {mse:.4f}")
```

## Explicación del código:

**1. Generación de datos**: Aquí utilizamos make_regression de sklearn.datasets para generar un conjunto de datos sintético adecuado para la regresión, con una única característica y un poco de ruido.

**2. División de datos**: Los datos se dividen en un conjunto de entrenamiento y un conjunto de prueba, con un 20% de los datos reservados para pruebas.

**3. Creación del modelo**: Se crea un modelo KNeighborsRegressor con n_neighbors = 5, lo que significa que el valor predicho para cada punto en el conjunto de prueba se basará en el promedio de los valores de sus 5 vecinos más cercanos en el conjunto de entrenamiento.

**4. Predicción y evaluación**: Después de entrenar el modelo, se realizan predicciones sobre el conjunto de prueba. El rendimiento del modelo se evalúa utilizando el **error cuadrático medio (MSE)**, que mide la diferencia promedio entre los valores predichos y los valores reales en el conjunto de prueba.

### Ponderación en KNN para regresión

Al igual que en KNN para clasificación, es posible utilizar un enfoque ponderado en KNN para regresión, donde los valores de los vecinos más cercanos se ponderan de acuerdo con su proximidad al punto de datos que se está prediciendo. En este caso, en lugar de tomar un simple promedio de los valores de los K vecinos, se tomaría un promedio ponderado.

### Ejemplo de código para KNN ponderado en regresión:

```
# KNN para regresión con ponderación basada en la distancia
knn_regressor_weighted = KNeighborsRegressor(n_neighbors=5,
weights='distance')
knn_regressor_weighted.fit(X_train, y_train)

# Hacer predicciones
y_pred_weighted = knn_regressor_weighted.predict(X_test)

# Calcular el error cuadrático medio
mse_weighted = mean_squared_error(y_test, y_pred_weighted)
print(f"Error Cuadrático Medio con Ponderación: {mse_weighted:.4f}")
```

**Explicación del código:**

**1. Modelo ponderado**: Se especifica weights='distance' para que los vecinos más cercanos tengan una mayor influencia en la predicción.

**2. Comparación de resultados**: Finalmente, se calcula el error cuadrático medio y se compara con el modelo sin ponderación para ver cuál ofrece un mejor rendimiento en este contexto.

**¿Cuándo Usar KNN para regresión?** 🤔

KNN para regresión es útil en escenarios donde:

• **Relaciones No Lineales**: El modelo puede captar relaciones no lineales complejas entre las características y el valor objetivo sin necesidad de especificar una forma funcional explícita.

• **Pequeños Conjuntos de Datos**: KNN tiende a funcionar bien en pequeños conjuntos de datos, donde el sobreajuste no es un gran problema.

• **Alta Dimensionalidad**: Aunque KNN para regresión puede ser computacionalmente costoso en espacios de alta dimensionalidad, puede manejar bien las interacciones complejas entre muchas características, siempre y cuando se utilicen técnicas de reducción de dimensionalidad o de selección de características.

KNN para regresión es una técnica poderosa y flexible que, a pesar de su simplicidad, puede ofrecer un rendimiento excelente en una variedad de tareas de predicción numérica.

# 6. Manejo de características categóricas en KNN

El algoritmo K-Nearest Neighbors (KNN) funciona de manera intuitiva y directa con características numéricas, ya que la noción de “cercanía” se basa en cálculos de distancia como la Euclidiana o Manhattan, que son fácilmente aplicables a datos numéricos. Sin embargo, cuando los datos incluyen **características categóricas** (por ejemplo, “color”, “tipo de producto”, “estado civil”), la situación se vuelve más compleja, ya que no es posible calcular directamente distancias entre categorías como lo hacemos con números.

Para manejar esta situación, es necesario transformar las características categóricas en una forma que permita al algoritmo KNN realizar sus cálculos de distancia. Existen varios métodos para convertir datos categóricos en formatos numéricos que KNN pueda manejar:

## One-Hot Encoding

El **One-Hot Encoding** es una técnica que convierte características categóricas en variables binarias. Por ejemplo, si tienes una característica “color” con tres categorías posibles (“rojo”, “verde”, “azul”), One-Hot Encoding crearía tres nuevas columnas, una para cada color, que contienen 1 si la observación pertenece a esa categoría y 0 en caso contrario.

**Ejemplo:**
Característica Original: Color = [Rojo, Verde, Azul]

- Rojo = [1, 0, 0]
- Verde = [0, 1, 0]
- Azul = [0, 0, 1]

## Implementación en Python:

```python
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Crear un DataFrame con características categóricas y numéricas
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4],
    'feature2': ['A', 'B', 'A', 'C'],
    'target': [0, 1, 0, 1]
})

# Identificar las características categóricas y numéricas
categorical_features = ['feature2']
numeric_features = ['feature1']

# Crear el transformador de columnas para aplicar One-Hot Encoding
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numeric_features),
        ('cat', OneHotEncoder(), categorical_features)
    ])

# Aplicar la transformación y separar en conjunto de características y
etiquetas
X = data.drop('target', axis=1)
y = data['target']
X_encoded = preprocessor.fit_transform(X)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_encoded, y,
test_size=0.3, random_state=42)

# Crear y entrenar el modelo KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión después de One-Hot Encoding: {precision:.4f}")
```

**Explicación del código:**

**1. One-Hot Encoding**: Se utiliza OneHotEncoder de sklearn.preprocessing para convertir la característica categórica “feature2” en varias columnas binarias.

**2. Transformación de Datos**: El ColumnTransformer se encarga de aplicar One-Hot Encoding solo a las características categóricas, mientras que las características numéricas pasan sin cambios (‘passthrough’).

**3. Entrenamiento y Evaluación**: Después de transformar las características, el conjunto de datos se divide en entrenamiento y prueba, y se entrena un modelo KNN utilizando las características transformadas.

## Label Encoding

El **Label Encoding** convierte las categorías en números enteros. Por ejemplo, “rojo”, “verde”, “azul” pueden convertirse en 1, 2 y 3 respectivamente. Aunque este método es más simple que One-Hot Encoding, puede introducir relaciones ordinales artificiales entre las categorías que no existen en la realidad.

**Ejemplo:**

Característica Original: Color = [Rojo, Verde, Azul]

Color = [1, 2, 3]

## Implementación en Python:

```python
from sklearn.preprocessing import LabelEncoder

# Crear un DataFrame con características categóricas y numéricas
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4],
    'feature2': ['A', 'B', 'A', 'C'],
    'target': [0, 1, 0, 1]
})

# Aplicar Label Encoding a las características categóricas
label_encoder = LabelEncoder()
data['feature2_encoded'] = label_encoder.fit_transform(data['feature2'])

# Separar las características y la etiqueta
X = data[['feature1', 'feature2_encoded']]
y = data['target']

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear y entrenar el modelo KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión después de Label Encoding: {precision:.4f}")
```

## Explicación del código:

**1. Label Encoding**: LabelEncoder convierte la característica categórica “feature2” en valores numéricos.

**2. Precisión**: Este enfoque es simple, pero como se mencionó antes, debe usarse con precaución en casos donde no hay una relación

inherente entre las categorías, ya que podría influir en cómo el modelo interpreta las distancias.

**Consideraciones y buenas prácticas**

• **Escalado de características**: Cuando se mezclan características numéricas y categóricas, es importante escalar las características numéricas para que no dominen las distancias calculadas por KNN.

• **Compatibilidad del modelo**: Algunas transformaciones, como One-Hot Encoding, aumentan el número de dimensiones en los datos, lo que puede afectar el rendimiento de KNN, especialmente en conjuntos de datos grandes.

• **Pruebas y validación**: Al igual que con otros aspectos de KNN, es fundamental probar diferentes enfoques de codificación y validar cuál funciona mejor en tu conjunto de datos específico.

Manejar características categóricas de manera adecuada en KNN es súper importante para asegurar que el modelo pueda captar patrones relevantes sin introducir sesgos o interpretaciones erróneas.

# 7. Maldición de la dimensionalidad en KNN 😈

La **maldición de la dimensionalidad** es un fenómeno que ocurre cuando trabajamos con conjuntos de datos que tienen un gran número de características o dimensiones. A medida que aumenta el número de dimensiones en un espacio de características, los datos se vuelven más dispersos, lo que puede afectar negativamente el rendimiento de algoritmos como K-Nearest Neighbors (KNN).

**¿Por qué ocurre la maldición de la dimensionalidad?**

En un espacio de baja dimensión (por ejemplo, 2D o 3D), es relativamente fácil encontrar puntos de datos cercanos entre sí. Sin embargo, a medida que aumentamos el número de dimensiones, la distancia entre los puntos de datos tiende a aumentar. Esto se debe a que los puntos de datos ocupan un volumen más grande en el espacio multidimensional, lo que hace que todos los puntos estén, en promedio, más lejos unos de otros.

Esto presenta varios problemas para KNN:

**1. Distancias similares**: A medida que aumenta la dimensionalidad, las distancias entre los puntos comienzan a parecerse más, lo que significa que no hay una clara diferencia entre los vecinos cercanos y lejanos. Esto dificulta la capacidad de KNN para identificar los puntos más cercanos y, por lo tanto, realizar predicciones precisas.

**2. Espacio escasamente poblado**: En espacios de alta dimensión, la densidad de los datos disminuye, lo que significa que es probable que haya menos puntos de datos cercanos en cualquier vecindario local. Esto puede llevar a un mayor error en la predicción, ya que KNN depende de tener vecinos cercanos para hacer inferencias.

**3. Sobrecarga computacional**: Calcular distancias en un espacio de alta dimensión es computacionalmente costoso, lo que puede hacer que KNN sea ineficiente para grandes conjuntos de datos con muchas características.

**Ejemplo ilustrativo:**
Imaginemos un conjunto de datos con diferentes números de dimensiones y cómo la distancia promedio entre puntos cambia al aumentar la dimensionalidad:

```python
import numpy as np
import matplotlib.pyplot as plt

# Número de puntos de datos
n_points = 1000

# Lista para almacenar distancias promedio
avg_distances = []

# Calcular distancias en diferentes dimensiones
dims = range(1, 101)
for dim in dims:
    points = np.random.rand(n_points, dim)
    distances = []

    # Calcular la distancia Euclidiana entre puntos aleatorios
    for i in range(n_points):
        for j in range(i + 1, n_points):
            distance = np.linalg.norm(points[i] - points[j])
            distances.append(distance)

    # Almacenar la distancia promedio para la dimensión actual
    avg_distances.append(np.mean(distances))

# Graficar la relación entre la dimensionalidad y la distancia promedio
plt.plot(dims, avg_distances)
plt.xlabel('Número de Dimensiones')
plt.ylabel('Distancia Promedio')
plt.title('Maldición de la Dimensionalidad')
plt.show()
```

**Explicación del código:**

**1. Generación de puntos aleatorios**: Generamos puntos de datos aleatorios en espacios de diferentes dimensiones (de 1 a 100).

**2. Cálculo de distancias**: Para cada dimensión, calculamos las distancias euclidianas entre pares de puntos aleatorios y almacenamos la distancia promedio.

**3. Visualización**: Graficamos la relación entre el número de dimensiones y la distancia promedio entre puntos. La gráfica suele mostrar que, a medida que aumentan las dimensiones, la distancia promedio también aumenta, lo que ilustra el efecto de la maldición de la dimensionalidad.

**Mitigación de la maldición de la dimensionalidad**

Existen varias técnicas para mitigar los efectos de la maldición de la dimensionalidad en KNN:

**1. Selección de características**: Reducir el número de características seleccionando solo las más relevantes. Técnicas como la selección de características basada en la importancia o la eliminación de características redundantes pueden ayudar a reducir la dimensionalidad del problema

**2. Reducción de dimensionalidad**: Aplicar técnicas de reducción de dimensionalidad, como el **Análisis de Componentes Principales (PCA)** o **t-SNE**, puede transformar los datos a un espacio de menor dimensión sin perder demasiada información.

**Checa un ejemplo sencillo de como usar PCA para reducir dimensionalidad:**

```python
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

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

# Aplicar PCA para reducir las dimensiones
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y,
test_size=0.3, random_state=42)

# Crear y entrenar el modelo KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Hacer predicciones y calcular la precisión
y_pred = knn.predict(X_test)
precision = accuracy_score(y_test, y_pred)
print(f"Precisión después de PCA: {precision:.4f}")
```

En este pequeño ejemplo, se aplica PCA para reducir las dimensiones del conjunto de datos Iris de 4 a 2. Luego, entrenamos un modelo KNN en este espacio reducido. Esto puede ayudar a mejorar el rendimiento y reducir los efectos negativos de la alta dimensionalidad.

**3. Normalización y estandarización**: Escalar las características puede ayudar a asegurar que ninguna característica domine la métrica de distancia debido a diferencias en la escala. Esto es especialmente importante cuando se trabaja con datos de alta dimensión.

**4. Usar Distancias Aproximadas**: Para conjuntos de datos extremadamente grandes y de alta dimensionalidad, técnicas como **Locality Sensitive Hashing (LSH)** pueden aproximar las distancias de manera eficiente, lo que reduce la carga computacional.

## Conclusión

La maldición de la dimensionalidad es un problema crítico que afecta a KNN, especialmente cuando se trabaja con datos de alta dimensionalidad. Sin embargo, mediante la aplicación de técnicas adecuadas de selección de características, reducción de dimensionalidad y preprocesamiento de datos, es posible mitigar sus efectos y mejorar el rendimiento del modelo.

# 8. Escalado de características para KNN

El **escalado de características** es un paso importantísimo en el preprocesamiento de datos cuando se utiliza el algoritmo K-Nearest Neighbors (KNN). Esto se debe a que KNN se basa en cálculos de distancia, y cuando las características de los datos tienen escalas diferentes, algunas características pueden dominar el cálculo de la distancia, lo que puede sesgar el modelo.

### ¿Por qué es necesario el escalado?

Considera un conjunto de datos donde tienes dos características: “altura” en metros y “peso” en kilogramos. Supongamos que la altura varía entre 1.5 y 2 metros, mientras que el peso varía entre 50 y 100 kilogramos. En este caso, la característica “peso” tendrá un rango mucho mayor y, por lo tanto, dominará las distancias calculadas por KNN. Esto podría hacer que el modelo ignore en gran medida la información contenida en la característica “altura”, lo que podría afectar negativamente el rendimiento del modelo.

El escalado de características asegura que todas las características contribuyan de manera equitativa al cálculo de la distancia, permitiendo que el modelo KNN tome en cuenta todas las variables relevantes en su toma de decisiones.

### Técnicas comunes de escalado

Existen varias técnicas de escalado de características que pueden utilizarse con KN pero las más comunes son:

**1. Normalización (Min-Max scaling)**:

La normalización escala las características a un rango específico, típicamente entre 0 y 1.

La fórmula es:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Después de la normalización, todas las características tendrán el mismo rango, lo que equilibra su influencia en el cálculo de la distancia.

**2. Estandarización (Standardization o Z-score scaling):**

La estandarización transforma las características para que tengan una media de 0 y una desviación estándar de 1.

La fórmula es:

$$x' = \frac{x - \mu}{\sigma}$$

donde $\mu$ es la media de la característica y sigma es la desviación estándar. Esto puede ser útil cuando los datos tienen diferentes escalas pero no tienen un rango conocido.

**Ejemplo de escalado en KNN**

En la próxima pagina por que ya no cabe en esta, te dejo cómo implementar el escalado de características en Python.

Aquí esta el ejemplo del código :)

```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear un pipeline que incluya el escalado de características y el
modelo KNN
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Estandarización
    ('knn', KNeighborsClassifier(n_neighbors=3))  # KNN
])

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

# Hacer predicciones
y_pred = knn_pipeline.predict(X_test)

# Calcular la precisión
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con escalado de características: {precision:.4f}")
```

**Explicación del código**:

**1. Pipeline**: Utilizamos un pipeline para encadenar el escalado de características (StandardScaler) y el modelo KNeighborsClassifier. Esto asegura que las características se escalen adecuadamente antes de aplicar KNN.

**2. Estandarización**: StandardScaler estandariza las características para que tengan una media de 0 y una desviación estándar de 1. Esto es particularmente útil cuando las características tienen distribuciones diferentes.

**3. Precisión del modelo**: Después de aplicar el escalado, entrenamos el modelo y evaluamos su precisión. En muchos casos, escalar las características mejora significativamente el rendimiento de KNN.

**¿Cuándo usar normalización vs. estandarización?** 🤔

- **Normalización**: Es preferible cuando las características están en diferentes rangos pero se espera que todas tengan una distribución similar. Por ejemplo, si estás trabajando con datos que miden diferentes tipos de cantidades físicas (metros, kilogramos, segundos), la normalización es una buena opción.

- **Estandarización**: Es preferible cuando las características tienen distribuciones diferentes o cuando se espera que algunas características tengan una mayor variabilidad que otras. La estandarización es también la técnica de elección cuando no hay un rango específico de valores para las características.

El escalado de características es un paso de cuidado para asegurar que KNN funcione correctamente. Sin un escalado adecuado, KNN puede producir resultados sesgados o inexactos, especialmente en conjuntos de datos donde las características tienen diferentes escalas o distribuciones. Mediante la aplicación de técnicas de normalización o estandarización, es posible mejorar significativamente la precisión y robustez del modelo.

# 9. Uso de PCA para la reducción de dimensionalidad en KNN

El **análisis de componentes principales (PCA)** es una técnica estadística que se utiliza para reducir la dimensionalidad de un conjunto de datos al transformar un conjunto de variables posiblemente correlacionadas en un conjunto más pequeño de variables no correlacionadas, llamadas componentes principales. PCA es especialmente útil en KNN para abordar el problema de la **maldición de la dimensionalidad**, como ya lo habíamos explicado unas paginas atrás pero considero importante profundizar un poco en esto.

**¿Cómo funciona PCA?** 🤔

PCA transforma los datos originales en un nuevo conjunto de variables, que son combinaciones lineales de las variables originales. Estos nuevos componentes están ordenados de manera que el primer componente principal captura la mayor parte de la varianza en los datos, el segundo componente principal captura la mayor parte de la varianza restante, y así sucesivamente.

La reducción de dimensionalidad se logra seleccionando solo los primeros componentes principales, lo que permite representar los datos en un espacio de menor dimensión sin perder demasiada información.

**Beneficios de usar PCA en KNN**

**1. Reducción de dimensionalidad:** Al reducir el número de dimensiones, se puede mitigar el problema de la maldición de la dimensionalidad, lo que mejora la eficiencia del algoritmo y puede mejorar la precisión.

**2. Eliminación de ruido**: PCA puede ayudar a eliminar el ruido al filtrar las dimensiones menos importantes, lo que hace que KNN se enfoque en las dimensiones que realmente importan.

**3. Visualización**: PCA facilita la visualización de datos en espacios de baja dimensión, lo que puede ser útil para el análisis exploratorio de datos.

### Implementación de PCA para reducir la dimensionalidad

```
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Crear un pipeline que incluya PCA y el modelo KNN
pca_knn_pipeline = Pipeline([
    ('pca', PCA(n_components=2)),  # Reducir a 2 componentes principales
    ('knn', KNeighborsClassifier(n_neighbors=3))  # KNN
])

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

# Hacer predicciones
y_pred = pca_knn_pipeline.predict(X_test)

# Calcular la precisión
precision = accuracy_score(y_test, y_pred)
print(f"Precisión con PCA y KNN: {precision:.4f}")
```

**Como ya es costumbre vamos a explicar un poco el código:**

**1. Pipeline con PCA y KNN**: Se crea un pipeline que primero aplica PCA para reducir las dimensiones a 2 componentes principales, y luego aplica KNN. Esto asegura que los datos se transformen correctamente antes de entrenar el modelo KNN.

**2. Reducción de Dimensionalidad**: PCA(n_components = 2) indica que reduciremos las dimensiones a solo 2 componentes principales. Este número se puede ajustar según las necesidades del problema y la cantidad de varianza que se desee retener.

**3. Entrenamiento y Predicción**: Después de reducir las dimensiones, entrenamos el modelo KNN y realizamos predicciones. La precisión del modelo después de aplicar PCA se evalúa y se compara con la precisión original.

**Consideraciones al Usar PCA** 🧐

• **Varianza retenida**: Es importante seleccionar un número adecuado de componentes principales para retener la mayor parte de la varianza original en los datos. Esto se puede determinar utilizando la relación de varianza explicada (explained variance ratio) que proporciona PCA.

• **Dimensionalidad vs. interpretabilidad**: Mientras que PCA reduce la dimensionalidad, también puede hacer que las características sean menos interpretables, ya que los componentes principales son combinaciones lineales de las características originales.

• **Adecuación de PCA**: PCA es más adecuado cuando las características originales están correlacionadas. Si las características no están correlacionadas, otras técnicas de reducción de dimensionalidad, como **t-SNE** o **UMAP**, podrían ser mucho más efectivas.

### Ejemplo avanzado: Selección del número óptimo de componentes

Es posible que te hayas preguntado como determinar cuántos componentes principales retienen la mayor parte de la varianza en los datos (Y si no, de todos modos te lo voy a explicar):

```
import matplotlib.pyplot as plt
import numpy as np

# Ajustar PCA al conjunto de datos para ver la varianza explicada
pca = PCA().fit(X)

# Graficar la varianza explicada acumulada
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Número de Componentes')
plt.ylabel('Varianza Explicada Acumulada')
plt.title('Selección del Número de Componentes')
plt.grid(True)
plt.show()

# Seleccionar el número de componentes que retiene el 95% de la varianza
pca_opt = PCA(n_components=0.95)
X_reduced = pca_opt.fit_transform(X)
print(f"Número de Componentes que retienen el 95% de la varianza:
{pca_opt.n_components_}")
```

### Expliquemos el código:

**1. Varianza explicada acumulada**: El gráfico que te va a salir al ejecutar este código muestra la relación de varianza explicada acumulada por cada número de componentes principales. Esto te permitirá ver cuántos componentes necesitas para retener un porcentaje específico de la varianza total.

**2. Selección automática de componentes**: PCA(n_components = 0.95) seleccionamos automáticamente el número de componentes que retiene al menos el 95% de la varianza. Este es un enfoque bien útil cuando no estás seguro de cuántos componentes elegir.

Usar PCA para la reducción de dimensionalidad en KNN puede ser extremadamente bueno, especialmente cuando se trabaja con datos de alta dimensionalidad. Al reducir las dimensiones, PCA no solo ayuda a mitigar la maldición de la dimensionalidad, sino que también mejora la eficiencia computacional y, en muchos casos, la precisión del modelo, considéralo cada que hagas un análisis con este algoritmo.

# 10. Validación cruzada para la optimización de hiperparámetros en KNN

La **validación cruzada** es una técnica barbara que se utiliza para evaluar el rendimiento de un modelo de aprendizaje automático y para optimizar sus hiperparámetros, como el valor de K en KNN. La validación cruzada permite estimar de manera más precisa cómo se comportará un modelo en datos no vistos, lo que ayuda a evitar el sobreajuste y a seleccionar los mejores hiperparámetros.

**¿Qué es la validación cruzada?** 🤔

En lugar de dividir el conjunto de datos en un único conjunto de entrenamiento y un conjunto de prueba, la validación cruzada divide los datos en varios subconjuntos o **folds**. El modelo se entrena en algunos de estos folds y se prueba en los restantes. Este proceso se repite varias veces, asegurando que cada fold se utilice tanto para entrenamiento como para prueba en algún punto.

El esquema más común de validación cruzada es la **validación cruzada k-fold**, la cual se divide en los siguiente sencillos 4 pasos:

1. El conjunto de datos se divide en k subconjuntos de aproximadamente el mismo tamaño.

2. El modelo se entrena en k-1 de estos subconjuntos y se prueba en el fold restante.

3. Este proceso se repite k veces, utilizando un fold diferente como conjunto de prueba en cada iteración.

4. El rendimiento del modelo se promedia a lo largo de todas las iteraciones para obtener una estimación más robusta.

**Optimización de hiperparámetros con validación cruzada**

En KNN, uno de los hiperparámetros más críticos es el valor de K, que determina el número de vecinos considerados para hacer una predicción. Otros hiperparámetros incluyen la métrica de distancia (Euclidiana, Manhattan, etc.) y el tipo de peso (uniforme o basado en la distancia) como ya lo hemos visto durante todo este documento.

La validación cruzada se puede utilizar para probar diferentes combinaciones de estos hiperparámetros y seleccionar la combinación que produce el mejor rendimiento promedio.

## Veamos un ejemplo de validación cruzada en KNN

```python
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Definir el modelo KNN
knn = KNeighborsClassifier()

# Definir el rango de hiperparámetros que se quieren probar
param_grid = {
    'n_neighbors': range(1, 31),
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Configurar la validación cruzada con GridSearchCV
grid_search = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')

# Entrenar el modelo utilizando GridSearchCV
grid_search.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params = grid_search.best_params_
print(f"Mejores hiperparámetros encontrados: {best_params}")

# Evaluar el modelo con los mejores hiperparámetros en el conjunto de
prueba
best_knn = grid_search.best_estimator_
test_accuracy = best_knn.score(X_test, y_test)
print(f"Precisión en el conjunto de prueba: {test_accuracy:.4f}")
```

## Explicando el código:

**1. GridSearchCV**: Utilizamos GridSearchCV para realizar una búsqueda exhaustiva de los mejores hiperparámetros en KNN. Este enfoque prueba todas las combinaciones de los valores especificados en param_grid y utiliza validación cruzada para evaluar el rendimiento de cada combinación.

**2. Parámetros a probar**: En este ejemplo, estamos probando 30 valores diferentes de K, dos tipos de pesos (uniform y distance), y dos métricas de distancia (euclidean y manhattan).

**3. Validación cruzada**: cv = 10 indica que estamos utilizando validación cruzada de 10-folds. El modelo se entrenará y evaluará 10 veces para cada combinación de hiperparámetros, y la precisión promedio se utilizará para seleccionar los mejores hiperparámetros.

**4. Mejores hiperparámetros**: Al final del proceso, grid_search.best_params_ devuelve la combinación de hiperparámetros que produjo el mejor rendimiento en la validación cruzada. Esta configuración se utiliza para entrenar el modelo final.

**5. Evaluación en el conjunto de prueba**: Finalmente, evaluamos el modelo con los mejores hiperparámetros en el conjunto de prueba independiente para obtener una estimación del rendimiento en datos no vistos.

**Ventajas de la validación cruzada**

- **Estimación robusta**: La validación cruzada proporciona una estimación más confiable del rendimiento del modelo, ya que utiliza múltiples subconjuntos de datos para entrenar y evaluar el modelo.

- **Prevención del sobreajuste**: Al probar el modelo en múltiples folds, la validación cruzada ayuda a identificar configuraciones de hiperparámetros que generalizan mejor a datos nuevos, reduciendo el riesgo de sobreajuste.

- **Optimización de múltiples hiperparámetros**: GridSearchCV permite optimizar varios hiperparámetros simultáneamente, lo que es esencial para encontrar la configuración óptima de KNN.

**Consideraciones al usar validación cruzada**

• **Costo computacional**: La validación cruzada puede ser computacionalmente costosa, especialmente cuando se trabaja con grandes conjuntos de datos o cuando se prueban muchas combinaciones de hiperparámetros. Para reducir la carga computacional, se puede considerar el uso de una versión aleatoria de búsqueda (RandomizedSearchCV) o técnicas de validación cruzada más rápidas, como la validación cruzada con reducción de varianza.

• **Balance entre precisión y complejidad**: Si bien es importante encontrar la mejor configuración de hiperparámetros, también es crucial asegurarse de que el modelo no se vuelva demasiado complejo o específico para el conjunto de datos de entrenamiento, ya que esto podría llevar a un sobreajuste.

La validación cruzada es una herramienta esencial en la optimización de modelos KNN, permitiendo una selección informada de los hiperparámetros y una evaluación más robusta del rendimiento del modelo. Mediante el uso de GridSearchCV u otras técnicas de búsqueda y validación cruzada, es posible mejorar significativamente la precisión y generalización del modelo.

# 11. Manejo de grandes conjuntos de datos en KNN

K-Nearest Neighbors (KNN) es conocido por su simplicidad, pero este don también viene con una maldición significativa: a medida que el tamaño del conjunto de datos crece, el costo computacional de calcular distancias entre todos los puntos de datos se vuelve prohibitivo. Además, KNN es un algoritmo de aprendizaje perezoso, lo que significa que todo el trabajo pesado se realiza en la fase de predicción, lo que puede hacer que las predicciones sean lentas cuando se manejan grandes volúmenes de datos.

**Problemas con grandes conjuntos de datos en KNN**

**1. Costo computacional alto:** KNN requiere calcular la distancia entre el punto de prueba y todos los puntos de entrenamiento. Esto lleva un tiempo de ejecución de $O(n \times d)$, donde n es el número de puntos en el conjunto de entrenamiento y d es la dimensión del espacio de características.

**2. Consumo de memoria:** El modelo KNN debe almacenar todos los puntos de entrenamiento en memoria, lo que puede ser un desafío cuando el conjunto de datos es grande.

**3. Latencia en la predicción:** Dado que KNN no tiene una fase de entrenamiento real, las predicciones pueden ser lentas cuando el conjunto de datos de entrenamiento es grande, lo que afecta la capacidad del modelo para hacer predicciones en tiempo real.

## Estrategias para manejar grandes conjuntos de datos

Existen varias estrategias y técnicas para manejar estos problemas y hacer que KNN sea más eficiente en grandes conjuntos de datos:

### 1. Reducción de dimensionalidad:

• **PCA:** Creo que ya se dieron cuenta que me encanta este método pero es porqué es super efectivo, el análisis de componentes principales (PCA) puede reducir el número de dimensiones, lo que no solo ayuda con la maldición de la dimensionalidad sino que también reduce el costo computacional.

• **t-SNE o UMAP:** Para visualización y reducción de dimensionalidad, t-SNE y UMAP son técnicas que pueden ayudar a comprimir los datos en menos dimensiones manteniendo la estructura interna de los datos.

### 2. Algoritmos aproximados de KNN:

• **Locality Sensitive Hashing (LSH):** LSH es una técnica que permite encontrar vecinos cercanos en alta dimensión de manera más eficiente. En lugar de calcular la distancia exacta entre todos los puntos, LSH utiliza una función de hash para agrupar puntos similares, reduciendo el número de cálculos necesarios.

• **Approximate Nearest Neighbors (ANN):** Implementaciones como Annoy o FAISS (Facebook AI Similarity Search) utilizan técnicas avanzadas para encontrar vecinos más cercanos en grandes conjuntos de datos de manera más rápida que los métodos exactos.

## Ejemplo con FAISS:

```
import faiss
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

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

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Escalar las características
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Crear el índice de FAISS para búsqueda aproximada de vecinos
d = X_train_scaled.shape[1]  # Dimensión del espacio
index = faiss.IndexFlatL2(d)  # Índice para distancia Euclidiana
index.add(X_train_scaled)  # Añadir los datos al índice

# Búsqueda de los k vecinos más cercanos
k = 3
D, I = index.search(X_test_scaled, k)  # D son las distancias, I son los
índices de los vecinos

# Predicción basada en los vecinos más cercanos
y_pred = np.array([np.argmax(np.bincount(y_train[indices])) for indices
in I])

# Calcular la precisión
accuracy = np.mean(y_pred == y_test)
print(f"Precisión con FAISS: {accuracy:.4f}")or_
test_accuracy = best_knn.score(X_test, y_test)
print(f"Precisión en el conjunto de prueba: {test_accuracy:.4f}")
```

## Expliquemos el código:

• **FAISS**: Utiliza FAISS, una biblioteca optimizada para la búsqueda de vecinos más cercanos en grandes conjuntos de datos. En lugar de hacer una búsqueda exhaustiva, FAISS ofrece una búsqueda aproximada que es mucho más rápida y escalable.

• **Búsqueda aproximada**: Aunque FAISS utiliza una aproximación, la precisión sigue siendo alta en muchos casos, con un tiempo de predicción significativamente reducido.

**3. Submuestreo inteligente:**

• **Submuestreo estratificado**: En lugar de utilizar todo el conjunto de datos, se puede realizar un submuestreo inteligente, manteniendo la distribución original de las clases en el conjunto de datos.

• **Condensed Nearest Neighbors (CNN)**: Es un algoritmo que reduce el tamaño del conjunto de entrenamiento al eliminar puntos redundantes sin reducir la precisión del modelo. Esto se logra eliminando puntos de datos que no afectan la clasificación de otros puntos.

**4. Algoritmos de búsqueda de vecinos eficientes:**

• **Ball Tree y KD-Tree**: Estas estructuras de datos aceleran el proceso de búsqueda de vecinos en KNN al organizar los puntos de datos de manera que las búsquedas de proximidad sean más eficientes. Estas estructuras son especialmente útiles en dimensiones más bajas.

**Aquí te dejo un ejemplo con Ball Tree:**

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import BallTree

# Crear un modelo KNN utilizando Ball Tree
knn_ball_tree = KNeighborsClassifier(algorithm='ball_tree',
n_neighbors=3)
knn_ball_tree.fit(X_train_scaled, y_train)

# Hacer predicciones y calcular la precisión
y_pred_ball_tree = knn_ball_tree.predict(X_test_scaled)
accuracy_ball_tree = np.mean(y_pred_ball_tree == y_test)
print(f"Precisión con Ball Tree: {accuracy_ball_tree:.4f}")
```

**Explicación del código:**

• **Ball Tree**: algorithm = 'ball_tree' indica que se utilizará la estructura Ball Tree para acelerar la búsqueda de vecinos en el modelo KNN.

• **Eficiencia**: Aunque Ball Tree es más eficiente que el algoritmo de fuerza bruta en muchos casos, es más efectivo en espacios de baja a media dimensionalidad.

KNN puede ser una herramienta poderosa incluso para grandes conjuntos de datos cuando se aplican las técnicas adecuadas para manejar la complejidad computacional. Desde la reducción de dimensionalidad con PCA hasta el uso de algoritmos aproximados como FAISS o estructuras de datos eficientes como Ball Tree, existen múltiples enfoques para optimizar el rendimiento de KNN en escenarios de big data.

# 12. Interpretación de resultados y evaluación del rendimiento en KNN

Después de entrenar un modelo K-Nearest Neighbors (KNN), es fundamental evaluar su rendimiento para comprender qué tan bien está funcionando y si está listo para ser implementado en un entorno real. La interpretación de los resultados incluye no solo medir la precisión del modelo, sino también analizar cómo y por qué el modelo toma ciertas decisiones.

### Métricas de evaluación para clasificación

Para tareas de clasificación, las métricas de evaluación más comunes son las siguientes:

#### 1. Precisión (Accuracy):

La precisión es la proporción de predicciones correctas realizadas por el modelo en relación con el total de predicciones.

**Fórmula:**

$$\text{Precisión} = \frac{\text{Predicciones Correctas}}{\text{Total de Predicciones}}$$

- **Pros:** Es fácil de interpretar y calcular.
- **Contras:** Puede ser engañosa en conjuntos de datos desbalanceados, donde algunas clases son mucho más frecuentes que otras.

## 2. Matriz de confusión:

La matriz de confusión es una tabla que se utiliza para describir el rendimiento de un modelo de clasificación. Muestra las verdaderas etiquetas frente a las predicciones del modelo.

### Componentes clave:

• **TP (True Positives):** Predicciones correctas para la clase positiva.
• **TN (True Negatives):** Predicciones correctas para la clase negativa.
• **FP (False Positives):** Predicciones incorrectas donde se predice la clase positiva incorrectamente.
• **FN (False Negatives):** Predicciones incorrectas donde se predice la clase negativa incorrectamente.

### Ejemplo en Python:

```python
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Hacer predicciones con el modelo KNN
y_pred = knn.predict(X_test)

# Calcular la matriz de confusión
cm = confusion_matrix(y_test, y_pred)

# Mostrar la matriz de confusión usando Seaborn
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicción')
plt.ylabel('Valor Verdadero')
plt.title('Matriz de Confusión')
plt.show()

# Calcular y mostrar el reporte de clasificación
print(classification_report(y_test, y_pred))
```

**Explicación del ejemplo:**

• **Matriz de confusión:** La matriz de confusión se muestra como un mapa de calor, lo que facilita la identificación de patrones de error específicos en el modelo.

• **Reporte de clasificación:** El reporte de clasificación incluye otras métricas como Precisión (Precision), Recuperación (Recall) y Puntaje F1 (F1 Score) para cada clase.

**3. Precisión, Recall y F1 Score:**

**Precisión (Precision):** La precisión mide cuántas de las predicciones positivas hechas por el modelo son correctas.

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

**Recall:** El recall mide cuántas de las instancias positivas reales fueron capturadas por el modelo.

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

**F1 Score:** El F1 Score es la media armónica entre la precisión y el recall, proporcionando un equilibrio entre ambos.

$$F1 = 2 \times \frac{\text{Precisión} \times \text{Recall}}{\text{Precisión} + \text{Recall}}$$

**Uso:** Estas métricas son especialmente útiles en conjuntos de datos desbalanceados.

### 4. Curva ROC y AUC (Área bajo la curva):

• La curva **ROC (Receiver Operating Characteristic)** es un gráfico que muestra la relación entre el TPR (True Positive Rate) y el FPR (False Positive Rate) para diferentes umbrales de decisión.

• El **AUC (Área bajo la curva)** mide el área total debajo de la curva ROC y proporciona una única métrica de rendimiento del modelo.

**Ejemplo en Python:**

```
from sklearn.metrics import roc_curve, auc

# Calcular las probabilidades de predicción
y_prob = knn.predict_proba(X_test)[:, 1]

# Calcular la curva ROC
fpr, tpr, thresholds = roc_curve(y_test, y_prob, pos_label=1)

# Calcular el AUC
roc_auc = auc(fpr, tpr)

# Graficar la curva ROC
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Curva ROC')
plt.legend(loc='lower right')
plt.show()
```

**Explicación del código:**

• **Curva ROC**: La curva ROC ayuda a visualizar el rendimiento del modelo en diferentes umbrales de clasificación.
• **AUC**: Un valor de AUC más alto indica un mejor rendimiento del modelo.

## Métricas de evaluación para regresión

Para tareas de regresión, donde el objetivo es predecir un valor continuo, se utilizan diferentes métricas:

### 1. Error Cuadrático Medio (MSE):

El MSE mide el promedio de los cuadrados de los errores, es decir, la diferencia entre los valores predichos y los valores reales.

**Fórmula:**

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

**Uso:** Es sensible a grandes errores, ya que los penaliza más fuertemente debido al cuadrado.

### 2. Raíz del Error Cuadrático Medio (RMSE):

La RMSE es la raíz cuadrada del MSE y tiene la misma unidad que la variable objetivo, lo que facilita la interpretación.

**Fórmula:**

$$RMSE = \sqrt{MSE}$$

**3. Error Absoluto Medio (MAE):**
El MAE mide el promedio de los errores absolutos.

**Fórmula:**

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

**Uso:** Proporciona una medida más robusta cuando se desea minimizar la influencia de grandes errores.

**4. Coeficiente de Determinación ($R^2$ Score):**
El $R^2$ mide la proporción de la varianza total en la variable objetivo que es explicada por el modelo.

**Fórmula:**

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}i)^2}{\sum i = 1^n (y_i - \bar{y})^2}$$

**Uso:** Un $R^2$ de 1 indica un modelo perfecto, mientras que un $R^2$ de 0 indica que el modelo no explica ninguna varianza.

**Ejemplo en Python para métricas de regresión:**

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score

# Hacer predicciones con el modelo KNN para regresión
y_pred = knn_regressor.predict(X_test)

# Calcular métricas de evaluación
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Mostrar las métricas
print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAE: {mae:.4f}")
print(f"R^2 Score: {r2:.4f}")
```

**Explicación del código:**

**Métricas clave**: Se calculan el MSE, RMSE, MAE y $R^2$ Score para evaluar el rendimiento del modelo de regresión. Estas métricas ofrecen una visión completa del rendimiento del modelo en términos de precisión y capacidad de explicación.

**Interpretación de la importancia de las características**

Aunque KNN no proporciona directamente un ranking de importancia de las características como lo hacen otros algoritmos (por ejemplo, Random Forest), existen métodos que pueden ser utilizados para interpretar cómo las características afectan las predicciones del modelo KNN:

## Permutación de importancia

Este método mide la importancia de cada característica al evaluar cuánto se deteriora el rendimiento del modelo cuando se permutan los valores de una característica específica.

La idea es que, si la permutación de una característica deteriora significativamente la precisión del modelo, esa característica es importante para el modelo.

### Ejemplo en Python:

```python
from sklearn.inspection import permutation_importance

# Realizar la permutación de importancia
perm_importance = permutation_importance(knn, X_test, y_test,
n_repeats=10, random_state=42)

# Mostrar la importancia de las características
feature_importance = perm_importance.importances_mean
for i, v in enumerate(feature_importance):
    print(f'Característica {i+1}: Importancia {v:.4f}')
```

### Explicación del pequeño código:

• **Permutación de importancia**: permutation_importance evalúa la importancia de cada característica basada en la pérdida de precisión al permutar sus valores. Esto proporciona una forma práctica de interpretar la importancia de las características en un modelo KNN.

La interpretación de los resultados de KNN y la evaluación de su rendimiento son pasos cruciales para asegurar que el modelo esté bien ajustado y sea confiable en aplicaciones reales

# 13. KNN para detección de anomalías

La **detección de anomalías** es un proceso en el que se identifican datos que no siguen el patrón esperado o común dentro de un conjunto de datos. Estos datos anómalos pueden ser errores, fraudes, comportamientos inusuales, o simplemente observaciones raras que no encajan con la mayoría de los datos. KNN es una técnica efectiva para la detección de anomalías debido a su capacidad para medir la proximidad entre puntos de datos.

**¿Cómo funciona KNN en la detección de anomalías?** 🤔

El principio básico detrás del uso de KNN para la detección de anomalías es que las anomalías (o outliers) tienden a estar lejos de la mayoría de los otros puntos de datos en el espacio de características. En otras palabras, un punto de datos se considera una anomalía si sus K vecinos más cercanos están relativamente lejos de él. Esto se puede cuantificar de diferentes maneras:

**1. Distancia al vecino más cercano:**

Un punto de datos se considera una anomalía si la distancia a su vecino más cercano es mayor que un umbral definido. Este es un método simple pero efectivo para detectar outliers aislados.

**2. Promedio de distancias a los K vecinos más cercanos:**

Aquí, calculamos el promedio de las distancias a los K vecinos más cercanos. Si este promedio es significativamente mayor que el promedio general, el punto podría ser una anomalía.

### 3. Número de vecinos dentro de una distancia específica:

Se considera que un punto de datos es una anomalía si tiene menos de un número mínimo de vecinos dentro de una distancia específica.

### Implementación en Python

```python
from sklearn.neighbors import NearestNeighbors
import numpy as np
import matplotlib.pyplot as plt

# Generar un conjunto de datos de ejemplo
np.random.seed(42)
X_inliers = 0.3 * np.random.randn(100, 2)
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
X = np.r_[X_inliers + 2, X_inliers - 2, X_outliers]

# Ajustar el modelo KNN para detección de anomalías
knn = NearestNeighbors(n_neighbors=5)
knn.fit(X)

# Calcular las distancias al 5º vecino más cercano
distances, indices = knn.kneighbors(X)

# Utilizar la distancia al 5º vecino más cercano como puntuación de
anomalía
anomaly_scores = distances[:, 4]

# Determinar un umbral para clasificar como anomalías (por ejemplo, el
percentil 95)
threshold = np.percentile(anomaly_scores, 95)
anomalies = X[anomaly_scores > threshold]

# Graficar los datos y las anomalías detectadas
plt.scatter(X[:, 0], X[:, 1], color='b', label='Datos Normales')
plt.scatter(anomalies[:, 0], anomalies[:, 1], color='r',
label='Anomalías')
plt.legend()
plt.title('Detección de Anomalías con KNN')
plt.show()
```

**Explicación del código:**

**1. Generación de datos**: Se generan datos normales (X_inliers) y anomalías (X_outliers). Estos datos se combinan en un solo conjunto X.

**2. Ajuste del Modelo KNN**: Se utiliza NearestNeighbors de sklearn.neighbors para entrenar un modelo KNN que encuentra los 5 vecinos más cercanos para cada punto en el conjunto de datos.

**3. Cálculo de Puntuaciones de Anomalía**: La distancia al 5º vecino más cercano se utiliza como una medida de cuán "anómalo" es un punto. Cuanto mayor sea la distancia, más probable es que el punto sea una anomalía.

**4. Umbral para Anomalías**: Se establece un umbral para clasificar puntos como anomalías. En este caso, se utiliza el percentil 95 de las puntuaciones de anomalía.

**5. Visualización**: Los puntos de datos se grafican junto con las anomalías detectadas, se deberían resaltar en rojo.

**Ventajas y desventajas del uso de KNN para detección de anomalías**

**Ventajas**:

• **Simplicidad**: KNN es intuitivo y fácil de implementar para la detección de anomalías.

• **No paramétrico**: No asume ninguna distribución específica para los datos, lo que lo hace flexible en una variedad de aplicaciones.

**Desventajas:**

• **Costo computacional**: Como en otros usos de KNN, la detección de anomalías puede ser computacionalmente costosa, especialmente en grandes conjuntos de datos.

• **Sensibilidad a la dimensionalidad**: La maldición de la dimensionalidad también afecta la detección de anomalías, ya que las distancias pueden volverse menos discriminativas en espacios de alta dimensionalidad.

**Aplicaciones comunes**

• **Detección de fraude**: Identificar transacciones que no se alinean con el comportamiento normal de un usuario.

• **Análisis de redes**: Detectar comportamientos inusuales en el tráfico de red que podrían indicar un ataque.

• **Control de calidad**: Identificar productos defectuosos que no cumplen con las especificaciones estándar.

KNN es una herramienta poderosa para la detección de anomalías, capaz de identificar puntos de datos que se desvían significativamente del comportamiento esperado. Aunque tiene algunas limitaciones en términos de escalabilidad y sensibilidad a la dimensionalidad, su flexibilidad y simplicidad lo convierten en una opción popular para este tipo de tareas.

# 14. KNN para clasificación de imágenes

La **clasificación de imágenes** es una tarea de aprendizaje automático en la que se asigna una etiqueta a una imagen basada en su contenido. Aunque los algoritmos de redes neuronales, como las CNNs (Convolutional Neural Networks), son comúnmente utilizados para este tipo de tareas, KNN también puede ser una opción efectiva, especialmente en casos donde el conjunto de datos es pequeño o cuando se busca una solución rápida y sencilla.

### ¿Cómo funciona KNN en la clasificación de imágenes? 🤔

El proceso de clasificación de imágenes con KNN sigue los mismos principios básicos que cualquier otra aplicación de KNN. La diferencia principal es cómo se manejan las imágenes como entradas.

**1. Representación de las imágenes:**

Cada imagen se convierte en un vector de características. Esto puede ser tan sencillito como aplanar la imagen en un vector unidimensional (concatenando las filas o columnas de píxeles) o utilizar descriptores de características más complejos, como SIFT, SURF, o HOG (Histogram of Oriented Gradients).

**2. Cálculo de distancias:**

Una vez que las imágenes están representadas como vectores, KNN calcula la distancia entre la imagen de prueba y todas las imágenes del conjunto de entrenamiento. Las distancias comunes incluyen la distancia Euclidiana o Manhattan.

### 3. Clasificación basada en vecinos:

La imagen de prueba se clasifica en la categoría que es más común entre sus K vecinos más cercanos.

### Ejemplo de implementación en Python con el conjunto de datos MNIST

El conjunto de datos **MNIST** es un famoso conjunto de datos que contiene imágenes de dígitos escritos a mano (0-9).

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt

# Cargar el conjunto de datos MNIST
mnist = fetch_openml('mnist_784')

# Separar las imágenes y las etiquetas
X, y = mnist.data, mnist.target.astype(int)

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

# Crear y entrenar el modelo KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Hacer predicciones en el conjunto de prueba
y_pred = knn.predict(X_test)

# Calcular la precisión del modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Precisión en el conjunto de prueba: {accuracy:.4f}")

# Mostrar algunas imágenes y sus predicciones
for i in range(5):
    plt.imshow(X_test.iloc[i].values.reshape(28, 28), cmap='gray')
    plt.title(f"Etiqueta Verdadera: {y_test.iloc[i]} - Predicción:
{y_pred[i]}")
    plt.show()
```

**Explicación del código:**

**1. Conjunto de Datos MNIST**: Este conjunto de datos contiene imágenes de dígitos escritos a mano de tamaño 28x28 píxeles, aplanadas en vectores de 784 dimensiones (28x28).

**2. Entrenamiento del Modelo**: Se entrena un modelo KNN utilizando las imágenes aplanadas como entradas. El valor de K se establece en 3, lo que significa que la predicción se basa en los 3 vecinos más cercanos.

**3. Evaluación del Modelo**: La precisión del modelo se calcula en el conjunto de prueba, y se visualizan algunas imágenes junto con sus predicciones.

**Ventajas y Desventajas de Usar KNN para Clasificación de Imágenes**

**Ventajas:**

• **Simplicidad**: KNN es fácil de implementar y no requiere una gran cantidad de ajustes de hiperparámetros.

• **No paramétrico**: No hace suposiciones sobre la distribución de los datos, lo que lo hace flexible para diferentes tipos de imágenes.

• **Rápido para prototipado**: Es útil para experimentos rápidos cuando se necesita una solución simple para un problema de clasificación de imágenes.

**Desventajas:**

• **Escalabilidad**: KNN puede ser ineficiente en conjuntos de datos grandes, tanto en términos de tiempo de cómputo como de memoria.

• **Sensibilidad a la dimensionalidad**: La clasificación de imágenes generalmente implica vectores de características de alta dimensionalidad, lo que puede llevar a la maldición de la dimensionalidad.

• **Menor precisión en comparación con modelos de Deep Learning**: KNN no captura patrones complejos tan bien como las CNNs u otros modelos de deep learning, lo que puede resultar en una menor precisión, especialmente en conjuntos de datos complicados.

**Mejoras potenciales en la clasificación de imágenes con KNN**

**Reducción de dimensionalidad:**

Aplicar técnicas como PCA o t-SNE antes de utilizar KNN puede ayudar a reducir la dimensionalidad y mejorar el rendimiento.

**Ingeniería de características:**

En lugar de utilizar píxeles brutos, emplear descriptores de características como HOG, SIFT o SURF puede mejorar significativamente el rendimiento de KNN en tareas de clasificación de imágenes.

**Algoritmos aproximados de KNN:**

En grandes conjuntos de datos, utilizar versiones aproximadas de KNN, como Annoy o FAISS, puede hacer que el proceso de clasificación sea más rápido y escalable.

# 15. Conclusiones y mejores prácticas para el uso de KNN

A lo largo de este documento, hemos platicado en profundidad cómo funciona el algoritmo K-Nearest Neighbors (KNN) y sus aplicaciones en diversas áreas, desde la clasificación básica hasta la detección de anomalías y la clasificación de imágenes. Vamos a resumir los puntos clave y las mejores prácticas para utilizar KNN de manera efectiva en nuestros proyectos.

**Puntos clave**

**1. Simples pero potentes**: KNN es un algoritmo simple y no paramétrico que puede ser utilizado para tareas de clasificación y regresión. Es fácil de implementar y entender, lo que lo convierte en una excelente opción para prototipos rápidos y experimentos.

**2. Importancia de los hiperparámetros**: Elegir el valor adecuado de K es crucial para el rendimiento del modelo. Un K pequeño puede llevar al sobreajuste, mientras que un K grande puede resultar en subajuste. La validación cruzada es una herramienta fundamental para optimizar K y otros hiperparámetros.

**3. Manejo de características categóricas y escalado**: Es esencial convertir características categóricas en un formato adecuado y escalar las características numéricas para asegurar que todas contribuyan equitativamente al cálculo de las distancias.

**4. Maldición de la dimensionalidad**: A medida que aumenta la dimensionalidad de los datos, KNN puede volverse ineficiente y menos preciso. La reducción de dimensionalidad mediante PCA o técnicas similares puede mitigar este problema.

**5. Aplicaciones especializadas:**

• **Detección de anomalías**: KNN es efectivo para detectar outliers mediante el análisis de distancias en un espacio de características.

• **Clasificación de imágenes**: Aunque no es tan potente como los modelos de deep learning, KNN puede ser útil para la clasificación de imágenes en escenarios donde se requieren soluciones rápidas y sencillas.

**6. Desempeño en Grandes Conjuntos de Datos**: KNN puede ser costoso en términos computacionales cuando se trabaja con grandes volúmenes de datos. Utilizar técnicas como algoritmos aproximados (FAISS, Annoy) y estructuras de datos eficientes (Ball Tree, KD-Tree) puede mejorar significativamente la escalabilidad.

**Mejores Prácticas**

**Escalado de características**: Siempre escala tus datos antes de aplicar KNN, ya sea mediante normalización o estandarización, para evitar que algunas características dominen otras en el cálculo de distancias.

**Reducción de dimensionalidad**: Considera aplicar técnicas de reducción de dimensionalidad, como PCA, especialmente cuando trabajas con datos de alta dimensionalidad. Esto no solo mejorará la eficiencia, sino que también puede mejorar la precisión del modelo.

**Validación cruzada**: Utiliza validación cruzada para seleccionar el mejor valor de K y otros hiperparámetros. Esto te ayudará a construir un modelo que generalice bien en datos no vistos.

**Ingeniería de características**: Para la clasificación de imágenes o cualquier otra tarea compleja, considera utilizar descriptores de características avanzados en lugar de características simples, como píxeles crudos.

**Uso de algoritmos aproximados**: Si estás trabajando con grandes conjuntos de datos, considera utilizar versiones aproximadas de KNN para reducir el tiempo de cómputo sin sacrificar demasiado la precisión.

**Interpretación de resultados**: No te limites a la precisión como única métrica de evaluación. Utiliza otras métricas relevantes para tu tarea específica, como el F1 Score en clasificación de clases desbalanceadas, o el $R^2$ en tareas de regresión.

**Para terminar…**

KNN es un algoritmo que me ha ayudado muchísimo y que, con el ajuste y preprocesamiento adecuados, puede ser extremadamente útil en una variedad de aplicaciones. Aunque es simple en su esencia, la correcta implementación y optimización pueden convertirlo en una herramienta poderosa en el inventario de cualquier científico de datos o ingeniero de ML.

Espero de todo corazón, que este documento te haya proporcionado una comprensión profunda y clara de cómo utilizar KNN en tus proyectos. Si tienes alguna otra pregunta o necesitas más detalles sobre un tema específico, estaré encantado de ayudarte, solo vista mi sitio.

¡Mucho éxito! :)