CAPÍTULO IV

SERIES DE TIEMPO

En este capítulo trabajaremos con datos estructurados, pero que tienen una condición especial: la dependencia de sus registros en relación con la línea de tiempo. ¿Qué significa esto? Que lo que sucede en un momento específico está fuertemente relacionado con lo que aconteció en tiempos anteriores. En este tipo de datasets existe una columna particular que contiene la información de la fecha (e incluso la hora) del registro, y esta información se convierte en la base para la construcción de nuevas características y la aplicación de técnicas de ingeniería de características.

Una serie de tiempo es, formalmente, un conjunto de observaciones registradas en instantes sucesivos, donde el orden temporal es esencial para comprender el comportamiento de la variable estudiada. A diferencia de los datasets estructurados tradicionales (en los que las filas pueden reordenarse sin alterar su significado), en una serie de tiempo cada valor depende del momento en que ocurrió y, en muchos casos, de los valores que le precedieron.

En términos simples, una serie de tiempo nos permite observar cómo evoluciona una variable a través del tiempo. Ejemplos comunes incluyen:

- El número diario de casos confirmados de COVID-19.
- La temperatura registrada por hora en una ciudad.
- El precio del dólar minuto a minuto.
- El número de ventas mensuales de un producto.

- La frecuencia cardíaca registrada segundo a segundo por un dispositivo wearable.

Este tipo de información es fundamental para tareas como analizar patrones, identificar tendencias, detectar anomalías y predecir valores futuros mediante técnicas de pronóstico (*forecasting*).

Las series de tiempo poseen características particulares que las diferencian de otros tipos de datos:

- Secuencia temporal obligatoria: no se puede alterar el orden de los registros sin perder información esencial.
- Dependencia entre observaciones: un valor puede influir sobre los siguientes.
- Contexto histórico: comprender el pasado es clave para interpretar o predecir el futuro.

Por estas razones, el análisis de series de tiempo requiere métodos específicos que respeten la naturaleza temporal del fenómeno. Cada decisión, desde la visualización inicial hasta el modelamiento, debe apoyarse en esta estructura para obtener resultados coherentes y útiles.

Nota aclaratoria sobre el dataset de COVID-19. Aunque el daraset de casos de COVID-19 de Bogotá fue abordado en el Capítulo 3, constituye, estrictamente hablando, una serie de tiempo. La razón de trabajarlo como una dataset estructurado básico es que nos permitió introducir de manera gradual los conceptos fundamentales de limpieza de datos y EDA, en un contexto real y familiar para el lector.

4.1. Conceptos Básicos de Series de Tiempo

Antes de iniciar nuestros ejemplos con Series de Tiempo, es necesario conocer algunos concentos básicos, los cuales se presentan a continuación.

4.1.1. Componentes fundamentales de una Serie de Tiempo

Una serie de tiempo puede analizarse como la combinación de varios

componentes que describen distintos tipos de comportamiento temporal. Identificar estos componentes permite comprender mejor la dinámica de los datos y constituye un paso clave antes de cualquier proceso de modelamiento o pronóstico.

De forma general, una serie de tiempo puede descomponerse en los siguientes elementos:

- **Tendencia**: representa el comportamiento global de largo plazo de la serie. Indica si la variable presenta una dirección creciente, decreciente o estable a lo largo del tiempo. Como ejemplo, se puede mencionar el aumento progresivo del precio de venta de una criptomoneda.

- **Estacionalidad**: corresponde a patrones que se repiten cada cierto periodo de tiempo, por ejemplo, días, semanas o meses. Un ejemplo es el incremento de ventas en el mes de diciembre, que se repite cada año.

- **Ciclo**: a diferencia de la estacionalidad, en este caso las fluctuaciones que se repiten en el tiempo no tienen una duración constante, y su inicio y fin no son perfectamente predecibles. Un ejemplo son los ciclos de “burbuja” y recesión en un país.

- **Ruido o componente irregular**: corresponde a variaciones aleatorias, errores de medición o perturbaciones imprevistas que no pueden explicarse por la tendencia, la estacionalidad ni el ciclo.

Nota aclaratoria 1: en este capítulo se trabaja bajo el supuesto de una descomposición aditiva de la serie de tiempo, una aproximación común en el análisis exploratorio; sin embargo, en aplicaciones reales pueden existir descomposiciones multiplicativas o mixtas.

Nota aclaratoria 2: en este capítulo, el término estacionariedad se utiliza en el sentido de estacionariedad débil o de segundo orden, en la cual la media y la varianza de la serie permanecen constantes en el tiempo y la covarianza depende únicamente del retardo temporal.

4.1.2. Clasificación de Series de Tiempo

Las series de tiempo pueden clasificarse de distintas formas según la cantidad de variables involucradas, la naturaleza de los datos, su comportamiento estadístico y la regularidad con la que son registradas. Esta clasificación es importante porque condiciona las técnicas de análisis y los modelos que pueden aplicarse (Ver Figura 36).

Figura 36. Clasificación Series de Tiempo.

A continuación, se presentan los tipos más comunes de series de tiempo.

Series de tiempo de una y varias variables

- Serie de tiempo de una sola variable: está conformada por una sola variable que se observa a lo largo del tiempo. Por ejemplo, el precio diario del dólar, la temperatura por hora o la cantidad de ventas mensuales.

- Serie de tiempo multivariada: está compuesta por dos o más variables observadas simultáneamente en el tiempo. Por ejemplo, un dataset de criptomoneda con las siguientes variables: precio de apertura, precio de cierre, cantidad de transacciones, precio más alto. Las series multivariadas permiten analizar relaciones entre variables, pero también presentan una mayor complejidad en el modelamiento.

Series de tiempo continuas y discretas

- Series continuas: toman valores en “cualquier instante” de tiempo.

- Series discretas: la mayoría de los datasets utilizados en ciencia de datos corresponden a este tipo. Registra valores específicos del fenómeno observado, con espaciamientos, por ejemplo, de minutos, horas, o días.

Series de tiempo estacionarias y no estacionarias

- Serie estacionaria: presenta media constante, varianza constante y una estructura de correlación que no cambia en el tiempo. Este tipo de series es especialmente importante porque muchos modelos estadísticos clásicos asumen estacionariedad.

- Serie no estacionaria: presenta cambios en su media, varianza o estructura de dependencia a lo largo del tiempo. La mayoría de las series reales pertenecen a esta categoría y requieren transformaciones antes de su modelamiento.

Series regulares e irregulares

- Series regulares: el espaciamiento en tiempo entre registros es fijo. Por ejemplo, existe un registro cada minuto.

- Series irregulares: los datos no siguen un patrón fijo de tiempo entre observaciones, lo cual dificulta su análisis directo y, en muchos casos, exige procesos previos de interpolación o re-muestreo.

4.1.3. Autocorrelación y memoria temporal (en series financieras no estacionarias)

En una serie de tiempo, los valores no suelen ser independientes entre sí: lo que ocurre en un instante determinado está, en mayor o menor medida, influenciado por lo que ocurrió en momentos anteriores. A este fenómeno se le conoce como autocorrelación y está directamente relacionado con el concepto de memoria temporal.

En el contexto de este capítulo, donde se trabaja principalmente con series financieras no estacionarias (como precios de acciones y criptomonedas), la autocorrelación juega un papel clave para entender la dinámica del mercado y para la construcción de variables predictoras basadas en el pasado.

- **Autocorrelación**

Mide el grado de relación entre el valor actual de una serie y sus valores pasados desplazados en el tiempo, conocidos como retardos o *lags*. Por ejemplo, permite responder preguntas como:

¿El precio de hoy está relacionado con el precio de ayer?

¿Existe relación entre el valor actual y el observado hace un mes?

En series financieras es frecuente encontrar autocorrelaciones débiles, inestables y variables en el tiempo, debido a la alta volatilidad y a la influencia de múltiples factores externos. Aun así, estos patrones temporales son fundamentales para el diseño de modelos predictivos.

- **Memoria temporal**

Hace referencia al horizonte de tiempo durante el cual los valores pasados influyen sobre el presente. Dependiendo del fenómeno, una serie puede presentar:

Memoria corta: solo los valores más recientes tienen impacto significativo.

Memoria larga: valores antiguos siguen influyendo en el comportamiento actual.

En datos financieros suele predominar una memoria corta, donde los últimos días u horas contienen la mayor parte de la información relevante para la predicción.

Nota aclaratoria sobre las implicaciones prácticas para este libro. En los ejemplos desarrollados en este capítulo, la autocorrelación y la memoria temporal se utilizarán principalmente para definir retardos temporales (*lags*) como variables de entrada, construir ventanas deslizantes (*sliding windows*), calcular promedios móviles como mecanismos de suavizado y alimentar modelos de

aprendizaje automático (*machine learning*) que aprendan directamente de estas dependencias temporales.

4.1.4. Diferencias entre el análisis exploratorio clásico y el EDA en series de tiempo

El análisis exploratorio de datos (EDA) en series de tiempo requiere un enfoque diferente al utilizado en datasets estructurados tradicionales. Aunque ambos comparten objetivos como comprender la distribución, detectar patrones y encontrar anomalías, la dimensión temporal introduce retos y técnicas específicas. En este capítulo, además, trabajamos con datos financieros no estacionarios, lo cual refuerza la necesidad de un EDA adaptado a este tipo de señales.

A continuación, se presentan las diferencias más relevantes.

- **El orden de los datos sí importa**

En un EDA clásico, las filas pueden reorganizarse sin afectar la interpretación de los resultados. En series de tiempo, reordenar los registros destruye la estructura temporal, impidiendo analizar tendencias, patrones o rupturas.

Por esta razón, el primer paso del EDA temporal es asegurar que los datos estén correctamente ordenados por fecha y hora.

- **Las visualizaciones se basan en el tiempo**

En EDA tradicional predominan histogramas, diagramas de caja y gráficos de barras. En series de tiempo, la herramienta más importante es la gráfica temporal, que permite visualizar:

  - ❖ Tendencias
  - ❖ Cambios abruptos
  - ❖ Periodos de alta volatilidad
  - ❖ Comportamientos inusuales

Para datos financieros (ej. como los precios de acciones o criptomonedas), estas curvas son esenciales para identificar patrones reales del mercado.

- **Análisis de estabilidad temporal**

Mientras que en EDA clásico se evalúan distribuciones completas, en series de tiempo se revisan cómo cambian esas distribuciones a lo largo del tiempo.

Por ejemplo:

- ❖ ¿Aumenta la varianza en ciertos periodos?
- ❖ ¿La volatilidad del precio cambia según el año o el mes?
- ❖ ¿Existen rupturas estructurales evidentes?

Este análisis es especialmente relevante en datos no estacionarios.

- **Uso de estadísticas móviles**

En vez de trabajar con una única media o desviación estándar global, en series de tiempo se emplean *promedios móviles*, *desviaciones móviles* y *ventanas deslizantes*, que permiten estudiar cómo evolucionan estos indicadores en el tiempo.

Esto es particularmente útil para series financieras que presentan:

- ❖ Cambios de volatilidad
- ❖ Tendencias marcadas
- ❖ Fluctuaciones cortas y abruptas

- **Identificación de dependencias temporales**

En el EDA clásico no se consideran relaciones temporales entre registros. En series de tiempo, es indispensable evaluar cómo los valores pasados influyen en los presentes, a través de:

- ❖ Retardos (*lags*)
- ❖ Ventanas temporales
- ❖ Transformaciones basadas en memoria corta

En este capítulo, esta idea se utilizará para la creación de features y no para construir modelos estadísticos clásicos como ARIMA.

- **Tratamiento de valores faltantes**

En datasets tradicionales se puede imputar con media, mediana o moda sin mayores consecuencias en el orden de los datos. En series de tiempo, la imputación debe preservar el comportamiento temporal, por ejemplo, mediante:

- ❖ Relleno hacia adelante (*forward fill*)
- ❖ Promedios móviles

En datos financieros (ej. donde no se generan observaciones en fines de semana o festivos), es frecuente realizar procesos de re-muestreo en vez de imputación directa.

- **Enfoque orientado al comportamiento y no a la distribución**

El EDA clásico se centra en cómo están distribuidos los valores. El EDA temporal se centra en cómo cambian (evolucionan) esos valores a lo largo del tiempo, lo cual es esencial para comprender fenómenos dinámicos, volátiles y no estacionarios.

Realizar un EDA adecuado en series de tiempo permite construir *features* robustas, detectar patrones clave y preparar la señal para modelos que aprovechan su estructura temporal. Esta preparación es indispensable antes de avanzar hacia los ejemplos prácticos del capítulo.

4.2. Ingeniería de características en Series de Tiempo

A continuación, se presentarán las técnicas de construcción de características más comúnes en series de tiempo.

4.2.1. Características con retardos (Lag Features)

Las características con retardo, o *lag features*, consisten en utilizar valores pasados de una variable como nuevas variables de entrada del modelo. En el contexto de series de tiempo, un retardo de orden "*k"* corresponde al valor de la serie observado "*k"* instantes antes.

Por ejemplo, un retardo de un día en el precio de cierre representa el valor del precio del día anterior, mientras que un retardo de cinco días corresponde al precio observado cinco días atrás.

Este tipo de características permite al modelo capturar **dependencias temporales directas**, bajo el supuesto de que el comportamiento pasado de la serie puede aportar información sobre su evolución futura.

De forma conceptual:

- close_lag_1: precio de cierre del día anterior
- close_lag_5: precio de cierre de cinco días atrás

4.2.2. Ventanas deslizantes (Rolling Window Features)

Las características basadas en ventanas deslizantes, o *rolling features*, resumen el comportamiento de la serie en un intervalo de tiempo reciente mediante estadísticas simples, como el promedio o la desviación estándar.

Una ventana deslizante de tamaño "*w"* se define como el conjunto de los últimos "*w"* valores observados en la serie. A partir de esta ventana, se pueden calcular distintas métricas que describen el comportamiento local de la serie.

Ejemplos comunes incluyen:

- promedio móvil (*rolling mean*), asociado a tendencias locales,
- desviación estándar móvil (*rolling std*), asociada a volatilidad.

A diferencia de los retardos individuales, las ventanas deslizantes permiten capturar **patrones agregados**, reduciendo la sensibilidad al ruido puntual.

4.2.3. Ventanas expansivas (Expanding Window Features)

Las ventanas expansivas consideran todos los valores históricos de la serie desde su inicio hasta el instante actual. A medida que avanza el tiempo, la ventana se va ampliando, incorporando progresivamente más información.

Este tipo de características permite calcular estadísticas acumuladas, como:

- promedio histórico acumulado,
- desviación estándar histórica acumulada.

Las ventanas expansivas son útiles para capturar el **comportamiento global de largo plazo** de la serie y proporcionar al modelo un contexto histórico más amplio.

4.3. Caso de Estudio 1 de Series de Tiempo: clima

Para este primer caso de estudio se utiliza un dataset que contiene registros climáticos horarios obtenidos a partir de diferentes sensores atmosféricos.

```
import pandas as pd
clima_df = pd.read_csv("/content/clima.csv")
clima_df.head()
```

	tiempo_hora_local	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima
0	01-10-2019 00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	partly-cloudy-night
1	01-10-2019 01:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	partly-cloudy-night
2	01-10-2019 02:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	partly-cloudy-night
3	01-10-2019 03:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	partly-cloudy-night
4	01-10-2019 04:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	partly-cloudy-night

Figura 37. Pre-visualización del dataset: Clima.

La variable de salida está explícita y corresponde a “estado_clima”, mientras que las demás columnas del dataframe son los *features* iniciales (excepto tipo_hora_local al cual debemos realizarle un tratamiento especial).

Antes de empezar a trabajar con el dataset es necesario conocer el tipo de datos de cada columna,

```
clima_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 648 entries, 0 to 647
Data columns (total 12 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   tiempo_hora_local        499 non-null    object
 1   temperatura              499 non-null    float64
 2   humedad                  499 non-null    float64
 3   punto_rocio              499 non-null    float64
 4   direccion_viento         499 non-null    float64
 5   velocidad_viento         499 non-null    float64
 6   rafagas_viento           499 non-null    float64
 7   presion                  499 non-null    float64
 8   indice_UV                499 non-null    float64
 9   ozono                    499 non-null    float64
 10  intensidad_precipitacion 499 non-null    float64
 11  estado_clima             499 non-null    object
dtypes: float64(10), object(2)
memory usage: 60.9+ KB
```

Figura 38. Información del dataset: Clima.

Observaciones iniciales:

- Aunque el dataset tiene **648 registros**, solo **499 cuentan con valores no nulos**, lo que indica la presencia de **datos faltantes** que deberán ser tratados en etapas posteriores. No obstante, en este caso, los registros faltantes corresponden a intervalos temporales completos sin medición, y no a valores aislados dentro de una observación.

- La mayoría de las variables son **numéricas continuas**, lo cual es consistente con la naturaleza de las mediciones climáticas.

- Existen **dos variables categóricas** (tiempo_hora_local y estado_clima) que requieren transformación antes de ser utilizadas en un modelo de aprendizaje automático.

- La variable tiempo_hora_local, aunque aparece como tipo object, representa información temporal y deberá convertirse a un **tipo datetime** para permitir análisis y transformaciones propias de series de tiempo.

4.3.1. EDA en el dataset Clima

La primera acción a realizar en este dataset consiste en la conversión de la columna “tiempo_hora_local” al índice del dataframe. Inicialmente debemos hacer la transformación del tipo de formato de object a *datetime* y posteriormente llevar esta columna al índice.

```
clima_df['tiempo_hora_local'] = pd.to_datetime(clima_df['tiempo_hora_
local'], format='%d-%m-%Y %H:%M')

# Asignación de la columna temporal como índice del dataframe
clima_df = clima_df.set_index('tiempo_hora_local')

# Ordenar por índice temporal (buena práctica en series de tiempo)
clima_df = clima_df.sort_index()
clima_df.head()
```

tiempo_hora_local	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima
2019-10-01 00:00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	partly-cloudy-night
2019-10-01 01:00:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	partly-cloudy-night
2019-10-01 02:00:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	partly-cloudy-night
2019-10-01 03:00:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	partly-cloudy-night
2019-10-01 04:00:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	partly-cloudy-night

Figura 39. Dataset Clima: conversión de la columna tiempo_hora_local.

La segunda tarea a realizar consiste en responder al siguiente interrogante:

¿qué hacer con los datos faltantes?

En este caso, la respuesta difiere ligeramente de la estrategia empleada para los datos estructurados analizados en el capítulo anterior. Lo primero que debe tenerse en cuenta es que, en este tipo de datasets, los registros mantienen una relación temporal, por lo que, a priori, no resulta una buena estrategia romper dicha dependencia eliminando filas con valores faltantes de forma indiscriminada. Sin embargo, el análisis cambia cuando se trata de la variable de salida, ya que no es válido “suponer” o imputar su valor. Hacerlo introduciría etiquetas artificiales en el proceso de aprendizaje y daría lugar a un problema de *label leakage*, afectando negativamente la validez del modelo.

Por lo cual, vamos a eliminar los registros en los cuales existe dato faltante en la salida a predecir, que en nuestro caso es ”estado_clima”:

```
clima_df = clima_df.dropna(subset=['estado_clima'])
```

Y revisamos qué ocurrió en nuestro dataset:

```
clima_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 499 entries, 2019-10-01 00:00:00 to 2019-10-21 18:00:00
Data columns (total 11 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   temperatura              499 non-null    float64
 1   humedad                  499 non-null    float64
 2   punto_rocio              499 non-null    float64
 3   direccion_viento         499 non-null    float64
 4   velocidad_viento         499 non-null    float64
 5   rafagas_viento           499 non-null    float64
 6   presion                  499 non-null    float64
 7   indice_UV                499 non-null    float64
 8   ozono                    499 non-null    float64
 9   intensidad_precipitacion 499 non-null    float64
 10  estado_clima             499 non-null    object
dtypes: float64(10), object(1)
memory usage: 46.8+ KB
```

Figura 40. Dataset Clima: conversión de la columna tiempo_hora_local.

En este caso particular, se observa que los registros con valores faltantes en la variable objetivo coinciden con aquellos que también presentan datos faltantes en las variables de entrada. Esto sugiere la ausencia completa de observaciones en determinados intervalos temporales. Por esta razón, al eliminar los registros sin valor en la variable de salida, se obtiene simultáneamente un daraset sin valores faltantes, sin necesidad de aplicar técnicas adicionales de imputación.

Como tercera tarea, se procede a **transformar la variable de salida**. El primer paso consiste en identificar cuántas categorías diferentes presenta la columna "*estado_clima*". Para ello, se ejecuta el siguiente comando:

```
unicos_clima = clima_df['estado_clima'].unique()
print(unicos_clima)
```

Y obtenemos:

['partly-cloudy-night' 'partly-cloudy-day' 'clear-night' 'clear-day'

'cloudy']

A partir de esta información, se opta por definir un **diccionario de codificación manual**, en el cual se asigna un valor entero a cada categoría de la variable *estado_clima*. Esta codificación permite representar la variable de salida de forma compacta y adecuada para su uso en modelos de clasificación supervisada.

Es importante destacar que en este escenario **no se aplica one-hot encoding**, ya que esta técnica está diseñada para variables de entrada (*features*) y no para la variable de salida. En problemas de clasificación multiclase, la salida debe representarse como una única columna numérica que identifique la clase asociada a cada registro. El uso de one-hot encoding en la variable objetivo rompería la formulación estándar del problema y dificultaría la interpretación y evaluación del modelo.

```
# Diccionario de codificación del estado del clima
clima_dict = {
    'clear-day': 1,
    'clear-night': 2,
    'partly-cloudy-day': 3,
    'partly-cloudy-night': 4,
    'cloudy': 5
}
```

Y continuamos con la codificación de la salida:

```
# Codificación de la variable de salida
clima_df['estado_clima_enc'] = clima_df['estado_clima'].map(clima_
dict)

# Verificar que no existan valores sin codificar
print("Valores no codificados:",
      clima_df['estado_clima_enc'].isna().sum())

# Eliminar la columna "estado_clima"
del clima_df['estado_clima']

clima_df.head()
```

tiempo_hora_local	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima_enc
2019-10-01 00:00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	4
2019-10-01 01:00:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	4
2019-10-01 02:00:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	4
2019-10-01 03:00:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	4
2019-10-01 04:00:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	4

Figura 41. Codificación de la salida en el dataset Clima.

4.3.2. Modelamiento del dataset Clima (baseline)

Ya tenemos el dataset listo para realizar modelamiento. Sin embargo, es recomendado conocer si el problema de clasificación (que en este caso es multi-clase) es balanceado o no. Para ello, vamos a utilizar el siguiente código en Python:

```
import matplotlib.pyplot as plt

# Diccionario inverso (número → nombre)
inv_clima_dict = {v: k for k, v in clima_dict.items()}

# Conteo por clase
counts = (
    clima_df["estado_clima_enc"]
    .value_counts()
    .sort_index()
    .rename(index=inv_clima_dict)
)

# Gráfica
plt.figure(figsize=(8,5))
bars = plt.bar(
    counts.index,
    counts.values
)

# Etiquetas encima de cada barra
for bar in bars:
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height,
        f"{int(height)}",
        ha="center",
        va="bottom"
    )
```

```
plt.xlabel("Estado del clima")
plt.ylabel("Número de observaciones")
plt.title("Distribución de clases del dataset de Clima")
plt.xticks(rotation=30)
plt.tight_layout()
plt.show()
```

Y obtenemos:

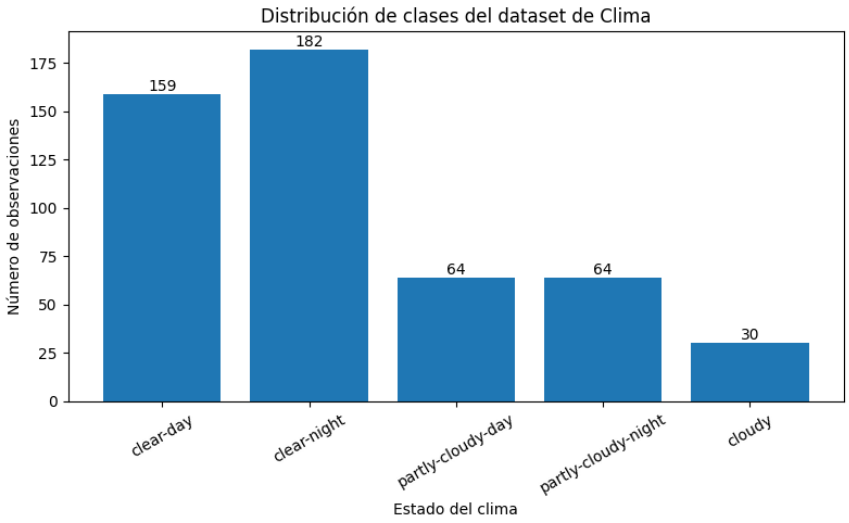

Figura 42. Distribución de datos por clase, dataset Clima.

De acuerdo con los resultados, el dataset no es balanceado, existiendo una diferencia significativa entre la clase mayoritaria (182 casos) y la minoritaria (30 casos).

Cuando realicemos el modelamiento, entonces debemos utilizar la opción "weighted" en el cálculo de las métricas precisión, recall y F1. Utilizamos split temporal, para garantizar que datos consecutivos queden en entrenamiento y otro grupo de datos consecutivos queden en test, dada la dependencia temporal entre los registros. Esta es una diferencia importante en términos de Split con datos estructurados que no son series de tiempo. Y como modelo utilizamos RandomForestClassifier

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report
)
# ==========================
# 1) Definir X e y
# ==========================
X = clima_df.drop(columns=["estado_clima_enc"], errors="ignore")
y = clima_df["estado_clima_enc"]  # objetivo multiclase (1..5)
# ==========================
# 2) Split temporal (70%-30%)
# ==========================
split = int(len(clima_df) * 0.7)
X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]
# ==========================
# 3) Entrenar modelo (baseline sin FE)
# ==========================
clf = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    class_weight="balanced"
)
clf.fit(X_train, y_train)
# ==========================
# 4) Predicción
# ==========================
y_pred = clf.predict(X_test)
# ==========================
# 5) Métricas
# ==========================
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="weighted", zero_
```

```
division=0)
recall = recall_score(y_test, y_pred, average="weighted", zero_
division=0)
f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)
print(f"Accuracy:  {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall:    {recall:.4f}")
print(f"F1-score:  {f1:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred,
 zero_division=0))
# ==========================
# 6) Matriz de confusión
# ==========================
labels_num = [1, 2, 3, 4, 5]
labels_txt = ['clear-day', 'clear-night', 'partly-cloudy-day', 'partly-cloudy-
night', 'cloudy']
conf_matrix = confusion_matrix(y_test, y_pred, labels=labels_num)
plt.figure(figsize=(7, 5))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=labels_txt,
    yticklabels=labels_txt
)
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.title("Matriz de Confusión (Baseline, sin FE)")
plt.show()
```

```
Accuracy:  0.6867
Precision: 0.6549
Recall:    0.6867
F1-score:  0.6589

Classification Report:
              precision    recall  f1-score   support

           1       0.93      0.78      0.85        68
           2       0.62      0.89      0.74        56
           3       0.00      0.00      0.00         3
           4       0.00      0.00      0.00        23
           5       0.00      0.00      0.00         0

    accuracy                           0.69       150
   macro avg       0.31      0.33      0.32       150
weighted avg       0.65      0.69      0.66       150
```

Figura 43. Resultados de modelamiento del dataset de Clima, sin FE.

Las clases 3, 4 y 5 (que son minoritarias), presentan un desempeño muy bajo, con ausencia de aciertos en el conjunto de prueba, dado que, no tienen ningún acierto.

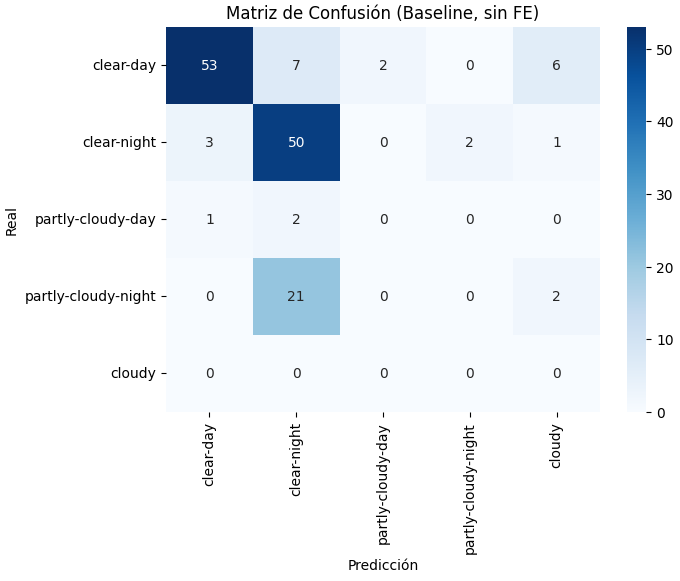

Figura 44. Matriz de confusión del dataset Clima, sin FE.

4.3.3. Ingeniería de Características aplicada al dataset de Clima

Matriz de correlación

En esta sección se analiza la **correlación entre las variables de entrada**, con el objetivo de apoyar de forma sistemática el proceso de **ingeniería de características**. La matriz de correlación permite identificar relaciones lineales entre los diferentes atributos climáticos, lo cual resulta clave para la toma de decisiones en la construcción de nuevos *features*.

Para ello, se calcula la matriz de correlación considerando únicamente las variables numéricas del dataset, excluyendo la variable objetivo codificada.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Seleccionar solo variables numéricas
corr_matrix = clima_df.drop(columns=['estado_clima_enc']).corr()

# Graficar heatmap de correlación
plt.figure(figsize=(10, 6))
sns.heatmap(
    corr_matrix,
    annot=True,
    cmap="coolwarm",
    fmt=".2f",
    square=True
)
plt.title("Matriz de correlación entre variables climáticas")
plt.show()
```

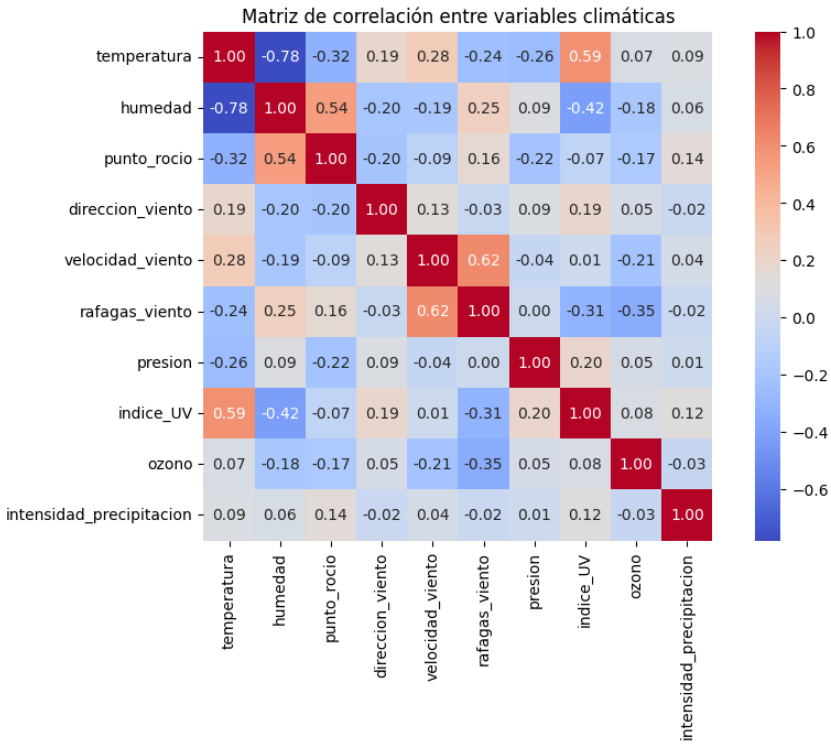

Figura 45. Matriz de correlación entre las variables del dataset de Clima.

Dado que la temperatura es una variable central del sistema climático y presenta correlaciones relevantes con otras variables del dataset, se utiliza como referencia inicial para orientar el proceso de ingeniería de características. Con el fin de evitar la generación indiscriminada de características y reducir redundancia, se seleccionan únicamente las variables con mayor correlación con la temperatura para aplicar transformaciones temporales.

Nos enfocaremos en la primera columna de la matriz de correlación y vemos que, en su orden, los tres primeros features más correlados con temperatura son: humedad (0.78), índice_UV (0.59) y punto_rocio (0.32). Entonces, inicialmente realizaremos FE con estos cuatro features, aplicando lags, Rolling y expanding a cada uno de ellos, así:

```
import pandas as pd

# Asegurar orden temporal (muy importante para lags/rolling/expanding)
clima_df = clima_df.sort_index()

# Copia para crear features sin tocar el original
clima_fe = clima_df.copy()

# ----------------------------
# 1) Lag features (retardos)
# ----------------------------
fe_cols = ["temperatura", "humedad", "indice_UV", "punto_rocio"]
sizes = [1, 2, 3]  # horas

for col in fe_cols:
    for s in sizes:
        clima_fe[f"{col}_lag{s}"] = clima_fe[col].shift(s)

# ----------------------------
# 2) Rolling features (ventanas móviles)
# ----------------------------
windows = [3, 6]  # horas

for col in fe_cols:
    for w in windows:
     clima_fe[f"{col}_roll_mean_{w}h"] = clima_fe[col].rolling(window=w).
mean()
       clima_fe[f"{col}_roll_std_{w}h"]  = clima_fe[col].rolling(window=w).
std()
      clima_fe[f"{col}_roll_min_{w}h"]  = clima_fe[col].rolling(window=w).
min()
     clima_fe[f"{col}_roll_max_{w}h"]  = clima_fe[col].rolling(window=w).
max()

# ----------------------------
# 3) Expanding features (acumuladas)
```

```
# ----------------------------
for col in fe_cols:
    clima_fe[f"{col}_exp_mean"] = clima_fe[col].expanding().mean()
    clima_fe[f"{col}_exp_max"]  = clima_fe[col].expanding().max()

# ----------------------------
# 4) Limpieza por NaNs creados por lags/rolling
# ----------------------------
clima_fe = clima_fe.dropna().copy()

# ----------------------------
# 5) Separar X e y (listo para modelamiento con FE)
# ----------------------------
X_fe = clima_fe.drop(columns=["estado_clima_enc"], errors="ignore")
y_fe = clima_fe["estado_clima_enc"]

print("Shape original:", clima_df.shape)
print("Shape con FE:", clima_fe.shape)
print("Nuevas columnas creadas:", X_fe.shape[1] - (clima_df.shape[1]
- 1))
```

Y obtenemos:

Shape original: (499, 11)
Shape con FE: (494, 63)
Nuevas columnas creadas: 52

Entrenamos el nuevo dataframe, así

```
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report
```

```
)

# ==========================
# 1) Definir X e y (con FE)
# ==========================
X = X_fe.copy()
y = y_fe.copy()   # objetivo multiclase (1..5)

# ==========================
# 2) Split temporal (70%-30%)
# ==========================
split = int(len(X) * 0.7)

X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]

# ==========================
# 3) Entrenar modelo (con FE)
# ==========================
clf_fe = RandomForestClassifier(
    n_estimators=200,         # un poco más robusto que el baseline
    random_state=42,
    class_weight="balanced",
    n_jobs=-1
)

clf_fe.fit(X_train, y_train)

# ==========================
# 4) Predicción
# ==========================
y_pred = clf_fe.predict(X_test)

# ==========================
# 5) Métricas
# ==========================
```

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="weighted", zero_
division=0)
recall = recall_score(y_test, y_pred, average="weighted", zero_
division=0)
f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)

print("Resultados del modelo con Feature Engineering\n")
print(f"Accuracy:  {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall:    {recall:.4f}")
print(f"F1-score:  {f1:.4f}")

print("\nClassification Report:\n")
print(classification_report(y_test, y_pred, zero_division=0))

# ==========================
# 6) Matriz de confusión
# ==========================
labels_num = [1, 2, 3, 4, 5]
labels_txt = [
    "clear-day",
    "clear-night",
    "partly-cloudy-day",
    "partly-cloudy-night",
    "cloudy"
]
conf_matrix = confusion_matrix(y_test, y_pred, labels=labels_num)

plt.figure(figsize=(7, 5))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=labels_txt,
```

```
        yticklabels=labels_txt
    )
    plt.xlabel("Predicción")
    plt.ylabel("Real")
    plt.title("Matriz de Confusión (Modelo con Feature Engineering)")
    plt.tight_layout()
    plt.show()
```

Y obtenemos la siguiente matriz de confusión:

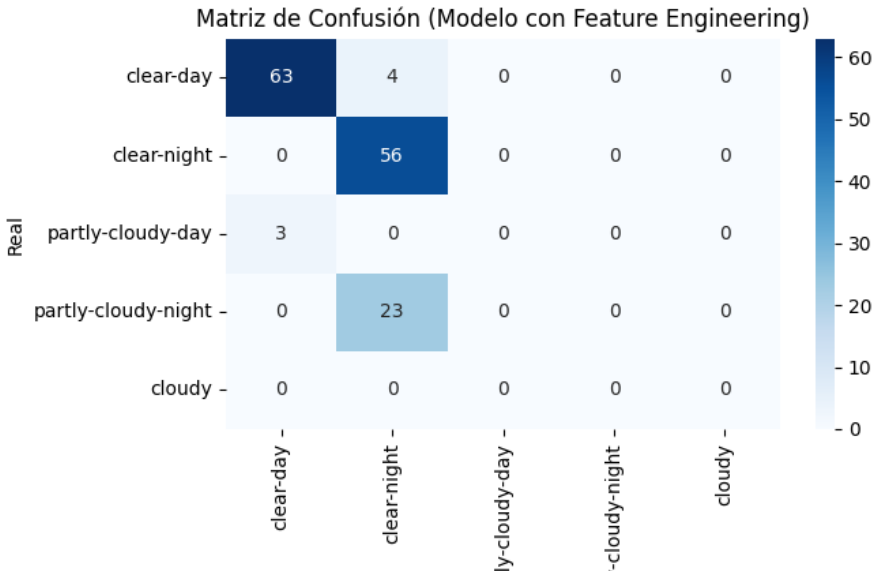

Figura 46. Matriz de confusión del dataset Clima, con FE (opción 1).

Al comparar la matriz de confusión de la Figura 46 con la presentada en la Figura 44, se observa un incremento en el número de casos correctamente clasificados para las clases *clear-day* y *clear-night*. En el escenario sin ingeniería de características, estas clases registraban 53 y 50 aciertos, respectivamente, mientras que al incorporar las características derivadas (opción 1) los aciertos aumentan a 63 y 56. El modelo con ingeniería de características alcanza una exactitud (*accuracy*) de 0.7987.

Si bien la mejora se concentra principalmente en las clases mayoritarias, este resultado evidencia que la ingeniería de características permite incorporar información adicional relevante al modelo, mejorando su desempeño sin necesidad de modificar la arquitectura del clasificador.

Finalmente, visualizaremos de los features originales y los constuidos, la correlación de ellos con la salida, así:

```
import pandas as pd
import matplotlib.pyplot as plt

# Obtener importancias
importances = clf_fe.feature_importances_

# Crear DataFrame
feature_importance_df = pd.DataFrame({
    "feature": X_fe.columns,
    "importance": importances
}).sort_values(by="importance", ascending=False)

# Mostrar las 15 más importantes
feature_importance_df.head(15)
```

El atributo feature_importances_ forma parte interna del objeto RandomForestClassifier y se genera automáticamente durante el entrenamiento del modelo.

Y para visualización:

```
plt.figure(figsize=(8,5))
top_n = 15
plt.barh(
    feature_importance_df["feature"].head(top_n)[::-1],
    feature_importance_df["importance"].head(top_n)[::-1]
)
plt.xlabel("Importancia")
plt.title("Importancia de las características (Random Forest)")
plt.tight_layout()
plt.show()
```

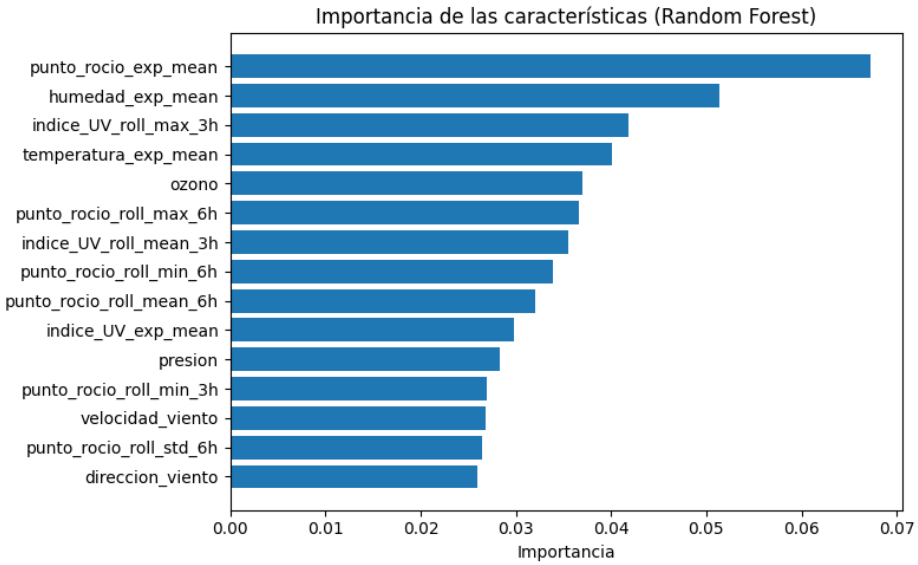

Figura 47. Importancia de las características (Random Forest) para el dataset Clima con FE.

Quiero resaltar que de los 5-top features de mayor importancia para el modelo de clasificación de estado del clima, los cuatro primeros corresponden a features construidos con FE, lo cual permitió que mejoráramos el desempeño del modelo *baseline*. Para este caso de estudio, expanding fue el método que generó features con mayor importancia.

4.4. Caso de Estudio 2 de Series de Tiempo: Twlo prices

A continuación, se trabajará con el dataset twlo_prices.csv, el cual contiene precios intradía de la acción de Twilio (TWLO) con una resolución de un minuto. El daraset incluye el precio de cierre (close), el volumen negociado (volume) y la marca temporal (date). Este tipo de series de tiempo de alta frecuencia es representativo de datos financieros reales, caracterizados por alta variabilidad, ausencia de estacionariedad y una fuerte dependencia temporal.

4.4.1. Análisis Exploratorio y construcción de la variable de salida

Iniciamos con la lectura del dataset y su pre-visualización:

```
import pandas as pd
price_df = pd.read_csv("/content/twlo_prices.csv")
price_df.head(10)
```

	close	volume	date
0	99.9800	93417.0	2020-01-02 14:30:00+00:00
1	99.7800	16685.0	2020-01-02 14:31:00+00:00
2	100.1400	21998.0	2020-01-02 14:32:00+00:00
3	100.3500	18348.0	2020-01-02 14:33:00+00:00
4	100.5500	22181.0	2020-01-02 14:34:00+00:00
5	100.6100	14573.0	2020-01-02 14:35:00+00:00
6	100.9500	13960.0	2020-01-02 14:36:00+00:00
7	100.9875	20219.0	2020-01-02 14:37:00+00:00
8	101.0500	10588.0	2020-01-02 14:38:00+00:00
9	101.1400	14118.0	2020-01-02 14:39:00+00:00

Figura 48. Pre-visualización de dataset twlo_prices.csv.

Adicionalmente, podemos conocer el tamaño del dataset, así:

```
price_df.shape
(146502, 3)
```

Es decir, la cantidad de registros es de 146,502, con las columnas: *close*, *volume* y *date*.

Tratamiento de la variable temporal

Antes de aplicar cualquier técnica de ingeniería de características (FE) en series de tiempo, es indispensable que la variable temporal sea tratada

explícitamente como tal. En el dataset twlo_prices.csv, la columna date contiene la información temporal de cada observación y debe convertirse en un índice de tipo datetime.

Este paso permite:

- ❖ preservar el orden cronológico de la serie,
- ❖ habilitar operaciones temporales (ventanas móviles, rezagos, resampling),
- ❖ y evitar errores conceptuales al tratar la fecha como una variable categórica o numérica común.

```
# Conversión de la columna date a formato datetime
price_df['date'] = pd.to_datetime(price_df['date'])
# Definir la fecha como índice temporal
price_df = price_df.set_index('date')
# Verificar estructura final
price_df.head()
```

date	close	volume
2020-01-02 14:30:00+00:00	99.9800	93417.0
2020-01-02 14:31:00+00:00	99.7800	16685.0
2020-01-02 14:32:00+00:00	100.1400	21998.0
2020-01-02 14:33:00+00:00	100.3500	18348.0
2020-01-02 14:34:00+00:00	100.5500	22181.0
2020-01-02 14:35:00+00:00	100.6100	14573.0
2020-01-02 14:36:00+00:00	100.9500	13960.0
2020-01-02 14:37:00+00:00	100.9875	20219.0
2020-01-02 14:38:00+00:00	101.0500	10588.0
2020-01-02 14:39:00+00:00	101.1400	14118.0

Figura 49. Pre-visualización de twlo_prices.csv, posterior a conversión *datetime*.

Al establecer la columna date como índice temporal, por medio de set_index('date'), esta deja de aparecer como una variable del dataset. No es necesario eliminarla explícitamente, ya que su información se conserva en el índice y será utilizada como referencia temporal para el análisis y la ingeniería de características.

De forma opcional, se puede asegurar que el dataset esté ordenado cronológicamente:

```
price_df.sort_index(inplace=True)
print(price_df.index)
```

Visualización de la serie temporal

Gracias a que el índice del *DataFrame* ahora contiene información temporal completa (*año, mes, día, hora y minuto*), es posible generar visualizaciones donde la dimensión temporal corresponde directamente al eje horizontal.

Para visualizar simultáneamente el precio de cierre y el volumen negociado, se puede utilizar el siguiente código:

```
import matplotlib.pyplot as plt

# Crear subgráficos
fig, ax1 = plt.subplots(figsize=(10, 6))

# Graficar 'close' en el primer eje
ax1.plot(price_df.index, price_df['close'], color='tab:blue', label='Close',
linewidth=2)
ax1.set_xlabel('Fecha')
ax1.set_ylabel('Precio de Cierre', color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')

# Crear un segundo eje para 'volume'
ax2 = ax1.twinx()
ax2.plot(price_df.index, price_df['volume'], color='tab:orange',
 label='Volume', linewidth=2)
ax2.set_ylabel('Volumen', color='tab:orange')
```

```
        ax2.tick_params(axis='y', labelcolor='tab:orange')

        # Añadir título
        plt.title('Precio de Cierre y Volumen de TWLO (Intradía)', fontsize=10)

        # Mostrar gráfico
        plt.tight_layout()
        plt.show()
```

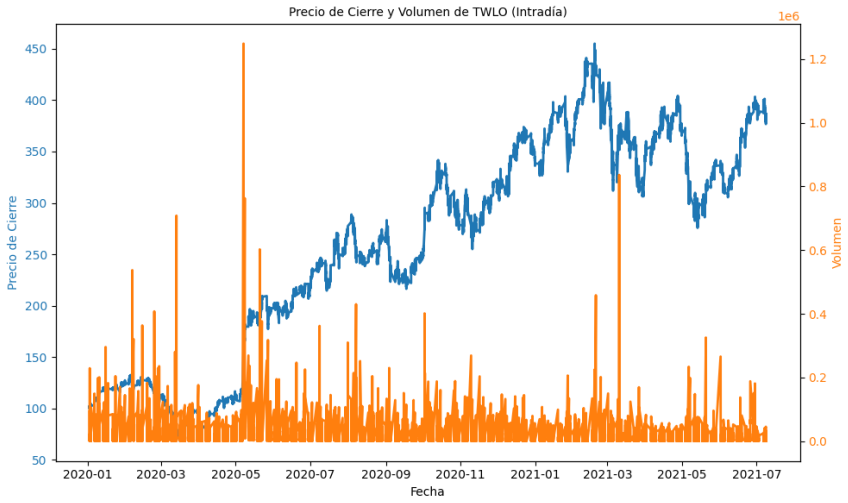

Figura 50. Gráfica de "volumen" y "close" de twlo_prices.csv

Alternativamente, las variables pueden visualizarse por separado:

```
        price_df['close'].plot()
        price_df['volume'].plot()
```

Construcción de la variable de salida

En problemas de series de tiempo financieras, la construcción de la variable objetivo es una de las decisiones más críticas del proceso de modelamiento. A diferencia de otros dominios, no basta con utilizar el valor actual de la serie, ya que esto conduciría a un problema trivial o a un escenario de *data*

leakage. En este caso, el objetivo es predecir si el precio de cierre diario del día siguiente presenta un incremento significativo, utilizando únicamente información disponible hasta el instante actual.

Dado que el dataset contiene precios intradía con resolución de un minuto, es necesario definir explícitamente el horizonte temporal de la predicción. En este estudio de caso, se plantea el siguiente problema:

Determinar si el precio de cierre diario del día siguiente presenta un incremento significativo respecto al día actual.

Esta formulación permite:

- evitar el uso de información futura,
- reducir el ruido intradía,
- y trabajar con una definición de salida alineada con decisiones reales de análisis financiero.

Para lo cual, el primer paso consiste en obtener el precio de cierre diario a partir de los datos intradía. Para ello, se selecciona el último valor disponible de la variable close para cada día:

```
daily_close = price_df['close'].resample('1D').last().dropna()
daily_close.head(10)
```

Obteniendo:

	close
date	
2020-01-02 00:00:00+00:00	103.15
2020-01-03 00:00:00+00:00	103.52
2020-01-06 00:00:00+00:00	107.46
2020-01-07 00:00:00+00:00	108.09
2020-01-08 00:00:00+00:00	109.38
2020-01-09 00:00:00+00:00	113.02
2020-01-10 00:00:00+00:00	115.75
2020-01-13 00:00:00+00:00	120.32
2020-01-14 00:00:00+00:00	119.02
2020-01-15 00:00:00+00:00	119.88

Figura 51. Precio de cierre diario, dataset twlo_prices.csv.

Este proceso transforma la serie intradía en una serie diaria, conservando la información relevante para la definición del evento a predecir.

A partir del precio de cierre diario, se calcula el retorno porcentual entre días consecutivos:

```
#Retorno diario
daily_ret = daily_close.pct_change()
```

donde *pct_change* calcula el cambio porcentual entre un valor y el valor anterior en la serie. La fórmula que se aplica internamente, es:

$$pct\_change_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

Es decir, al precio del día actual se le resta el precio del día anterior, y el resultado se divide por el precio del día anterior. El valor obtenido representa el retorno relativo asociado al día actual. Por ejemplo, al precio de cierre del 3 de enero de 2020 se le resta el precio de cierre del 2 de enero de 2020 y el resultado se divide por el precio de cierre del 2 de enero de 2020, obteniendo (103.52-103.15)/103.15 = 0.003587.

```
#Retorno diario
daily_ret = daily_close.pct_change()
print(daily_ret)
```

```
date
2020-01-02 00:00:00+00:00         NaN
2020-01-03 00:00:00+00:00    0.003587
2020-01-06 00:00:00+00:00    0.038060
2020-01-07 00:00:00+00:00    0.005863
2020-01-08 00:00:00+00:00    0.011934
                               ...
2021-07-01 00:00:00+00:00   -0.018971
2021-07-02 00:00:00+00:00    0.004886
2021-07-06 00:00:00+00:00    0.014639
2021-07-07 00:00:00+00:00   -0.014985
2021-07-08 00:00:00+00:00   -0.009885
```

Figura 52. Precio de retorno, dataset twlo_prices.csv.

El primer valor del retorno diario es NaN, ya que no existe un día previo con el cual comparar el precio de cierre. El uso de retornos, en lugar de precios absolutos, permite trabajar con una variable más estable y comparable a lo largo del tiempo.

Posteriormente, creamos una variable objetivo, así:

```
# Umbral mínimo de variación (0.2%)
threshold = 0.002

# Variable objetivo: ¿sube el precio al día siguiente?
daily_target = (daily_ret.shift(-1) > threshold).astype(int)

print(daily_target)

daily_target.value_counts()
```

Este umbral (*threshold = 0.002*, equivalente a una variación del 0.2 %) se define con el fin de filtrar fluctuaciones pequeñas asociadas al ruido propio del mercado, y concentrar el análisis en movimientos de precio con mayor relevancia práctica.

La expresión daily_ret.shift(-1) desplaza la serie de retornos una posición hacia atrás, de modo que a cada registro se le asocia el retorno observado en el día siguiente. De esta forma, el valor del primer registro corresponde al retorno del segundo día, y así sucesivamente.

Posteriormente, se evalúa si dicho retorno supera el umbral definido. En caso afirmativo, se asigna el valor entero **1** a la variable objetivo daily_target, indicando un incremento relevante del precio al día siguiente; en caso contrario, se asigna el valor entero **0**.

```
date
2020-01-02 00:00:00+00:00    1
2020-01-03 00:00:00+00:00    1
2020-01-06 00:00:00+00:00    1
2020-01-07 00:00:00+00:00    1
2020-01-08 00:00:00+00:00    1
                            ..
2021-07-01 00:00:00+00:00    1
2021-07-02 00:00:00+00:00    1
2021-07-06 00:00:00+00:00    0
2021-07-07 00:00:00+00:00    0
2021-07-08 00:00:00+00:00    0
Name: close, Length: 382, dtype: int64
```

	count
close	

Figura 53. Valor daily_target del dataset twlo_prices.csv.

Finalmente, la variable objetivo diaria se asigna a cada observación intradía correspondiente al mismo día. De esta forma, cada registro intradía queda asociado con el evento que ocurrirá al cierre del día siguiente.

```
# Crear columna auxiliar con el día
price_df['day'] = price_df.index.floor('D')
```

Con price_df.index.floor a cada uno de los registros del dataset, se les obtendrá la información únicamente del día en formato (año-mes-día) y dejando en 00:00:00 la información de la hora y minuto del registro.

```
# Crear columna auxiliar con el día
price_df['day'] = price_df.index.floor('D')

price_df.head(10)
```

date	close	volume	day
2020-01-02 14:30:00+00:00	99.9800	93417.0	2020-01-02 00:00:00+00:00
2020-01-02 14:31:00+00:00	99.7800	16685.0	2020-01-02 00:00:00+00:00
2020-01-02 14:32:00+00:00	100.1400	21998.0	2020-01-02 00:00:00+00:00
2020-01-02 14:33:00+00:00	100.3500	18348.0	2020-01-02 00:00:00+00:00
2020-01-02 14:34:00+00:00	100.5500	22181.0	2020-01-02 00:00:00+00:00
2020-01-02 14:35:00+00:00	100.6100	14573.0	2020-01-02 00:00:00+00:00
2020-01-02 14:36:00+00:00	100.9500	13960.0	2020-01-02 00:00:00+00:00
2020-01-02 14:37:00+00:00	100.9875	20219.0	2020-01-02 00:00:00+00:00
2020-01-02 14:38:00+00:00	101.0500	10588.0	2020-01-02 00:00:00+00:00
2020-01-02 14:39:00+00:00	101.1400	14118.0	2020-01-02 00:00:00+00:00

Figura 54. Creación de la columna "day" del dataset twlo_prices.csv.

Posteriormente, con price_df.merge mezclamos el *dataframe* original con *daily_target* (que hemos renombrado como "target") a partir de la columna day.

```
# Unir la variable objetivo diaria al dataset intradía
price_df = price_df.merge(
    daily_target.rename('target'),
    left_on='day',
    right_index=True,
    how='inner'
)
# Eliminar columna auxiliar
price_df.drop(columns=['day'], inplace=True)

price_df[['close', 'target']].head()
```

date	close	volume	target
2020-01-02 14:30:00+00:00	99.98	93417.0	1
2020-01-02 14:31:00+00:00	99.78	16685.0	1
2020-01-02 14:32:00+00:00	100.14	21998.0	1
2020-01-02 14:33:00+00:00	100.35	18348.0	1
2020-01-02 14:34:00+00:00	100.55	22181.0	1

Figura 55. Dataset twlo_prices.csv con columna de salida denominada "target".

A partir de este punto, el dataset contiene explícitamente la variable target, que será utilizada en los experimentos de modelamiento.

Con info podemos verificar que el dataset contiene 146.502 registros, con dos columnas de entrada numéricas de tipo float64 correspondientes a "close" y "volumen" y una columna de salida de tipo int64 correspondiente a "target". Todas las celdas están diligenciadas.

4.2.2. Modelamiento en twlo_prices dataset antes de FE

Ahora podemos proceder con el modelamiento. Es importante aclarar que el modelo obtenido en este punto corresponde al **modelo base (baseline)**, el cual será utilizado posteriormente como referencia para compararlo con el modelo resultante de aplicar técnicas de **ingeniería de características (Feature Engineering, FE)** al dataset.

Paso 1: Importe de librerías y definición de la entrada del modelo (X) y de la salida (y)

```
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_
score, f1_score, confusion_matrix, classification_report
```

```
import seaborn as sns

X = price_df.drop(columns=["target"])
y = price_df["target"]
```

Paso 2: Identificación de tipo de dataset (balanceado/desbalanceado)

```
y.value_counts(normalize=True)
```

target	proportion
1	0.524273
0	0.475727

dtype: float64

Figura 56. Distribución del dataset twlo_prices.csv.

Como se observa, las clases se encuentran prácticamente balanceadas, con proporciones cercanas al 50 % para ambas categorías. Esto permite interpretar métricas globales como *acc* sin el sesgo que normalmente introducen *datasets* altamente desbalanceados.

Paso 3: Split temporal

A diferencia del *split* que hacíamos en el capítulo anterior de datos estructurados, en este caso debemos tener en cuenta que los datos tienen un **orden cronológico** y no pueden mezclarse arbitrariamente. El *split* temporal divide el dataset en dos subconjuntos **respetando ese orden**: uno para entrenamiento (pasado) y otro para prueba (futuro).

```
# Split temporal (70%-30%)
split = int(len(price_df) * 0.7)
X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]
```

De esta forma, los datos comprendidos desde el inicio del dataset hasta el índice definido por *split* se utilizan para el **entrenamiento**, mientras que los

datos posteriores a dicho índice y hasta el final de la serie se emplean para la **evaluación del modelo**.

Paso 4: Entrenamiento del modelo

Teniendo en cuenta el tipo de salida binaria, el modelo lo trabajaremos de clasificación binaria. Utilizaremos en este caso un DecisionTreeClassifier, así:

```
# Modelamiento

from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt

# Modelo base: Árbol de decisión
model = DecisionTreeClassifier(criterion="entropy", max_depth=2,
random_state=42)
model.fit(X_train, y_train)

# Predicción
y_pred = model.predict(X_test)

# Métricas
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nMatriz de Confusión:")
print(confusion_matrix(y_test, y_pred))

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred, target_names=["No subió",
"Subió"]))

# Visualización del árbol
plt.figure(figsize=(12, 8))
plot_tree(
```

```
        model,
        feature_names=X.columns,
        class_names=["No subió", "Subió"],
        filled=True,
        rounded=True
    )
    plt.show()
```

Este código es muy similar al utilizado en el Capítulo anterior. En este caso las etiquetas son: "No subió", "Subió".

Y obtenemos el siguiente resultado:

```
Accuracy: 0.56

Matriz de Confusión:
[[ 8207 15185]
 [ 4078 16481]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.67      0.35      0.46     23392
       Subió       0.52      0.80      0.63     20559

    accuracy                           0.56     43951
   macro avg       0.59      0.58      0.55     43951
weighted avg       0.60      0.56      0.54     43951
```

Figura 57. Información del dataset twlo_prices.csv, después del proceso de EDA.

Dado que la clase mayoritaria corresponde aproximadamente al 52% de las observaciones, el accuracy de referencia (que ocurre cuando el clasificador siempre predice la clase mayoritaria) es cercano al 52 %. El árbol de decisión entrenado sin ingeniería de características obtiene un *accuracy* de 56%, apenas 4% por encima del *baseline nulo*. Esto indica que, en términos globales, el modelo apenas supera una predicción aleatoria informada por la distribución de clases, lo cual evidencia una capacidad predictiva muy limitada.

4.4.3. Ingeniería de características con selección estadística

A partir de este punto, el objetivo es incorporar **contexto temporal** al dataset *twlo_prices*, de modo que el modelo pueda capturar patrones dinámicos que no están presentes en los valores instantáneos de las variables *close* y *volume*.

Si bien las técnicas de ingeniería de características empleadas en este capítulo son ampliamente utilizadas en el análisis de series de tiempo, la selección de sus parámetros (ej. el número de retardos o el tamaño de las ventanas temporales) no es trivial. Elegir estos valores de forma arbitraria puede:

- introducir ruido innecesario en los datos,
- aumentar la dimensionalidad del problema, y
- degradar el desempeño del modelo.

Por esta razón, en este estudio se emplean criterios estadísticos para guiar la selección de retardos y tamaños de ventana, con el objetivo de reducir el costo computacional y mejorar la interpretabilidad del proceso de ingeniería de características.

El procedimiento que se presenta a continuación corresponde a un enfoque avanzado de ingeniería de características guiado por criterios estadísticos. No se espera que el lector memorice el código, sino que comprenda la lógica general que sustenta la selección de retardos y ventanas temporales.

```
import numpy as np
import pandas as pd
from statsmodels.tsa.stattools import acf
from sklearn.feature_selection import mutual_info_classif

def add_statistical_fe(
    price_df: pd.DataFrame,
    max_lag: int = 60,
    top_k_lags: int = 8,
    acf_nlags: int = 200,
    acf_threshold: float = 0.05,
```

```
    use_expanding: bool = True,
):
    df = price_df.copy()

    # --- 0) Orden y verificación ---
    df = df.sort_index()
    required = {“close”, “volume”, “target”}
    missing = required - set(df.columns)
    if missing:
        raise ValueError(f”Faltan columnas requeridas: {missing}”)

    # --- 1) Selección de ventanas rolling usando ACF -> tiempo de
decorrelación (tau) ---
    r = df[“close”].dropna()
    if len(r) < 50:
        raise ValueError(“Muy pocos datos para estimar ACF de forma
 estable.”)

    acf_vals = acf(r, nlags=min(acf_nlags, len(r) - 1), fft=True)
    # Primer lag donde |ACF| cae por debajo del umbral
    tau = next((i for i in range(1, len(acf_vals)) if abs(acf_vals[i]) < acf_
threshold), 20)

  # Ventanas alrededor de tau (redondeadas y con mínimos razonables)
   windows = sorted({max(3, int(round(tau / 2))), max(3, int(round(tau))),
max(5, int(round(2 * tau)))})
    # Si tau es muy pequeño, añade una ventana un poco más larga
para estabilidad
    if windows[-1] < 10:
        windows = sorted(set(windows + [10, 20]))

    # --- 2) Selección de lags usando Mutual Information (MI) ---
    # Creamos candidatos de lags sobre retornos (ret)
    X_lags = pd.concat({f”close_lag_{k}”: df[“close”].shift(k) for k in
range(1, max_lag + 1)}, axis=1)
    tmp_mi = pd.concat([X_lags, df[“target”]], axis=1).dropna()
```

```
            if len(tmp_mi) < 200:
                # Si hay pocos datos (por NaNs), baja max_lag automáticamente
                new_max_lag = max(10, min(max_lag, len(df) // 20))
                X_lags = pd.concat({f"close_lag_{k}": df["close"].shift(k) for k in
        range(1, new_max_lag + 1)}, axis=1)
                tmp_mi = pd.concat([X_lags, df["target"]], axis=1).dropna()

            mi = mutual_info_classif(
                tmp_mi.drop(columns=["target"]),
                tmp_mi["target"],
                random_state=42
            )
            mi_rank = (
                pd.Series(mi, index=tmp_mi.drop(columns=["target"]).columns)
                .sort_values(ascending=False)
            )

            # Top-k lags (extrae el número del lag desde el nombre "ret_lag_k")
            selected_lags = [int(name.split("_")[-1]) for name in mi_rank.
        head(top_k_lags).index]
            selected_lags = sorted(set(selected_lags))

            # --- 3) Construcción de features con lags seleccionados ---
            # Lags de Close
            for k in selected_lags:
                df[f"close_lag_{k}"] = df["close"].shift(k)

            # (Opcional) Lags de volumen en los mismos k (a veces ayuda, a
        veces no; útil para docencia)
            for k in selected_lags:
                df[f"vol_lag_{k}"] = df["volume"].shift(k)

            # --- 4) Rolling features con ventanas elegidas (sobre Close y volume)
---
            for w in windows:
                # retornos
```

```
        df[f"close_roll_mean_{w}"] = df["close"].rolling(window=w).mean()
        df[f"close_roll_std_{w}"] = df["close"].rolling(window=w).std()

        # volumen (se suele beneficiar de suavizado)
        df[f"vol_roll_mean_{w}"] = df["volume"].rolling(window=w).mean()
        df[f"vol_roll_std_{w}"] = df["volume"].rolling(window=w).std()

    # --- 5) Expanding (acumulado histórico sobre Close y volume) ---
    if use_expanding:
        df["close_exp_mean"] = df["close"].expanding().mean()
        df["close_exp_std"]  = df["close"].expanding().std()
        df["vol_exp_mean"] = df["volume"].expanding().mean()
        df["vol_exp_std"]  = df["volume"].expanding().std()

    # --- 6) Dataset final ---
    df_fe = df.dropna().copy()

    # X / y listos para modelar (sin leakage de target)
    X = df_fe.drop(columns=["target"])
    y = df_fe["target"]

    # Parámetros seleccionados
    meta = {
        "tau_acf": tau,
        "acf_threshold": acf_threshold,
        "rolling_windows": windows,
        "selected_lags_mi": selected_lags,
        "top_k_lags": top_k_lags,
        "max_lag_considered": max_lag,
    }
    return X, y, df_fe, meta
```

Este código realiza tres tareas principales:
(1) selecciona lags de forma informada usando *Mutual Information (MI)*,
(2) selecciona ventanas rolling a partir del tiempo de decorrelación estimado mediante la *función de autocorrelación (ACF)*, y
(3) genera *features* del tipo lags + rolling mean/std + expanding.

1. **Inicialmente se importan las librerías de trabajo**: *NumPy* y *Pandas* se utilizan para la manipulación de datos y *dataframes*; *acf* se emplea para estimar cuánta **memoria temporal** presenta la serie; y *mutual_info_classif* permite medir qué tanto una variable explica la salida (*target* en este caso), incluso cuando la relación no es lineal.

2. **Como segundo paso se definen los parámetros de diseño**, entre ellos: el número máximo de lags candidatos (*max_lag*), la cantidad de lags finales seleccionados (*top_k_lags*), el número máximo de retardos utilizados para estimar la ACF (*acf_nlags*), el umbral a partir del cual se considera que la serie se ha decorrelacionado (*acf_threshold*), y si se agregan o no métricas acumuladas históricas (*use_expanding*).

3. **Sección 0 – Orden y verificaciones**: el *dataframe* se ordena cronológicamente para respetar la naturaleza temporal de la serie y se verifica la existencia de las columnas requeridas.

4. **Sección 1 – Selección de ventanas rolling usando ACF → tiempo de decorrelación (τ)**: se calcula la función de autocorrelación de la serie close y se identifica el primer *lag* para el cual el valor absoluto de la ACF cae por debajo del umbral definido. Este valor se interpreta como el **tiempo de decorrelación** de la serie. A partir de τ se definen las ventanas *rolling* principales: τ/2, τ y 2τ, incorporando mínimos razonables para garantizar estabilidad numérica.

5. **Sección 2 – Selección de lags usando Mutual Information (MI)**: se generan múltiples *lags* candidatos de la variable close y se evalúa qué tan informativo resulta cada uno para predecir la variable objetivo. Los *lags* se ordenan de mayor a menor según su puntaje de MI y se seleccionan los *top_k_lags* más relevantes.

6. **Sección 3 – Construcción de *features* con lags seleccionados**: se crean nuevos *features* de tipo *lag* para las columnas close y volume, utilizando únicamente los retardos seleccionados en la sección anterior.

7. **Sección 4 – *Rolling features* con ventanas elegidas**: para cada ventana definida en la Sección 1 se calculan nuevas variables de **promedio móvil** y **desviación estándar móvil**, tanto para close como para volume, capturando tendencia y volatilidad local.

8. **Sección 5 – *Expanding* (acumulado histórico)**: se generan *features* acumuladas desde el inicio de la serie hasta el instante actual, calculando promedio y desviación estándar de forma *expanding*. En este caso, el tamaño efectivo de la ventana crece progresivamente a medida que se avanza en el *dataframe*.

9. **Sección 6 – Dataset final (limpieza, X/y y metadatos)**: Se eliminan los registros iniciales que contienen valores NaN generados por las operaciones de *lags*, *rolling* y *expanding*. Posteriormente, se separan las variables de entrada (X) y la variable objetivo (y). Adicionalmente, se construye un diccionario de **metadatos** (meta) que almacena información clave del proceso, como el valor de τ, las ventanas *rolling* seleccionadas y los *lags* finales utilizados, facilitando la trazabilidad y reproducibilidad del experimento.

Nota aclaratoria: En este caso se calcula la ACF sobre la serie de precios (*close*) con el fin de estimar la memoria temporal del nivel de precios, y no de los retornos, dado que el objetivo es capturar estructuras de dependencia de mediano plazo.

Ahora, aplicamos la función y obtenemos:

```
X_fe, y_fe, price_df_fe, fe_meta = add_statistical_fe(price_df)

print("Selección FE (estadística):")
print(fe_meta)
print("Shape con FE:", X_fe.shape, y_fe.shape)
```

Donde:

- X_fe: es el conjunto de features resultante de aplicar ingeniería de características. Incluye nuevas características tipo: retardo (*lags*), estadísticas móviles (*rolling*), estadísticas acumuladas (*expanding*).

- y_fe: corresponde a la variable objetivo, que en este caso es target, alineada con X_fe. Ambas tienen la misma cantidad de registros y el mismo índice temporal.

- price_df_fe: es el dataframe completo que contiene las variables originales, las nuevas variables obtenidas con FE, y la salida. Es útil para fines de inspección, EDA, o visualización de las características construidas.

- fe_meta: es un diccionario con los parámetros seleccionados automáticamente durante el proceso de FE, como los retardos elegidos mediante *mutual information* y los tamaños de ventana derivados de ACF. Este objeto permite documentar y reproducir las decisiones tomadas durante la ingeniería de características.

Obtenemos como resultado:

Selección FE (estadística):
{'tau_acf': 20, 'acf_threshold': 0.05, 'rolling_windows': [10, 20, 40], 'selected_lags_mi': [1, 2, 3, 4, 6, 9, 10, 12], 'top_k_lags': 8, 'max_lag_considered': 60}

Shape con FE: (146463, 34) (146463,)

Es decir, que nuestro dataset contiene ahora 34 columnas, el valor de tau encontrado fue de 20, por lo que el tamaño de ventanas para Rolling es de 10, 20 y 40 (por la regla explicada anteriormente). Se trabajaron con ochos valores de lags comprendidos entre 1 y 12.

Después de haber aplicada FE es importante verificar que la distribución del dataset no haya sido drásticamente modificada, para ello:

```
y_fe.value_counts(normalize=True)
```

Y obtenemos que la clase “1” tiene el 52% y la clase “0” tiene el 48% de los datos, al igual que el dataset original.

Finalmente, podemos realizar algunas gráficas de los nuevos features. Por ejemplo, de tipo Rolling para la variable close, en relación con el feature original, así:

```
cols = ['close', 'close_roll_mean_10', 'close_roll_mean_20']

# Seleccionar un tramo continuo y eliminar NaNs
plot_df = price_df_fe[cols].dropna().iloc[1200:1500]

plot_df.plot(figsize=(10, 4))
plt.title("Ventanas rolling del feature Close")
plt.xlabel("Tiempo")
plt.ylabel("Valor")
plt.tight_layout()
plt.show()
```

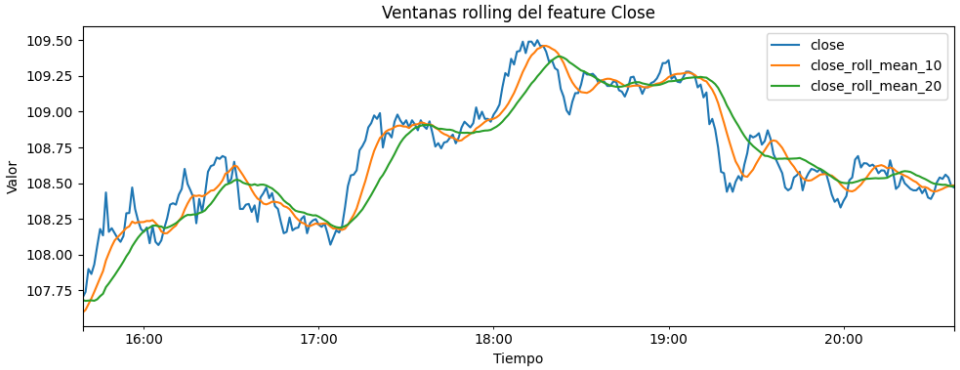

Figura 58. Ejemplo de nuevos *features* tipo Rolling, a partir de Close.

La información del nuevo dataset se presenta a continuación:

```
price_df_fe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 146463 entries, 2020-01-02 15:09:00+00:00 to 2021-07-08 19:59:00+00:00
Data columns (total 35 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   close               146463 non-null  float64
 1   volume              146463 non-null  float64
 2   target              146463 non-null  int64
 3   close_lag_1         146463 non-null  float64
 4   close_lag_2         146463 non-null  float64
 5   close_lag_3         146463 non-null  float64
 6   close_lag_4         146463 non-null  float64
 7   close_lag_6         146463 non-null  float64
 8   close_lag_9         146463 non-null  float64
 9   close_lag_10        146463 non-null  float64
 10  close_lag_12        146463 non-null  float64
 11  vol_lag_1           146463 non-null  float64
 12  vol_lag_2           146463 non-null  float64
 13  vol_lag_3           146463 non-null  float64
 14  vol_lag_4           146463 non-null  float64
 15  vol_lag_6           146463 non-null  float64
 16  vol_lag_9           146463 non-null  float64
 17  vol_lag_10          146463 non-null  float64
 18  vol_lag_12          146463 non-null  float64
 19  close_roll_mean_10  146463 non-null  float64
 20  close_roll_std_10   146463 non-null  float64
 21  vol_roll_mean_10    146463 non-null  float64
 22  vol_roll_std_10     146463 non-null  float64
 23  close_roll_mean_20  146463 non-null  float64
 24  close_roll_std_20   146463 non-null  float64
 25  vol_roll_mean_20    146463 non-null  float64
 26  vol_roll_std_20     146463 non-null  float64
 27  close_roll_mean_40  146463 non-null  float64
 28  close_roll_std_40   146463 non-null  float64
 29  vol_roll_mean_40    146463 non-null  float64
 30  vol_roll_std_40     146463 non-null  float64
 31  close_exp_mean      146463 non-null  float64
 32  close_exp_std       146463 non-null  float64
 33  vol_exp_mean        146463 non-null  float64
 34  vol_exp_std         146463 non-null  float64
```

Figura 59. Información del dataset twlo_prices.csv, posterior a FE.

4.4.4. Entrenamiento del modelo y predicción con FE

Vamos a utilizar el mismo tipo de modelo utilizado antes de aplicar FE. El único cambio que hacemos es que nuestra entrada ahora es X_fe.

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

# 1) Split temporal (70/30)
split = int(len(X_fe) * 0.7)
X_train, X_test = X_fe.iloc[:split], X_fe.iloc[split:]
y_train, y_test = y_fe.iloc[:split], y_fe.iloc[split:]

# 2) Entrenar y predecir
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# 3) Métricas
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nMatriz de Confusión:")
cm = confusion_matrix(y_test, y_pred)
print(cm)

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred, target_names=["No subió",
"Subió"]))

# 4) Heatmap simple (sin seaborn)
plt.figure(figsize=(6, 5))
plt.imshow(cm)
plt.title("Matriz de Confusión")
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.xticks([0, 1], ["No subió", "Subió"])
plt.yticks([0, 1], ["No subió", "Subió"])
```

```
# Valores dentro de la matriz
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, cm[i, j], ha="center", va="center")

plt.colorbar()
plt.tight_layout()
plt.show()
```

Y obtenemos como resultado:

```
Accuracy: 0.53

Matriz de Confusión:
[[23380     0]
 [20559     0]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.53      1.00      0.69     23380
       Subió       0.00      0.00      0.00     20559

    accuracy                           0.53     43939
   macro avg       0.27      0.50      0.35     43939
weighted avg       0.28      0.53      0.37     43939
```

Figura 60. Resultado del modelo posterior a la ingeniería de características, utilizando un árbol de decisión con dos niveles de profundidad.

El modelo resultante presenta un comportamiento claramente **sesgado hacia la clase "No subió"**, clasificando todos los registros dentro de esta categoría. Sin embargo, este resultado no debe interpretarse como un error metodológico, sino como una **limitación del modelo utilizado**.

Tras la etapa de ingeniería de características, el nuevo *dataset* contiene un número significativamente mayor de *features* en comparación con el daraset original. Un árbol de decisión con únicamente **dos niveles de profundidad** no tiene la capacidad suficiente para explotar esta mayor complejidad, ya que

dispone de muy pocas divisiones (preguntas) para separar adecuadamente el espacio de características.

Como una mejora sencilla desde un enfoque **model-centric**, se incrementa la profundidad máxima del árbol de decisión, por ejemplo, a **cuatro niveles**, permitiendo al modelo capturar relaciones más complejas entre las variables de entrada. Para ello, se define el siguiente modelo:

```
model = DecisionTreeClassifier(
    criterion="entropy",
    max_depth=4,
    min_samples_leaf=50,
    random_state=42
)
```

Posteriormente, se ejecuta nuevamente el proceso de modelamiento utilizando el daraset con ingeniería de características (X_fe), obteniendo los resultados mostrados en la Figura 50.

```
Accuracy: 0.56

Matriz de Confusión:
[[ 8188 15192]
 [ 4043 16516]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.67      0.35      0.46     23380
       Subió       0.52      0.80      0.63     20559

    accuracy                           0.56     43939
   macro avg       0.60      0.58      0.55     43939
weighted avg       0.60      0.56      0.54     43939
```

Figura 61. Resultado del modelo posterior a la ingeniería de características, utilizando un árbol de decisión con cuatro niveles de profundidad.

Al comparar el Modelo 1 (sin ingeniería de características) con **el Modelo 3 (con ingeniería de características y árbol de cuatro niveles)**, se observan los siguientes resultados:

- El *accuracy* de ambos modelos es de **0.56**.
- Las métricas de **precisión (P)**, **recall (R)** y **F1-score** son iguales en ambos casos.
- El número de aciertos para la clase **"Subió"** en el Modelo 3 es de **16 516**, mientras que en el Modelo 1 es de **16 481**.
- El número de aciertos para la clase **"No subió"** en el Modelo 3 es de **8 188**, frente a **8 207** en el Modelo 1.
- La cantidad total de aciertos del Modelo 3 es de **24 704**, mientras que en el Modelo 1 es de **24 688**, lo que representa una diferencia de **19 registros**.

Aunque la diferencia entre ambos modelos es pequeña en términos absolutos, su interpretación depende del contexto del problema. Por ejemplo, si cada registro representara una transacción o un usuario, estos 19 casos adicionales correctamente clasificados podrían traducirse en decisiones más acertadas en un entorno real, lo cual ***no resulta despreciable en aplicaciones prácticas***.

4.5. Conclusiones de cierre del capitulo

En este capítulo se abordó el análisis de series de tiempo a través de dos casos de estudio con características y niveles de complejidad distintos. En el primer caso, basado en un dataset climático, se mostró cómo la incorporación sistemática de ingeniería de características temporales puede mejorar de forma significativa el desempeño de un modelo de clasificación, incluso sin modificar su arquitectura. Este ejemplo permitió introducir conceptos fundamentales como dependencia temporal, autocorrelación y construcción de características basadas en retardos, ventanas móviles y estadísticas acumuladas, enfatizando la importancia de una representación adecuada de los datos.

En contraste, el segundo caso de estudio, basado en datos financieros de alta frecuencia, evidenció que la ingeniería de características no garantiza necesariamente una mejora automática en el desempeño del modelo. A pesar de emplear criterios estadísticos para la selección de retardos y ventanas, y de aplicar un proceso cuidadoso para evitar *data leakage*, los resultados muestran que la naturaleza altamente ruidosa y no estacionaria de las series financieras impone limitaciones inherentes al proceso de modelamiento. Este contraste refuerza una idea central del capítulo: en ciencia de datos, las decisiones metodológicas deben estar guiadas tanto por el análisis de los datos como por el contexto del problema, y no todas las técnicas producen los mismos beneficios en todos los escenarios.