Sistemas Inteligentes

Alumno: Santiago Vietto

Docente: Mariano Aliaga

DNI: 42654882

Institución: UCC

Año: 2021

Proyecto de Machine Learning de principio a fin

Introducción

_ A continuación, listamos los pasos para llevar a cabo un proyecto de machine learning:

1. Mirar el panorama general.
2. Conseguir los datos.
3. Descubrir y visualizar los datos para ganar entendimiento.
4. Preparar los datos para los algoritmos de Machine Learning.
5. Seleccionar el modelo y entrenarlo.
6. Ajuste del modelo.
7. Presentar la solución.
8. Lanzarla, monitorearla y mantenerla.

Panorama general

Mirar el panorama general

_ Mirar el panorama general o "frame the problem" se refiere al proceso de definir claramente el problema que se está intentando resolver con el modelo de aprendizaje automático. En esta etapa, es importante comprender el problema y sus objetivos, identificar las limitaciones y restricciones relevantes, y determinar qué tipo de modelo de aprendizaje automático es apropiado para abordar el problema. Algunas preguntas que se pueden hacer al "enmarcar" un problema de aprendizaje automático incluyen:

- ¿Cuál es el objetivo final del modelo? ¿Qué tipo de resultado se espera?
- ¿Qué tipo de datos estarán disponibles para el modelo? ¿Son datos etiquetados o no etiquetados? ¿Qué tan limpios y estructurados están los datos?
- ¿Qué limitaciones o restricciones existen para el modelo? ¿Hay limitaciones de tiempo, recursos o presupuesto?
- ¿Cómo la compañía espera usar y beneficiarse de nuestro modelo?
- ¿Qué tipo de modelo de aprendizaje automático es más apropiado para abordar el problema? ¿Se requiere un modelo de clasificación, regresión o agrupamiento?

_ Al enmarcar el problema de esta manera, se puede ayudar a definir claramente los objetivos y limitaciones del modelo, lo que a su vez puede ayudar a guiar el proceso de construcción y ajuste del modelo. Además, una definición clara del problema puede ayudar a garantizar que el modelo se diseñe de manera efectiva y que sea útil en la resolución del problema en cuestión.

Aprendizaje automático: también conocido como Machine Learning, es una rama de la inteligencia artificial que se enfoca en el diseño y desarrollo de algoritmos y modelos estadísticos que permiten a los ordenadores aprender a partir de datos, sin necesidad de ser programados de manera explícita. En otras palabras, se trata de enseñar a las máquinas a aprender por sí mismas a partir de los datos y a hacer predicciones o tomar decisiones basadas en esos datos. El aprendizaje automático tiene una amplia variedad de aplicaciones, como la detección de fraudes, la clasificación de imágenes, la predicción del comportamiento del usuario, entre muchas otras.

Pipeline: se refiere a la técnica de encadenamiento de múltiples etapas de procesamiento de datos en un flujo de trabajo único y automatizado. En el contexto del aprendizaje automático, un pipeline es un conjunto de pasos que se ejecutan en secuencia para transformar y preparar los datos antes de aplicar un modelo de aprendizaje automático. Un pipeline típico consta de tres etapas principales, la preparación de datos, la selección de características y la construcción del modelo. Cada una de estas etapas puede estar compuesta por varias transformaciones de datos diferentes. Por ejemplo, en la etapa de preparación de datos, se pueden realizar transformaciones como la normalización, la eliminación de valores faltantes y la codificación de características categóricas. En la etapa de selección de características, se pueden aplicar técnicas como la selección basada en modelos o la selección basada en la correlación. Finalmente, en la etapa de construcción del modelo, se pueden ajustar y evaluar diferentes modelos de aprendizaje automático.

_ La ventaja de utilizar un pipeline es que automatiza el flujo de trabajo de preparación de datos y construcción del modelo, lo que puede ahorrar tiempo y reducir errores. Además, un pipeline bien diseñado permite realizar cambios y ajustes de forma fácil y rápida, lo que facilita la experimentación con diferentes configuraciones de modelo y preparación de datos. En resumen, un pipeline es una técnica importante en el aprendizaje automático que permite automatizar y simplificar el proceso de preparación de datos y construcción del modelo. Al encadenar varias etapas de procesamiento de datos en un flujo de trabajo único, se puede reducir el tiempo de desarrollo y minimizar los errores, lo que puede resultar en modelos de aprendizaje automático más precisos y eficientes.

Selección de medida desempeño

_ Seleccionar una medida de rendimiento o “Select a Performance Measure” se refiere a la elección de una métrica para evaluar el rendimiento de un modelo de aprendizaje automático. En otras palabras, es necesario definir una medida que permita medir la calidad del modelo y comparar diferentes modelos para determinar cuál es el mejor.

_ La elección de una medida de rendimiento adecuada depende del tipo de problema que se esté abordando. Por ejemplo:

Problema de clasificación: es un tipo de problema en el que se intenta predecir una etiqueta discreta (clase) para una determinada entrada. Por ejemplo, predecir si un correo electrónico es spam o no spam, o si una imagen es un gato o un perro. En este caso se pueden utilizar medidas como:

- Precisión: que mide la proporción de muestras clasificadas correctamente.
- Sensibilidad: que mide la proporción de muestras positivas clasificadas correctamente.
- Especificidad: que mide la proporción de muestras negativas clasificadas correctamente.

Problema de regresión: se refiere a un tipo de problema en el que se intenta predecir un valor numérico para una determinada entrada. Por ejemplo, predecir el precio de una casa en función de sus características, o predecir la cantidad de ventas de un producto en función de sus características. En este caso, una medida común es el error cuadrático medio (MSE), que mide la distancia media cuadrada entre los valores predichos y los valores reales. Otras medidas de rendimiento comunes en la regresión son el coeficiente de determinación (R2), que mide la proporción de la variabilidad en la variable dependiente que se explica por el modelo, y el error absoluto medio (MAE), que mide la distancia media absoluta entre los valores predichos y los valores reales.

- Raíz del error cuadrático medio (RMSE): mide la raíz cuadrada de la media de los errores cuadrados entre las predicciones del modelo y los valores reales. Es una medida de la distancia entre los valores reales y los valores predichos del modelo, y se expresa en las mismas unidades que la variable de destino. La fórmula del RMSE es la siguiente:

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

  En donde:
  - h(x): son las predicciones del modelo.
  - y: son los valores reales de la variable de destino.
  - m: es el número total de muestras en el conjunto de datos.
- Error Absoluto medio (MAE): mide la media de la distancia absoluta entre las predicciones del modelo y los valores reales. Es una medida de la magnitud de los errores del modelo, y se expresa en las mismas unidades que la variable de destino. La fórmula del MAE es la siguiente:

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

- En donde:
  - h(x): son las predicciones del modelo.
  - y: son los valores reales de la variable de destino.
  - m: es el número total de muestras en el conjunto de datos.

_ En general, RMSE y MAE son medidas de rendimiento similares, pero RMSE tiende a ser más sensible a valores atípicos en los datos que MAE, ya que eleva los errores al cuadrado antes de calcular la media. Por lo tanto, RMSE puede ser una mejor medida de rendimiento si se desea penalizar más los errores grandes en el modelo.

_ La elección de una medida de rendimiento adecuada es importante porque puede influir en la elección del modelo y en la forma en que se ajusta. Por ejemplo, si se utiliza una medida de rendimiento que no es adecuada para el problema en cuestión, es posible que el modelo no se ajuste correctamente o que se seleccionen parámetros subóptimos. Por lo tanto, es importante elegir una medida de rendimiento adecuada que refleje los objetivos y requisitos del problema. En resumen, seleccionar una medida de rendimiento adecuada es fundamental para evaluar la calidad de un modelo de aprendizaje automático y comparar diferentes modelos. La elección de una medida de rendimiento adecuada depende del tipo de problema que se esté abordando, y es importante elegir una medida de rendimiento que refleje los objetivos y requisitos del problema.

Chequeo de suposiciones

_ Cuando se utiliza un modelo de regresión, se asumen ciertas condiciones y suposiciones sobre los datos, como la linealidad, la independencia, la homocedasticidad y la normalidad de los residuos. Es importante verificar si estas suposiciones se cumplen antes de confiar en las predicciones del modelo. Para verificar estas suposiciones, se pueden realizar diversas técnicas de análisis y visualización de datos. Algunas de ellas incluyen:

- Diagrama de dispersión: para evaluar la relación entre las variables independientes y la variable dependiente y detectar cualquier patrón no lineal en los datos.
- Gráficos de residuos: para evaluar la distribución de los residuos y detectar cualquier patrón en los errores del modelo, como la heterocedasticidad o la presencia de valores atípicos.
- Pruebas estadísticas: para evaluar la normalidad y la independencia de los residuos utilizando pruebas como la prueba de normalidad de Shapiro-Wilk o la prueba de autocorrelación de Durbin-Watson.
- Transformaciones de datos: para transformar las variables independientes y/o la variable dependiente si no cumplen con las suposiciones requeridas por el modelo, como la transformación logarítmica o la transformación de Box-Cox.

_ Es importante tener en cuenta que las suposiciones subyacentes en un modelo de regresión pueden variar según el tipo de modelo y los datos utilizados. Por lo tanto, es fundamental realizar un análisis exhaustivo de los datos antes de aplicar un modelo de regresión y verificar que se cumplan todas las suposiciones necesarias para obtener predicciones precisas y confiables antes de empezar para poder evitar posibles inconvenientes más adelante. Es necesario estar en contacto con el resto de los equipos para dispersar este tipo de dudas.

Obtener los datos

_ Antes de construir un modelo de aprendizaje automático, es fundamental tener acceso a datos de calidad que sean relevantes para el problema que se está tratando de resolver. La calidad de los datos puede afectar significativamente la precisión y el rendimiento del modelo, por lo que es importante asegurarse de que los datos estén limpios, sean completos y estén bien estructurados. Para obtener los datos, es posible que necesite considerar las opciones que van a ser nombradas a continuación.

_ Una vez que se tenga los datos, es importante preprocesarlos para asegurarse de que estén limpios y estructurados adecuadamente. Algunas técnicas comunes de preprocesamiento de datos incluyen la eliminación de valores faltantes, la normalización de las características y la codificación de variables categóricas. Es importante tener en cuenta que el proceso de obtener y preparar los datos puede ser uno de los aspectos más desafiantes y que requiere más tiempo del aprendizaje automático. Sin embargo, es un paso crucial para garantizar que su modelo sea preciso y útil.

Descargar los datos

_ Descargar datos es una tarea crucial en el aprendizaje automático, ya que los modelos de aprendizaje automático se entrenan en datos y, por lo tanto, la calidad de los resultados del modelo depende en gran medida de la calidad de los datos de entrada. Los datos pueden estar disponibles en diferentes formatos, como CSV, JSON, TXT, XML, etc. Además, los datos pueden estar almacenados en diferentes ubicaciones, como en servidores web, bases de datos, sistemas de almacenamiento en la nube, etc. En Python, hay varias bibliotecas que se pueden utilizar para descargar y leer datos, como urllib, requests, wget, entre otras. Es importante tener en cuenta que la descarga de datos de fuentes externas puede presentar algunos desafíos, como la falta de control sobre la calidad de los datos, la presencia de datos faltantes o valores atípicos, y la necesidad de asegurarse de que los datos sean legalmente accesibles.

Conjunto de datos de entrenamiento: es un subconjunto de los datos totales u originales disponibles, que se utiliza para entrenar un modelo de aprendizaje automático. Este conjunto de datos se utiliza para ajustar los parámetros del modelo de manera que pueda hacer predicciones precisas sobre datos no vistos. En general, el conjunto de datos de entrenamiento se divide en dos partes: los datos de entrada (características) y los datos de salida (etiquetas) que el modelo intentará predecir.

_ A continuación, se muestra un ejemplo de cómo descargar conjunto de datos, mediante una función llamada fetch_housing_data() que descarga un archivo tar.gz que contiene un conjunto de datos de viviendas de California y lo extrae en un directorio local utilizando la biblioteca “tarfile”:

```
import os
import tarfile
import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-
ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

- El primer bloque de código define tres constantes, una es DOWNLOAD_ROOT, que es la URL base de donde se descargarán los datos; otra es HOUSING_PATH, que es la ubicación en la que se almacenarán los datos descargados; y por último HOUSING_URL, que es la URL específica del archivo comprimido que contiene los datos.
- La función fetch_housing_data() comienza creando el directorio HOUSING_PATH si aún no existe utilizando la función os.makedirs(). La opción exist_ok=True indica que no se debe producir un error si el directorio ya existe.
- A continuación, la función utiliza la función urllib.request.urlretrieve() para descargar el archivo comprimido de viviendas de California en la ubicación especificada por HOUSING_URL y lo almacena localmente en la ubicación especificada por tgz_path.
- Después de descargar el archivo, la función abre el archivo tar.gz utilizando la función tarfile.open() y extrae todos los archivos contenidos en él en el directorio especificado por HOUSING_PATH utilizando la función housing_tgz.extractall(). Finalmente, se cierra el archivo tar.gz utilizando housing_tgz.close().

_ Una vez que se han descargado los datos, es importante explorarlos y analizarlos para comprender mejor su estructura y características. En Python, la biblioteca pandas se utiliza a menudo para la exploración de datos y proporciona una serie de funciones para analizar y manipular datos. A continuación vemos como cargar los datos anteriores:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

- El código que se muestra utiliza la biblioteca Pandas y define una función llamada load_housing_data() que carga los datos de viviendas de California desde un archivo CSV en una estructura de datos llamada DataFrame.
- La función load_housing_data() toma una ruta housing_path como argumento, que es la ruta al directorio que contiene el archivo CSV. Si no se especifica una ruta, se utiliza la constante HOUSING_PATH definida previamente en el código.
- Luego, la función utiliza la función os.path.join() para unir el directorio housing_path con el nombre del archivo CSV ("housing.csv"), creando así la ruta completa del archivo CSV.
- Finalmente, la función carga los datos desde el archivo CSV utilizando la función pd.read_csv() de Pandas y devuelve el DataFrame resultante.

_ Una vez que se hayan cargado los datos en un DataFrame de Pandas, se pueden ver los datos utilizando las funciones y métodos proporcionados por la biblioteca Pandas. Algunas son:

- Método head(): para ver las primeras filas del DataFrame.
- Método info(): para obtener información sobre el DataFrame, como el número de filas y columnas, el tipo de datos de cada columna, y si hay valores faltantes.
- Método describe(): para obtener estadísticas básicas sobre las columnas numéricas del DataFrame.

Creación un dataset de evaluación los datos

_ Es importante dividir los datos en un conjunto de entrenamiento y un conjunto de prueba para evaluar el rendimiento del modelo de aprendizaje automático. Si no se realiza esta división, el modelo puede ajustarse demasiado a los datos de entrenamiento y no generalizar bien a nuevos datos.

Conjunto de prueba: (también conocido como conjunto de validación) se refiere a una parte del conjunto de datos que se separa del conjunto de entrenamiento y se utiliza para evaluar el rendimiento del modelo después de que se haya entrenado con el conjunto de entrenamiento. Es importante que el conjunto de prueba sea independiente del conjunto de entrenamiento para que el modelo no haya visto los datos de prueba antes de su evaluación y la evaluación sea una verdadera medida de su capacidad para generalizar a nuevos datos. En general, se utiliza una proporción del 20-30% del conjunto de datos para formar el conjunto de prueba, dependiendo del tamaño del conjunto de datos total.

_ Para crear el conjunto de prueba, se puede utilizar una función de división proporcionada por “scikit-learn”, una biblioteca de aprendizaje automático en Python. Por ejemplo, la función train_test_split() divide aleatoriamente los datos en un conjunto de entrenamiento y un conjunto de prueba. También se puede establecer la proporción de datos que se desean en cada conjunto. Por defecto, el 25% de los datos se asignan al conjunto de prueba.

_ Es importante tener en cuenta que la división aleatoria puede resultar en diferentes conjuntos de entrenamiento y prueba cada vez que se ejecute el código. Para evitar esto, se puede fijar la semilla aleatoria utilizando el parámetro random_state.

_ Una vez que se tiene el conjunto de entrenamiento y el conjunto de prueba, es importante no mirar el conjunto de prueba hasta que se haya ajustado el modelo completamente en el conjunto de entrenamiento. De esta manera, se puede evaluar el rendimiento del modelo en datos nuevos y desconocidos.

_ A continuación, dividimos el conjunto de datos:

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2,
random_state=42)
```

- Este código utiliza la función train_test_split de la biblioteca scikit-learn para dividir el conjunto de datos housing en dos conjuntos, uno para entrenamiento (train_set) y otro para prueba (test_set).
- La función train_test_split toma tres argumentos principales, el primer argumento (housing en este caso) es el conjunto de datos que se desea dividir. El segundo argumento (test_size=0.2) indica el tamaño del conjunto de prueba, que se establece en un 20% del tamaño total del conjunto de datos. Esto significa que el 80% restante se utilizará para el conjunto de entrenamiento. Y el tercer argumento (random_state=42) establece una semilla aleatoria para asegurar que los mismos puntos de datos se asignen al conjunto de entrenamiento y al conjunto de prueba cada vez que se ejecute el código. Esto ayuda a mantener los resultados reproducibles.
- El resultado de la función train_test_split son dos conjuntos de datos, train_set y test_set. En este caso, el 80% del conjunto de datos original se asignará a train_set, mientras que el 20% restante se asignará a test_set. Podremos ver la cantidad de datos que se han asignado a cada conjunto de entrenamiento y prueba.
- Se coloca 42 porque es un valor comúnmente utilizado en la comunidad de aprendizaje automático como una forma de obtener resultados reproducibles. Al fijar random_state a un número específico, garantizamos que cada vez que se ejecute el código, se generará la misma secuencia de números aleatorios, lo que produce resultados consistentes. Pero en realidad podría haber sido cualquier otro número. La elección del número en sí no es importante, lo que importa es que sea un número fijo y que sea utilizado consistentemente en todo el código.

_ Creamos una nueva variable categórica en el DataFrame, para luego utilizarla mas tarde para dividir aleatoriamente los datos en conjuntos de entrenamiento y prueba estratificados por ingresos.:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

- Este código crea una nueva columna llamada "income_cat" en el DataFrame "housing". Esta nueva columna se crea utilizando la función "pd.cut()" de la biblioteca pandas.
- La función "pd.cut()" se utiliza para segmentar los valores en diferentes rangos. En este caso, se segmenta el valor de la columna "median_income" en diferentes categorías utilizando los siguientes argumentos:
  - "bins" especifica los límites de los diferentes intervalos de segmentación.
  - "labels" especifica los nombres de las categorías resultantes.
- En este caso, los valores de "median_income" se dividen en los siguientes cinco intervalos:
  - De 0 a 1.5
  - De 1.5 a 3.0
  - De 3.0 a 4.5
  - De 4.5 a 6.0
  - Más de 6.0
- Cada uno de estos intervalos se etiqueta con un número del 1 al 5, correspondiente a su posición en la lista "labels".
- La nueva columna "income_cat" se agrega al DataFrame "housing" y se utiliza para dividir el conjunto de datos en conjuntos de entrenamiento y prueba estratificados.

_ Los datos de entrenamiento y prueba son dos conjuntos de datos utilizados en el aprendizaje automático para evaluar y ajustar un modelo.

- Datos de entrenamiento: se utilizan para ajustar los parámetros del modelo.
- Datos de prueba: se utilizan para evaluar la capacidad de generalización del modelo, es decir, su capacidad para predecir de manera efectiva datos que nunca ha visto antes.

_ Por lo general, se divide el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba en una proporción determinada, como 80-20 o 70-30. Los datos de entrenamiento se utilizan para entrenar el modelo y los datos de prueba se utilizan para evaluar su desempeño.

_ A continuación, dividimos los datos en un conjunto de prueba y entrenamiento:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

- Este código utiliza la clase StratifiedShuffleSplit de Scikit-Learn para dividir los datos en un conjunto de entrenamiento y un conjunto de prueba estratificados.
- En la primera línea, se importa la clase StratifiedShuffleSplit de la biblioteca sklearn.model_selection. Luego, se crea una instancia de StratifiedShuffleSplit con tres argumentos, uno es n_splits que indica el número de iteraciones de división, el otro test_size que indica el tamaño del conjunto de prueba en relación al conjunto completo, y por último random_state que establece la semilla aleatoria para que la salida sea reproducible.
- En el siguiente bloque de código, se utiliza un bucle for para iterar sobre las divisiones generadas por StratifiedShuffleSplit. En cada iteración, se obtienen los índices de entrenamiento y prueba correspondientes y se asignan a strat_train_set y strat_test_set, respectivamente, utilizando el método loc de pandas.
- La división estratificada se realiza en base a la variable income_cat que se creó previamente, lo que asegura que la distribución de los datos en la variable income_cat sea similar en los conjuntos de entrenamiento y prueba.
- En resumen, este código realiza una división estratificada de los datos en conjuntos de entrenamiento y prueba para utilizarlos en el modelo de aprendizaje automático.

_ Con el siguiente comando:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

- Este código recorre dos conjuntos de datos, strat_train_set y strat_test_set, y elimina la columna "income_cat" de cada uno de ellos utilizando el método drop() del objeto DataFrame de Pandas.
- El parámetro axis=1 se utiliza para especificar que se desea eliminar una columna en lugar de una fila, y inplace=True se utiliza para aplicar los cambios directamente en los objetos DataFrame en lugar de crear nuevos objetos.

Obtener los datos

_ Lo primero que hacemos antes de continuar es ejecutar los siguiente:

```
housing = strat_train_set.copy()
```

- Este código crea una copia del conjunto de datos de entrenamiento strat_train_set y lo almacena en la variable housing. La función copy() se utiliza para evitar que los cambios realizados en housing afecten al conjunto de datos original strat_train_set. Al hacer esto, podemos realizar modificaciones en housing sin afectar los datos originales y preservar la integridad del conjunto de entrenamiento original.

Descubrir y visualizar los datos para ganar entendimiento

_ Esta sección hace referencia a la exploración de los datos mediante técnicas de visualización para obtener una comprensión más profunda de las relaciones entre las diferentes variables y cómo afectan a la variable objetivo. Se utilizan diversas técnicas de visualización, como gráficos de dispersión, histogramas, gráficos de correlación y mapas de calor para identificar patrones, tendencias y relaciones entre variables. Esto puede ayudar a los científicos de datos y a los analistas a identificar características importantes de los datos, descubrir variables que tengan una fuerte relación con la variable objetivo y comprender mejor los factores que pueden afectar los resultados del modelo. Además, se pueden utilizar técnicas de preprocesamiento de datos, como el tratamiento de valores atípicos, la normalización y la imputación de valores faltantes para preparar los datos antes de aplicar modelos de aprendizaje automático.

Visualización de datos geográficos

_ Visualizar datos geográficos es una tarea común en el análisis de datos. Continuando con el ejemplo de las viviendas, en primer lugar, se utiliza el paquete Matplotlib para crear un gráfico de dispersión de las coordenadas geográficas de los datos de vivienda. Luego, se usa el mapa base de California para visualizar las áreas con alta densidad de población de acuerdo con el tamaño de los círculos en el gráfico de dispersión. Además, se puede utilizar el paquete Folium para crear mapas interactivos que permiten la exploración de datos geográficos. En el conjunto de datos de vivienda de California, se puede utilizar Folium para crear un mapa interactivo de California que muestra la ubicación de las viviendas.

_ La visualización de datos geográficos puede proporcionar información importante sobre patrones y tendencias geográficas en los datos, lo que puede ser útil para la toma de decisiones en el análisis de datos. A continuación vemos algunos ejemplos:

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

- Este código es una forma de visualizar datos geográficos en un gráfico de dispersión. En particular, está graficando la longitud de la ubicación de cada punto en el eje x y la latitud en el eje y. Cada punto en el gráfico representa una ubicación en el conjunto de datos de viviendas.
- El parámetro kind="scatter" especifica el tipo de gráfico que se debe crear. En este caso, es un gráfico de dispersión. Los parámetros x="longitude" e y="latitude" especifican qué variables deben representarse en los ejes x e y, respectivamente.
- Este gráfico puede ser útil para visualizar la distribución geográfica de las viviendas y detectar patrones o agrupaciones. Por ejemplo, es posible que se observe que las viviendas más caras se encuentran cerca de la costa o en ciertas áreas urbanas.

_ Una variación del anterior seria:

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

- Este es igual al anterior solo que “alpha” define la transparencia de los puntos. En este caso, alpha=0.1 significa que los puntos serán muy transparentes, lo que permite ver mejor la densidad de los datos en áreas con mayor concentración.

_ Una mejor variación seria:

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

- Este código representa una gráfica de dispersión (scatter plot) de la longitud (longitude) y la latitud (latitude) de los datos de vivienda (housing), con algunos parámetros adicionales para visualizar mejor la información.
  - kind="scatter": especifica que se desea un gráfico de dispersión.
  - x="longitude", y="latitude": especifica los ejes X e Y de la gráfica.
  - alpha=0.4: especifica la transparencia de los puntos en la gráfica.
  - s=housing["population"]/100: especifica el tamaño de los puntos en la gráfica, que se basa en la población (population) dividida por 100.
  - label="population": agrega una etiqueta a la leyenda de la gráfica.
  - figsize=(10,7): especifica el tamaño de la figura que se va a crear.
  - c="median_house_value": especifica que se desea utilizar la columna de valor mediano de vivienda (median_house_value) para determinar el color de los puntos.
  - cmap=plt.get_cmap("jet"): especifica una paleta de colores (jet) que se utilizará para asignar un color a cada punto, en función de su valor de median_house_value.
  - colorbar=True: muestra una barra de colores que indica la relación entre los colores y los valores.
  - plt.legend(): agrega una leyenda a la gráfica para explicar los diferentes elementos de la misma.

Buscando correlaciones

_ Buscar correlaciones en los datos, se refiere a la exploración de la relación entre diferentes características en los mismos, y cómo estas relaciones pueden ayudar a entender mejor el conjunto de datos y cómo afectan al objetivo del análisis. Para buscar correlaciones, se suelen utilizar gráficos de dispersión o matrices de correlación para visualizar las relaciones entre diferentes variables. También se pueden utilizar técnicas estadísticas como la correlación de Pearson o el coeficiente de correlación de Spearman para medir la fuerza de las relaciones entre variables.

_ El objetivo de buscar correlaciones es identificar qué variables pueden estar más relacionadas con el objetivo del análisis, lo que puede ayudar a elegir las características más importantes para un modelo de machine learning o para la toma de decisiones basada en los datos.

Correlación de Spearman: es una medida estadística no paramétrica que evalúa la relación monótona entre dos variables, es decir, no asume una distribución específica de los datos. A diferencia del coeficiente de Pearson, no mide una relación lineal entre variables, sino una relación de rango. Puede tener valores entre -1 y 1, de manera similar al coeficiente de Pearson. Sin embargo, a diferencia del coeficiente de Pearson, el coeficiente de Spearman se utiliza mejor para evaluar la relación entre dos variables que no tienen una relación lineal. También es menos sensible a los valores atípicos que el coeficiente de Pearson.

Correlación lineal o coeficiente de Pearson: es una medida estadística que evalúa la relación lineal entre dos variables continuas. El coeficiente de correlación puede tener valores que oscilan entre -1 y 1. Cuando el valor es cercano a 1, indica una correlación positiva fuerte, lo que significa que cuando una variable aumenta, la otra también lo hace. Por otro lado, cuando el valor es cercano a -1, indica una correlación negativa fuerte, lo que significa que cuando una variable aumenta, la otra disminuye. Si el valor es cercano a 0, no hay una correlación lineal aparente entre las dos variables.

_ A continuación vemos un ejemplo de su ejecución, en donde primero corremos:

```
corr_matrix = housing.corr()
```

- Este código calcula la matriz de correlación entre todas las columnas numéricas en el dataframe "housing". La función ".corr()" es un método que se aplica a un dataframe en Pandas y devuelve una matriz de correlación de Pearson que muestra la relación lineal entre cada par de variables. El resultado se almacena en la variable "corr_matrix".
- Al ejecutarlo surge un error que se debe a que la columna "ocean_proximity" en el dataframe housing tiene valores de tipo string que no pueden ser convertidos directamente a tipo float. Una forma de resolver esto es eliminar esa columna del dataframe antes de convertirlo a tipo float.

- De esta forma, se elimina la columna que causa el problema y se convierten las demás columnas a tipo float para poder calcular la matriz de correlación.

Matriz de correlación: es una matriz cuadrada que muestra el coeficiente de correlación entre cada par de variables en un conjunto de datos, y es útil para detectar patrones y relaciones entre variables en un conjunto de datos

- Coeficientes de correlación: miden la fuerza y la dirección de la relación lineal entre dos variables.

_Una vez creada la matriz, ejecutamos:

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

- Este código se utiliza para calcular la correlación de la variable objetivo "median_house_value" con todas las demás variables y ordenar los resultados en orden descendente.
- Primero, corr_matrix["median_house_value"] selecciona la columna "median_house_value" de la matriz de correlación corr_matrix. Luego, sort_values(ascending=False) ordena los valores de correlación de mayor a menor.
- Esto significa que la variable más correlacionada con "median_house_value" aparecerá en la parte superior de la lista, seguida de las variables menos correlacionadas. Esto puede ayudarnos a identificar las variables que tienen una fuerte relación con la variable objetivo y, por lo tanto, pueden ser útiles para predecir los valores de la variable objetivo.

_ Seguimos procesando el resultado anterior:

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

- El código anterior importa la función scatter_matrix desde el paquete pandas.plotting, la cual es utilizada para crear una matriz de dispersión entre varias variables.
- A continuación, se define una lista attributes con los nombres de cuatro variables (columnas) que se desean comparar: median_house_value, median_income, total_rooms y housing_median_age.
- Luego se utiliza la función scatter_matrix para crear una matriz de dispersión que muestra la relación entre estas cuatro variables. El argumento housing[attributes] indica que se deben tomar solo estas cuatro columnas del dataframe housing.
- El argumento figsize=(12, 8) define el tamaño de la figura que se va a generar. En este caso, se define una figura de 12 pulgadas de ancho y 8 pulgadas de alto.

_ Para ver un gráfico en particular ejecutamos:

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

- El código muestra una gráfica de dispersión (scatter plot) utilizando los datos de un DataFrame llamado "housing". La gráfica se crea utilizando el método "plot" de Pandas, y se especifica el tipo de gráfica como "scatter".
- Los valores de la columna "median_income" se utilizan como el eje x (horizontal), mientras que los valores de la columna "median_house_value" se utilizan como el eje y (vertical).
- El parámetro "alpha" se establece en 0.1, lo que significa que los puntos en la gráfica se mostrarán con un grado de transparencia del 10%, lo que permite ver la densidad de puntos en las áreas más pobladas.
- En resumen, la gráfica muestra la relación entre el ingreso medio y el valor medio de las casas en un área geográfica determinada, lo que puede ayudar a identificar patrones o tendencias en los datos.

Experimentando con combinaciones de atributos

_ Esta sección se enfoca en la creación de nuevos atributos a partir de la combinación de dos o más atributos existentes. El objetivo es descubrir patrones o relaciones que no son obvias a simple vista. A continuación, se prueban diferentes combinaciones de atributos para ver cuál de ellas tiene una fuerte correlación con la variable objetivo, en este caso, el valor medio de las viviendas, según como se viene trabajando. Por ejemplo, se pueden crear nuevas variables como la cantidad de habitaciones por hogar o la cantidad de habitaciones por población. Luego se evalúa la correlación de estas nuevas variables con la variable objetivo y se determina si tienen una relación significativa. La experimentación con combinaciones de atributos es un proceso iterativo que puede llevar a la creación de nuevas variables predictivas y, en última instancia, a modelos más precisos y efectivos.

_ A continuación, tenemos un ejemplo:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] =
housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households
"]
```

- Este código agrega tres nuevas columnas al conjunto de datos housing utilizando algunas combinaciones de atributos ya existentes:
  - rooms_per_household: esta columna calcula la cantidad promedio de habitaciones por hogar al dividir el número total de habitaciones en una casa entre el número de hogares.

    - bedrooms_per_room: esta columna calcula la relación entre el número total de dormitorios y el número total de habitaciones en una casa. Esta relación puede ser un indicador de la densidad de población en un área determinada.
    - population_per_household: esta columna calcula la cantidad promedio de personas por hogar al dividir la población total por el número de hogares.
- El objetivo de crear estas nuevas características es simplificar el conjunto de datos y obtener una mejor comprensión de la información que contienen.

_ Visualizamos la matriz de correlación nuevamente:

```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

- Este código está calculando la matriz de correlación entre todas las columnas del conjunto de datos housing usando el método corr() de Pandas, y luego muestra los valores de correlación de la columna "median_house_value" en orden descendente utilizando el método sort_values().
- La segunda línea de código muestra las correlaciones de todas las demás columnas con respecto a la columna "median_house_value" de manera ordenada, de mayor a menor.
- Esto puede ayudar a identificar las características que tienen una correlación más fuerte con la variable objetivo y, por lo tanto, pueden ser más importantes para la predicción del precio de la vivienda.

Preparar los datos para algoritmos de Machine Learning

_ Esta sección se enfoca en la preparación de los datos para los algoritmos de aprendizaje automático. Esta es una de las etapas más importantes en el proceso de aprendizaje automático y puede afectar significativamente el rendimiento del modelo. Se deben realizar diversas tareas, tales como la limpieza de los datos, la manipulación y transformación de las variables, la selección de características relevantes y la división de los datos en conjuntos de entrenamiento y prueba. En vez de preparar los manualmente es mejor hacerlo en funciones por lo siguiente:

- Permite reproducir las transformaciones fácilmente en cualquier conjunto de datos.
- Permite construir gradualmente una librería de transformación de funciones qué puedes utilizar en otros proyectos de machine learning.
- Permite utilizar esas funciones una vez deployado el algoritmo para transformar la nueva información.
- Facilitará probar varias transformaciones para entender qué combinación es mejor.

_ En general, la preparación de los datos es un proceso iterativo que puede requerir varias iteraciones antes de que los datos estén listos para ser utilizados en un modelo de aprendizaje automático. Es importante dedicar tiempo y esfuerzo a esta etapa para obtener resultados precisos y confiables del modelo.

Preparando los datos

_ En el contexto de la preparación de datos para algoritmos de aprendizaje automático, la limpieza de datos se refiere al proceso de identificar, corregir o eliminar errores, inconsistencias o imprecisiones en los datos. El objetivo de la limpieza de datos es asegurarse de que los datos sean precisos, completos y coherentes antes de utilizarlos en un modelo de aprendizaje automático. Algunos de los pasos comunes de limpieza de datos incluyen:

- Identificación y manejo de valores atípicos o nulos.
- Normalización o estandarización de datos para reducir la variabilidad.
- Codificación de variables categóricas en formato numérico.
- Remoción de duplicados.
- Verificación de la coherencia de los datos entre diferentes fuentes.

_ La limpieza de datos es un paso crítico en la preparación de datos y puede ser un proceso iterativo. Es importante asegurarse de que los datos sean de alta calidad antes de utilizarlos en un modelo de aprendizaje automático, ya que los errores o inconsistencias en los datos pueden resultar en modelos imprecisos o sesgados.

_ Para comenzar el proceso, lo primero que hacemos es volver a un conjunto de entrenamiento limpio (copiando nuevamente strat_train_set). También separaremos los predictores y las etiquetas, ya que no necesariamente queremos aplicar las mismas transformaciones a los predictores y los valores objetivo (observe que drop() crea una copia de los datos y no afecta a strat_train_set):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

- En este código, se está separando el conjunto de datos en dos variables, housing y housing_labels.
- La variable housing contendrá todas las columnas del conjunto de datos original strat_train_set, excepto la columna "median_house_value". Esto se logra utilizando el método drop() de Pandas, que devuelve una copia del conjunto de datos sin la columna especificada. Se utiliza axis=1 para especificar que se desea eliminar una columna en lugar de una fila.
- La variable housing_labels contendrá solamente la columna "median_house_value" del conjunto de datos original. Esto se logra seleccionando esa columna con la notación de corchetes y utilizando el método copy() para crear una copia de la columna en lugar de una vista de la misma.

- En resumen, housing contiene todas las características de entrada del modelo, mientras que housing_labels contiene la variable de salida o la etiqueta a predecir.

_ Cuando nos topamos con valores faltantes en nuestros dataset tenemos tres opciones:

- Deshacerse de filas correspondientes.
- Deshacerse todo el atributo (columna).
- Reemplazar los valores faltantes con otro valor (media, mediana, cero, etc)

_ Vimos anteriormente que el atributo total_bedrooms tiene algunos valores faltantes, así que podemos arreglar esto con:

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

- Estas líneas de código son una posible solución para manejar los valores faltantes en la columna "total_bedrooms" del conjunto de datos de viviendas.
- La primera línea elimina las filas que tienen un valor faltante en la columna "total_bedrooms" utilizando el método "dropna" con el argumento "subset" que indica en qué columnas buscar valores faltantes.
- La segunda línea elimina completamente la columna "total_bedrooms" utilizando el método "drop" con el argumento "axis" que indica que se desea eliminar una columna en lugar de una fila.
- La tercera línea calcula la mediana de la columna "total_bedrooms" y la almacena en la variable "median".
- La cuarta línea utiliza el método "fillna" para reemplazar los valores faltantes en la columna "total_bedrooms" con la mediana calculada anteriormente. El argumento "inplace=True" indica que se desea realizar la operación directamente sobre el conjunto de datos "housing" en lugar de devolver una copia modificada.

_ Estas son tres posibles opciones para manejar los valores faltantes, y la opción adecuada dependerá del conjunto de datos y del problema específico que se esté abordando. Si se elige la opción 3, se debe calcular el valor mediano en el conjunto de entrenamiento y usarlo para rellenar los valores faltantes en el conjunto de entrenamiento. No hay que olvidarse de guardar el valor mediano que se calculó calculado. Ya que se lo necesitará más adelante para reemplazar los valores faltantes en el conjunto de prueba cuando quieras evaluar tu sistema, y también una vez que el sistema esté en funcionamiento para reemplazar los valores faltantes en nuevos datos.

_ Scikit-Learn proporciona una clase útil para manejar valores faltantes mediante SimpleImputer. Para usarlo, primero, necesitamos crear una instancia de

SimpleImputer, especificando que quieres reemplazar los valores faltantes de cada atributo con la mediana de ese atributo:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

- Este código importa la clase SimpleImputer del módulo impute de la biblioteca Scikit-Learn y luego crea una instancia de esta clase llamada imputer.
- La clase SimpleImputer proporciona una manera de tratar los valores faltantes en los datos. Al crear una instancia de esta clase, se especifica la estrategia que se utilizará para imputar los valores faltantes. En este caso, se ha especificado la estrategia "median" para que el valor faltante se impute con la mediana del atributo correspondiente.

_ Dado que la mediana solo se puede calcular en atributos numéricos, es necesario crear una copia de los datos sin el atributo de texto ocean_proximity:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

_ Ahora podemos ajustar la instancia del imputador a los datos de entrenamiento usando el método fit():

```
imputer.fit(housing_num)
```

- El método fit() del objeto imputer calcula el valor mediano de cada atributo numérico y lo almacena en su variable de instancia statistics_. En otras palabras, el imputer se ajusta (fit) a los datos de entrenamiento para calcular el valor mediano de cada atributo numérico y lo guarda en la variable de instancia.
- Después de llamar a fit(), el imputer está listo para reemplazar los valores faltantes en cualquier conjunto de datos utilizando la mediana que se calculó durante la llamada a fit().

Imputador o SimpleImputer: como vimos, es una clase en la biblioteca Scikit-Learn de Python que se utiliza para manejar los valores faltantes en los datos. Su función principal es reemplazar los valores faltantes por una estrategia específica, como la media, mediana, valor más frecuente, etc. El imputador se "entrena" con los datos de entrenamiento para calcular la estrategia de reemplazo, y luego se utiliza para transformar tanto los datos de entrenamiento como los de prueba.

Datos de entrenamiento: son un conjunto de datos utilizado para entrenar un modelo de aprendizaje automático. Estos datos se utilizan para ajustar los parámetros del modelo, para que pueda aprender a realizar predicciones precisas en nuevos datos. El objetivo es que el modelo pueda generalizar su conocimiento más allá de los datos de entrenamiento y realizar predicciones precisas en datos nunca antes vistos. Los datos

de entrenamiento suelen estar etiquetados, lo que significa que se proporciona una respuesta deseada o una salida correcta para cada entrada.

_ El imputador simplemente ha calculado la mediana de cada atributo y ha almacenado el resultado en su variable de instancia “statistics_”. Solo el atributo total_bedrooms tenía valores faltantes, pero no podemos estar seguros de que no habrá valores faltantes en los nuevos datos después de que el sistema se active, por lo que es más seguro aplicar el imputador a todos los atributos numéricos.

_ Ahora podemos usar este imputador "entrenado" para transformar el conjunto de entrenamiento reemplazando los valores faltantes con las medianas aprendidas:

```
X = imputer.transform(housing_num)
```

- El código X = imputer.transform(housing_num) utiliza el método transform() de la clase SimpleImputer para aplicar la estrategia de imputación previamente ajustada en el conjunto de datos housing_num.
- La variable X contiene la matriz NumPy resultante de la transformación de housing_num aplicando la estrategia de imputación. Los valores faltantes son reemplazados por los valores aprendidos previamente por el imputador.
- Después de la transformación, se pueden utilizar los datos transformados para entrenar un modelo de aprendizaje automático.

_ El resultado es una matriz NumPy simple que contiene las características transformadas. Si desea volver a ponerlo en un DataFrame de pandas, es simple:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

- Este código crea un nuevo DataFrame llamado housing_tr, que contiene los datos transformados obtenidos después de reemplazar los valores faltantes en housing_num.
- pd.DataFrame(X): crea un DataFrame de Pandas a partir del arreglo NumPy X.
- columns=housing_num.columns: especifica que los nombres de las columnas del nuevo DataFrame deben ser los mismos que los nombres de las columnas del DataFrame original housing_num.
- index=housing_num.index: especifica que los índices del nuevo DataFrame deben ser los mismos que los índices del DataFrame original housing_num.

_ Podemos ver el resultado de housing_tr simplemente imprimiendo el objeto usando la función print() o escribiendo su nombre en una celda de código y ejecutándola. Esto mostrará la versión transformada de housing_num con los valores faltantes llenados con la mediana de cada columna numérica.

Matriz NumPy: es una estructura de datos multidimensional que permite almacenar y manipular datos numéricos de manera eficiente. Es similar a una lista o array en Python, pero se caracteriza por tener un tamaño fijo y homogéneo, lo que permite realizar operaciones vectorizadas de forma rápida y eficiente. Además, las matrices

NumPy ofrecen una amplia variedad de funciones y métodos especializados para trabajar con datos numéricos.

Tratando con texto y variables categóricas

Atributo categórico: es una característica de un conjunto de datos que describe una variable con un número finito de valores posibles, que generalmente representan diferentes categorías o clases. Por ejemplo, el color de los ojos de una persona, el tipo de automóvil, la marca de un producto, son ejemplos de atributos categóricos. Estos atributos son diferentes a los atributos numéricos, que son variables que pueden tomar valores continuos o discretos.

_ El manejo de atributos de texto y categóricos es una parte importante en el preprocesamiento de datos para el aprendizaje automático. Los algoritmos de aprendizaje automático generalmente esperan que las entradas sean números, por lo que es necesario convertir los atributos de texto y categóricos en valores numéricos antes de alimentarlos a un modelo. Hay dos técnicas principales para manejar atributos categóricos, uno es la codificación one-hot y el otro la codificación de etiquetas.

Codificación one-hot: convierte cada valor categórico en un vector binario que indica si el valor está presente o no en la muestra.

Codificación de etiquetas: asigna un número entero único a cada valor categórico.

_ Además, a menudo es necesario manipular los atributos de texto, como eliminar puntuación, convertir a minúsculas y tokenizar para separar el texto en palabras individuales.

_ A continuación tenemos lo siguiente:

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)
```

- Este código extrae la columna "ocean_proximity" del conjunto de datos "housing" y la almacena en una nueva variable llamada "housing_cat".
- Luego, muestra las primeras 10 filas de esta variable utilizando el método "head()". La columna "ocean_proximity" es una variable categórica que indica la proximidad del océano para cada registro en el conjunto de datos.

_ No es un texto arbitrario, ya que hay un número limitado de valores posibles, cada uno de los cuales representa una categoría. Por lo tanto, esta característica es una característica categórica. La mayoría de algoritmos de machine learning espera como atributos variables numéricas. Para tratar estos casos se pueden utilizar clases de sklearn en función de la variable categórica con la que estamos tratando.

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

- En este código, se está importando la clase OrdinalEncoder de la biblioteca scikit-learn para convertir las categorías de texto en números.
- Luego se crea una instancia del objeto OrdinalEncoder y se almacena en la variable 'ordinal_encoder'. A continuación, se utiliza el método fit_transform() de esta instancia para ajustar y transformar los datos de la columna 'ocean_proximity' de housing en números enteros codificados en orden.
- El resultado se almacena en la variable housing_cat_encoded, que muestra los primeros 10 elementos de la matriz de datos codificados en orden.

_ Como resultado, se obtiene una representación numérica del atributo categórico ocean_proximity. Los valores originales de ocean_proximity se han sustituido por valores numéricos de 0 a 4.

_ Podemos obtener la lista de categorías usando la variable de instancia categories_. Es una lista que contiene una matriz de 1 dimensión, de categorías para cada atributo categórico (en este caso, una lista que contiene una sola matriz ya que solo hay un atributo categórico).

```
ordinal_encoder.categories_
```

- El código ordinal_encoder.categories_ muestra las categorías aprendidas por el codificador ordinal en forma de una lista de matrices NumPy, donde cada matriz contiene las categorías de una característica categórica. Por ejemplo, si se tiene una característica categórica "color" con tres categorías posibles (rojo, verde y azul), la lista de matrices devuelta tendrá una matriz con tres elementos, donde cada elemento corresponderá a una de las categorías ("rojo", "verde" y "azul", en este caso). En este caso mostraría los países.

_ Un problema con esta representación es que los algoritmos de machine learning supondrán que dos valores cercanos son más similares que dos valores lejanos. Esto puede estar bien en algunos casos (por ejemplo, para categorías ordenadas como "malo", "promedio", "bueno" y "excelente"), pero obviamente no es el caso para la columna de "ocean_proximity" (por ejemplo, las categorías 0 y 4 son claramente más similares que las categorías 0 y 1). Para solucionar este problema, una solución común es crear un atributo binario por categoría, en donde un atributo será igual a 1 cuando la categoría es "<1H OCEAN" y 0 los demás, otro atributo igual a 1 cuando la categoría es "INLAND" y 0 los demás, y así sucesivamente. Esto se llama codificación one-hot, porque solo un atributo será igual a 1 (hot), mientras que los demás serán 0 (cold). Los nuevos atributos a veces se llaman atributos ficticios.

Atributos ficticios: este término se refiere a una técnica de codificación de variables categóricas, también conocida como "codificación one-hot" como ya vimos. En donde, en lugar de representar la variable categórica con un solo número entero (como se hace en la codificación ordinal), se crea una nueva columna para cada categoría posible y se le asigna un valor binario de 1 o 0 para indicar si esa categoría está presente o no en la observación. Entonces, estas nuevas columnas se llaman "atributos ficticios" porque no representan una medida real de la variable original, sino que son

simplemente una forma de representar categóricamente las observaciones en un formato que puede ser entendido por los algoritmos de aprendizaje automático.

_ Scikit-Learn proporciona una clase OneHotEncoder para convertir valores categóricos en vectores one-hot:

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
```

- Este código importa la clase OneHotEncoder del módulo preprocessing de Scikit-Learn, que se utiliza para convertir atributos categóricos en vectores one-hot.
- Luego, se crea una instancia de OneHotEncoder y se almacena en la variable cat_encoder.
- A continuación, se utiliza el método fit_transform() de la instancia cat_encoder para transformar el atributo categórico housing_cat en un conjunto de vectores one-hot, que se almacena en la variable housing_cat_1hot.
- El resultado es una matriz dispersa (sparse matrix) que representa los vectores one-hot. El número de filas es igual al número de instancias en el conjunto de datos (16512 en este caso), y el número de columnas es igual al número de categorías diferentes en el atributo categórico (5 en este caso). La matriz dispersa se almacena en formato comprimido de fila (Compressed Sparse Row format) para ahorrar espacio en memoria.
- Cada columna corresponde a una categoría de la variable categórica "ocean_proximity". Las celdas de la matriz contienen valores 0 o 1, dependiendo de si la vivienda pertenece a esa categoría o no. La representación de matriz dispersa se utiliza porque la mayoría de los elementos de la matriz son ceros, lo que ahorra espacio de almacenamiento en memoria.

Matriz dispersa: es una matriz que contiene principalmente ceros. Cuando una matriz es grande y contiene principalmente ceros, almacenarla como una matriz densa (es decir, una matriz normal que contiene todos sus elementos) puede ser ineficiente en términos de memoria. Por lo tanto, en lugar de almacenar todos los valores cero, se puede utilizar una representación comprimida de la matriz que solo almacena los valores no cero y su ubicación en la matriz.

_ Podemos observar que la salida es una matriz dispersa de SciPy en lugar de una matriz de NumPy. Esto es muy útil cuando tenemos atributos categóricos con miles de categorías. Después de la codificación one-hot, obtenemos una matriz con miles de columnas y la matriz está llena de ceros excepto por un solo 1 por fila. Usar mucha memoria principalmente para almacenar ceros sería muy derrochador, entonces, en lugar de eso, una matriz dispersa solo almacena la ubicación de los elementos no nulos. Podemos usarla principalmente como una matriz de 2 dimensiones normal,

pero si realmente queremos convertirla en una matriz de NumPy (densa), debemos llamar al método toarray(). Para visualizar el resultado hacemos:

```
housing_cat_1hot.toarray()
```

- Este código convierte la matriz dispersa housing_cat_1hot en una matriz NumPy densa. Como se mencionó anteriormente, después de aplicar la codificación one-hot, obtenemos una matriz con miles de columnas y la matriz está llena de ceros excepto por un solo 1 por fila. Esto es ineficiente en términos de memoria, por lo que se utiliza una matriz dispersa que solo almacena la ubicación de los elementos no nulos. La llamada toarray() convierte esta matriz dispersa en una matriz NumPy densa para poder manipularla como una matriz normal.

Matriz NumPy densa: es una matriz que almacena todos sus elementos en la memoria, incluyendo ceros y valores no nulos. Esto significa que todos los elementos ocupan espacio en la memoria, lo que puede ser costoso en términos de recursos cuando se trabaja con matrices grandes.

_ Ahora, podemos obtener la lista de categorías utilizando la variable de instancia categories_ del codificador:

```
cat_encoder.categories_
```

- Este código devuelve las categorías de las variables categóricas después de haber sido transformadas por el codificador de una sola codificación (OneHotEncoder) y almacenadas en el objeto cat_encoder. El resultado es una lista de arrays NumPy, donde cada array representa las categorías de una variable categórica transformada en un vector de codificación. Por ejemplo, si tenemos una variable categórica "ocean_proximity" con cinco categorías, el resultado de este código sería una lista que contiene un array con las cinco categorías codificadas en un vector binario para esa variable.

Codificador: se refiere a un objeto o función que se utiliza para transformar los datos de una forma a otra. En el contexto de aprendizaje automático, se pueden utilizar codificadores para transformar datos categóricos en valores numéricos que puedan ser procesados por algoritmos de aprendizaje automático. Un ejemplo común de un codificador es el OrdinalEncoder de Scikit-Learn, que convierte las categorías categóricas en valores numéricos ordinales. Otro ejemplo es el OneHotEncoder, que transforma las categorías categóricas en vectores binarios (es decir, "unos" y "ceros").

Transformaciones personalizadas

Transformers personalizados: son una característica útil en Scikit-Learn que nos permiten personalizar el preprocesamiento de los datos y la construcción de modelos de aprendizaje automático. Los transformers personalizados son esencialmente clases de Python que implementan dos métodos principales:

- Método fit() toma los datos de entrenamiento como entrada y los utiliza para aprender los parámetros necesarios para la transformación.
- Método transform(): toma los datos de entrada y los transforma en una nueva representación.

_ Para construir un transformer personalizado, se debe crear una clase Python que herede de la clase BaseEstimator y TransformerMixin, y luego definir los métodos fit() y transform() personalizados. También es posible añadir un método adicional, fit_transform(), que se utiliza para entrenar y transformar los datos de entrada de una sola vez. Una vez que se ha definido la clase personalizada, se puede utilizar junto con otros transformers y modelos en un pipeline de Scikit-Learn para realizar tareas de preprocesamiento y modelado complejas de manera eficiente y efectiva.

_ A continuación vemos un ejemplo en donde hay una pequeña clase de transformador que agrega los atributos combinados que mencionamos anteriormente:

```
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]

        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

- Este código define una clase personalizada de transformador de Scikit-Learn llamada CombinedAttributesAdder que agrega características adicionales a los datos. Esta clase tiene dos métodos, fit y transform.
- El método init inicializa la clase y toma un parámetro add_bedrooms_per_room con un valor predeterminado de True.
- El método fit no hace nada, simplemente devuelve la instancia de la clase.

- El método transform realiza la transformación real. Primero, calcula dos nuevas características usando los índices de columnas definidos anteriormente: rooms_per_household y population_per_household. Luego, si el parámetro add_bedrooms_per_room es verdadero, calcula una tercera característica, bedrooms_per_room. Finalmente, utiliza la función np.c_ de NumPy para concatenar estas nuevas características con las características originales en un solo conjunto de datos.
- En el último bloque de código, se crea una instancia de la clase CombinedAttributesAdder con el parámetro add_bedrooms_per_room establecido en False, y luego se llama al método transform en los datos de vivienda para agregar las nuevas características y almacenar los resultados en housing_extra_attribs.

_ Si ejecutamos este código, se creará una instancia de la clase CombinedAttributesAdder con el hiperparámetro add_bedrooms_per_room establecido en False. Luego, se aplicará el método transform() de la instancia a los valores de housing.values y se asignará el resultado a la variable housing_extra_attribs. El resultado será un array NumPy que contiene las características originales de housing y dos nuevas características: rooms_per_household y population_per_household. La característica bedrooms_per_room no se incluirá porque el hiperparámetro add_bedrooms_per_room se estableció en False.

_ Cuanto más se automatice estos pasos de preparación de datos, más combinaciones puede probar automáticamente, lo que hace mucho más probable que encuentre una gran combinación (y le ahorra mucho tiempo).

Escalado de características

_ El escalado de características (Feature Scaling) es un proceso importante en el preprocesamiento de datos para el aprendizaje automático. Este proceso implica transformar las características de entrada de tal manera que estén en una escala común. Esto se hace porque muchos algoritmos de aprendizaje automático no funcionan bien cuando las características tienen diferentes escalas. Por ejemplo, los algoritmos que se basan en la distancia euclidiana, como la regresión logística y el K-Vecinos más cercanos (KNN), pueden verse afectados negativamente por las diferentes escalas de las características.

_ Existen dos técnicas comunes de escalado de características:

Normalización: o Min-max scaling, escala los valores de las características a un rango de 0 a 1. Internamente funciona como:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

- Este código es utilizado para escalar características de una matriz X de entrada. En la primera línea, se calcula la versión estandarizada de la matriz X. Lo que hace esta fórmula es restar el valor mínimo de cada columna de X y luego dividir cada valor por la diferencia entre el valor máximo y el valor mínimo en esa columna. Esto da como resultado una versión estandarizada de la matriz X en la que cada columna tiene una media de cero y una desviación estándar de uno.
- Luego, se ejecuta la segunda línea para escalar los valores de la matriz estandarizada X_std a un rango específico definido por los valores max y min. Esto se hace multiplicando cada valor en X_std por el rango deseado (max - min) y luego sumando el valor mínimo deseado (min).

Estandarización: escala los valores de las características para que tengan una media de cero y una desviación estándar de uno. Este tipo de escalado es menos sensible a valor atípicos.

$$z = (x - u) / s$$

_ Ambas técnicas son útiles en diferentes situaciones, y se debe seleccionar la técnica adecuada según el conjunto de datos y el algoritmo de aprendizaje automático utilizado. La normalización es adecuada cuando la distribución de los datos no sigue una distribución gaussiana (normal) y cuando se requiere que todos los valores estén en un rango específico, por ejemplo, entre 0 y 1.La estandarización, por otro lado, es adecuada cuando la distribución de los datos es gaussiana y se desea que los datos tengan una media de cero y una desviación estándar de uno. En resumen, el escalado de características es un paso importante en el preprocesamiento de datos para el aprendizaje automático, que ayuda a mejorar el rendimiento y la precisión de los modelos de aprendizaje automático al asegurarse de que las características estén en una escala común.

Pipelines de transformación

Pipeline de transformación: es una secuencia de transformaciones que se aplican a los datos. En muchas ocasiones, se necesita aplicar una secuencia de transformaciones en un conjunto de datos, como por ejemplo, reemplazar valores faltantes, escalar características y crear nuevas características a partir de las existentes. Tenemos que recordar, que se aplican transformaciones a los datos por varias razones, algunas de las razones son:

- Escalado: a veces, los datos pueden tener diferentes escalas. Por ejemplo, en un conjunto de datos, una variable puede tener un rango de valores entre 1 y 10, mientras que otra variable puede tener un rango de valores entre 100 y 1000. En tales casos, es necesario escalar los datos para que todas las variables tengan un rango similar. De esta manera, se evita que una variable tenga más peso que otra en los cálculos.

- Normalización: a veces, es necesario normalizar los datos para que tengan una distribución normal. Esto es importante para algunos algoritmos de aprendizaje automático, que asumen que los datos tienen una distribución normal.
- Eliminación de valores atípicos: los valores atípicos son valores extremos que se encuentran en los datos y que no siguen la distribución normal. A veces, es necesario eliminar estos valores atípicos antes de realizar cualquier análisis o modelado.
- Ingeniería de características: a veces, es necesario crear nuevas características a partir de las características existentes para mejorar el modelo. Por ejemplo, se puede crear una nueva característica a partir de dos características existentes, como la relación entre ellas.
- Preprocesamiento: el preprocesamiento es una etapa importante en el análisis de datos. Se utiliza para limpiar y preparar los datos antes de realizar cualquier análisis o modelado. Esto puede incluir la eliminación de valores faltantes, la eliminación de duplicados y la corrección de errores.

_ Una pipeline de transformación ayuda a automatizar este proceso al encadenar las transformaciones en un orden específico. Además, una pipeline también es útil para incluir la etapa final de entrenamiento del modelo en la misma secuencia. Esto significa que la pipeline completa se puede tratar como un solo transformador y se puede utilizar para preprocesar nuevos datos antes de alimentarlos al modelo.

_ En Scikit-Learn, podemos construir una pipeline utilizando la clase Pipeline. Esta clase toma una lista de pares (nombre, transformador), donde el nombre es una cadena que identifica el paso de la pipeline y el transformador es un objeto que implementa los métodos fit() y transform(). Una vez que se ha construido la pipeline, se puede usar el método fit_transform() para aplicar todas las transformaciones de una vez en los datos de entrenamiento. La pipeline también se puede utilizar para transformar nuevos datos en el futuro utilizando el método transform(). También, cuando se realizan múltiples transformaciones sobre las mismas columnas utilizamos la clase Pipeline.

_ A continuación vemos un pequeño pipeline para los atributos numéricos:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- Este código crea un pipeline de transformación de datos para los atributos numéricos de un conjunto de datos llamado housing.

- Primero, se importa la clase Pipeline de Scikit-Learn, que es una herramienta que ayuda a encadenar diferentes transformaciones y estimadores. También se importa la clase StandardScaler, que normaliza los datos escalando cada atributo para que tenga una media cero y una desviación estándar de uno.
- A continuación, se define un num_pipeline que consiste en tres transformaciones:
    - El primer paso es aplicar la transformación SimpleImputer con la estrategia de "mediana" para imputar los valores faltantes en los datos numéricos.
    - El segundo paso es aplicar la transformación personalizada CombinedAttributesAdder que agrega tres nuevas características a los datos numéricos: "rooms_per_household", "population_per_household" y "bedrooms_per_room".
    - El tercer paso es aplicar la transformación StandardScaler para normalizar los datos numéricos.
- Finalmente, se aplica el pipeline al conjunto de datos numéricos housing_num usando el método fit_transform(), que aplica las transformaciones secuencialmente al conjunto de datos y devuelve el resultado transformado. El resultado final es una matriz de características normalizadas y sin valores faltantes.

_ El código que se presenta se realiza con el objetivo de preprocesar los datos numéricos en un conjunto de datos de viviendas. El resultado de la ejecución de ese código es la transformación de los datos numéricos del conjunto de datos de viviendas (housing_num) mediante la secuencia de transformaciones definidas en el pipeline (num_pipeline). Específicamente, las transformaciones que se aplican son:

- Imputación de los valores faltantes (SimpleImputer(strategy="median")): se reemplazan los valores faltantes en los atributos numéricos por la mediana de cada atributo.
- Adición de atributos personalizados (CombinedAttributesAdder()): se agrega dos nuevos atributos numéricos calculados a partir de los atributos existentes.
- Escalado de los atributos (StandardScaler()): se escala cada atributo para que tenga media cero y desviación estándar uno.

_ El resultado final de num_pipeline.fit_transform(housing_num) es un array NumPy con los datos transformados, en el mismo orden de las columnas del dataframe original housing_num. Este array puede ser utilizado posteriormente en modelos de aprendizaje automático.

Uniendo múltiples Pipelines

_ Hasta ahora, hemos manejado las columnas categóricas y numéricas por separado. Sería más conveniente tener un único transformador capaz de manejar todas las columnas, aplicando las transformaciones apropiadas a cada una de ellas. En la versión 0.20, Scikit-Learn introdujo ColumnTransformer para este propósito, y la buena noticia es que funciona muy bien con los DataFrames de pandas. Vamos a usarlo para aplicar todas las transformaciones a los datos de vivienda:

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

- En este código se está utilizando la clase ColumnTransformer de Scikit-Learn para aplicar transformaciones a todas las columnas de un conjunto de datos.
- Primero, se crean dos listas:
  - num_attribs contiene los nombres de las columnas numéricas del conjunto de datos housing_num.
  - cat_attribs: contiene el nombre de la columna categórica "ocean_proximity".
- Luego, se crea un full_pipeline que utiliza el ColumnTransformer para aplicar las transformaciones necesarias a las columnas numéricas y categóricas.
- El ColumnTransformer toma una lista de tuplas, donde cada tupla especifica un nombre de columna, un transformador y una lista de nombres de columnas en las que se aplicará el transformador. En este caso, se utiliza el transformador num_pipeline en las columnas numéricas (especificadas por num_attribs) y el transformador OneHotEncoder() en la columna categórica (especificada por cat_attribs).
- Finalmente, se aplica el full_pipeline al conjunto de datos housing para obtener housing_prepared, que es una matriz NumPy que contiene todas las transformaciones aplicadas a las columnas numéricas y categóricas del conjunto de datos original.

_ Este código se utiliza para aplicar todas las transformaciones necesarias a los datos de vivienda. Y el resultado es una matriz de datos transformados y preparados para su uso en el modelo de aprendizaje automático.

Seleccionar el modelo y entrenarlo

_ La selección y entrenamiento de un modelo es un paso clave en el proceso de aprendizaje automático. En esta etapa, se elige un modelo y se lo entrena con los datos disponibles para que pueda hacer predicciones precisas en nuevos datos.

1)_ El primer paso en la selección de un modelo es determinar qué tipo de problema se está tratando de resolver, si es un problema de clasificación o de regresión. En el aprendizaje supervisado, se entrena un modelo con datos de entrada y salida conocidos para que pueda predecir la salida correcta para nuevos datos de entrada.

2)_ Una vez que se ha determinado el tipo de problema, se selecciona un modelo que sea adecuado para el problema. Hay muchos modelos diferentes disponibles, desde simples regresiones lineales hasta complejos modelos de redes neuronales.

3)_ Después de seleccionar un modelo, se entrena con los datos disponibles. En esta etapa, los datos se dividen en conjuntos de entrenamiento y prueba. El modelo se entrena con los datos de entrenamiento y se evalúa su rendimiento utilizando los datos de prueba.

4)_ Una vez que se ha entrenado y evaluado el modelo, se pueden ajustar los hiperparámetro del modelo para mejorar su rendimiento en los datos de prueba. Los hiperparámetros son valores que no se aprenden durante el entrenamiento del modelo, pero que afectan el rendimiento del modelo.

5)_ Finalmente, una vez que se ha seleccionado y entrenado un modelo final, se puede utilizar para hacer predicciones en nuevos datos.

Entrenando el modelo

_ El entrenamiento y evaluación en el Conjunto de Entrenamiento (Training Set), es una etapa importante del proceso de aprendizaje automático. Después de preparar los datos y crear un conjunto de entrenamiento, el siguiente paso es elegir un modelo y entrenarlo en el conjunto de entrenamiento. El objetivo del entrenamiento es ajustar los parámetros del modelo para que pueda hacer predicciones precisas en nuevos datos. Para hacer esto, el modelo utiliza un algoritmo de aprendizaje que compara sus predicciones con las respuestas reales en el conjunto de entrenamiento y ajusta sus parámetros para minimizar la diferencia entre las predicciones y las respuestas reales.

_ Después del entrenamiento, es importante evaluar el modelo en el conjunto de entrenamiento para medir su rendimiento y estimar qué tan bien generaliza a nuevos datos. El modelo se evalúa calculando su capacidad para hacer predicciones precisas en los datos que no ha visto antes. Para evaluar el modelo, se utilizan varias métricas de evaluación, como el error cuadrático medio (MSE) y la raíz cuadrada del error cuadrático medio (RMSE) para modelos de regresión y la precisión, la recall y la F1-score para modelos de clasificación.

_ Si el rendimiento del modelo en el conjunto de entrenamiento es insatisfactorio, es posible que se deba ajustar los hiperparámetros del modelo o probar diferentes algoritmos de aprendizaje. Si el rendimiento es satisfactorio, se puede proceder a evaluar el modelo en el conjunto de prueba para obtener una evaluación más realista de su rendimiento.

Modelo de regresión lineal: es un modelo estadístico que se utiliza para describir la relación entre una variable continua (la variable dependiente) y una o más variables predictoras (variables independientes), asumiendo que esta relación es lineal. El modelo busca ajustar una línea recta que se ajuste lo mejor posible a los datos, de modo que pueda utilizarse para hacer predicciones sobre la variable dependiente para valores desconocidos de las variables predictoras. El objetivo de la regresión lineal es minimizar la distancia entre los valores reales y los valores predichos por el modelo.

_ Entonces, una vez realizado todos los pasos anteriores solo queda elegir el modelo que vamos a utilizar. Podríamos empezar por probar un modelo simple como es una regresión lineal, por ende ejecutamos:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- Este código se encarga de entrenar un modelo de regresión lineal utilizando los datos preparados housing_prepared y las etiquetas housing_labels.
- Primero se importa la clase LinearRegression del módulo sklearn.linear_model.
- Luego, se crea una instancia de esta clase y se asigna a la variable lin_reg.
- Finalmente, se entrena el modelo utilizando el método fit(), pasando los datos preparados y las etiquetas como argumentos. Después de que se haya ajustado el modelo a los datos de entrenamiento, estará listo para hacer predicciones sobre nuevos datos.

_ El modelo entrenado se puede utilizar posteriormente para hacer predicciones en nuevos datos. Entonces, una vez ejecutado el código ahora tenemos un modelo de Regresión Lineal funcional. Por un lado, podemos ver los resultados utilizando las métricas de evaluación para modelos de regresión, como el error cuadrático medio (MSE) o el error absoluto medio (MAE). También podemos probar algunas instancias del conjunto de entrenamiento como:

```
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions:", lin_reg.predict(some_data_prepared))
print("Labels:", list(some_labels))
```

- Este código selecciona las primeras 5 instancias del conjunto de datos housing y las etiquetas correspondientes del conjunto de etiquetas housing_labels.

Luego, aplica el pipeline completo (con todas las transformaciones) a esas 5 instancias utilizando el método transform() de ColumnTransformer y lo almacena en some_data_prepared.

- Finalmente, el código utiliza el modelo de regresión lineal lin_reg para hacer predicciones en some_data_prepared utilizando el método predict() y muestra las predicciones en la pantalla. También muestra las etiquetas correspondientes para comparar.

_ Funciona, aunque las predicciones no son exactas (por ejemplo, la primera predicción está fuera por cerca del 40%). Volvemos a medir el RMSE de este modelo de regresión en todo el conjunto de entrenamiento usando la función mean_squared_error() de Scikit-Learn:

```
from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

- Este código calcula la raíz del error cuadrático medio (RMSE) de las predicciones del modelo de regresión lineal en todo el conjunto de entrenamiento.
- Primero, se utiliza el método "predict" para realizar predicciones sobre el conjunto de entrenamiento, que se almacenan en "housing_predictions". Luego, se utiliza la función "mean_squared_error" de Scikit-Learn para calcular el error cuadrático medio (MSE) entre las etiquetas reales "housing_labels" y las predicciones "housing_predictions".
- Finalmente, se calcula la raíz cuadrada del MSE para obtener el RMSE, que se almacena en "lin_rmse".

_ Este valor se utiliza para evaluar el rendimiento del modelo de regresión lineal, cuanto menor sea el valor del RMSE, mejor será el rendimiento del modelo en el conjunto de datos de entrenamiento.

_ El puntaje obtenido por el modelo de regresión lineal en el conjunto de entrenamiento no es satisfactorio, con un error de predicción típico de $68628 aproximadamente, lo que indica que el modelo no se ajusta bien a los datos de entrenamiento, ya que teniendo en cuenta que los precios de la casas van entre $120000 y $265000 la error en la estimación es bastante grande. Esto puede deberse a que las características no proporcionan suficiente información o que el modelo no es lo suficientemente potente. Las formas de solucionar este problema son seleccionar un modelo más potente, proporcionar mejores características o reducir las restricciones en el modelo. En este caso, el modelo no está regularizado, lo que descarta la última opción. Una solución sería agregar más características al modelo, pero primero se probará con un modelo más complejo para ver si mejora el rendimiento. En este caso podríamos estar frente a un caso de underfitting,

Caso de underfitting: ocurre cuando un modelo no puede aprender lo suficiente de los datos de entrenamiento y, por lo tanto, no puede hacer predicciones precisas incluso en el conjunto de entrenamiento. Esto puede deberse a que el modelo es demasiado simple o que los datos de entrenamiento son ruidosos o insuficientes. En general, el modelo es incapaz de capturar la complejidad de los datos y, por lo tanto, no se ajusta bien a los datos.

_ Entonces, podemos probar un modelo más complejo, usando DesitionTreeRegressor:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

- Este código importa la clase DecisionTreeRegressor del módulo sklearn.tree de Scikit-Learn, que es una biblioteca popular de aprendizaje automático en Python.
- Luego crea una instancia de la clase DecisionTreeRegressor() y la almacena en la variable tree_reg. Después, ajusta el modelo a los datos de entrenamiento utilizando el método fit(), que acepta dos argumentos, housing_prepared, que es una matriz de NumPy que contiene las características de entrada del modelo, y housing_labels, que es una matriz NumPy que contiene las etiquetas de salida del modelo correspondientes a cada instancia de housing_prepared.

DecisionTreeRegressor: es un modelo de regresión no paramétrico que utiliza un árbol de decisión para predecir el valor de una variable de respuesta continua. En lugar de ajustar una función paramétrica, este modelo divide recursivamente el conjunto de datos en subconjuntos más pequeños basados en las variables predictoras, y utiliza la media de las variables de respuesta en cada subconjunto como la predicción. Esto permite capturar relaciones no lineales y complejas entre las variables predictoras y la variable de respuesta.

_ Ahora que el modelo está entrenado, lo evaluamos en el conjunto de entrenamiento:

```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

_ El resultado es 0 en housing_predictions y tree_mse, y esto indica que el modelo de árbol de decisión pudo ajustarse perfectamente al conjunto de entrenamiento, lo cual puede parecer una buena noticia. Sin embargo, es muy probable que el modelo haya sobreajustado (overfitting) los datos de entrenamiento, lo que significa que es poco probable que generalice bien a nuevos datos. Para confirmar esto, es necesario evaluar el modelo en un conjunto de datos de prueba separado. Como vimos anteriormente, no se debe tocar el conjunto de prueba hasta que se esté listo para lanzar un modelo

en el que se tenga confianza, por lo que es necesario utilizar parte del conjunto de entrenamiento para entrenamiento y parte para validación del modelo.

Validación cruzada

_ En esta sección se aborda el problema de evaluar la eficacia de un modelo de aprendizaje automático de manera más fiable. En lugar de simplemente dividir los datos en un conjunto de entrenamiento y un conjunto de prueba, se utiliza la validación cruzada (cross-validation) para evaluar el modelo en varias divisiones diferentes de los datos de entrenamiento y obtener una puntuación media.

_ La validación cruzada implica dividir el conjunto de entrenamiento en un número determinado de "pliegues" (folds en inglés), por ejemplo, 10. El modelo se entrena en nueve pliegues y se evalúa en el pliegue restante. Esto se repite para cada pliegue, de modo que cada pliegue se utiliza una vez como conjunto de prueba y nueve veces como conjunto de entrenamiento. Se calcula una puntuación de evaluación para cada pliegue y se promedian para obtener una puntuación media de evaluación.

_ Este enfoque proporciona una evaluación más fiable del rendimiento del modelo, ya que se evalúa en varios subconjuntos diferentes de los datos de entrenamiento y no solo en uno. Además, permite aprovechar al máximo los datos disponibles para el entrenamiento y la evaluación, en lugar de desperdiciar una parte de ellos para un conjunto de prueba separado.

_ En resumen, la validación cruzada es una técnica útil para evaluar la eficacia de un modelo de aprendizaje automático, ya que proporciona una evaluación más fiable al evaluar el modelo en varias divisiones diferentes de los datos de entrenamiento.

_ Una gran alternativa es usar la función de validación cruzada K-fold de Scikit-Learn.

Modelo de árbol de decisión: es un tipo de modelo de aprendizaje supervisado que se utiliza tanto para problemas de regresión como de clasificación. Como su nombre indica, el modelo se construye en forma de un árbol, donde cada nodo interno representa una pregunta o una característica del conjunto de datos, cada rama representa una posible respuesta a esa pregunta, y cada hoja representa una predicción o una clase. El objetivo es dividir el conjunto de datos en ramas más pequeñas y homogéneas, de manera que se puedan tomar decisiones precisas sobre la predicción o clasificación de nuevas instancias.

_ El siguiente código divide aleatoriamente el conjunto de entrenamiento en 10 subconjuntos distintos llamados folds, luego entrena y evalúa el modelo de árbol de decisión 10 veces, eligiendo un fold diferente para la evaluación cada vez y entrenando en los otros 9 folds. El resultado es una matriz que contiene las 10 puntuaciones de evaluación.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

- Este código utiliza la función cross_val_score de la librería sklearn.model_selection para evaluar el modelo de DecisionTreeRegressor (tree_reg) utilizando la técnica de validación cruzada. En este caso, se utiliza cv=10, lo que significa que se divide el conjunto de entrenamiento en 10 partes iguales, se entrena el modelo en 9 partes y se evalúa en la restante, repitiéndose este proceso 10 veces, de manera que todas las partes hayan sido utilizadas como conjunto de evaluación en algún momento.
- Se utiliza scoring="neg_mean_squared_error" para especificar que se va a utilizar el error cuadrático medio negativo como métrica de evaluación. La función devuelve un array con los resultados obtenidos en cada una de las 10 iteraciones de la validación cruzada.
- Finalmente, se utiliza np.sqrt(-scores) para obtener las puntuaciones de raíz cuadrada del error cuadrático medio negativo, lo que representa el error de cada iteración del modelo en la validación cruzada. Estos resultados se almacenan en tree_rmse_scores.

_ Vemos los resultados:

```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

- La función display_scores imprime en pantalla los scores de evaluación de un modelo y su promedio y desviación estándar.
- La función toma como argumento una lista de scores y primero imprime la lista completa de scores. Luego, calcula el promedio de los scores y lo imprime en pantalla. Finalmente, calcula la desviación estándar de los scores y lo imprime en pantalla.
- Esta función puede ser útil para obtener una idea de la variabilidad de los scores de evaluación de un modelo.

_ Entonces podemos ver los puntajes obtenidos en cada una de las iteraciones del proceso de validación cruzada, la media de los puntajes y la desviación estándar. Estos resultados nos permiten tener una idea más clara del rendimiento del modelo y su nivel de consistencia en las diferentes iteraciones del proceso.

_ Ahora el árbol de decisiones no parece tan bueno como antes. De hecho, parece funcionar peor que el modelo de regresión lineal. Vemos que la validación cruzada permite obtener no solo una estimación del rendimiento del modelo, sino también una medida de cuán precisa es esta estimación (es decir, su desviación estándar). El árbol de decisiones tiene una puntuación de aproximadamente 71,407, generalmente ± 2,439. No tendría esta información si solo usara un conjunto de validación. Pero la validación cruzada tiene el costo de entrenar el modelo varias veces, por lo que no

siempre es posible. Entonces calculamos las mismas puntuaciones para el modelo de Regresión Lineal por las dudas:

```
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)

lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

- El código realiza validación cruzada con 10 folds para el modelo de regresión lineal y se utiliza el error cuadrático medio negativo como medida de desempeño.
- Luego, se calcula la raíz cuadrada del valor negativo del error cuadrático medio para obtener el puntaje de validación cruzada para cada fold.
- Finalmente, se utiliza la función "display_scores" para imprimir los puntajes obtenidos, así como la media y la desviación estándar de los mismos.

_ El modelo de Árbol de Decisión está sobreajustando tanto que funciona peor que el modelo de Regresión Lineal. Probemos ahora un último modelo qué cae dentro de la categoría de algoritmos de ensamble, el RandomForestRegressor. Los algoritmos de ensamble se construyen en base a algoritmos más simples. En este caso RandomForest se construye en base a múltiples DesitionTree.

Modelo RandomForestRegressor: es un algoritmo de aprendizaje automático que utiliza el método de Random Forest, que es una técnica de Ensemble Learning que se basa en la construcción de múltiples árboles de decisión y la combinación de sus resultados para obtener una predicción más precisa y estable.

_ En lugar de entrenar un solo árbol de decisión, el modelo RandomForestRegressor entrena un conjunto de árboles de decisión en diferentes subconjuntos de características y utiliza un promedio ponderado de las predicciones de cada árbol para hacer la predicción final. Esto ayuda a reducir el riesgo de overfitting y mejora la precisión del modelo.

_ El modelo RandomForestRegressor es muy flexible y se puede utilizar para resolver problemas de regresión en una amplia variedad de dominios, incluyendo finanzas, comercio electrónico, salud y muchos otros. Es uno de los algoritmos más populares en el campo del aprendizaje automático debido a su alta precisión, capacidad de manejar grandes conjuntos de datos y facilidad de uso con la biblioteca Scikit-Learn.

```
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)
[...]
forest_rmse

display_scores(forest_rmse_scores)
```

- Este código utiliza el algoritmo RandomForestRegressor de la librería Scikit-Learn para ajustar un modelo de regresión a los datos de entrenamiento housing_prepared y housing_labels.
- Luego, se utiliza la métrica de error RMSE para evaluar el rendimiento del modelo sobre los mismos datos de entrenamiento, lo que puede dar lugar a una sobreestimación del rendimiento real del modelo.
- Para obtener una evaluación más precisa del rendimiento del modelo, se utiliza la técnica de validación cruzada con 10 folds para calcular la puntuación RMSE promedio y su desviación estándar. Esto se realiza con la función cross_val_score() de Scikit-Learn, que toma como entrada el modelo ajustado, los datos de entrenamiento, la métrica de error a utilizar y el número de folds a utilizar. Los resultados de la validación cruzada se almacenan en un array forest_rmse_scores, que se utiliza para calcular la puntuación media y la desviación estándar mediante la función display_scores().

_ Los RandomForests parecen muy prometedores. Sin embargo, hay que tener en cuenta que la puntuación en el conjunto de entrenamiento sigue siendo mucho más baja que en los conjuntos de validación, lo que significa que el modelo todavía está sobreajustando el conjunto de entrenamiento. Posibles soluciones para el sobreajuste son simplificar el modelo, restringirlo (es decir, regularizarlo) u obtener muchos más datos de entrenamiento.

Fine-tuning del modelo

_ En esta fase de ajuste de modelos, el objetivo es mejorar el rendimiento de los modelos existentes mediante la optimización de sus hiperparámetros. Los hiperparámetros son valores que no se aprenden directamente del entrenamiento, sino que deben ser establecidos por el usuario antes de entrenar el modelo. Una forma de ajustar los hiperparámetros es mediante:

Búsqueda de cuadrícula (Grid Search): que es una técnica que consiste en definir una cuadrícula de valores posibles para cada hiperparámetro y evaluar el rendimiento del modelo con cada combinación de valores de los hiperparámetros. Se selecciona el conjunto de hiperparámetros que da como resultado el mejor rendimiento en el conjunto de validación.

Búsqueda aleatoria (Random Search): que en lugar de probar todas las combinaciones posibles de hiperparámetros, selecciona un número determinado de combinaciones de forma aleatoria. Esta técnica es más eficiente que la búsqueda de cuadrícula si hay muchos hiperparámetros para ajustar.

_ También es posible utilizar optimizadores bayesianos (Bayesian Optimization) para ajustar los hiperparámetros. En este enfoque, un modelo probabilístico se utiliza para estimar el rendimiento esperado de cada combinación de hiperparámetros, y se busca la combinación que maximiza el rendimiento esperado.

_ Una vez que se han ajustado los hiperparámetros, es importante evaluar el modelo final en el conjunto de prueba para obtener una estimación más precisa del rendimiento del modelo en datos no vistos.

Hiperparámetros: son parámetros que no se aprenden directamente del conjunto de datos de entrenamiento, sino que se establecen antes del entrenamiento. Estos parámetros son críticos para el rendimiento del modelo y pueden afectar significativamente su capacidad para generalizar a datos nuevos y no vistos. El proceso de ajustar y optimizar estos hiperparámetros se llama "fine-tuning" o "ajuste fino" y es una parte crucial del proceso de modelado de machine learning.

Grid search

_ Esta es una técnica utilizada en Machine Learning para buscar los mejores valores de hiperparámetros de un modelo, con el objetivo de maximizar su rendimiento. Los hiperparámetros son parámetros del modelo que no se aprenden automáticamente durante el entrenamiento y que deben ser establecidos por el usuario antes de entrenar el modelo. Algunos ejemplos de hiperparámetros son la tasa de aprendizaje, la profundidad máxima del árbol de decisiones, el número de estimadores en un modelo de bosques aleatorios, entre otros.

_ El proceso de Grid Search consiste en definir una cuadrícula de valores posibles para cada hiperparámetro, y luego entrenar y evaluar el modelo para cada combinación de valores de hiperparámetros. Al final del proceso, se selecciona la combinación de hiperparámetros que obtuvo el mejor rendimiento en la evaluación.

_ Grid Search es una técnica efectiva para optimizar hiperparámetros, pero puede ser computacionalmente costosa si se tienen muchos hiperparámetros o muchos valores posibles para cada hiperparámetro. Por lo tanto, es importante seleccionar cuidadosamente los hiperparámetros y valores a incluir en la cuadrícula, y se pueden utilizar técnicas como la búsqueda aleatoria o la búsqueda bayesiana para hacer el proceso más eficiente.

_ Una opción sería ajustar manualmente los hiperparámetros, hasta que encuentre una gran combinación de valores de hiperparámetros. Este sería un trabajo muy tedioso, y es posible que no tenga tiempo para explorar muchas combinaciones. En su lugar, debería utilizar GridSearchCV de Scikit-Learn para buscar por nosotros. Lo único que debemos hacer es indicar qué hiperparámetros deseamos experimentar y qué valores probar, y utilizará la validación cruzada para evaluar todas las posibles combinaciones de valores de hiperparámetros. Por ejemplo, el siguiente código busca la mejor combinación de valores de hiperparámetros para RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3,
4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

- Este código utiliza la función GridSearchCV de la biblioteca sklearn.model_selection para buscar los mejores hiperparámetros para un modelo de regresión de bosques aleatorios (RandomForestRegressor). GridSearchCV toma un modelo, un diccionario de hiperparámetros y una medida de rendimiento como entrada, y luego busca a través de todas las combinaciones de hiperparámetros posibles para encontrar los mejores hiperparámetros que maximizan la medida de rendimiento dada.
- En este ejemplo, se definen dos conjuntos de hiperparámetros para n_estimators y max_features, y se crea un diccionario de parámetros param_grid con estos conjuntos. Además, se definen algunos parámetros adicionales como bootstrap y cv.
- Luego, se crea un modelo RandomForestRegressor y se utiliza GridSearchCV para ajustar este modelo a los datos de entrenamiento (housing_prepared y housing_labels) utilizando los hiperparámetros definidos en param_grid. Se utiliza una validación cruzada de 5 pliegues (cv=5) y la métrica de evaluación es el error cuadrático medio negativo (neg_mean_squared_error).
- Finalmente, el modelo ajustado se almacena en la variable grid_search.

_ Este param_grid le indica a Scikit-Learn que primero evalúe todas las 3x4=12 combinaciones de valores de hiperparámetros n_estimators y max_features especificados en el primer diccionario, luego pruebe las 2x3=6 combinaciones de valores de hiperparámetros en el segundo diccionario, pero esta vez con el hiperparámetro bootstrap establecido en False en lugar de True (que es el valor predeterminado para este hiperparámetro). La búsqueda en cuadrícula explorará 12 + 6 = 18 combinaciones de valores de hiperparámetros RandomForestRegressor, y entrenará cada modelo 5 veces (ya que estamos usando validación cruzada de cinco pliegues). En otras palabras, en total habrá 18x5=90 rondas de entrenamiento. Cuando termine la ejecución, podemos obtener la mejor combinación de parámetros de esta manera:

```
grid_search.best_params_
```

- grid_search.best_params_ es un atributo del objeto GridSearchCV que devuelve un diccionario con los mejores valores de los hiperparámetros que se encontraron durante la búsqueda de cuadrícula.
- Después de que se haya completado la búsqueda de cuadrícula, este atributo se puede usar para recuperar los mejores hiperparámetros encontrados. Por ejemplo, si param_grid incluía un diccionario con el parámetro max_features con valores [2, 4, 6, 8], y la mejor combinación encontrada fue con max_features=6, grid_search.best_params_ devolverá {'max_features': 6} junto con los mejores valores para cualquier otro parámetro que se haya incluido en param_grid.

_ Se puede obtener directamente el mejor estimador (es decir, el modelo con los mejores hiperparámetros encontrados durante la búsqueda en la cuadrícula) mediante:

```
grid_search.best_estimator_
```

- grid_search.best_estimator_ es un atributo del objeto GridSearchCV que retorna el mejor estimador encontrado durante la búsqueda de la rejilla. En este caso, como se ha utilizado RandomForestRegressor() como estimador base, grid_search.best_estimator_ retorna el mejor modelo de bosque aleatorio encontrado durante la búsqueda de la rejilla, que ha sido entrenado en todo el conjunto de datos (ya que GridSearchCV se ajusta automáticamente al mejor conjunto de hiperparámetros encontrados en todo el conjunto de datos).

_ Y por supuesto, también están disponibles las puntuaciones de evaluación.

```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

- Este código utiliza el objeto grid_search creado anteriormente para acceder a los resultados de la búsqueda en la rejilla (grid search) que se acaba de realizar.
- Primero, se almacenan todos los resultados de la búsqueda en la rejilla en la variable cvres usando el atributo cv_results_ del objeto grid_search.
- Luego, se itera a través de los resultados y se imprimen en pantalla la raíz cuadrada del opuesto del puntaje medio de prueba (np.sqrt(-mean_score)) y los valores de los parámetros correspondientes (params) para cada combinación de hiperparámetros. El uso de -mean_score en la fórmula se debe a que la búsqueda en la rejilla se optimizó en función del error cuadrático medio negativo (neg_mean_squared_error). Al negar los valores, el error cuadrático medio ahora es positivo y se puede calcular la raíz cuadrada.
- De esta manera, se puede ver la puntuación de evaluación de cada combinación de hiperparámetros en una lista ordenada, lo que ayuda a identificar la combinación de hiperparámetros que produjo el mejor modelo.

_ Como resultado se observan las combinaciones de hiperparámetros evaluadas por la búsqueda en cuadrícula (GridSearchCV) y el puntaje de evaluación (RMSE) de cada combinación. El puntaje de evaluación se muestra como la raíz cuadrada del negativo del puntaje de la función de pérdida (mean_test_score), que es el opuesto del puntaje de validación cruzada promedio de los modelos en cada combinación de hiperparámetros. Por lo tanto, se observan los hiperparámetros y el rendimiento correspondiente de cada modelo en las diferentes combinaciones evaluadas.

Búsqueda aleatoria

_ Esta es una técnica de optimización de hiperparámetros que se utiliza para encontrar los mejores valores de los hiperparámetros de un modelo de aprendizaje automático. A diferencia de la búsqueda en cuadrícula (Grid Search), la búsqueda aleatoria no evalúa todos los posibles valores de los hiperparámetros, sino que evalúa un número aleatorio de combinaciones de valores. En lugar de especificar explícitamente todos los valores posibles para cada hiperparámetro, el usuario define una distribución de probabilidad para cada uno. Luego, se eligen valores aleatorios de cada distribución para crear una combinación aleatoria de valores de hiperparámetros, que luego se utiliza para entrenar y evaluar el modelo. Este proceso se repite un número determinado de veces (definido por el usuario) y los resultados se comparan para encontrar la mejor combinación de hiperparámetros. La búsqueda aleatoria es útil cuando hay muchos hiperparámetros para ajustar y la búsqueda en cuadrícula tardaría demasiado tiempo en evaluar todas las combinaciones posibles.

_ A continuación tenemos un ejemplo:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

- Este código muestra un ejemplo de cómo hacer una búsqueda aleatoria de hiperparámetros usando la clase RandomizedSearchCV de Scikit-Learn. En este caso, se está buscando la mejor combinación de hiperparámetros para un RandomForestRegressor.
- El diccionario param_distribs especifica las distribuciones de probabilidad para cada hiperparámetro que se desea ajustar. En este ejemplo, se especifica que n_estimators debe ser un número entero entre 1 y 200, y que max_features debe ser un número entero entre 1 y 8.
- Se crea una instancia de RandomForestRegressor y otra de RandomizedSearchCV. La clase RandomizedSearchCV realiza una búsqueda aleatoria en lugar de una búsqueda en cuadrícula. Los hiperparámetros de esta clase incluyen param_distributions, que toma el diccionario param_distribs como argumento, n_iter, que especifica el número de combinaciones de hiperparámetros que se probarán, cv, que especifica el número de pliegues para la validación cruzada, y scoring, que especifica la métrica de evaluación.

- Finalmente, se llama al método fit de la instancia de RandomizedSearchCV para realizar la búsqueda aleatoria de hiperparámetros en el conjunto de datos housing_prepared con las etiquetas housing_labels.

RandomizedSearchCV: es una técnica de búsqueda de hiperparámetros en el aprendizaje automático que, al igual que GridSearchCV, busca el conjunto óptimo de hiperparámetros para un modelo dado. Sin embargo, en lugar de buscar exhaustivamente todas las combinaciones posibles, RandomizedSearchCV realiza una búsqueda aleatoria de un número predefinido de combinaciones de hiperparámetros. Este enfoque puede ser útil cuando el espacio de búsqueda de hiperparámetros es grande y exhaustivamente buscar todas las combinaciones no es práctico en términos de tiempo y recursos. RandomizedSearchCV también utiliza la validación cruzada para evaluar la calidad del modelo para cada combinación de hiperparámetros.

_ A continuación, vemos los resultados mediante:

```
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

- Este código se utiliza para imprimir los resultados del proceso de búsqueda aleatoria (RandomizedSearchCV). Se obtienen los resultados del objeto cv_results_ de la búsqueda aleatoria (rnd_search), que contiene información sobre los puntajes y parámetros para cada combinación de parámetros probados.
- Luego, con un bucle for, se itera a través de cada puntuación media (mean_test_score) y los parámetros correspondientes (params) para imprimirlos. Dentro del bucle, se utiliza np.sqrt(-mean_score) para calcular la puntuación RMSE (error cuadrático medio) para cada modelo.
- En resumen, se están mostrando los resultados de la evaluación de todas las combinaciones de hiperparámetros que se especificaron en la búsqueda aleatoria, ordenados por su desempeño en términos de la métrica de evaluación (en este caso, la raíz cuadrada del error cuadrático medio negativo).

Evaluar el sistema en el test set

_ Después de haber entrenado y ajustado el modelo de aprendizaje automático en los datos de entrenamiento y validado el modelo con los datos de validación, es importante evaluar el desempeño del modelo en un conjunto de datos completamente nuevo que nunca ha visto antes. Para hacer esto, se utiliza un conjunto de prueba que se reserva desde el principio y no se utiliza para entrenar ni ajustar el modelo. El proceso de evaluación del modelo en el conjunto de prueba es sencillo. Se hace una predicción en cada instancia del conjunto de prueba utilizando el modelo entrenado y se compara la predicción con el valor real correspondiente. Luego, se calcula una

medida de error para todas las predicciones, como el error cuadrático medio (MSE) o la raíz del error cuadrático medio (RMSE).

_ Es importante tener en cuenta que el modelo no debe ser ajustado nuevamente en el conjunto de prueba, ya que esto puede hacer que el modelo se sobreajuste a los datos de prueba y que no sea generalizable a nuevos datos. El conjunto de prueba solo se utiliza para evaluar la capacidad del modelo para generalizar a nuevos datos.

_ Entonces, después de ajustar nuestros modelos por un tiempo, eventualmente tendremos un sistema que funciona lo suficientemente bien. Ahora es el momento de evaluar el modelo final en el conjunto de prueba. No hay nada especial en este proceso; solo obtén los predictores y las etiquetas de tu conjunto de prueba, ejecuta tu full_pipeline para transformar los datos (llama a transform(), no fit_transform() - no quieres ajustar el conjunto de prueba!), y evalúa el modelo final en el conjunto de prueba:

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,730.2
```

- En este código, primero se asigna el mejor estimador encontrado por la búsqueda en cuadrícula (grid_search.best_estimator_) a la variable final_model.
- Luego, se divide el conjunto de datos de prueba en predictores X_test y etiquetas y_test, utilizando drop para eliminar la columna objetivo median_house_value del conjunto de datos.
- A continuación, se transforma el conjunto de datos de prueba con el mismo full_pipeline utilizado para el conjunto de datos de entrenamiento, utilizando transform() y no fit_transform() para evitar ajustar el conjunto de datos de prueba.
- Luego, se realizan predicciones sobre el conjunto de datos de prueba utilizando el modelo final_model, que son asignadas a la variable final_predictions.
- Finalmente, se calcula el error cuadrático medio entre las etiquetas reales y las predicciones (mean_squared_error(y_test, final_predictions)), y se calcula la raíz cuadrada de este error cuadrático medio (np.sqrt(final_mse)) para obtener la raíz del error cuadrático medio o RMSE.
- El valor del RMSE obtenido, 47,730.2, es una medida de la calidad de las predicciones del modelo en el conjunto de datos de prueba.

_ En algunos casos, una estimación puntual del error de generalización no será suficiente para convencerte de lanzar, ya que si es solo un 0.1% mejor que el modelo actualmente en producción, es posible que queramos tener una idea de cuán precisa es esta estimación. Para ello, podemos calcular un intervalo de confianza del 95% para el error de generalización utilizando scipy.stats.t.interval():

```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

- Este código calcula un intervalo de confianza del 95% para el error de generalización de un modelo de aprendizaje automático.
- Primero, se establece la variable "confidence" en 0.95, lo que significa que se desea un intervalo de confianza del 95%.
- Luego, se calculan los errores cuadráticos del modelo en el conjunto de prueba. Esto se hace restando las predicciones del modelo (almacenadas en "final_predictions") de las etiquetas reales del conjunto de prueba (almacenadas en "y_test"), elevando al cuadrado la diferencia y almacenando el resultado en "squared_errors".
- Finalmente, se utiliza la función stats.t.interval de la librería scipy para calcular el intervalo de confianza del 95% para el error de generalización. Se le pasan los siguientes argumentos: el nivel de confianza, el número de grados de libertad (en este caso, la longitud de los errores cuadráticos menos uno), la media de los errores cuadráticos y el error estándar de la media de los errores cuadráticos (calculado mediante la función stats.sem).
- El resultado es un array que contiene los límites inferior y superior del intervalo de confianza del 95% para el error de generalización.

_ Si se realizaron muchos ajustes de hiperparámetros, el rendimiento generalmente será ligeramente peor que lo que mediste utilizando la validación cruzada (porque el sistema termina ajustado para rendir bien en los datos de validación y es probable que no se desempeñe tan bien en conjuntos de datos desconocidos). No es el caso en este ejemplo, pero cuando esto suceda, debemos resistir la tentación de ajustar los hiperparámetros para que los números luzcan bien en el conjunto de prueba; las mejoras es poco probable que se generalicen a nuevos datos.

Clasificación

Dataset MNIST

_ MNIST se enfoca en el reconocimiento de dígitos escritos a mano. Es un conjunto de datos muy utilizado para entrenar y evaluar modelos de aprendizaje automático. El conjunto de datos MNIST contiene 70,000 imágenes de dígitos escritos a mano, que han sido preprocesadas y normalizadas para facilitar el uso en modelos de aprendizaje automático. Cada imagen es en escala de grises y tiene una resolución de 28 x 28 píxeles. Los dígitos van del 0 al 9 y hay 10 clases en total.

_ El objetivo es entrenar un modelo de clasificación que pueda identificar correctamente el dígito en una imagen de entrada. Este modelo puede ser utilizado en diversas aplicaciones, como sistemas de reconocimiento de escritura a mano o sistemas de reconocimiento de caracteres en OCR.

OCR (Reconocimiento Óptico de Caracteres): se refiere al proceso de identificar caracteres individuales en una imagen digitalizada y convertirlos en texto editable. En el contexto de clasificación, OCR se utiliza para reconocer y clasificar caracteres en imágenes de texto para su posterior procesamiento.

_ El siguiente código descarga el conjunto de datos MNIST:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

- Este código descarga y carga el conjunto de datos MNIST en una variable llamada mnist, utilizando la función fetch_openml de Scikit-Learn.
- El argumento 'mnist_784' indica el nombre del conjunto de datos a descargar, que contiene 70,000 imágenes de dígitos escritos a mano por estudiantes de secundaria y empleados del Censo de EE. UU., cada imagen con una resolución de 28x28 píxeles. La versión del conjunto de datos se especifica con el argumento version=1.

- La función fetch_openml devuelve un diccionario que contiene los datos cargados, incluidos los datos de imagen y las etiquetas. Al llamar al método keys() de la variable mnist, se imprimen las claves del diccionario, que incluyen data, target, frame, feature_names, target_names, DESCR, y otros.

_ Los conjuntos de datos cargados por Scikit-Learn generalmente tienen una estructura de diccionario similar, que incluye lo siguiente:

- Una clave DESCR que describe el conjunto de datos.
- Una clave data que contiene una matriz con una fila por instancia y una columna por característica.
- Una clave target que contiene una matriz con las etiquetas.

_ Vemos las matrices:

```
X, y = mnist["data"], mnist["target"]
X.shape
y.shape
```

- Este código carga el conjunto de datos MNIST utilizando el diccionario devuelto por la función fetch_openml de Scikit-Learn. Luego, se almacena la matriz de características en X y la matriz de etiquetas en y.
- Finalmente, se imprimen las formas de X y y. Vemos que X es una matriz con 70000 filas y 784 columnas, donde cada fila representa una imagen de 28x28 píxeles (cada píxel se representa como un valor de intensidad de 0 a 255). La matriz y es un vector unidimensional con 70000 elementos, que contiene las etiquetas de las imágenes, es decir, los dígitos que representan (0, 1, 2, ..., 9).

_ Lo que necesitamos hacer es obtener el vector de características de una instancia, remodelarlo en una matriz de 28 × 28 y mostrarlo usando la función imshow() de Matplotlib.

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```

- Este código importa las librerías de matplotlib y pyplot para graficar imágenes. Luego, selecciona la primera imagen de los datos MNIST, que se encuentra almacenada en la variable X, y la almacena en la variable some_digit. Después, se redimensiona la imagen some_digit en una matriz de 28 x 28 píxeles, que es lo que corresponde a su tamaño original.

- Por último, se utiliza la función imshow de Matplotlib para mostrar la imagen some_digit_image. La opción "binary" se utiliza para que se muestre la imagen en blanco y negro. La función axis se utiliza para ocultar los ejes de coordenadas en el gráfico y la función show para mostrar el gráfico en la pantalla.
- En resumen, este código muestra la imagen de un dígito MNIST en una ventana emergente utilizando Matplotlib.

_ Si ejecutamos la siguiente línea vemos que Esto parece un 5, y en efecto es lo que la etiqueta nos dice:

```
y[0]
```

- El código y[0] accede al primer elemento del arreglo y.
- En el contexto de la clasificación de dígitos MNIST, y es un arreglo que contiene las etiquetas de las imágenes. Cada etiqueta corresponde a un número entero del 0 al 9 que representa el dígito que se muestra en la imagen correspondiente en el arreglo X.
- Entonces, y[0] devuelve la etiqueta del primer dígito en el conjunto de datos.

_ Debemos tener en cuenta que la etiqueta es una cadena de texto. La mayoría de los algoritmos de ML esperan números, así que vamos a convertir y a entero.

```
y = y.astype(np.uint8)
```

- El código y = y.astype(np.uint8) convierte el tipo de datos de la variable y a un entero sin signo de 8 bits (unsigned 8-bit integer). La función astype() es un método de la clase NumPy ndarray que permite convertir el tipo de datos de un array a otro tipo de datos especificado como argumento.
- En este caso, y es un array de etiquetas que originalmente fue cargado como un array de cadenas de texto. Para poder utilizar estas etiquetas con la mayoría de los algoritmos de aprendizaje automático, es necesario convertirlas a valores enteros, y en este caso se decidió utilizar un entero sin signo de 8 bits para reducir el espacio de almacenamiento requerido.

_ Siempre debemos crear un conjunto de prueba y apartarlo antes de inspeccionar los datos detalladamente. El conjunto de datos MNIST ya está dividido en un conjunto de entrenamiento (las primeras 60000 imágenes) y un conjunto de prueba (las últimas 10000 imágenes):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000],
y[60000:]
```

- Este código está dividiendo el conjunto de datos MNIST en conjuntos de entrenamiento y prueba. X_train es el conjunto de entrenamiento, que contiene las primeras 60000 imágenes de dígitos escritos a mano, y X_test es el conjunto de prueba, que contiene las últimas 10000 imágenes.

- El mismo proceso se sigue para dividir las etiquetas en conjuntos de entrenamiento y prueba. y_train es el conjunto de etiquetas de entrenamiento que corresponden a X_train, y y_test es el conjunto de etiquetas de prueba que corresponden a X_test.
- La sintaxis utilizada en este código es conocida como "slicing" en Python, que permite seleccionar un subconjunto de elementos de una matriz. En este caso, los primeros 60,000 elementos de X se seleccionan como el conjunto de entrenamiento, y los últimos 10,000 elementos se seleccionan como el conjunto de prueba. De manera similar, los primeros 60,000 elementos de y se seleccionan como las etiquetas de entrenamiento, y los últimos 10,000 elementos se seleccionan como las etiquetas de prueba.

_ Tenemos en cuenta que también existe esta forma:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)
```

- Este código divide los datos de entrada y los objetivos en conjuntos de entrenamiento y prueba para su uso en el modelo.
- X y y son los datos de entrada y los objetivos respectivamente.
- test_size especifica el tamaño de la muestra de prueba, en este caso 0.15 (o el 15% de los datos se reservarán para la prueba).
- random_state se utiliza para fijar la semilla de generación de números aleatorios, lo que garantiza que la división de los datos sea siempre la misma en cada ejecución del código.

_ Ambos fragmentos de código dividen un conjunto de datos en conjuntos de entrenamiento y prueba, pero difieren en cómo se realiza esa división. La primera línea de código divide los datos manualmente en un 60% de entrenamiento y un 40% de prueba. Se utilizan las primeras 60,000 instancias para entrenamiento y las restantes 10,000 instancias para pruebas. La segunda línea de código utiliza la función train_test_split de Scikit-Learn para dividir aleatoriamente los datos en conjuntos de entrenamiento y prueba. En este caso, se ha especificado que el tamaño de la prueba sea el 15% del tamaño total de los datos, y se ha fijado una semilla aleatoria (random_state=42) para asegurar que la división aleatoria sea reproducible.

_ Ambos enfoques pueden ser útiles en diferentes situaciones. La división manual puede ser útil si se desea tener un control preciso sobre la composición de los conjuntos de entrenamiento y prueba, por ejemplo, si se está trabajando con un conjunto de datos desequilibrado y se desea asegurarse de que ambas partes contengan una cantidad suficiente de instancias de cada clase. Por otro lado, el uso de train_test_split puede ser más conveniente en general, ya que permite dividir los datos de forma aleatoria y reproducible. Además, se pueden especificar otras opciones, como la proporción de datos utilizados para entrenamiento y prueba, y la estratificación para mantener la proporción de clases en ambas partes.

_ Para el caso que estamos trabajando, también es importante el escalado. Si no normalizamos los datos, puede haber algunos problemas al entrenar nuestro modelo de aprendizaje automático. Primero, algunos algoritmos son sensibles a la escala de los datos, lo que significa que la diferencia en las escalas de los atributos puede afectar la calidad del modelo. Por ejemplo, los algoritmos de regresión logística y SVM pueden verse afectados por la escala de los datos. En segundo lugar, normalizar los datos puede mejorar la velocidad de entrenamiento del modelo. Cuando los datos están en una escala similar, puede ser más fácil para el modelo encontrar la solución óptima más rápidamente. Por lo tanto, normalizar los datos puede mejorar la precisión de tus modelos y, en algunos casos, acelerar el tiempo de entrenamiento del modelo.

_ Para este caso ejecutamos:

```
from sklearn.preprocessing import MinMaxScaler

MinMaxScaler()
scaler = MinMaxScaler()
print(scaler.fit(X_train))
print(scaler.transform(X_train))
```

- Este código utiliza la clase MinMaxScaler de la biblioteca scikit-learn para normalizar los datos de entrenamiento X_train entre un rango específico. La normalización de los datos es una técnica común en el procesamiento de datos que se utiliza para escalar los datos a un rango específico. Esto puede ser necesario si los datos de entrada no están en la misma escala, lo que puede afectar el rendimiento del modelo.
- La primera línea importa la clase MinMaxScaler de sklearn.preprocessing, que se utiliza para escalar los datos entre un rango específico. Luego, se crea una instancia de la clase MinMaxScaler y se almacena en la variable scaler. La función fit() se llama en la instancia scaler y se pasa el conjunto de datos X_train. Esta función calcula la media y la desviación estándar de X_train y las almacena en la instancia scaler. Luego, se llama a la función transform() en la instancia scaler y se pasa el conjunto de datos X_train. Esta función normaliza los datos en X_train entre un rango específico utilizando la media y la desviación estándar previamente calculadas.
- En resumen, este código normaliza los datos de entrenamiento X_train utilizando la clase MinMaxScaler de sklearn.preprocessing.

Entrenamiento de un clasificador binario

Clasificador Binario

_ El entrenamiento de un clasificador binario se refiere a la tarea de enseñar a un modelo a clasificar instancias en dos clases diferentes. En el caso del conjunto de datos MNIST, podemos utilizar un clasificador binario para detectar si una imagen representa un número específico o no. El proceso de entrenamiento del clasificador binario implica seleccionar un algoritmo de aprendizaje de máquina adecuado y entrenarlo en el conjunto de entrenamiento. Para el problema de clasificación binaria, una opción popular es el clasificador de descenso de gradiente estocástico (SGD).

_ En términos generales, el algoritmo SGD ajusta los pesos del modelo de forma iterativa utilizando un subconjunto de los datos de entrenamiento en cada iteración. Durante cada iteración, el modelo calcula la diferencia entre las predicciones y las etiquetas reales, y ajusta los pesos para minimizar esta diferencia.

_ El proceso de entrenamiento implica iterar sobre todo el conjunto de entrenamiento varias veces (épocas) hasta que el modelo haya aprendido a clasificar correctamente las instancias del conjunto de entrenamiento. Luego, se evalúa la precisión del modelo en el conjunto de prueba para asegurarse de que no haya sobreajuste.

Clasificador binario: es un modelo de aprendizaje automático que se entrena para distinguir entre dos clases diferentes. Tiene como objetivo predecir una variable objetivo binaria, es decir, una variable que puede tomar solo dos valores posibles, como verdadero/falso, sí/no, 0/1, etc. En general, el clasificador binario toma un conjunto de características de entrada y produce una predicción binaria como salida, basada en un modelo entrenado previamente. En este caso, se trata de entrenar un modelo para distinguir entre dos dígitos específicos en el conjunto de datos MNIST, el número 5 y cualquier otro número. Es decir, el modelo debe clasificar cada imagen como "es un 5" o "no es un 5".

_ A continuación creamos los vectores de destino para esta tarea de clasificación:

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

- Este código se utiliza para crear dos vectores de destino (target vectors) binarios para la tarea de clasificación que estamos abordando. En este caso, queremos clasificar imágenes como "5" o "no 5".
- El primer vector, y_train_5, se crea a partir del vector de destino original y_train, donde se establece en True para todas las instancias cuyo objetivo es "5" y en False para todas las demás instancias. De esta manera, el vector y_train_5 contendrá solo valores binarios, True o False.
- El segundo vector, y_test_5, se crea de manera similar, pero a partir del vector de destino y_test, que contiene las etiquetas de las imágenes de prueba.

_ Ahora elegimos un clasificador y lo entrenamos. Un buen punto de partida es utilizar un clasificador de descenso de gradiente estocástico (SGD), utilizando la clase SGDClassifier de Scikit-Learn. Este clasificador tiene la ventaja de ser capaz de manejar conjuntos de datos muy grandes de manera eficiente. Esto se debe en parte a que SGD maneja las instancias de entrenamiento de manera independiente, una a la vez (lo que también hace que SGD sea adecuado para el aprendizaje en línea), como veremos más adelante. Entonces creamos un SGDClassifier y lo entrenamos en todo el conjunto de entrenamiento:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

- Este código importa la clase SGDClassifier del módulo linear_model de la librería Scikit-Learn. Luego se crea una instancia de SGDClassifier llamada sgd_clf con el parámetro random_state establecido en 42. El parámetro random_state asegura que los resultados sean reproducibles en diferentes ejecuciones del mismo código.
- Posteriormente se entrena el modelo sgd_clf en el conjunto de entrenamiento (X_train e y_train_5) utilizando el método fit(). Este método ajusta el modelo a los datos de entrenamiento, encontrando los parámetros del modelo que mejor se ajusten a los datos. En este caso, se está entrenando el modelo para clasificar imágenes como 5s o no-5s.

_ Ahora podemos usarlo para detectar imágenes del número 5:

```
sgd_clf.predict([some_digit])
```

- Este código utiliza el modelo de clasificación binaria SGDClassifier entrenado previamente para predecir si una imagen específica (representada por some_digit) corresponde al número 5 o no.
- La función predict() del modelo toma una matriz de características (en este caso, una sola instancia de imagen) como entrada y devuelve una matriz de etiquetas correspondientes a cada instancia. En este caso, la salida será un array con un solo valor booleano que indica si la imagen se clasifica como un 5 o no.

_ Por ende, el clasificador adivina que la imagen mostrada representa un 5 (True).

Medición de Rendimiento

_ Las medidas de rendimiento se refieren a cómo se evalúa la eficacia de un modelo de aprendizaje automático en términos de su capacidad para realizar predicciones precisas en nuevos datos. Para evaluar el rendimiento de un modelo, se utilizan diferentes medidas, que dependen del tipo de problema de aprendizaje (clasificación o regresión) y del tipo de datos en cuestión.

_ En el caso de un problema de clasificación binaria, las medidas más comunes son la precisión (accuracy), la sensibilidad (recall) y la especificidad (specificity). La precisión mide la proporción de predicciones correctas realizadas por el modelo en relación con el total de predicciones realizadas. La sensibilidad mide la proporción de casos positivos (ejemplos de la clase que se desea predecir) que el modelo predijo correctamente, mientras que la especificidad mide la proporción de casos negativos (ejemplos de otras clases) que el modelo predijo correctamente.

_ Estas medidas a menudo se complementan con la matriz de confusión, que muestra el número de predicciones correctas e incorrectas realizadas por el modelo en cada clase. La matriz de confusión es especialmente útil para evaluar el rendimiento del modelo en relación con clases desequilibradas, es decir, cuando hay muchas más instancias de una clase que de otra.

_ Una parte importante del flujo de trabajo de Machine Learning es la evaluación del desempeño del modelo. En el mundo real, por lo general, es imposible tener un clasificador perfecto. Debemos entender el problema y entender que es lo importante del mismo, por ejemplo:

- Si estamos tratando de determinar si un tumor es maligno o benigno, nos interesa que la predicción incorrecta de que un tumor es maligno sea lo más pequeña posible.
- Si estamos tratando de determinar que transacciones son fraudulentas, podríamos estar interesados en perder la menor cantidad posible de este tipo de transacciones.

Midiendo “Accuracy” usando validación cruzada

_ Acá vemos cómo evaluar el rendimiento de un modelo de aprendizaje automático utilizando validación cruzada.

Validación cruzada: es una técnica que implica dividir el conjunto de datos en varios subconjuntos o "folds" y entrenar el modelo en algunos subconjuntos mientras se prueba en otros. De esta manera, se pueden obtener varias medidas de rendimiento y reducir la posibilidad de sobreajuste del modelo.

_ Aquí se detalla cómo se puede utilizar la validación cruzada para medir la precisión de un clasificador binario. Se explica cómo se divide el conjunto de datos en "folds", cómo se entrena el modelo en cada iteración y cómo se evalúa la precisión del modelo

utilizando la matriz de confusión. También se menciona la importancia de tener en cuenta la distribución de las clases en los subconjuntos y cómo se puede utilizar la estratificación para garantizar que cada subconjunto contenga una proporción similar de muestras positivas y negativas.

_ En resumen, se aborda una técnica importante para evaluar el rendimiento de un modelo de aprendizaje automático de manera robusta y precisa, lo que es esencial para garantizar que el modelo pueda generalizar bien a nuevos datos.

_ Se define accuracy como:

$$accuracy = \frac{\text{predicciones correctas}}{\text{total de predicciones}}$$

_ Un modelo con un alto accuracy (exactitud) puede no ser un buen modelo por diversas razones, por ejemplo:

- El modelo puede estar sobreajustado (overfitting) a los datos de entrenamiento y por lo tanto no generalizar bien a nuevos datos. En este caso, aunque el modelo tenga un alto accuracy en los datos de entrenamiento, su precisión en nuevos datos puede ser baja.
- El modelo puede estar infravalorando una clase minoritaria, es decir, la clase menos frecuente en el conjunto de datos. Esto puede suceder cuando las clases están desequilibradas y el modelo se entrena con una gran cantidad de muestras de una clase y muy pocas muestras de la otra clase. En este caso, el accuracy puede ser alto, pero el modelo no está haciendo un buen trabajo en la identificación de la clase minoritaria.
- El modelo puede estar haciendo predicciones aleatorias o siguiendo patrones espurios en los datos de entrenamiento, lo que lleva a un alto accuracy en los datos de entrenamiento pero una baja precisión en nuevos datos.

_ En resumen, el accuracy por sí solo no es una métrica suficiente para evaluar la calidad de un modelo. Se deben considerar otras métricas como la precisión, la recall, la F1-score, etc., así como realizar pruebas en datos de validación o de prueba para asegurar que el modelo generalice bien a nuevos datos.

_ Podemos usar la función cross_val_score() para evaluar nuestro modelo SGDClassifier, utilizando validación cruzada K-fold con tres pliegues. La validación cruzada K-fold significa dividir el conjunto de entrenamiento en K pliegues (en este caso, tres), luego hacer predicciones y evaluarlas en cada pliegue utilizando un modelo entrenado en los pliegues restantes.

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
```

- Este código utiliza la función cross_val_score de la librería Scikit-Learn para evaluar el rendimiento del modelo sgd_clf mediante validación cruzada (cross-

validation) con tres pliegues (folds). Se le proporciona el conjunto de entrenamiento X_train y las etiquetas de entrenamiento y_train_5 del problema de clasificación binaria de detectar números 5 en el conjunto MNIST.

- La función cross_val_score devuelve un array con los puntajes de cada fold de la validación cruzada, calculados mediante la métrica de "exactitud" (accuracy), que es la proporción de instancias clasificadas correctamente.

_ Vemos que el accuracy de un resultado por encima del 93%. Utilizando un clasificador que siempre predice la clase más frecuente (en este caso no 5):

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

_ Observamos la precisión de este modelo:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
```

- El código crea una instancia de la clase Never5Classifier, que es un clasificador que siempre predice "no es un 5". Luego, utiliza la función cross_val_score de la biblioteca sklearn para calcular el puntaje de precisión del clasificador.
- La función cross_val_score divide los datos de entrenamiento en "folds" y entrena el modelo en cada combinación de "folds", lo que permite una evaluación más precisa del modelo. En este caso, se especifica cv = 3, lo que significa que la función cross_val_score utilizará validación cruzada de 3 veces.
- La puntuación se calcula utilizando la métrica de precisión, que compara la cantidad de predicciones correctas con el número total de predicciones.

_ El resultado nos demuestra que tiene más del 90% de precisión. Esto se debe simplemente a que solo alrededor del 10% de las imágenes son 5, por lo que si siempre supone que una imagen no es un 5, tendrá razón alrededor del 90% del tiempo. Esto demuestra porqué la precisión o accuracy generalmente no es la medida de rendimiento preferida para los clasificadores, especialmente cuando se trata de conjuntos de datos sesgados (es decir, cuando algunas clases son mucho más frecuentes que otras).

_ El siguiente código hace aproximadamente lo mismo que la función cross_val_score() de Scikit-Learn, e imprime el mismo resultado, mediante la clase StratifiedKFold que realiza un muestreo estratificado para producir pliegues que contienen una proporción representativa de cada clase:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.9502, 0.96565, and
0.96495
```

- Este código implementa la validación cruzada utilizando la clase StratifiedKFold de Scikit-Learn.
- En la primera línea se importa la clase StratifiedKFold. En la segunda línea se importa la función clone desde la clase base de Scikit-Learn para hacer una copia del clasificador para cada iteración de la validación cruzada. Se instancia un objeto skfolds de la clase StratifiedKFold con n_splits=3, lo que significa que se dividirá el conjunto de entrenamiento en 3 pliegues.
- Se itera a través de cada pliegue utilizando el método split() de skfolds. Dentro del ciclo for se crea una copia del clasificador con la función clone(), se asignan los datos de entrenamiento y de prueba para el pliegue actual y se entrena el clasificador en los datos de entrenamiento.
- A continuación, se hacen predicciones en los datos de prueba y se calcula el número de predicciones correctas.
- Finalmente, se imprime la tasa de aciertos para cada pliegue de la validación cruzada.

Matriz de confusión

_ La matriz de confusión es una herramienta que se utiliza para evaluar el rendimiento de un modelo de clasificación. Básicamente, es una tabla que muestra la cantidad de veces que el modelo predijo correctamente cada clase y la cantidad de veces que predijo incorrectamente cada clase. Una clase se refiere a una categoría o etiqueta en la que se pueden clasificar los datos. Por ejemplo, en un conjunto de datos de imágenes de frutas, las clases podrían ser manzanas, naranjas, plátanos, etc. En el caso de un clasificador binario, solo habría dos clases posibles, positivo y negativo (o 1 y 0).

_ La matriz de confusión se divide en cuatro partes:

- Verdaderos positivos (TP): representan los casos en los que el modelo predijo correctamente que la instancia pertenece a una determinada clase.
- Falsos positivos (FP): son los casos en los que el modelo predijo incorrectamente que la instancia pertenece a una determinada clase.
- Falsos negativos (FN): son los casos en los que el modelo predijo incorrectamente que la instancia no pertenece a una determinada clase.
- Verdaderos negativos (TN): representan los casos en los que el modelo predijo correctamente que la instancia no pertenece a una determinada clase.

_ La matriz de confusión es útil porque permite evaluar la calidad de la clasificación de cada clase. Por ejemplo, si el modelo tiene muchos falsos positivos, es posible que esté clasificando incorrectamente muchas instancias como pertenecientes a una determinada clase, lo que puede ser problemático en ciertas aplicaciones. Por otro lado, si el modelo tiene muchos falsos negativos, es posible que esté perdiendo muchos casos en los que las instancias realmente pertenecen a una determinada clase, lo que también puede ser problemático en ciertas aplicaciones.

_ Para calcular la matriz de confusión, primero necesitamos tener un conjunto de predicciones para que puedan ser comparadas con los objetivos reales. Podríamos hacer predicciones en el conjunto de prueba, pero por ahora lo mantenemos intacto (ya que solo queremos usar el conjunto de prueba al final del proyecto, una vez que tengamos un clasificador listo para lanzar). En su lugar, podemos utilizar la función cross_val_predict():

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

- Este código utiliza la función cross_val_predict de Scikit-Learn para realizar una validación cruzada en el conjunto de entrenamiento X_train con sus etiquetas correspondientes y_train_5 y un clasificador sgd_clf.
- La validación cruzada divide el conjunto de datos en tres pliegues y entrena el modelo en dos de ellos, realizando predicciones en el tercero. Esto se repite tres veces para que cada pliegue actúe como conjunto de pruebas una vez. La función cross_val_predict devuelve las predicciones de todas las instancias en el conjunto de entrenamiento, utilizando el modelo entrenado en cada una de las tres iteraciones de la validación cruzada.
- Las predicciones resultantes se almacenan en y_train_pred.

_ Ahora, para obtener la matriz de confusión usamos la función confusion_matrix(). Simplemente pasamos las clases objetivo (y_train_5) y las clases predichas (y_train_pred):

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

- El código utiliza la función confusion_matrix de la librería sklearn.metrics para obtener la matriz de confusión a partir de las clases reales de los datos de entrenamiento y_train_5 y las predicciones generadas por el modelo y_train_pred.
- La matriz de confusión es una tabla que muestra la cantidad de verdaderos positivos (TP), falsos positivos (FP), verdaderos negativos (TN) y falsos negativos (FN) para cada clase.
- En este caso, la matriz de confusión mostrará la cantidad de veces que el modelo predijo correctamente que una imagen no es un 5 (TN) y la cantidad de veces que predijo incorrectamente que una imagen era un 5 (FP). También mostrará la cantidad de veces que el modelo predijo correctamente que una imagen era un 5 (TP) y la cantidad de veces que predijo incorrectamente que una imagen no era un 5 (FN).

_ Cada fila en una matriz de confusión representa una clase real, mientras que cada columna representa una clase predicha. La primera fila de esta matriz considera imágenes que no son 5, ellas fueron clasificadas correctamente como no-5 (se llaman verdaderos negativos), mientras que las restantes fueron clasificadas erróneamente como 5s (falsos positivos). La segunda fila considera las imágenes de 5s y fueron clasificadas erróneamente como no-5s (falsos negativos), mientras que las restantes 4, fueron clasificadas correctamente como 5s (verdaderos positivos).

_ Un clasificador perfecto tendría solo verdaderos positivos y verdaderos negativos, por lo que su matriz de confusión tendría valores diferentes de cero solo en su diagonal principal (de arriba a la izquierda a abajo a la derecha):

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached
perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
```

- El código asigna a la variable y_train_perfect_predictions los valores de y_train_5. Luego, se usa la función confusion_matrix de Scikit-Learn para generar la matriz de confusión entre y_train_5 y y_train_perfect_predictions.
- Dado que y_train_perfect_predictions contiene los valores correctos para todas las instancias, la matriz de confusión resultante tiene valores no nulos solamente en la diagonal principal, indicando que todos los valores fueron clasificados correctamente, lo que es propio de un clasificador perfecto.

_ La matriz de confusión proporciona mucha información, pero a veces se puede necesitar una métrica más concisa.

Precision y Recall

_ Precision y Recall son dos métricas comúnmente utilizadas para evaluar el desempeño de un modelo de clasificación. A partir de la matriz de confusión puede obtenerse estas métricas.

Precision: se refiere a la proporción de instancias clasificadas como positivas que son verdaderamente positivas (verdaderos positivos) en comparación con el total de instancias clasificadas como positivas (tanto verdaderos positivos como falsos positivos):

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

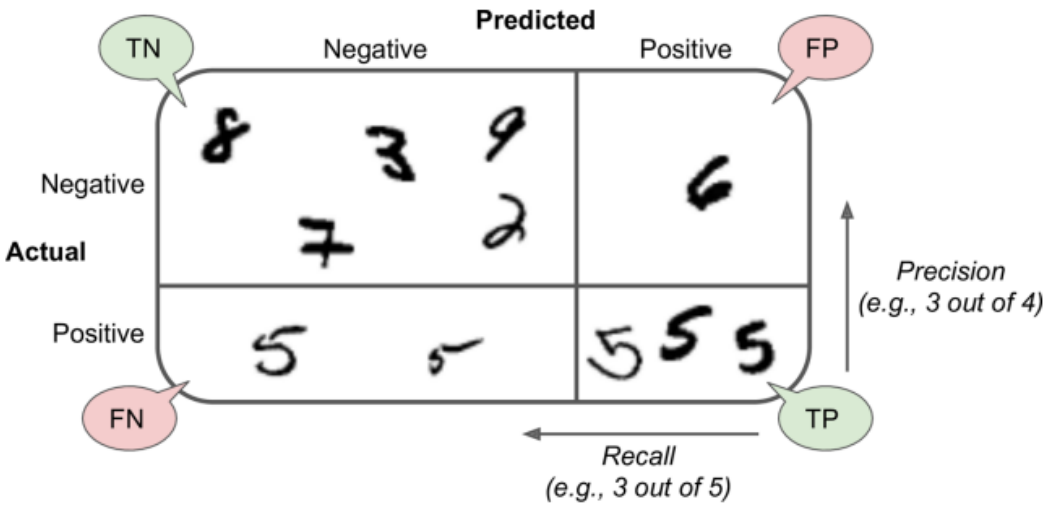
_ La precision es útil cuando se desea minimizar el número de falsos positivos, es decir, minimizar el número de casos en los que el modelo predice que una instancia pertenece a una clase positiva cuando en realidad pertenece a una clase negativa.

Recall: se refiere a la proporción de instancias verdaderamente positivas que fueron correctamente identificadas por el modelo (verdaderos positivos) en comparación con el total de instancias que deberían haber sido clasificadas como positivas (verdaderos positivos y falsos negativos):

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

_ El Recall es útil cuando se desea minimizar el número de falsos negativos, es decir, minimizar el número de casos en los que el modelo predice que una instancia pertenece a una clase negativa cuando en realidad pertenece a una clase positiva.

_ En general, la elección entre precisión y recall depende del problema específico y de las implicaciones de cada tipo de error. Por ejemplo, en un problema médico, puede ser más importante maximizar el recall para asegurarse de que todos los pacientes con una enfermedad sean correctamente identificados, aunque esto signifique un mayor número de falsos positivos. Por otro lado, en un problema de detección de spam, puede ser más importante maximizar la precisión para minimizar el número de correos electrónicos legítimos que se identifican erróneamente como spam, aunque esto signifique un mayor número de falsos negativos.



_ Scikit-Learn provee varias funciones para calcular métricas de clasificadores, incluyendo precisión y recuperación (recall):

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
```

- Este código utiliza las funciones precision_score y recall_score de Scikit-Learn para calcular las métricas de precisión y recall de un clasificador.
- precision_score se utiliza para calcular la precisión del clasificador, es decir, la proporción de verdaderos positivos (TP) entre la suma de verdaderos positivos y falsos positivos (FP). Se llama "precision" porque mide la precisión del clasificador al identificar los verdaderos positivos.
- recall_score se utiliza para calcular el recall del clasificador, es decir, la proporción de verdaderos positivos (TP) entre la suma de verdaderos positivos y falsos negativos (FN). Se llama "recall" porque mide la capacidad del clasificador para "recordar" o detectar los verdaderos positivos.
- En ambos casos, se pasan como parámetros las etiquetas reales (y_train_5) y las etiquetas predichas (y_train_pred) para calcular las métricas. El resultado de cada función es un número que representa la precisión o recall del clasificador.

_ Ahora el detector de 5 no parece tan bueno como parecía cuando se observó su exactitud. Cuando afirma que una imagen representa un 5, solo es correcto el 72,9% del tiempo. Además, solo detecta el 75,6% de los 5s. A menudo es conveniente combinar precision y el recall en una sola métrica llamada F1 score, en particular si necesita una forma sencilla de comparar dos clasificadores.

F1 score: es la media armónica de la precisión y el recall. Mientras que la media regular trata todos los valores por igual, la media armónica da mucho más peso a los valores bajos, y varía entre 0 y 1, donde 1 es la mejor puntuación. Para aumentar la precisión, el clasificador tiene que ser más conservador al asignar positivos. Por el contrario, para aumentar el recall, el clasificador tiene que detectar más instancias positivas reduciendo el umbral de decisión. Este compromiso entre precisión y recall se conoce como la compensación entre precisión y recall. Como resultado, el clasificador solo obtendrá un alto F1 score si tanto el recall como la precisión son altos.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

_ Para calcular el F1 score, simplemente llamamos a la función f1_score():

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

- El código importa la función f1_score de la biblioteca scikit-learn y la aplica a los conjuntos de entrenamiento y predicciones para calcular el puntaje F1, que es una medida de la precisión general del modelo en la clasificación de los datos.
- El primer argumento de la función es el conjunto de valores verdaderos (y_train_5) y el segundo argumento es el conjunto de valores predichos (y_train_pred) por el modelo para el mismo conjunto de datos.
- La puntuación resultante es un valor único que representa la precisión general del modelo en términos de precisión y recall.
- La puntuación F1 es un promedio ponderado de la precisión y recall, y se calcula como la media armónica de estos dos valores.

_ Desafortunadamente, no se puede tener ambas cosas al mismo tiempo, es decir, aumentar la precision disminuye el recall, y viceversa.

_ No necesariamente F1 es la mejor métrica que se puede utilizar al momento de optimizar un modelo. La F1 score es una medida útil para problemas de clasificación binaria cuando se desea equilibrar la precisión y el recall del modelo. Sin embargo, dependiendo del problema, puede haber otras métricas más importantes que deban ser consideradas. Por ejemplo, si el objetivo es minimizar los falsos positivos, la precisión puede ser la métrica más importante. Si el objetivo es minimizar los falsos negativos, el recall puede ser la métrica más importante. Además, es importante considerar otras medidas como la exactitud, el área bajo la curva ROC o el índice J de Youden, dependiendo de las necesidades específicas del problema en cuestión. En resumen, la elección de la métrica adecuada depende del problema específico y de las necesidades y prioridades del usuario.

Precision/recall trade off

_ Acá nos referimos a la relación inversa entre precision y recall de un clasificador binario. La precision se refiere a la proporción de instancias clasificadas como positivas que son realmente positivas, mientras que el recall se refiere a la proporción de instancias positivas que son correctamente identificadas por el clasificador.

- Cuanto más alta sea la precision, menos falsos positivos habrá, es decir, menos instancias negativas serán clasificadas erróneamente como positivas.
- Cuanto más alta sea la exhaustividad, menos falsos negativos habrá, es decir, menos instancias positivas serán clasificadas erróneamente como negativas.

_ En general, aumentar precision reducirá el recall y viceversa. Esto se debe a que al aumentar el umbral de clasificación, se disminuirá la cantidad de verdaderos positivos y aumentará la cantidad de falsos negativos, lo que aumentará el precision y disminuirá el recall. Por otro lado, al disminuir el umbral de clasificación, se aumentará la cantidad de verdaderos positivos y disminuirá la cantidad de falsos negativos, lo que aumentará la exhaustividad y disminuirá la precisión.

_ La elección del umbral de clasificación dependerá del problema en cuestión y de las necesidades del usuario. En algunos casos, puede ser más importante minimizar los falsos positivos, mientras que en otros puede ser más importante minimizar los falsos negativos. Por lo tanto, es importante evaluar las métricas de precisión y exhaustividad para determinar el umbral de clasificación adecuado y lograr un equilibrio óptimo entre ambas.

_ Entonces, si existe un trade-off entre precisión y recall porque al ajustar un modelo de clasificación, es posible mejorar la precisión a expensas del recall o viceversa. Precisión y recall son medidas que describen el rendimiento de un modelo de clasificación. La precisión se refiere a la proporción de verdaderos positivos (TP) entre los casos positivos que el modelo ha predicho, mientras que el recall se refiere a la proporción de verdaderos positivos entre todos los casos positivos reales. Un modelo con alta precisión y baja recall se enfoca en minimizar los falsos positivos, lo que significa que es poco probable que clasifique un caso negativo como positivo, pero puede perder algunos casos positivos. Por otro lado, un modelo con alta recall y baja precisión se enfoca en minimizar los falsos negativos, lo que significa que es poco probable que pierda casos positivos, pero puede clasificar algunos casos negativos como positivos. Por lo tanto, al optimizar un modelo de clasificación, es importante tener en cuenta el trade-off entre precisión y recall, y seleccionar el punto óptimo en función de las necesidades específicas del problema. En algunos casos, es posible que se prefiera una mayor precisión, mientras que en otros se puede preferir un mayor recall, dependiendo de las consecuencias de los errores de clasificación.

_ Scikit-Learn no permite establecer directamente el umbral de decisión, pero sí proporciona acceso a las puntuaciones de decisión que utiliza para realizar las predicciones. En lugar de llamar al método predict() del clasificador, podemos llamar al método decision_function(), que devuelve una puntuación para cada instancia, y luego usar cualquier umbral que desees para realizar las predicciones basadas en esas puntuaciones:

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

- Este código utiliza el clasificador SGD (Stochastic Gradient Descent) previamente entrenado para hacer una predicción en un solo ejemplo, representado por some_digit. En lugar de la función predict(), que devuelve la clase predicha para este ejemplo, se utiliza la función decision_function() del clasificador para obtener los puntajes de decisión para el ejemplo.
- Los puntajes de decisión son valores numéricos que indican cuán positivo es la clasificación del ejemplo como la clase positiva (5 en este caso) en comparación con la clasificación de la clase negativa (no-5 en este caso). Si el puntaje de decisión es positivo, la predicción del modelo será la clase positiva, y si es negativo, será la clase negativa.

- En este código, los puntajes de decisión se calculan para el ejemplo some_digit utilizando la función decision_function() del clasificador entrenado, y se almacenan en la variable y_scores. Luego, estos puntajes de decisión se pueden utilizar para determinar la clase predicha para some_digit utilizando un umbral específico.

_ Y luego:

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
```

- En este código se establece un umbral (threshold) para hacer la predicción de la clase de una instancia de prueba (some_digit) basada en los scores de decisión obtenidos por el clasificador SGD. Si el score de decisión para la instancia de prueba es mayor que el umbral, la predicción será positiva, y si es menor que el umbral, la predicción será negativa.
- En este caso, el umbral se establece en 0 y se utiliza la operación de comparación ">". Por lo tanto, si el score de decisión es mayor que 0, la predicción será positiva, y si es menor o igual que 0, la predicción será negativa. La predicción se guarda en la variable "y_some_digit_pred".

_ El SGDClassifier utiliza un umbral igual a 0, por lo que el código anterior devuelve el mismo resultado que el método predict() (es decir, True). Levantamos el umbral:

```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

- Este código establece un valor de umbral de 8000 y luego aplica la regla del umbral a los puntajes de decisión del modelo (en este caso, y_scores) para predecir si la imagen representada por some_digit es un 5 o no.
- Si el puntaje de decisión es mayor que el umbral, se predice que la imagen es un 5 (True), de lo contrario, se predice que no es un 5 (False). La variable y_some_digit_pred contiene la predicción resultante después de aplicar la regla del umbral.

_ Esto confirma que aumentar el umbral disminuye el recall. La imagen en realidad representa un 5, y el clasificador lo detecta cuando el umbral es 0, pero lo pierde cuando el umbral se aumenta a 8000.

_ Para decidir que umbral usar, primero, usamos la función cross_val_predict() para obtener los puntajes de todas las instancias en el conjunto de entrenamiento, pero esta vez especifique que desea devolver puntajes de decisión en lugar de predicciones:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

- El código calcula las decisiones (scores) del clasificador SGD en lugar de hacer predicciones para cada instancia en el conjunto de entrenamiento utilizando la función cross_val_predict() de Scikit-Learn. En lugar de hacer predicciones binarias, esta función devuelve los valores de score (puntuación) para cada instancia.
- El parámetro method="decision_function" se usa para especificar que se deben devolver los scores de decisión en lugar de las predicciones del clasificador. Los parámetros X_train y y_train_5 corresponden a los datos de entrenamiento de entrada y las etiquetas correspondientes a los 5. cv=3 indica que se está utilizando un esquema de validación cruzada de 3 pliegues.

_ Con estos puntajes, utilizamos la función precision_recall_curve() para calcular la precisión y la recuperación para todos los umbrales posibles. Y finalmente, utilizamos Matplotlib para graficar la precisión y la recuperación como funciones del valor del umbral:

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    [...] # highlight the threshold and add the legend, axis label, and grid

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

- Este código calcula la precisión y el recall para diferentes valores de umbral (threshold) utilizando la función precision_recall_curve de Scikit-Learn. Luego, define una función llamada plot_precision_recall_vs_threshold para graficar las curvas de precisión y recall en función del umbral.
- La función plot_precision_recall_vs_threshold utiliza los valores de precisión, recall y umbral para graficar dos curvas: una para la precisión (en color azul) y otra para el recall (en color verde). La función también resalta el umbral seleccionado y agrega una leyenda, etiquetas de los ejes y una cuadrícula para la visualización.
- Finalmente, se llama a la función plot_precision_recall_vs_threshold con los valores de precisión, recall y umbral calculados previamente y se muestra la gráfica utilizando la función plt.show() de Matplotlib.

_ Podemos ver que el precision comienza a disminuir abruptamente alrededor del 80% de recall. Suponiendo que decidimos apuntar a una precisión del 90%. Miramos la gráfica y encontramos que necesitamos usar un umbral de alrededor de 8000. Para ser más preciso, podemos buscar el umbral más bajo que nos dé al menos el 90% de precisión (np.argmax() te dará el primer índice del valor máximo, lo que en este caso significa el primer valor True).

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] #
~7816
```

- Este código busca el valor de umbral (threshold) que produce una precisión de al menos 0.90, es decir, del 90%.
- La precisión se almacena en el vector "precisions" y los valores de umbral en el vector "thresholds". Al aplicar la función "np.argmax(precisions >= 0.90)", se busca el índice del primer valor en el vector "precisions" que sea mayor o igual a 0.90. Luego, ese índice se utiliza para obtener el umbral correspondiente en el vector "thresholds".

_ Para hacer predicciones (solo en el conjunto de entrenamiento por ahora), en lugar de llamar al método predict() del clasificador, puedes ejecutar este código:

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

- Este código crea un conjunto de predicciones binarias para el conjunto de entrenamiento. Para ello, se compara el puntaje obtenido para cada instancia (almacenado en la variable y_scores) con un umbral determinado (threshold_90_precision) que se ha seleccionado previamente con el objetivo de obtener una precisión de al menos el 90%.
- Si el puntaje para una instancia es mayor o igual al umbral, se asigna el valor True en la predicción para esa instancia, y si es menor, se asigna el valor False.

_ Comprobamos la precision y el recall de estas predicciones:

```
precision_score(y_train_5, y_train_pred_90)

recall_score(y_train_5, y_train_pred_90)
```

- Estos dos códigos calculan la precisión y el recall del modelo que se ha creado con la variable y_train_pred_90 que contiene las predicciones basadas en el valor del umbral que se ha calculado para lograr una precisión del 90%.
- La función precision_score(y_train_5, y_train_pred_90) calcula la precisión del modelo comparando los valores predichos en y_train_pred_90 con los valores reales en y_train_5.
- La función recall_score(y_train_5, y_train_pred_90) calcula el recall del modelo comparando los valores predichos en y_train_pred_90 con los valores reales en y_train_5.

Curva ROC

_ La curva ROC (Receiver Operating Characteristic) es una herramienta útil para evaluar el rendimiento de un modelo de clasificación binaria en diferentes umbrales de decisión. Esta muestra cómo cambia la relación entre verdaderos positivos y falsos positivos a medida que varía el umbral de decisión. Un modelo ideal tendría una tasa de verdaderos positivos del 100% (es decir, todos los positivos se clasifican correctamente) y una tasa de falsos positivos del 0% (es decir, no hay negativos incorrectamente clasificados como positivos). En este caso, la curva ROC seguiría el borde superior izquierdo del gráfico.

_ El área bajo la curva ROC (AUC) es una medida útil de la calidad de un modelo de clasificación. Un modelo con un AUC de 0,5 es básicamente un clasificador aleatorio, mientras que un modelo con un AUC de 1,0 es un clasificador perfecto. En general, cuanto más cerca esté la curva ROC de la esquina superior izquierda del gráfico, mejor será el rendimiento del modelo, ya que significa que el modelo tiene una alta tasa de verdaderos positivos y una baja tasa de falsos positivos.

_ La curva ROC es útil para evaluar la calidad de un modelo de clasificación binaria y para comparar el rendimiento de diferentes modelos. También se utiliza para determinar el mejor umbral de clasificación para un modelo dado. Es similar a la curva precision/recall, pero utiliza las siguientes métricas:

- True positive rate (TPR): otro nombre para recall.
- False positive rate (FPR): se define como la tasa de instancia negativa que han sido clasificadas como positivas.

_ Para trazar la curva ROC, primero se utiliza la función roc_curve() para calcular el TPR (tasa de verdaderos positivos) y FPR (tasa de falsos positivos) para varios valores de umbral. Luego podemos graficar el FPR frente al TPR utilizando Matplotlib. Este código produce la gráfica:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagona
    [...] # Add axis labels and grid

plot_roc_curve(fpr, tpr)
plt.show()
```

- Este código es utilizado para graficar la curva ROC (Receiver Operating Characteristic). Primero, se importa la función roc_curve desde la biblioteca sklearn.metrics, que se utiliza para calcular la tasa de falsos positivos (FPR) y la tasa de verdaderos positivos (TPR) para varios valores umbral de clasificación.

Luego, se define una función llamada plot_roc_curve, que se utiliza para trazar la curva ROC.

- Esta función toma como entrada las tasas FPR y TPR, y opcionalmente una etiqueta para la curva. El código utiliza la función plot de Matplotlib para trazar la curva ROC. Además, se traza una línea diagonal punteada que representa un clasificador aleatorio.
- Finalmente, se llama a la función plot_roc_curve para trazar la curva ROC y se muestra el gráfico resultante utilizando la función show de Matplotlib.

_ Vemos que hay un compromiso, es decir, cuanto mayor es la recuperación (TPR), más falsos positivos (FPR) produce el clasificador. La línea punteada representa la curva ROC de un clasificador puramente aleatorio; un buen clasificador se mantiene lo más alejado posible de esa línea (hacia la esquina superior izquierda). Una forma de comparar clasificadores es medir el área bajo la curva (AUC). Un clasificador perfecto tendrá una ROC AUC igual a 1, mientras que un clasificador aleatorio tendrá una ROC AUC igual a 0.5. Scikit-Learn proporciona una función para calcular la ROC AUC:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

- El código importa la función roc_auc_score de la librería sklearn.metrics, la cual se utiliza para calcular el área bajo la curva ROC (ROC AUC, por sus siglas en inglés).
- Después, se llama a la función roc_auc_score pasando como argumentos las etiquetas verdaderas y_train_5 y los puntajes de decisión y_scores obtenidos a partir de un clasificador binario. La función devuelve el valor del área bajo la curva ROC, que es un valor numérico entre 0.0 y 1.0 que mide la capacidad del clasificador para distinguir entre clases positivas y negativas. Un valor de 1.0 indica un clasificador perfecto, mientras que un valor de 0.5 indica un clasificador que no es mejor que una elección aleatoria.

_ Ahora vamos a entrenar un RandomForestClassifier y comparar su curva ROC y puntuación AUC ROC con las del SGDClassifier. Primero, necesitamos obtener puntajes para cada instancia en el conjunto de entrenamiento. Pero debido a la forma en que funciona, la clase RandomForestClassifier no tiene un método decision_function(). En su lugar, tiene un método predict_proba(). Los clasificadores de Scikit-Learn generalmente tienen uno o ambos métodos. El método predict_proba() devuelve una matriz que contiene una fila por instancia y una columna por clase, cada una que contiene la probabilidad de que la instancia dada pertenezca a la clase dada (por ejemplo, 70% de probabilidad de que la imagen represente un 5):

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")

y_scores_forest = y_probas_forest[:, 1]   # score = proba of positive
class
fpr_forest, tpr_forest, thresholds_forest =
roc_curve(y_train_5,y_scores_forest)

plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```

- Este código entrena un clasificador RandomForestClassifier y luego lo utiliza para predecir las probabilidades de la clase positiva en el conjunto de entrenamiento utilizando cross_val_predict.
- Luego, utiliza las probabilidades de la clase positiva para calcular la tasa de verdaderos positivos (tpr_forest) y la tasa de falsos positivos (fpr_forest) mediante la función roc_curve.
- Finalmente, utiliza la función plot_roc_curve para trazar las curvas ROC del SGDClassifier y el RandomForestClassifier en un solo gráfico y mostrar el gráfico. Esto permite comparar la eficacia de los dos clasificadores en términos de su capacidad para discriminar la clase positiva de la clase negativa.

_ Como se puede ver en el gráfico, la curva ROC del RandomForestClassifier se ve mucho mejor que la del SGDClassifier, ya que se acerca mucho más a la esquina superior izquierda. Como resultado, su puntaje ROC AUC también es significativamente mejor:

```
roc_auc_score(y_train_5, y_scores_forest)
```

- Este código calcula el área bajo la curva ROC (ROC AUC) utilizando la función roc_auc_score de Scikit-Learn. Se utiliza para calcular el rendimiento del modelo de clasificación RandomForestClassifier en el conjunto de entrenamiento.
- Los parámetros de entrada son y_train_5, que contiene las etiquetas reales de las instancias, y y_scores_forest, que contiene las probabilidades de pertenecer a la clase positiva (5 en este caso) asignadas por el modelo de RandomForestClassifier.
- El resultado de este código es el valor de la ROC AUC, que mide la calidad del modelo de clasificación RandomForestClassifier. Un valor de 1.0 indica un modelo perfecto, mientras que un valor de 0.5 indica un modelo aleatorio.

Cuanto más alto sea el valor de la ROC AUC, mejor será el rendimiento del modelo.

Clasificación multiclase

_ La clasificación multiclase es una técnica de aprendizaje automático que se utiliza cuando se tienen más de dos clases en los datos de entrenamiento. En lugar de predecir una salida binaria (sí o no), la clasificación multiclase se utiliza para predecir una de varias opciones posibles. Hay varios algoritmos de clasificación multiclase, como Regresión Logística Multinomial, Árboles de Decisión, Random Forest, Naive Bayes, SVM y Redes Neuronales.

_ Hay dos enfoques principales para abordar el problema de la clasificación multiclase:

One-vs-All: se entrenan tantos clasificadores binarios como clases existan, y cada uno de ellos se entrena para distinguir una clase de todas las demás.

One-vs-One: se entrenan clasificadores binarios para cada par de clases posibles. En ambos casos, se utiliza la votación para predecir la clase final.

_ Una de las formas más comunes de evaluar la clasificación multiclase es a través de la matriz de confusión, que muestra cuántos elementos de cada clase se clasificaron correctamente y cuántos se clasificaron incorrectamente. Otras métricas de evaluación comunes incluyen la precision, el recall y la F1 score. En general, la clasificación multiclase es un desafío interesante en el aprendizaje automático, ya que involucra la predicción de múltiples clases y la elección del enfoque adecuado depende de los datos de entrenamiento y la elección del algoritmo de clasificación adecuado.

_ Scikit-Learn detecta cuando intentamos utilizar un algoritmo de clasificación binaria para una tarea de clasificación multiclase y automáticamente ejecuta OvR u OvO, dependiendo del algoritmo. Probamos esto con un clasificador de máquina de vectores de soporte, utilizando la clase sklearn.svm.SVC:

```
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit(X_train, y_train) # y_train, not y_train_5
svm_clf.predict([some_digit])
```

- Este código importa la clase SVC (Support Vector Machine) del módulo svm de Scikit-Learn y luego crea una instancia de esta clase llamada svm_clf. Luego, ajusta (fit) el modelo a los datos de entrenamiento X_train y y_train, donde y_train es un arreglo que contiene los valores de las etiquetas para todas las instancias de entrenamiento, no solo para la clase 5.
- Finalmente, el código hace una predicción (predict) de la clase de una instancia específica (some_digit) utilizando el modelo SVM entrenado.

_ Este código entrena la SVC en el conjunto de entrenamiento utilizando las clases de destino originales de 0 a 9 (y_train), en lugar de las clases de destino 5-versus-the-rest (y_train_5). Luego hace una predicción (una correcta en este caso). Bajo el hood, Scikit-Learn utilizó la estrategia OvO, entrenó 45 clasificadores binarios, obtuvo sus puntajes de decisión para la imagen y seleccionó la clase que ganó la mayoría de los duelos. Si llamamos al método decision_function(), veremos que devuelve 10 puntuaciones por instancia (en lugar de solo 1). Eso es una puntuación por clase:

```
some_digit_scores = svm_clf.decision_function([some_digit])
some_digit_scores
```

_ El puntaje más alto es de hecho el que corresponde a la clase 5:

```
np.argmax(some_digit_scores)

svm_clf.classes_

svm_clf.classes_[5]
```

_ Si queremos forzar a Scikit-Learn a utilizar la estrategia uno contra uno o uno contra el resto, podemos utilizar las clases OneVsOneClassifier o OneVsRestClassifier. Simplemente creamos una instancia y pasa un clasificador a su constructor (incluso no tiene que ser un clasificador binario). Por ejemplo, este código crea un clasificador multiclase utilizando la estrategia OvR, basado en un SVC:

```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC())
ovr_clf.fit(X_train, y_train)
ovr_clf.predict([some_digit])

len(ovr_clf.estimators_)
```

- El código crea una instancia de la clase OneVsRestClassifier de Scikit-Learn, que permite entrenar un clasificador multiclase utilizando la estrategia uno contra todos (OvR), a partir de un clasificador binario. En este caso se utiliza un clasificador SVM como clasificador binario.
- Luego se ajusta el clasificador multiclase a los datos de entrenamiento X_train e y_train. A continuación, se utiliza el clasificador para hacer una predicción sobre una única instancia some_digit, y se obtiene la etiqueta de clase predicha, que en este caso es 5.
- Por último, se imprime el número de clasificadores binarios que se han entrenado internamente en el clasificador multiclase, que en este caso es 10, uno por cada clase del conjunto de datos.

_ Entrenar un SGDClassifier (o un RandomForestClassifier) es igual de fácil:

```
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])
```

_ Esta vez, Scikit-Learn no tuvo que ejecutar OvR u OvO porque los clasificadores SGD pueden clasificar directamente instancias en múltiples clases. El método decision_function() ahora devuelve un valor por clase. Veamos el puntaje que el clasificador SGD asignó a cada clase:

```
sgd_clf.decision_function([some_digit])
```

_ Podemos ver que el clasificador tiene bastante confianza en su predicción, casi todas las puntuaciones son en gran medida negativas, mientras que la clase 5 tiene una puntuación como la que vemos. El modelo tiene una ligera duda con respecto a la clase 3, que obtiene una puntuación como la que vemos. Ahora, por supuesto, deseamos evaluar este clasificador. Como de costumbre, puede usar la validación cruzada. Use la función cross_val_score() para evaluar la precisión del SGDClassifier:

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

_ Obtenemos más del 84% en todas las pruebas. Si usáramos un clasificador aleatorio, obtendríamos una precisión del 10%, por lo que este no es un puntaje tan malo, pero aún puede hacer mucho mejor. Simplemente escalando las entradas, la precisión aumenta por encima del 89%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3,
scoring="accuracy")
```

Análisis de error

_ El análisis de errores es una técnica utilizada para examinar los errores cometidos por un modelo de aprendizaje automático durante el entrenamiento y la evaluación. Se utiliza para identificar los patrones de error en los datos y en el modelo, y para determinar las posibles mejoras que se pueden hacer. Para realizar un análisis de errores, se puede examinar la matriz de confusión, que muestra la cantidad de veces que el modelo ha clasificado correctamente o incorrectamente cada clase. A partir de la matriz de confusión, se pueden calcular diversas métricas, como la precisión, la exhaustividad y la puntuación F1, para evaluar el rendimiento del modelo.

_ También se puede examinar los casos de errores individuales para ver si hay patrones comunes, como características que son difíciles de clasificar para el modelo. Esto puede ayudar a identificar las áreas en las que se puede mejorar el modelo, como la recopilación de más datos, la adición de nuevas características o la modificación del algoritmo de aprendizaje automático.

_ Primero, miramos la matriz de confusión. Necesitamos hacer predicciones usando la función cross_val_predict(), y luego llamar a la función confusion_matrix(),como lo hiciste antes:

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train,
cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
```

- Este código primero, utiliza la función cross_val_predict() para hacer predicciones utilizando el modelo sgd_clf entrenado con los datos de entrenamiento escalados (X_train_scaled) y las etiquetas de entrenamiento (y_train). La función cross_val_predict() usa validación cruzada para obtener las predicciones y retorna un array con las predicciones para cada instancia.
- Luego, se llama a la función confusion_matrix() pasando las etiquetas de entrenamiento (y_train) y las predicciones (y_train_pred) como argumentos para crear una matriz de confusión. La matriz de confusión es una herramienta que muestra el número de veces que se clasificó cada instancia en una clase específica y el número de veces que se confundió con otras clases.

_ Vemos una representación gráfica de la matriz de confusión, utilizando la función matshow() de Matplotlib:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```

_ Esta matriz de confusión se ve bastante bien, ya que la mayoría de las imágenes se encuentran en la diagonal principal, lo que significa que fueron clasificadas correctamente. Los números 5 se ven ligeramente más oscuros que los demás dígitos, lo que podría significar que hay menos imágenes de 5 en el conjunto de datos o que el clasificador no funciona tan bien en los 5 como en otros dígitos.

_ Ahora, nos centramos en el gráfico de los errores. Primero, dividimos cada valor en la matriz de confusión por el número de imágenes en la clase correspondiente para que puedas comparar las tasas de error en lugar de los números absolutos de errores (lo que haría que las clases abundantes parezcan injustamente malas). Y llenamos la diagonal con ceros para mantener solo los errores y traza el resultado con la función matshow() de Matplotlib.

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

_ Las filas representan las clases reales, mientras que las columnas representan las clases predichas. La columna para la clase 8 es bastante brillante, lo que indica que muchas imágenes se clasifican incorrectamente como 8. Sin embargo, la fila para la clase 8 no es tan mala, lo que indica que los verdaderos 8 en general se clasifican correctamente como 8. Como podemos ver, la matriz de confusión no necesariamente es simétrica. También podemos ver que los 3 y los 5 a menudo se confunden (en ambas direcciones).

_ El análisis de la matriz de confusión a menudo brinda información sobre cómo mejorar nuestro clasificador. Al ver este gráfico, parece que deberíamos reducir los falsos 8. El análisis de errores individuales también puede ser una buena manera de comprender lo que está haciendo nuestro clasificador y por qué está fallando, pero es más difícil y consume más tiempo. Por ejemplo, graficamos ejemplos de 3 y 5 (la función plot_digits() solo usa la función imshow() de Matplotlib:

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```

- Este código se utiliza para analizar los errores del clasificador de imágenes. Primero se especifican las clases a analizar, en este caso la clase A es el número 3 y la clase B es el número 5. Luego, se filtran las imágenes de entrenamiento según la verdadera clase (cl_a o cl_b) y la clase predicha (cl_a o cl_b) utilizando máscaras booleanas.
- Se crean cuatro conjuntos de imágenes:
  - X_aa: imágenes de la clase A que fueron correctamente clasificadas como clase A.
  - X_ab: imágenes de la clase A que fueron incorrectamente clasificadas como clase B.
  - X_ba: imágenes de la clase B que fueron incorrectamente clasificadas como clase A.
  - X_bb: imágenes de la clase B que fueron correctamente clasificadas como clase B.
- Luego, se muestra un gráfico de 4 subplots usando la función plot_digits() que muestra 25 imágenes por subplot. En el primer subplot se muestran las imágenes de la clase A que fueron correctamente clasificadas como clase A, en el segundo subplot se muestran las imágenes de la clase A que fueron incorrectamente clasificadas como clase B, en el tercer subplot se muestran las imágenes de la clase B que fueron incorrectamente clasificadas como clase A, y en el cuarto subplot se muestran las imágenes de la clase B que fueron correctamente clasificadas como clase B.
- Este análisis ayuda a comprender mejor los errores del clasificador y a tomar decisiones sobre cómo mejorar su rendimiento.

Clasificación de múltiples etiquetas

_ La clasificación multietiqueta es una tarea de aprendizaje supervisado en la que cada instancia de entrada puede asignarse a múltiples categorías simultáneamente. En otras palabras, en lugar de asignar una sola etiqueta a cada instancia, se asignan varias etiquetas. Por ejemplo, en una aplicación de etiquetado automático de imágenes, una imagen puede tener varias etiquetas como "perro", "animal", "feliz".

_ Existen diferentes enfoques para realizar la clasificación multietiqueta, como el enfoque de:

- Partición: se crean múltiples clasificadores binarios, uno para cada etiqueta, y se combinan para predecir las múltiples etiquetas.
- Transformación: se transforma el problema de clasificación multietiqueta en un conjunto de problemas de clasificación binaria, y se utiliza un clasificador binario para cada problema.
- Construcción: se construye un clasificador específico para la tarea de clasificación multietiqueta.

_ Para evaluar el desempeño de un clasificador multietiqueta, se pueden utilizar diversas métricas como la precisión, el recall, la F1-score y la exactitud.

_ Tenemos el siguiente código:

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

_ Este código crea una matriz y_multilabel que contiene dos etiquetas objetivo para cada imagen de dígito, la primera indica si el dígito es grande (7, 8 o 9) o no, y la segunda indica si es impar o no. Las siguientes líneas crean una instancia de KNeighborsClassifier (que admite clasificación multietiqueta, aunque no todos los clasificadores lo hacen), y la entrenamos usando la matriz de múltiples objetivos. Es decir, se quiere saber si un número es mayor a 7 e impar. Ahora podemos hacer una predicción y notar que produce dos etiquetas:

```
knn_clf.predict([some_digit])
```

_ El resultado que obtenemos es correcto, ya que el dígito 5 no es mayor o igual que 7 (False) y es impar (True). Hay muchas formas de evaluar un clasificador multietiqueta, y seleccionar la métrica adecuada realmente depende de su proyecto. Un enfoque es medir el F1 score para cada etiqueta individual (o cualquier otra métrica de clasificador binario discutida anteriormente) y simplemente calcular la puntuación promedio. El siguiente código calcula el F1 score promedio en todas las etiquetas:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel,
cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
```

_ Esto asume que todas las etiquetas son igualmente importantes, lo cual puede no ser el caso.

Clasificación de múltiples salidas

_ La clasificación multi-output es un tipo de problema de aprendizaje supervisado en el que se pretende predecir varios valores de salida para cada instancia de entrada. En otras palabras, se trata de una extensión del problema de clasificación en el que cada instancia de entrada tiene múltiples etiquetas de salida. Un ejemplo de problema de clasificación multi-output podría ser la predicción de la posición de un objeto en una imagen. En este caso, se tendría que predecir las coordenadas X e Y del objeto en la imagen.

_ El uso de la clasificación multi-output puede ser útil en una variedad de situaciones, incluyendo la clasificación de imágenes y el procesamiento de señales.

_ Para implementar un modelo de clasificación multi-output se pueden utilizar algoritmos de regresión o clasificación, y el enfoque utilizado dependerá de la naturaleza del problema. La evaluación de un modelo de clasificación multi-output se realiza mediante el cálculo de la precisión de cada etiqueta de salida y la determinación de la precisión media de todas las etiquetas.

_ Es importante destacar que la clasificación multi-output no debe confundirse con la clasificación multiclase o multietiqueta, ya que en la clasificación multi-output se pretende predecir múltiples valores de salida para cada instancia de entrada, mientras que en la clasificación multiclase o multietiqueta se pretende predecir una sola etiqueta de salida.

_ Para explicar esto, vamos a construir un sistema que elimine el ruido de las imágenes. Tomará como entrada una imagen de dígito ruidosa y producirá como salida una imagen del dígito limpia, representada como un arreglo de intensidades de píxeles, al igual que las imágenes de MNIST. Vemos que la salida del clasificador es de múltiples etiquetas (una etiqueta por píxel) y cada etiqueta puede tener múltiples valores (la intensidad de los píxeles varía de 0 a 255):

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

- Este código tiene como objetivo crear un conjunto de datos de entrenamiento y prueba para un sistema de clasificación multioutput que tiene como objetivo remover el ruido de las imágenes MNIST.
- Primero se crea una matriz de ruido aleatorio con valores entre 0 y 100 utilizando la función randint() de NumPy. La forma de la matriz es (número de imágenes, 784), donde 784 representa los 28x28 píxeles de cada imagen MNIST.
- Luego, se agrega el ruido a las imágenes de entrenamiento y prueba sumando el ruido a las matrices de imágenes X_train y X_test. Las nuevas matrices de imágenes se almacenan en X_train_mod y X_test_mod.
- Finalmente, para crear las etiquetas objetivo para el sistema de clasificación multioutput, se utilizan las matrices originales de entrenamiento y prueba (y_train y y_test). Esto se debe a que el objetivo es restaurar las imágenes originales a partir de las imágenes ruidosas. Por lo tanto, y_train_mod y y_test_mod se inicializan con y_train e y_test, respectivamente.

_ Vemos en la imagen de la izquierda, que obtenemos como resultado, que está la imagen de entrada con ruido y en la de la derecha está la imagen objetivo limpia. Ahora entrenamos el clasificador y hacemos que limpie esta imagen.

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

- Este código entrena un clasificador K-vecinos más cercanos (KNN) en el conjunto de datos de entrenamiento X_train_mod y y_train_mod, que consiste en imágenes ruidosas de dígitos MNIST y sus etiquetas correspondientes. Luego, se utiliza el clasificador entrenado para limpiar una imagen específica (denotada por some_index) del conjunto de datos de prueba X_test_mod.
- El resultado limpio de la imagen se almacena en la variable clean_digit, que se traza en una figura utilizando la función plot_digit(). La función plot_digit() toma una imagen MNIST como entrada y la muestra en una trama matplotlib. En este caso, se muestra la imagen de dígito limpio pronosticado por el clasificador KNN.