

Pontificia Universidad Javeriana

Facultad de Ciencias, Matemáticas

Análisis de Datos con Python

Desarrollo de un Flujo de Trabajo Reproducible

para el Análisis de Datos

Autor:

Wolfgang Sánchez

Colombia, Bogotá D. C.

Octubre 2025

Resumen

Este proyecto tiene como objetivo diseñar e implementar, en Python, un flujo de trabajo reproducible para análisis de datos, aplicable a distintos conjuntos de datos públicos. Se cubren de manera integrada todas las etapas del proceso analítico: preparación del entorno y estructura de carpetas, adquisición programática de datos desde una API abierta, limpieza y transformación sistemática de las tablas, análisis exploratorio (EDA), modelado de series de tiempo y generación automatizada de productos (gráficas, tablas y archivos intermedios) listos para ser reutilizados.

El flujo de trabajo se organiza en scripts y notebooks que documentan cada paso, utilizando librerías como pandas, matplotlib, statsmodels y scikit-learn. Se enfatiza la reproducibilidad mediante: descarga programática de datos en bloques, criterios explícitos de limpieza (tratamiento de valores faltantes, duplicados y outliers), creación de variables derivadas, separación clara entre datos crudos y procesados, uso de estructuras de directorios consistentes (data/raw, data/processed, data/modelos, data/graficas) y exportación sistemática de resultados.

Sobre este flujo general, se implementa un módulo de análisis de series de tiempo que incluye construcción de rezagos, evaluación de estacionariedad, diferenciación, ajuste de modelos ARIMA/SARIMA, regresiones lineales en el tiempo y evaluación de pronósticos mediante métricas estándar (MAE, RMSE, MAPE). El proyecto se entrega acompañado de un informe técnico y de notebooks de Jupyter que permiten re-ejecutar todo el pipeline desde la descarga de los datos hasta la obtención de las figuras y modelos, de forma transparente y replicable por terceros. Un caso de estudio concreto, descrito en el apéndice, ilustra la aplicación de esta metodología a un dataset real.

Índice general

1. Introducción	5
1.1. Breve descripción de la propuesta	5
1.2. Relevancia y justificación	5
1.2.1. Pregunta de investigación	5
1.3. Objetivos	6
1.3.1. Objetivo General	6
1.3.2. Objetivos específicos	6
2. Marco Teórico y Revisión de Literatura	7
2.1. Ciencia de datos y análisis reproducible	7
2.1.1. Reproducibilidad y trazabilidad	7
2.2. Herramientas y estructuras de datos en Python	8
2.2.1. Arreglos de NumPy y estructuras de pandas	8
2.2.2. Series de tiempo y datos transversales	8
2.3. Adquisición de datos mediante APIs	9
2.3.1. Paginación y descarga programática	9
2.4. Limpieza y preparación de datos	9
2.4.1. Valores faltantes	10
2.4.2. Conversión de tipos de datos	10
2.4.3. Detección de duplicados	11
2.4.4. Detección de outliers	11
2.4.5. Creación de variables derivadas	11
2.5. Análisis exploratorio de datos (EDA)	12
2.5.1. Estadística descriptiva	12
2.5.2. Distribuciones univariadas y comparaciones	12
2.5.3. Correlaciones y relaciones bivariadas	12
2.5.4. Análisis temporal exploratorio	13
2.6. Visualización de datos	13

2.7.	Series de tiempo: conceptos fundamentales	14
2.7.1.	Estacionariedad	14
2.7.2.	Autocorrelación y autocorrelación parcial	14
2.8.	Modelos ARIMA y SARIMA	15
2.8.1.	Modelos AR y MA	15
2.8.2.	Modelo ARIMA	15
2.8.3.	Modelo SARIMA	15
2.8.4.	Selección de modelos y criterios de información	16
2.9.	Regresión lineal y métricas de evaluación	16
2.9.1.	Modelo de regresión lineal	16
2.9.2.	Métricas de desempeño	17
2.10.	Diseño de pipelines reproducibles	17
3.	Metodología Implementada	19
3.1.	Visión General del Pipeline	19
3.2.	Fase 1: Configuración del Entorno	19
3.2.1.	Instalación y Gestión de Librerías	19
3.2.2.	Importación de Librerías	20
3.2.3.	Entorno de Desarrollo	20
3.2.4.	Ventajas del Entorno Virtual	21
3.2.5.	Configuración Técnica del Entorno	21
3.2.6.	Extensiones de Visual Studio Code	21
3.2.7.	Estructura del Proyecto	22
3.2.8.	Configuración de Git y Control de Versiones	22
3.2.9.	Ventajas de la Estructura Implementada	23
3.2.10.	Integración con GitHub	23
3.3.	Fase 2: Selección y Adquisición de Datos	24
3.3.1.	Criterios de Selección del Dataset	24
3.3.2.	Implementación de la Descarga por API	24
3.4.	Fase 3: Limpieza y Transformación	26
3.4.1.	Análisis de Valores Nulos	26
3.4.2.	Manejo de Valores Faltantes	27
3.4.3.	Conversión de Tipos de Datos	29
3.4.4.	Detección de Outliers	30
3.4.5.	Creación de Variables Derivadas	31
3.4.6.	Eliminación de Duplicados	32

3.4.7. Validación y Exportación de Datos Limpios	33
3.5. Fase 4: Análisis Exploratorio y Visualización	35
3.5.1. Introducción al Análisis Exploratorio de Datos	35
3.5.2. Arquitectura del Pipeline de EDA	35
3.5.3. Análisis Estadístico Descriptivo	36
3.5.4. Análisis de Distribución	37
3.5.5. Análisis Temporal Avanzado	38
3.5.6. Análisis de Estacionalidad y Patrones Temporales	39
3.5.7. Análisis de Autocorrelación	40
3.5.8. Detección de Outliers y Anomalías	41
3.5.9. Análisis de Volatilidad y Correlaciones	43
3.5.10. Análisis Comparativo por Años	44
3.5.11. Análisis mediante tablas dinámicas (pivot tables)	45
3.5.12. Regresión lineal	45
3.5.13. Tipos de Estructuras de Datos	48
3.5.14. Series de Tiempo (Time Series)	48
3.5.15. Datos Transversales (Cross-Sectional)	48
3.5.16. Análisis de Series de Tiempo	49
3.5.17. Componentes de las Series de Tiempo	49
3.5.18. Visualización de Series de Tiempo	49
3.5.19. Análisis de Múltiples Series	50
3.5.20. Análisis de Estacionariedad	51
3.5.21. Prueba de Dickey-Fuller Aumentada	51
3.5.22. Modelos de Series de Tiempo	52
3.5.23. Fundamentos de los Modelos ARIMA	52
3.5.24. Modelo AR (AutoRegresivo)	54
3.5.25. Modelo MA (Media Móvil)	55
3.5.26. Modelo ARIMA	55
3.5.27. Modelo SARIMA	56
3.5.28. Selección y Evaluación de Modelos	57
3.5.29. Identificación de Parámetros	57
3.5.30. Evaluación de Pronósticos	58
3.5.31. Pipeline Integrado de Análisis de Series Temporales	59
3.5.32. Análisis de Datos Transversales	61
3.6. Fase 5: Documentación	61
3.6.1. Documentación en Jupyter Notebooks	61

3.6.2. Metodología de Documentación	62
3.6.3. Notebooks Implementados	62
3.7. Control de Versiones y Reproducibilidad	62
4. Conclusiones	64
Glosario de Términos	67
Glosario de Términos	68
A. Análisis exhaustivo y resultados completos del pipeline reproducible	
Caso de estudio: fondos de pensiones y cesantías Colombia (2016–2025 de noviembre de 2025)	76
A.1. Resumen Ejecutivo del Pipeline Ejecutado	76
A.2. Análisis Descriptivo Profundo	77
A.2.1. Estadísticas descriptivas generales (2016–2025)	77
A.2.2. Distribución y densidad	78
A.2.3. Evolución temporal y tendencias	79
A.3. Comparación con la Inflación (IPC Colombia)	81
A.4. Volatilidad, Drawdown y Riesgo	82
A.5. Correlaciones	83
A.6. Regresión Lineal en el Tiempo (Tendencia de Largo Plazo)	85
A.7. Diagnóstico previo al modelado ARIMA/SARIMA	85
A.7.1. Calidad de datos y comportamiento de nulos	86
A.7.2. Estacionalidad y descomposición clásica	87
A.7.3. Autocorrelación, PACF y motivación del modelo ARIMA	89
A.7.4. Prueba ADF y número de diferenciaciones: Cesantías Largo Plazo Col-fondos	89
A.8. Modelado Predictivo: SARIMA (mejor desempeño real)	90
A.9. Conclusiones Clave del Caso de Estudio	91

Capítulo 1

Introducción

1.1. Breve descripción de la propuesta

El análisis de datos se ha convertido en una competencia fundamental en investigación científica e industria. Python, como lenguaje de programación, ha emergido como una herramienta dominante en este campo debido a su ecosistema de librerías especializadas (pandas, NumPy, matplotlib, scikit-learn, SciPy) y su naturaleza de código abierto. El estado del arte en análisis de datos enfatiza la importancia de la reproducibilidad, la documentación exhaustiva y la aplicación de buenas prácticas en el flujo de trabajo analítico.

Este proyecto se enfoca en diseñar y ejecutar un flujo de trabajo (pipeline) completo de análisis de datos aplicado a un conjunto de datos público, abordando desde la configuración del entorno hasta la comunicación de hallazgos, con especial atención al análisis de distribuciones de variables numéricas. La relevancia del tema radica en la creciente demanda de profesionales capaces de implementar flujos de análisis robustos y reproducibles en diversos contextos.

1.2. Relevancia y justificación

1.2.1. Pregunta de investigación

¿Cómo diseñar y ejecutar un flujo reproducible de análisis de datos en Python, desde la limpieza hasta la comunicación de hallazgos, aplicado a un conjunto de datos público?

La motivación principal de este proyecto es consolidar competencias transferibles a investigación e industria, permitiendo la creación de un portafolio técnico presentable y el dominio de herramientas estándar en el análisis de datos.

1.3. Objetivos

1.3.1. Objetivo General

Desarrollar habilidades de programación para análisis de datos en Python y aplicarlas en un pipeline reproducible sobre un conjunto de datos público.

1.3.2. Objetivos específicos

- Configurar un entorno profesional de desarrollo con gestión de entornos virtuales y estructura de proyecto organizada.
- Investigar, seleccionar y adquirir un dataset público de interés personal.
- Implementar procesos de ingesta, depuración y transformación de datos usando `pandas` y `NumPy`.
- Realizar análisis exploratorio de datos (EDA) con estadísticas descriptivas, incluyendo el estudio de distribuciones de variables numéricas y visualizaciones utilizando `matplotlib`, `seaborn` y `SciPy`.
- Implementar y contrastar modelos paramétricos de series de tiempo, incluyendo componentes autoregresivos (AR), de medias móviles (MA), modelos ARIMA y SARIMA con estacionalidad, así como un modelo de regresión lineal simple, para predecir valores futuros utilizando las librerías `scikit-learn` y `statsmodels`.
- Documentar resultados efectivamente mediante notebooks de `Jupyter` y un informe técnico breve.
- Publicar un repositorio completo en GitHub con instrucciones claras de ejecución y reproducibilidad.

Capítulo 2

Marco Teórico y Revisión de Literatura

En esta sección se presentan los conceptos teóricos y antecedentes que sustentan el flujo de trabajo reproducible implementado en los capítulos posteriores. El objetivo es que cada decisión técnica descrita en la metodología (Capítulo 3) tenga una justificación conceptual clara, apoyada en la literatura especializada en análisis de datos, estadística y series de tiempo.

2.1. Ciencia de datos y análisis reproducible

La ciencia de datos puede entenderse como la intersección entre estadística, programación y conocimiento de dominio, con el propósito de extraer información útil a partir de datos estructurados y no estructurados. En este contexto, la reproducibilidad ocupa un lugar central: un análisis es reproducible cuando un tercero, con acceso al mismo código y los mismos datos, puede obtener los mismos resultados siguiendo los mismos pasos **casella2002; mckinney2018**.

2.1.1. Reproducibilidad y trazabilidad

Siguiendo a **casella2002**, la confianza en los resultados científicos depende de la posibilidad de verificar de manera independiente los procesos de cálculo y los supuestos subyacentes. En el ámbito computacional, **mckinney2018** enfatiza varios elementos clave para garantizar reproducibilidad:

- **Gestión explícita de dependencias:** especificar versiones de librerías y del intérprete de Python.

- **Separación entre datos crudos y procesados:** los datos originales nunca deben sobrescribirse.
- **Control de versiones:** registrar cada cambio en el código y la estructura del proyecto.
- **Documentación ejecutable:** uso de notebooks que combinan texto, fórmulas, código y resultados.

En el proyecto, estos principios se materializan mediante un entorno virtual (`venv`), un archivo `requirements.txt`, un repositorio Git organizado por ramas y una estructura de carpetas estandarizada, tal como se detalla en la Fase 1 del Capítulo 3.

2.2. Herramientas y estructuras de datos en Python

Python se ha consolidado como uno de los lenguajes de referencia para análisis de datos gracias a ecosistemas como NumPy, pandas y matplotlib, que proporcionan estructuras de datos eficientes y herramientas de análisis de alto nivel **mckinney2018**.

2.2.1. Arreglos de NumPy y estructuras de pandas

mckinney2018 destaca dos estructuras fundamentales:

- **ndarray de NumPy:** arreglo n -dimensional eficiente para operaciones numéricas vectorizadas.
- **DataFrame de pandas:** tabla bidimensional etiquetada, análoga a una hoja de cálculo o tabla SQL.

El DataFrame es la unidad de trabajo principal en este proyecto; todas las funciones de limpieza, EDA y modelado reciben y devuelven DataFrames, lo que facilita la composición de funciones en un pipeline.

2.2.2. Series de tiempo y datos transversales

Siguiendo la clasificación estándar en estadística aplicada:

- **Datos transversales** (cross-sectional): observaciones de múltiples unidades en un único instante de tiempo.

- **Series de tiempo:** observaciones de una misma unidad (o conjunto de unidades) recolectadas a lo largo del tiempo, en intervalos regulares.

datasciencedojo2023 enfatiza que las series temporales presentan dependencia entre observaciones consecutivas, lo que exige técnicas específicas de modelado (ARIMA, SARIMA) y diagnósticos de estacionariedad. En el proyecto, el `DataFrame` se indexa por fechas para facilitar la resampleación, el cálculo de medias móviles y el ajuste de modelos de series de tiempo.

2.3. Adquisición de datos mediante APIs

La obtención de datos a través de APIs REST es una práctica generalizada cuando se trabaja con fuentes públicas o institucionales. Una API expone endpoints que devuelven datos estructurados (típicamente en formato JSON) bajo un protocolo bien definido (por ejemplo, HTTP GET con parámetros de consulta).

2.3.1. Paginación y descarga programática

En datasets voluminosos, las APIs suelen implementar **paginación**: el usuario debe solicitar bloques de registros especificando parámetros como `limit` y `offset`. **mckinney2018** recomienda automatizar este proceso para evitar errores manuales, controlar tiempos de espera y manejar fallos de red.

En el pipeline implementado se adopta este enfoque:

- Verificación previa del número total de registros disponibles.
- Descarga iterativa por bloques con `offset` creciente.
- Control de errores mediante bloques `try/except` y tiempos de espera (`timeout`) configurables.

La función genérica de descarga por API, descrita en la Fase 2 del Capítulo 3, se basa en estas recomendaciones para asegurar que la adquisición de datos sea reproducible y robusta.

2.4. Limpieza y preparación de datos

La limpieza (data cleaning o data wrangling) es el proceso de transformar datos crudos en un formato coherente, consistente y analizable. **mckinney2018** y **dataCleaning** subrayan

que esta fase suele consumir la mayor parte del tiempo de un proyecto de datos y que debe implementarse de forma sistemática.

2.4.1. Valores faltantes

Los valores faltantes pueden surgir por errores de captura, cambios en la estructura de registro o ausencias legítimas. Las estrategias típicas incluyen:

- **Eliminación:** descartar filas o columnas cuando la proporción de valores faltantes es elevada.
- **Imputación simple:** reemplazo por media, mediana o moda, según el tipo de variable.
- **Imputación avanzada:** métodos basados en modelos, como imputación por vecinos más cercanos (KNN) o regresión múltiple.

En el proyecto, se implementa una función que identifica el porcentaje de valores nulos por columna y asigna una estrategia de imputación según umbrales (por ejemplo, menos del 5 %, entre 5 % y 30 %, más del 30 %), siguiendo recomendaciones prácticas similares a las discutidas en **dataCleaning**.

2.4.2. Conversión de tipos de datos

Los datos suelen llegar en formatos heterogéneos (número como cadena, fechas como texto, categorías como **object**). La literatura de **mckinney2018** enfatiza la importancia de:

- Convertir fechas a tipo **datetime**.
- Convertir variables numéricas a **float** o **int** eliminando símbolos no numéricos.
- Representar variables categóricas como **category**, lo que reduce consumo de memoria y facilita ciertas operaciones.

La función de conversión de tipos implementada en el Capítulo 3 aplica estas ideas de manera parametrizable, de modo que el usuario puede especificar el tipo objetivo de cada columna.

2.4.3. Detección de duplicados

Se distinguen dos tipos de duplicados:

- **Duplicados exactos:** filas idénticas en todas las columnas.
- **Duplicados conceptuales:** filas repetidas según un subconjunto de columnas clave (por ejemplo, fecha + entidad + tipo de fondo).

La eliminación de duplicados evita el sesgo en estadísticas descriptivas y modelos. El proyecto implementa funciones para detectar y eliminar ambos tipos, asegurando que cada combinación clave aparezca una sola vez en el dataset final.

2.4.4. Detección de outliers

Los valores atípicos (*outliers*) pueden distorsionar medidas agregadas y ajustar modelos inestablemente. Un criterio clásico basado en el rango intercuartílico (IQR) identifica como outliers las observaciones que se encuentran fuera del intervalo:

$$[Q_1 - 1,5 \text{ IQR}, Q_3 + 1,5 \text{ IQR}],$$

donde Q_1 y Q_3 son el primer y tercer cuartil, respectivamente. Este enfoque, recomendado en textos introductorios de estadística robusta y popularizado en librerías como **pandas mckinney2018**, es el que se implementa en la función de detección de outliers descrita en la Fase 3.

2.4.5. Creación de variables derivadas

La generación de *features* derivadas es una técnica fundamental para enriquecer el análisis. En datos temporales, es habitual extraer componentes como:

- Año, mes, trimestre.
- Día de la semana.
- Indicadores de fin/inicio de mes o de fin de trimestre.

La creación de estas variables permite agrupar y resumir el comportamiento de la serie temporal en diferentes escalas, lo cual se explota en el análisis comparativo por años y en tablas dinámicas en el Capítulo 3.

2.5. Análisis exploratorio de datos (EDA)

El análisis exploratorio de datos (EDA) fue formalizado por Tukey como una fase inicial en la que se exploran los datos para descubrir patrones, identificar anomalías y formular hipótesis *antes* de aplicar modelos formales. En la práctica moderna, el EDA combina resúmenes numéricos, visualizaciones y pruebas básicas de supuestos **mckinney2018; edaPandas**.

2.5.1. Estadística descriptiva

Las herramientas clásicas de EDA incluyen:

- Medidas de tendencia central: media, mediana.
- Medidas de dispersión: varianza, desviación estándar, rango intercuartílico.
- Medidas de forma: asimetría (*skewness*) y curtosis.

En el pipeline se implementa una función de análisis estadístico descriptivo que devuelve estadísticas básicas globales y desagregadas por categoría (por ejemplo, tipo de fondo), además de *skewness* y *kurtosis*, utilizando `pandas` y `scipy`.

2.5.2. Distribuciones univariadas y comparaciones

mckinney2018 y **pivotalstats2023** recomiendan inspeccionar la forma de las distribuciones mediante:

- Histogramas y estimadores de densidad (KDE).
- Diagramas de caja (boxplots).
- Gráficos de violín para comparaciones entre categorías.

La función de análisis de distribución del proyecto genera, de forma automatizada, histogramas y curvas de densidad para las variables numéricas principales, tanto a nivel global como desagregadas por tipo de fondo u otras categorías.

2.5.3. Correlaciones y relaciones bivariadas

Las matrices de correlación permiten cuantificar la relación lineal entre variables numéricas. Representarlas mediante *heatmaps* facilita la identificación de patrones de dependencia y posibles problemas de multicolinealidad. En el caso de series de tiempo, también es útil

analizar la correlación entre series asociadas a distintas entidades o categorías a lo largo del tiempo.

En el pipeline se implementa una función que construye tablas dinámicas (`pivot_table`) y calcula matrices de correlación por categorías, visualizadas con `seaborn.heatmap`.

2.5.4. Análisis temporal exploratorio

Cuando los datos poseen una dimensión temporal, el EDA incluye:

- Gráficos de evolución temporal (serie en niveles).
- Medias móviles (rolling mean) y desviaciones estándar móviles (rolling std).
- Análisis comparativo por años y por meses.

Estas herramientas, descritas en **datasciencedojo2023**, permiten distinguir entre variación de corto plazo, estacionalidad y tendencias de largo plazo. En el proyecto, se implementan funciones para calcular medias móviles de distintas ventanas y para generar gráficos comparativos por año, que se utilizan extensivamente en el apéndice de resultados.

2.6. Visualización de datos

La visualización es un componente esencial tanto del EDA como de la comunicación de resultados. Las buenas prácticas recomiendan:

- Etiquetas claras de ejes y unidades.
- Títulos informativos que describan la historia del gráfico.
- Escalas coherentes y consistentes entre figuras comparables.
- Uso moderado de color y énfasis visual sólo donde es necesario.

pivotalstats2023 muestra cómo librerías como `matplotlib`, `seaborn` y `plotly` pueden utilizarse para construir visualizaciones reproductibles. En este proyecto se estandariza el estilo de las figuras mediante `plt.style.use` y se guardan todas en carpetas organizadas, con nombres que codifican la variable y el tipo de análisis realizado. Esto facilita tanto la trazabilidad como su inclusión posterior en reportes y presentaciones.

2.7. Series de tiempo: conceptos fundamentales

Las series temporales se definen como secuencias ordenadas de observaciones X_t indexadas por un tiempo discreto $t = 1, 2, \dots$. **data sciencedojo 2023** y otros textos introductorios distinguen varios componentes:

- **Tendencia:** comportamiento de largo plazo (creciente, decreciente o estable).
- **Estacionalidad:** patrones que se repiten en intervalos regulares (por ejemplo, anual).
- **Ciclicidad:** fluctuaciones de mayor duración no necesariamente periódicas.
- **Componente irregular o residual:** variación aleatoria no explicada por los componentes anteriores.

La descomposición clásica aditiva asume que

$$X_t = T_t + S_t + R_t,$$

donde T_t es la tendencia, S_t la estacionalidad y R_t el residuo. Esta descomposición es la base de las funciones de *seasonal decomposition* utilizadas en el pipeline.

2.7.1. Estacionariedad

Muchos modelos clásicos de series de tiempo requieren que la serie sea estacionaria, es decir, que sus propiedades estadísticas (media, varianza y autocorrelación) no cambien con el tiempo. Una forma común de lograr estacionariedad es aplicar la **diferenciación**:

$$Y_t = X_t - X_{t-1}.$$

La prueba de Dickey–Fuller aumentada (ADF) es una herramienta estándar para evaluar si una serie posee raíz unitaria (no estacionaria) o si puede considerarse estacionaria. En el proyecto, se implementa una función que aplica la prueba ADF y reporta el estadístico de prueba, el p-valor y los valores críticos, indicando si la serie requiere diferenciación adicional.

2.7.2. Autocorrelación y autocorrelación parcial

Las funciones de autocorrelación (ACF) y autocorrelación parcial (PACF) describen la dependencia entre X_t y sus rezagos X_{t-k} :

- La **ACF** mide la correlación total entre X_t y X_{t-k} .

- La **PACF** mide la correlación entre X_t y X_{t-k} una vez removida la influencia de los rezagos intermedios.

Estas funciones son herramientas diagnósticas clave para seleccionar los órdenes p y q de modelos ARIMA, y se calculan y grafican con **statsmodels** en el pipeline descrito en el Capítulo 3.

2.8. Modelos ARIMA y SARIMA

Los modelos ARIMA (AutoRegressive Integrated Moving Average) y SARIMA (Seasonal ARIMA) constituyen una familia de modelos lineales para series de tiempo ampliamente utilizada en pronóstico **datasciencedojo2023**.

2.8.1. Modelos AR y MA

Un modelo autoregresivo de orden p , AR(p), tiene la forma

$$X_t = c + \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \varepsilon_t,$$

donde ε_t es un término de error blanco. Un modelo de media móvil de orden q , MA(q), se escribe como

$$X_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}.$$

2.8.2. Modelo ARIMA

El modelo ARIMA(p, d, q) combina componentes AR, integración (diferenciación) y MA. De manera informal, el modelo se aplica sobre la serie diferenciada d veces:

$$(1 - B)^d X_t = \text{ARMA}(p, q),$$

donde B es el operador de rezago. En el proyecto, la elección de d se guía por la prueba ADF, mientras que p y q se seleccionan a partir de ACF/PACF y de criterios de información como AIC y BIC.

2.8.3. Modelo SARIMA

Cuando existe estacionalidad significativa, los modelos ARIMA pueden extenderse a SARIMA($p, d, q) \times (P, D, Q)_s$, donde s es el período estacional (por ejemplo, $s = 12$ para

datos mensuales). El componente estacional captura patrones que se repiten cada s períodos. `statsmodels` implementa estos modelos a través de clases como `SARIMAX`, que son las utilizadas en el pipeline de series temporales del Capítulo 3.

2.8.4. Selección de modelos y criterios de información

La comparación entre modelos se realiza típicamente mediante:

- **AIC** (Criterio de Información de Akaike).
- **BIC** (Criterio de Información Bayesiano).

Estos criterios penalizan la complejidad del modelo (número de parámetros) y ayudan a evitar el sobreajuste. En la implementación se realiza una búsqueda sobre una rejilla de parámetros (grid search) y se selecciona el modelo con menor AIC, que posteriormente se valida sobre un conjunto de datos de prueba.

2.9. Regresión lineal y métricas de evaluación

Además de los modelos ARIMA/SARIMA, el proyecto utiliza regresión lineal como herramienta flexible para capturar tendencias de largo plazo, especialmente en series que muestran un crecimiento más o menos lineal en el tiempo.

2.9.1. Modelo de regresión lineal

En su forma más simple, la regresión lineal relaciona una variable respuesta Y con una variable explicativa X :

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i, \quad i = 1, \dots, n.$$

En el caso de múltiples predictores, se utiliza la formulación matricial

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

donde \mathbf{X} es la matriz de diseño. Los parámetros se estiman mediante mínimos cuadrados ordinarios, lo que equivale a minimizar la suma de cuadrados de los residuos.

En el proyecto, la regresión lineal se implementa con `scikit-learn` (`LinearRegression`), separando sistemáticamente los datos en conjuntos de entrenamiento y prueba para evaluar la capacidad de generalización del modelo.

2.9.2. Métricas de desempeño

Para evaluar la calidad de un modelo de regresión o de pronóstico, se utilizan métricas estándar basadas en la diferencia entre valores observados y_i y valores ajustados \hat{y}_i :

- **Error absoluto medio (MAE):**

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

- **Error cuadrático medio (MSE) y raíz del error cuadrático medio (RMSE):**

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2, \quad \text{RMSE} = \sqrt{\text{MSE}}.$$

- **Error porcentual absoluto medio (MAPE):**

$$\text{MAPE} = \frac{100}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|.$$

En el contexto de pronósticos financieros, MAE y RMSE ofrecen una medida en las unidades originales del problema, mientras que MAPE proporciona una interpretación porcentual fácilmente comunicable. Todas estas métricas se calculan en funciones específicas del pipeline para comparar modelos y cuantificar el desempeño en datos de prueba.

2.10. Diseño de pipelines reproducibles

Finalmente, la literatura reciente en ciencia de datos enfatiza la importancia de estructurar los proyectos como **pipelines** modulares, donde cada etapa (adquisición, limpieza, EDA, modelado, visualización) se implementa como un conjunto de funciones reutilizables y composable **mckinney2018**. Este enfoque aporta:

- **Claridad:** cada fase tiene responsabilidades bien delimitadas.
- **Reutilización:** las funciones pueden aplicarse a nuevos datasets con cambios mínimos.
- **Automatización:** es posible ejecutar el flujo completo con un solo script.
- **Mantenibilidad:** los cambios en una fase se encapsulan sin afectar innecesariamente al resto.

El proyecto adopta este paradigma mediante:

- Scripts separados en `scripts/` para descarga, limpieza, EDA y modelado.
- Notebooks en `notebooks/` que documentan y ejemplifican cada fase.
- Funciones genéricas (por ejemplo, `ejecutar_eda_completo`, `pipeline_series_temporales`) que encapsulan la lógica del flujo de trabajo.

De esta forma, el Capítulo 3 puede interpretarse como la *implementación concreta* de los conceptos teóricos expuestos en este capítulo, garantizando una correspondencia directa entre teoría y práctica a lo largo de todo el proyecto.

Capítulo 3

Metodología Implementada

3.1. Visión General del Pipeline

El pipeline reproducible implementado sigue una estructura de cinco fases, de las cuales las tres primeras han sido completadas en el avance actual:

1. **Fase 1:** Configuración del entorno
2. **Fase 2:** Selección y adquisición de datos
3. **Fase 3:** Limpieza y transformación
4. **Fase 4:** Análisis exploratorio y visualización
5. **Fase 5:** Documentación

3.2. Fase 1: Configuración del Entorno

3.2.1. Instalación y Gestión de Librerías

Se implementó un sistema robusto de gestión de dependencias:

Listing 3.1: Archivo requirements.txt para dependencias del proyecto

```
1 pandas >=2.0.0
2 numpy >=1.24.0
3 matplotlib >=3.7.0
4 seaborn >=0.12.0
5 scipy >=1.10.0
6 scikit-learn >=1.2.0
```

```
7 requests >=2.28.0
8 jupyter >=1.0.0
9 pathlib2 >=2.3.0
10 openpyxl >=3.0.0
```

3.2.2. Importación de Librerías

Se estableció una estructura estandarizada para la importación de librerías:

Listing 3.2: Estructura estandarizada de importaciones

```
1 # Librerías fundamentales para análisis de datos
2 import pandas as pd
3 import numpy as np
4
5 # Visualización
6 import matplotlib.pyplot as plt
7 import seaborn as sns
8
9 # Análisis estadístico
10 from scipy import stats
11 from scipy.stats import norm, skew, kurtosis
12
13 # Adquisición de datos
14 import requests
15 from pathlib import Path
16 import os
17 import time
18
19 # Configuración para reproducibilidad
20 np.random.seed(42)
21 pd.set_option('display.max_columns', None)
22 plt.style.use('seaborn-v0_8-whitegrid')
```

3.2.3. Entorno de Desarrollo

Se estableció un entorno de desarrollo profesional que incluye:

- Visual Studio Code como editor principal con extensiones para Python, Jupyter y Git

- Entorno virtual Python 3.12.3 con gestión de dependencias mediante `venv`
- Control de versiones con Git y repositorio en GitHub
- Estructura de proyecto organizada en directorios especializados

3.2.4. Ventajas del Entorno Virtual

La implementación de un entorno virtual Python 3.12.3 proporciona múltiples ventajas para la reproducibilidad:

- **Aislamiento de dependencias:** Las librerías instaladas no interfieren con el sistema global de Python
- **Portabilidad:** El entorno puede replicarse fácilmente en otras máquinas
- **Consistencia:** Garantiza que varios trabajen con las mismas versiones

3.2.5. Configuración Técnica del Entorno

El entorno virtual se configuró con las siguientes especificaciones:

Listing 3.3: Creación y activación del entorno virtual

```

1 # Crear entorno virtual
2 python -m venv venv
3
4 # Activar entorno virtual (Windows)
5 venv\Scripts\activate
6
7 # Activar entorno virtual (Linux/Mac)
8 source venv/bin/activate
9
10 # Instalar dependencias
11 pip install -r requirements.txt

```

3.2.6. Extensiones de Visual Studio Code

Para mejorar la productividad y calidad del código, se instalaron las siguientes extensiones:

- **Python:** Soporte completo para desarrollo en Python

- **Jupyter**: Integración con notebooks de Jupyter
- **GitLens**: Mejora la integración con Git

3.2.7. Estructura del Proyecto

La organización del proyecto sigue las mejores prácticas de **mckinney2018python**:

Listing 3.4: Estructura de directorios del proyecto

```

1 proyecto-analisis-datos/
2     venv/                      # Entorno virtual
3     data/
4         raw/                    # Datos originales
5         processed/              # Datos limpios
6         interim/                # Datos intermedios
7     notebooks/                 # Jupyter notebooks
8         01_adquisicion.ipynb
9         02_limpieza.ipynb
10        03_analisis.ipynb
11     scripts/                  # Scripts Python reutilizables
12         descarga_datos.py
13         limpieza_datos.py
14         utilidades.py
15     reports/                  # Reportes y visualizaciones
16         figuras/
17         tablas/
18     tests/                     # Pruebas unitarias
19     .gitignore                  # Archivos ignorados por Git
20     requirements.txt            # Dependencias del proyecto
21     README.md                  # Documentación principal
22     environment.yml            # Configuración de entorno

```

3.2.8. Configuración de Git y Control de Versiones

Se implementó una estrategia robusta de control de versiones:

Listing 3.5: Configuración inicial de Git

```

1 # Inicializar repositorio
2 git init

```

```

3
4 # Configurar usuario
5 git config user.name "wolfgang7"
6 git config user.email "wolesanleo@gmail.com"
7
8 # Crear ramas principales
9 git checkout -b main
10 git checkout -b develop
11
12 # Archivo .gitignore para proyectos de datos
13 echo "#_Datos" >> .gitignore
14 echo "data/raw/" >> .gitignore
15 echo "data/processed/" >> .gitignore
16 echo "#_Entornos_virtuales" >> .gitignore
17 echo "venv/" >> .gitignore
18 echo "__pycache__/" >> .gitignore

```

3.2.9. Ventajas de la Estructura Implementada

La configuración del entorno proporciona los siguientes beneficios para la reproducibilidad:

- **Transparencia:** Cualquier persona puede replicar exactamente el entorno
- **Trazabilidad:** Cada cambio en el código está documentado en Git
- **Escalabilidad:** La estructura permite agregar nuevos datasets y análisis
- **Colaboración:** Múltiples personas pueden trabajar simultáneamente
- **Mantenibilidad:** El código está organizado de forma lógica y modular

3.2.10. Integración con GitHub

El repositorio se configuró con las siguientes características:

- **README.md:** Documentación completa del proyecto
- **LICENSE:** Licencia MIT para código abierto
- **Badges:** Indicadores de estado del proyecto

- **Issues:** Seguimiento de bugs y mejoras
- **Releases:** Versiones estables del código

3.3. Fase 2: Selección y Adquisición de Datos

3.3.1. Criterios de Selección del Dataset

Para demostrar el pipeline, se seleccionó un dataset público basado en:

- Presencia de variables numéricas continuas
- Tamaño manejable para procesamiento
- Disponibilidad mediante API
- Documentación clara
- Potencial para análisis distribucional

3.3.2. Implementación de la Descarga por API

Se implementó un sistema robusto de descarga con paginación para datasets grandes:

Listing 3.6: Implementación genérica de descarga por API

```

1 import requests
2 import pandas as pd
3 import time
4 from pathlib import Path
5
6 def descargar_datos_api(url, parametros_paginacion, headers=None,
7     timeout=120):
8     """
9         Función genérica para descargar datos con paginación
10    Parameters:
11        url (str): URL de la API
12        parametros_paginacion (dict): Parámetros de paginación
13        headers (dict): Headers para la solicitud
14        timeout (int): Timeout en segundos
15

```

```

16     """Returns:  

17     pd.DataFrame: Dataset completo  

18     """  

19     datos = []  

20     offset = 0  

21  

22     print("Iniciando descarga de datos...")  

23  

24     while True:  

25         params = {**parametros_paginacion, 'offset': offset}  

26  

27         try:  

28             respuesta = requests.get(url, params=params, headers=  

29                                     headers, timeout=timeout)  

30             respuesta.raise_for_status()  

31  

32             lote_datos = respuesta.json()  

33             if not lote_datos:  

34                 print("Descarga completada.")  

35                 break  

36  

37             datos.extend(lote_datos)  

38             offset += parametros_paginacion['limit']  

39             print(f"Descargadas: {offset} filas...")  

40  

41             # Pequeña pausa para no saturar el servidor  

42             time.sleep(0.3)  

43  

44         except requests.exceptions.RequestException as e:  

45             print(f"Error en la descarga: {e}")  

46             break  

47  

48     df = pd.DataFrame(datos)  

49     print(f"Descarga finalizada. Total de registros: {len(df)}")  

50     return df  

51  

52 # Función para verificar el total de registros disponibles

```

```

53     """
54     Verifica el total de registros disponibles en la API
55     """
56     try:
57         respuesta = requests.get(f"{url}?$select=count(*)")
58         respuesta.raise_for_status()
59         total = int(respuesta.json()[0]["count"])
60         print(f"Total de registros reportados: {total}")
61         return total
62     except Exception as e:
63         print(f"Error al verificar total de registros: {e}")
64         return None

```

3.4. Fase 3: Limpieza y Transformación

3.4.1. Análisis de Valores Nulos

Siguiendo las recomendaciones de [mckinney2018python](#), se implementó un análisis exhaustivo de valores nulos:

Listing 3.7: Análisis sistemático de valores nulos

```

1 def analizar_valores_nulos(df):
2     """
3     Analiza valores nulos en el dataset y aplica estrategias de
4     manejo
5
6     Parameters:
7     df (pd.DataFrame): Dataset a analizar
8
9     Returns:
10    dict: Estrategias de manejo por columna
11    """
12    print("== AN LISIS DE VALORES NULOS ==")
13
14    # Porcentaje de nulos por columna
15    nulos_por_columna = df.isnull().mean() * 100
    nulos_por_columna = nulos_por_columna[nulos_por_columna > 0].
        sort_values(ascending=False)

```

```

16
17     if len(nulos_por_columna) > 0:
18         print("Columnas con valores nulos:")
19         for columna, porcentaje in nulos_por_columna.items():
20             print(f"- {columna}: {porcentaje:.2f}%")
21
22     # Estrategias específicas por columna
23     estrategias = {}
24     for columna, porcentaje in nulos_por_columna.items():
25         if porcentaje < 5: # Menos del 5% de nulos
26             if df[columna].dtype in ['float64', 'int64']:
27                 estrategias[columna] = 'mediana'
28             else:
29                 estrategias[columna] = 'moda'
30         elif porcentaje < 30: # Entre 5% y 30% de nulos
31             estrategias[columna] = 'imputacion_avanzada'
32         else: # Más del 30% de nulos
33             estrategias[columna] = 'analizar_eliminacion'
34
35     return estrategias
36 else:
37     print("No se encontraron valores nulos en el dataset")
38     return {}

```

3.4.2. Manejo de Valores Faltantes

Listing 3.8: Estrategias de imputación específicas

```

1 def manejar_valores_faltantes_avanzado(df, estrategias):
2     """
3         Aplica estrategias de imputación según tipo de variable y
4         porcentaje de nulos
5
6         Parameters:
7             df (pd.DataFrame): Dataset a procesar
8             estrategias (dict): Estrategias por columna
9
10        Returns:

```

```

10 pd.DataFrame: Dataset con valores imputados
11 """
12 print("\n====MANEJO DE VALORES FALTANTES====")
13
14 for columna, estrategia in estrategias.items():
15     valores_anteriores = df[columna].isnull().sum()
16
17     if estrategia == 'mediana':
18         df[columna].fillna(df[columna].median(), inplace=True)
19         print(f"-{columna}: {valores_anteriores} nulos imputados con mediana")
20
21     elif estrategia == 'moda':
22         df[columna].fillna(df[columna].mode()[0], inplace=True)
23         print(f"-{columna}: {valores_anteriores} nulos imputados con moda")
24
25     elif estrategia == 'eliminar':
26         filas_anteriores = len(df)
27         df.dropna(subset=[columna], inplace=True)
28         filas_eliminadas = filas_anteriores - len(df)
29         print(f"-{columna}: {filas_eliminadas} filas eliminadas")
30
31     elif estrategia == 'imputacion_avanzada':
32         # Imputación usando KNN o en todos los demás sofisticados
33         from sklearn.impute import KNNImputer
34         imputer = KNNImputer(n_neighbors=5)
35         df[columna] = imputer.fit_transform(df[[columna]])
36         print(f"-{columna}: {valores_anteriores} nulos imputados con KNN")
37
38     elif estrategia == 'analizar_eliminacion':
39         print(f"-{columna}: Requiere análisis manual (alto porcentaje de nulos: {valores_anteriores/len(df)*100:.2f} %)")
40
41 # Verificación final

```

```

42     nulos_finales = df.isnull().sum().sum()
43     print(f"\n    Valores nulos despues del tratamiento:{nulos_finales}")
44
45     return df

```

3.4.3. Conversión de Tipos de Datos

Listing 3.9: Función genérica para conversión de tipos

```

1 def convertir_tipos_datos(df, configuracion_tipos):
2     """
3         Convierte tipos de datos segun configuracion
4
5     Parameters:
6         df (pd.DataFrame): Dataset a convertir
7         configuracion_tipos (dict): Configuracion de tipos por columna
8
9     Returns:
10        pd.DataFrame: Dataset con tipos convertidos
11    """
12    print("==> CONVERSIÓN DE TIPOS DE DATOS ==>")
13
14    for columna, tipo in configuracion_tipos.items():
15        if columna not in df.columns:
16            print(f"!- Advertencia: Columna '{columna}' no encontrada")
17            continue
18
19        if tipo == 'fecha':
20            df[columna] = pd.to_datetime(df[columna], errors='coerce')
21            print(f"!- {columna}: convertida a datetime")
22
23        elif tipo == 'numerico':
24            # Limpieza de caracteres no numericos
25            df[columna] = (
26                df[columna]

```

```

27         .astype(str)
28         .str.replace(r'^\d\-,\.]', '', regex=True)
29         .str.replace(',', '.', regex=False)
30     )
31     df[columna] = pd.to_numeric(df[columna], errors='coerce',
32                                 )
33     print(f"uu-{columna}: convertida a num r ico")
34
35 elif tipo == 'categoria':
36     df[columna] = df[columna].astype('category')
37     print(f"uu-{columna}: convertida a categoria")
38
39 elif tipo == 'texto':
40     df[columna] = df[columna].astype(str)
41     print(f"uu-{columna}: convertida a texto")
42
43 return df

```

3.4.4. Detección de Outliers

Listing 3.10: Detección genérica de outliers

```

1 def detectar_outliers_iqr(df, columnas_numericas):
2     """
3     Detecta outliers usando m todo IQR
4
5     Parameters:
6     df (pd.DataFrame): Dataset a analizar
7     columnas_numericas (list): Lista de columnas num ricas
8
9     Returns:
10    dict: Mscaras de outliers por columna
11    """
12    print("== DETECCION DE OUTLIERS (M TODO IQR) ==")
13
14    outliers = []
15
16    for columna in columnas_numericas:

```

```

17     if columna not in df.columns:
18         continue
19
20     Q1 = df[columna].quantile(0.25)
21     Q3 = df[columna].quantile(0.75)
22     IQR = Q3 - Q1
23     limite_inferior = Q1 - 1.5 * IQR
24     limite_superior = Q3 + 1.5 * IQR
25
26     mascara_outliers = (df[columna] < limite_inferior) | (df[
27         columna] > limite_superior)
28     outliers[columna] = mascara_outliers
29
30     num_outliers = mascara_outliers.sum()
31     porcentaje_outliers = (num_outliers / len(df)) * 100
32
33     print(f"---{columna}: {num_outliers} outliers ({
34         porcentaje_outliers:.2f}%)")
35     print(f"mites: [{limite_inferior:.2f}, {limite_superior:.2f}]")
36
37     return outliers

```

3.4.5. Creación de Variables Derivadas

Listing 3.11: Creación de features derivadas

```

1 def crear_variables_derivadas(df):
2     """
3     Crea variables derivadas para análisis
4
5     Parameters:
6     df (pd.DataFrame): Dataset original
7
8     Returns:
9     pd.DataFrame: Dataset con variables derivadas
10    """
11    print("== CREACIÓN DE VARIABLES DERIVADAS ==")

```

```

12
13     # Componentes temporales
14     if 'fecha' in df.columns:
15         df['a_o'] = df['fecha'].dt.year
16         df['mes'] = df['fecha'].dt.month
17         df['trimestre'] = df['fecha'].dt.quarter
18         df['dia_semana'] = df['fecha'].dt.dayofweek
19         print("Variables temporales creadas: a_o, mes, trimestre, dia_semana")
20
21     # Optimización de memoria para columnas categóricas
22     columnas_categoricas = df.select_dtypes(include=['object']).columns
23     for columna in columnas_categoricas:
24         if df[columna].nunique() / len(df) < 0.5: # Menos del 50% de valores nulos
25             df[columna] = df[columna].astype('category')
26             print(f'{columna}: optimizada como categoría')
27
28     return df

```

3.4.6. Eliminación de Duplicados

Listing 3.12: Eliminación sistemática de duplicados

```

1 def eliminar_duplicados_completo(df, columnas_conceptuales=None):
2     """
3     Elimina duplicados exactos y conceptuales
4
5     Parameters:
6     df (pd.DataFrame): Dataset a limpiar
7     columnas_conceptuales (list): Columnas para duplicados conceptuales
8
9     Returns:
10    pd.DataFrame: Dataset sin duplicados
11    """
12    print("== ELIMINACIÓN DE DUPLICADOS ==")

```

```

13
14     # Duplicados exactos
15     duplicados_exactos = df.duplicated().sum()
16     print(f"---Duplicados exactos identificados:{duplicados_exactos}")
17
18     if duplicados_exactos > 0:
19         df = df.drop_duplicates()
20         print(f"--Dataset despues de eliminar duplicados exactos:{len(df)} filas")
21
22     # Duplicados conceptuales
23     if columnas_conceptuales:
24         duplicados_conceptuales = df.duplicated(subset=
25             columnas_conceptuales).sum()
26         print(f"---Duplicados conceptuales identificados:{duplicados_conceptuales}")
27
28         if duplicados_conceptuales > 0:
29             df = df.drop_duplicates(subset=columnas_conceptuales,
30                                     keep='first')
31             print(f"--Dataset final despues de limpieza:{len(df)} filas")
32
33     return df

```

3.4.7. Validación y Exportación de Datos Limpios

Listing 3.13: Sistema de validación y exportación

```

1 def validar_y_exportar_datos(df, ruta_exportacion):
2     """
3     Valida la calidad de los datos y exporta el dataset limpio
4
5     Parameters:
6     df (pd.DataFrame): Dataset validado
7     ruta_exportacion (str): Ruta de exportacion
8

```

```

9    """Returns:
10   dict: M tricas de calidad
11   """
12   print("\n==== VALIDACION FINAL DEL DATASET ===")
13
14   # M tricas de calidad
15   metricas_calidad = {
16       'filas_totales': len(df),
17       'columnas_totales': len(df.columns),
18       'valores_nulos_totales': df.isnull().sum().sum(),
19       'porcentaje_nulos': (df.isnull().sum().sum() / (len(df) *
20                           len(df.columns)) * 100),
21       'duplicados_exactos': df.duplicated().sum(),
22       'memoria_mb': df.memory_usage(deep=True).sum() / 1024**2,
23       'fecha_procesamiento': pd.Timestamp.now().strftime('%Y-%m-%d
24                           %H:%M:%S')
24   }
25
26   print("M tricas de calidad del dataset:")
27   for metrica, valor in metricas_calidad.items():
28       if metrica != 'fecha_procesamiento':
29           print(f"  {metrica}: {valor}")
30
31   # Validar consistencia de tipos de datos
32   print("\nTipos de datos finales:")
33   print(df.dtypes)
34
35   # Exportar dataset limpio
36   Path(ruta_exportacion).parent.mkdir(parents=True, exist_ok=True)
37   df.to_csv(ruta_exportacion, index=False, encoding='utf-8')
38   print(f"\n  Dataset exportado a: {ruta_exportacion}")
39
40   # Exportar resumen de limpieza
41   ruta_resumen = Path(ruta_exportacion).parent / "resumen_limpieza
42   .csv"
43   pd.Series(metricas_calidad).to_csv(ruta_resumen)
44   print(f"  Resumen de limpieza exportado a: {ruta_resumen}")

```

```
44     return metricas_calidad
```

3.5. Fase 4: Análisis Exploratorio y Visualización

3.5.1. Introducción al Análisis Exploratorio de Datos

El análisis exploratorio de datos (EDA) constituye una fase fundamental en cualquier pipeline de análisis de datos, permitiendo comprender las características principales de los datasets y formular hipótesis iniciales. Según [tukey1977exploratory](#), el EDA se centra en el descubrimiento de patrones, detección de anomalías y verificación de supuestos mediante técnicas estadísticas y visualizaciones.

En el contexto de un pipeline reproducible, el EDA se implementa de manera sistemática y automatizada, garantizando que los mismos procedimientos puedan aplicarse consistentemente a diferentes conjuntos de datos.

3.5.2. Arquitectura del Pipeline de EDA

Se implementó un sistema modular de análisis exploratorio compuesto por 16 tipos de análisis diferentes, organizados en una secuencia lógica que garantiza reproducibilidad y exhaustividad.

Listing 3.14: Pipeline principal de análisis exploratorio

```
1 # Estructura principal del pipeline de EDA
2 def ejecutar_eda_completo(df, configuracion_eda):
3     """
4         Pipeline reproducible de análisis exploratorio
5
6     Parameters:
7         df (pd.DataFrame): Dataset limpio y procesado
8         configuracion_eda (dict): Configuración de análisis
9
10    Returns:
11        dict: Resultados y métricas del EDA
12    """
13    resultados = []
14
15    # 1. Análisis descriptivo básico
```

```

16 resultados['estadisticas'] = analisis_estadistico_descriptivo(df)
17
18 # 2. An lisis de distribuci n
19 resultados['distribucion'] = analisis_distribucion(df,
20   configuracion_eda)
21
22 # 3. An lisis temporal avanzado
23 resultados['temporal'] = analisis_temporal_avanzado(df,
24   configuracion_eda)
25
26 # 4. An lisis de correlaciones
27 resultados['correlaciones'] = analisis_correlaciones(df)
28
29 # 5. Detecci n de outliers
30 resultados['outliers'] = deteccion_outliers_avanzada(df)
31
32 # 6. Exportaci n de resultados
33 exportar_resultados_eda(resultados, configuracion_eda['
      ruta_salida'])
34
35 return resultados

```

3.5.3. Análisis Estadístico Descriptivo

El análisis descriptivo proporciona una comprensión fundamental de las variables numéricas mediante estadísticas resumidas y comparativas por categorías.

Listing 3.15: Función de análisis estadístico descriptivo

```

1 def analisis_estadistico_descriptivo(df, columnas_numericas):
2     """An lisis estad stico comprehensivo de variables num ricas"""
3
4     print("==ESTAD STICAS DESCRIPTIVAS==")
5
6     # Estad sticas b sicas
7     estadisticas_basicas = df[columnas_numericas].describe()
8

```

```

9   # Estadísticas por categoría
10  if 'tipo_fondo' in df.columns:
11      stats_por_categoria = df.groupby('tipo_fondo', observed=True
12          )[
13              columnas_numericas].describe()
14
15      # Métricas adicionales de forma
16      skewness = df[columnas_numericas].skew()
17      kurtosis = df[columnas_numericas].kurtosis()
18
19      return {
20          'estadisticas_basicas': estadisticas_basicas,
21          'stats_por_categoria': stats_por_categoria,
22          'skewness': skewness,
23          'kurtosis': kurtosis
24      }

```

3.5.4. Análisis de Distribución

El análisis de distribución examina la forma, dispersión y características fundamentales de las variables numéricas mediante histogramas, gráficos de densidad y comparaciones entre categorías.

Listing 3.16: Análisis de distribución de variables

```

1 def analisis_distribucion(df, columnas_numericas, ruta_guardado):
2     """Análisis comprehensivo de distribuciones"""
3
4     # Histograma y densidad combinados
5     plt.figure(figsize=(15, 5))
6
7     plt.subplot(1, 2, 1)
8     df[columnas_numericas].hist(bins=50, alpha=0.7)
9     plt.title('Distribución de Variables Numéricas')
10
11    plt.subplot(1, 2, 2)
12    df[columnas_numericas].plot(kind='density')
13    plt.title('Densidad de Variables Numéricas')
14

```

```

15 plt.savefig(f"{ruta_guardado}/distribucion_completa.png")
16 plt.close()
17
18 # Análisis por categorías si existen
19 if 'tipo_fondo' in df.columns:
20     for columna in columnas_numericas:
21         plt.figure(figsize=(12, 6))
22         for tipo in df['tipo_fondo'].unique():
23             subset = df[df['tipo_fondo'] == tipo]
24             plt.hist(subset[columna], bins=50, alpha=0.5,
25                      label=tipo, density=True)
26         plt.legend()
27         plt.savefig(f"{ruta_guardado}/densidad_{columna}.png")
28         plt.close()

```

3.5.5. Análisis Temporal Avanzado

El análisis temporal examina patrones de evolución, tendencias y comportamientos a lo largo del tiempo, esencial para datos financieros y series temporales.

Listing 3.17: Análisis temporal avanzado

```

1 def analisis_temporal_avanzado(df, columna_fecha, columna_numerica):
2     """Análisis temporal comprehensivo con múltiples perspectivas"""
3
4     # Configurar fecha como índice
5     df_temp = df.set_index(columna_fecha)
6
7     # 1. Evolución anual con estadísticas
8     evolucion_anual = df_temp.groupby(df_temp.index.year)[
9         columna_numerica].agg(['mean', 'std', 'min', 'max'])
10
11    # 2. Gráfico de evolución con bandas de desviación
12    plt.figure(figsize=(12, 6))
13    evolucion_anual['mean'].plot(kind='line', marker='o')
14    plt.fill_between(evolucion_anual.index,
15                    evolucion_anual['mean'] - evolucion_anual['std'],
16                    evolucion_anual['mean'] + evolucion_anual['std'],
17                    color='gray', alpha=0.5)

```

```

16         evolucion_anual['mean'] + evolucion_anual['std',
17             ],
18             alpha=0.2)
19 plt.title('Evoluci uAnual u(media u desviaci n)')
20 plt.grid(True, alpha=0.3)
21
22 # 3. An lisis de tendencias con medias m viles
23 tendencia_30d = df_temp[columna_numerica].rolling(window=30).
24     mean()
25 tendencia_90d = df_temp[columna_numerica].rolling(window=90).
26     mean()
27
28 return {
29     'evolucion_anual': evolucion_anual,
30     'tendencias': {
31         '30_dias': tendencia_30d,
32         '90_dias': tendencia_90d
33     }
34 }
```

3.5.6. Análisis de Estacionalidad y Patrones Temporales

La descomposición estacional identifica componentes fundamentales en series temporales: tendencia, estacionalidad y residuos.

Listing 3.18: Análisis de estacionalidad y patrones temporales

```

1 def analisis_estacionalidad(df, columna_fecha, columna_numerica):
2     """An lisis de componentes estacionales y patrones temporales"""
3
4     # Preparar datos mensuales para descomposici n
5     serie_mensual = df.set_index(columna_fecha)[
6         columna_numerica].resample('ME').mean()
7
8     # Descomposici n estacional
9     try:
10         descomposicion = seasonal_decompose(
11             serie_mensual, model='additive', period=12)
```

```

12
13     # Gráfico de descomposición
14     fig = descomposicion.plot()
15     fig.set_size_inches(12, 8)
16
17     return {
18         'descomposicion': descomposicion,
19         'tendencia': descomposicion.trend,
20         'estacionalidad': descomposicion.seasonal,
21         'residuos': descomposicion.resid
22     }
23
24 except Exception as e:
25     print(f"Error en la descomposición estacional: {e}")
26     return None
27
28 def análisis_estacionalidad_mensual(df, columna_fecha,
29                                     columna_numerica):
30     """Análisis de patrones estacionales por mes"""
31
32     # Extraer mes para análisis estacional
33     df['mes'] = df[columna_fecha].dt.month
34     estacionalidad_mensual = df.groupby('mes')[columna_numerica].
35                               mean()
36
37     plt.figure(figsize=(10, 6))
38     estacionalidad_mensual.plot(kind='bar', color='skyblue', alpha
39                                 =0.7)
40     plt.title('Comportamiento Estacional Promedio por Mes')
41     plt.xlabel('Mes')
42     plt.ylabel('Valor Promedio')
43
44     return estacionalidad_mensual

```

3.5.7. Análisis de Autocorrelación

El análisis de autocorrelación examina la dependencia temporal en series, esencial para identificar patrones recurrentes y memoria en los datos.

Listing 3.19: Análisis de autocorrelación temporal

```
1 def analisis_autocorrelacion(serie_temporal, lags=30):
2     """Análisis de autocorrelación para series temporales"""
3
4     plt.figure(figsize=(15, 5))
5
6     # Autocorrelación simple
7     plt.subplot(1, 2, 1)
8     plot_acf(serie_temporal, lags=lags, ax=plt.gca())
9     plt.title('Función de Autocorrelación')
10
11    # Autocorrelación parcial
12    plt.subplot(1, 2, 2)
13    plot_pacf(serie_temporal, lags=lags, ax=plt.gca())
14    plt.title('Función de Autocorrelación Parcial')
15
16    plt.tight_layout()
17
18    # Calcular valores numéricos
19    acf_valores = acf(serie_temporal, nlags=lags)
20    pacf_valores = pacf(serie_temporal, nlags=lags)
21
22    return {
23        'acf': acf_valores,
24        'pacf': pacf_valores
25    }
```

3.5.8. Detección de Outliers y Anomalías

La detección sistemática de outliers identifica valores atípicos utilizando métodos estadísticos robustos como el rango intercuartílico (IQR).

Listing 3.20: Detección y visualización de outliers

```
1 def deteccion_outliers_avanzada(df, columnas_numericas):
2     """Detección comprehensiva de outliers usando múltiples
3         en todos"""
4
5     resultados_outliers = {}
```

```

5
6     for columna in columnas_numericas:
7         # M todo IQR
8         Q1 = df[columna].quantile(0.25)
9         Q3 = df[columna].quantile(0.75)
10        IQR = Q3 - Q1
11        limite_inferior = Q1 - 1.5 * IQR
12        limite_superior = Q3 + 1.5 * IQR
13
14        # Identificar outliers
15        outliers = df[
16            (df[columna] < limite_inferior) |
17            (df[columna] > limite_superior)
18        ]
19
20        resultados_outliers[columna] = {
21            'limite_inferior': limite_inferior,
22            'limite_superior': limite_superior,
23            'total_outliers': len(outliers),
24            'porcentaje_outliers': (len(outliers) / len(df)) * 100,
25            'outliers': outliers
26        }
27
28        # Crear columna flag para outliers
29        df['es_outlier'] = False
30        for columna in columnas_numericas:
31            limites = resultados_outliers[columna]
32            mask = (df[columna] < limites['limite_inferior']) | \
33                   (df[columna] > limites['limite_superior'])
34            df.loc[mask, 'es_outlier'] = True
35
36        return resultados_outliers
37
38 def visualizar_outliers_por_categoria(df, columna_categoria,
39                                       columna_numerica):
40     """Visualizaci n de outliers por categor as"""
41
42     plt.figure(figsize=(12, 6))

```

```

42     sns.boxplot(data=df, x=columna_categoria, y=columna_numerica)
43     plt.title('Distribución y Outliers por Categoría')
44     plt.xticks(rotation=45)
45
46     return plt.gcf()

```

3.5.9. Análisis de Volatilidad y Correlaciones

El análisis de volatilidad mide la variabilidad temporal, mientras que las matrices de correlación examinan relaciones entre variables.

Listing 3.21: Análisis de volatilidad y correlaciones

```

1 def analisis_volatilidad(serie_temporal, ventana=30):
2     """Análisis de volatilidad usando desviación estandar rolling
3
4     volatilidad = serie_temporal.rolling(window=ventana).std()
5
6     plt.figure(figsize=(12, 6))
7     plt.plot(volatilidad.index, volatilidad.values,
8             color='red', alpha=0.7)
9     plt.title(f'Volatilidad Rolling ({ventana} días)')
10    plt.xlabel('Fecha')
11    plt.ylabel('Volatilidad (Desviación Estándar)')
12    plt.grid(True, alpha=0.3)
13
14    return volatilidad
15
16 def matriz_correlacion_categorias(df, columna_fecha,
17                                   columna_categoria,
18                                   columna_numerica):
19     """Matriz de correlación entre diferentes categorías"""
20
21     # Crear pivot table para correlación
22     pivot_corr = df.pivot_table(
23         index=columna_fecha,
24         columns=columna_categoria,
25         values=columna_numerica,

```

```

25     observed=False
26     ).corr()
27
28     plt.figure(figsize=(8, 6))
29     sns.heatmap(pivot_corr, annot=True, cmap='coolwarm',
30                  center=0, fmt='.2f')
31     plt.title('Correlacin entre Diferentes Categoras')
32
33     return pivot_corr

```

3.5.10. Análisis Comparativo por Años

El análisis temporal desagregado por años permite identificar patrones específicos y evoluciones interanuales.

Listing 3.22: Análisis temporal comparativo por años

```

1 def analisis_temporal_por_a_o(df, columna_fecha, columna_numerica,
2                               columna_categoria):
3
4     """An lisisntemporaldesglosadopora os uycategori as"""
5
6
7     # Extraer a o para an lisis
8     df['a o'] = df[columna_fecha].dt.year
9     years = sorted(df['a o'].unique())
10
11
12     # Calcular dimensiones din micas para grid
13     n_cols = 3
14     n_rows = (len(years) + n_cols - 1) // n_cols
15
16
17     plt.figure(figsize=(15, 5 * n_rows))
18
19
20     for i, year in enumerate(years, 1):
21         plt.subplot(n_rows, n_cols, i)
22
23
24         data_year = df[df['a o'] == year]
25
26
27         # Graficar cada categoria
28         for categoria in data_year[columna_categoria].unique():
29             data_categoria = data_year[

```

```

23         data_year[columna_categoria] == categoria]
24 monthly_avg = data_categoria.groupby('mes')[
25     columna_numerica].mean()
26 plt.plot(monthly_avg.index, monthly_avg.values,
27             label=categoria, marker='o')
28
29 plt.title(f'Evoluci n mensual{year}')
30 plt.xlabel('Mes')
31 plt.ylabel('Valor Promedio')
32 plt.xticks(range(1, 13))
33 plt.grid(True, alpha=0.3)
34
35 # Leyenda solo en primer gr fico
36 if i == 1:
37     plt.legend()
38
39 plt.tight_layout()
40 return plt.gcf()

```

3.5.11. Análisis mediante tablas dinámicas (pivot tables)

Las tablas dinámicas representan una herramienta fundamental en análisis exploratorio, ya que permiten resumir grandes volúmenes de datos y explorar relaciones entre categorías y tiempo (McKinney, 2018). En este proyecto, se generan resúmenes interanuales mediante `pivot_table`, permitiendo una visión agregada del comportamiento de la variable numérica principal a través de entidades y tipos de fondo.

3.5.12. Regresión lineal

Antes de especializarse al caso de datos ordenados en el tiempo, es útil presentar la regresión lineal en su forma más general, como una herramienta para modelar la relación entre una variable de respuesta cuantitativa y un conjunto de variables explicativas. En términos generales, la regresión lineal busca aproximar la relación entre una variable aleatoria Y (respuesta) y un vector de predictores $\mathbf{X} = (X_1, \dots, X_p)$ mediante una combinación lineal de estos últimos.

Modelo de regresión lineal simple

En este proyecto, el concepto de regresión lineal se implementa utilizando `scikit-learn`, siguiendo la misma filosofía reproducible empleada en el resto del código: se define una función genérica que recibe un conjunto de datos, separa las variables predictoras de la variable objetivo, ajusta el modelo de regresión lineal y calcula métricas básicas de desempeño.

Listing 3.23: Función genérica para ajustar un modelo de regresión lineal

```
1 from sklearn.linear_model import LinearRegression
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import mean_absolute_error, mean_squared_error,
4     r2_score
5
6 import numpy as np
7
8 def ajustar_regresion_lineal(df, columnas_predictoras,
9     columna_objetivo, test_size=0.2, random_state=42):
10     """
11     Ajusta un modelo de regresión lineal y evalúa su desempeño.
12
13     Parameters:
14         df (pd.DataFrame): Conjunto de datos con predictores y variable objetivo.
15         columnas_predictoras (list): Nombres de las columnas usadas como X.
16         columna_objetivo (str): Nombre de la columna usada como y.
17         test_size (float): Proporción de datos para prueba.
18         random_state (int): Semilla para la partición de entrenamiento/prueba.
19
20     Returns:
21         dict: Parámetros del modelo y métricas de evaluación.
22
23     # 1. Definir X (predictores) e y (objetivo)
24     X = df[columnas_predictoras].values
25     y = df[columna_objetivo].values
26
27     # 2. Partición en entrenamiento y prueba
28     X_train, X_test, y_train, y_test = train_test_split(
29         X, y, test_size=test_size, random_state=random_state
```

```

27 )
28
29 # 3. Crear y ajustar el modelo de regresión lineal
30 modelo = LinearRegression()
31 modelo.fit(X_train, y_train)
32
33 # 4. Predicciones en entrenamiento y prueba
34 y_train_pred = modelo.predict(X_train)
35 y_test_pred = modelo.predict(X_test)
36
37 # 5. Cálculo de métricas
38 mae_train = mean_absolute_error(y_train, y_train_pred)
39 rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
40 r2_train = r2_score(y_train, y_train_pred)
41
42 mae_test = mean_absolute_error(y_test, y_test_pred)
43 rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
44 r2_test = r2_score(y_test, y_test_pred)
45
46 print("== REGRESIÓN LINEAL ==")
47 print("Coeficientes:", modelo.coef_)
48 print("Intercepto:", modelo.intercept_)
49 print("\nDesempeño en entrenamiento:")
50 print(f"MAE: {mae_train:.4f}")
51 print(f"RMSE: {rmse_train:.4f}")
52 print(f"R^2: {r2_train:.4f}")
53
54 print("\nDesempeño en prueba:")
55 print(f"MAE: {mae_test:.4f}")
56 print(f"RMSE: {rmse_test:.4f}")
57 print(f"R^2: {r2_test:.4f}")
58
59 resultados = {
60     "coeficientes": modelo.coef_,
61     "intercepto": modelo.intercept_,
62     "mae_train": mae_train,
63     "rmse_train": rmse_train,
64     "r2_train": r2_train,

```

```

65     "mae_test": mae_test,
66     "rmse_test": rmse_test,
67     "r2_test": r2_test
68 }
69
70 return modelo, resultados

```

3.5.13. Tipos de Estructuras de Datos

En el análisis de datos, es crucial reconocer la estructura fundamental de los datasets, ya que determina las técnicas analíticas apropiadas. Según [mckinney2018python](#), existen dos categorías principales:

3.5.14. Series de Tiempo (Time Series)

Una serie de tiempo es una secuencia de observaciones ordenadas cronológicamente, donde cada observación está asociada a un instante temporal específico. Estas series presentan características distintivas:

- **Ordenamiento temporal:** Las observaciones están indexadas por marcas de tiempo (días, meses, años)
- **Dependencia temporal:** Los valores presentes están influenciados por valores pasados (autocorrelación)
- **Estacionalidad:** Patrones que se repiten en intervalos regulares
- **Tendencia:** Comportamiento de largo plazo que muestra dirección general

3.5.15. Datos Transversales (Cross-Sectional)

Los datos transversales consisten en observaciones recolectadas en un único momento del tiempo para múltiples unidades de análisis:

- **Sin orden temporal:** Todas las observaciones corresponden al mismo período
- **Comparación entre unidades:** Permite analizar diferencias entre individuos, empresas o categorías
- **Independencia:** Generalmente se asume que las observaciones son independientes entre sí

3.5.16. Análisis de Series de Tiempo

Dado que muchos conjuntos de datos del mundo real presentan estructura temporal, el pipeline incorpora funcionalidades específicas para el análisis de series de tiempo.

3.5.17. Componentes de las Series de Tiempo

Las series temporales pueden descomponerse en componentes fundamentales que facilitan su análisis:

- **Tendencia:** Movimiento de largo plazo que muestra la dirección general de la serie
- **Estacionalidad:** Patrones que se repiten en intervalos regulares (diarios, semanales, anuales)
- **Ciclicidad:** Fluctuaciones no periódicas de largo plazo
- **Residuos:** Componente aleatorio no explicado por los otros componentes

3.5.18. Visualización de Series de Tiempo

La visualización es una herramienta esencial para comprender el comportamiento de las series temporales. El pipeline implementa funciones genéricas para la visualización sistemática:

Listing 3.24: Función genérica para visualización de series temporales

```
1 def visualizar_serie_temporal(df, columna_fecha, columna_valor,
2                               titulo, ruta_guardado):
3     """
4     Visualiza una serie temporal con configuraciones estandarizadas
5
6     Parameters:
7         df (pd.DataFrame): Dataset con la serie temporal
8         columna_fecha (str): Nombre de la columna de fecha
9         columna_valor (str): Nombre de la columna de valores
10        titulo (str): Título del gráfico
11        ruta_guardado (str): Ruta para guardar el gráfico
12
13
14        plt.figure(figsize=(14, 8))
15
16        plt.plot(df[columna_fecha], df[columna_valor],
```

```

15     linewidth=2, alpha=0.8, color='blue')
16
17 plt.title(titulo, fontsize=14, fontweight='bold')
18 plt.xlabel('Fecha')
19 plt.ylabel('Valor')
20 plt.grid(True, alpha=0.3)
21 plt.xticks(rotation=45)
22 plt.tight_layout()
23
24 # Guardar gráfico de manera reproducible
25 Path(ruta_guardado).parent.mkdir(parents=True, exist_ok=True)
26 plt.savefig(ruta_guardado, dpi=300, bbox_inches='tight')
27 plt.close()
28 print(f"Gráfico guardado en: {ruta_guardado}")

```

3.5.19. Análisis de Múltiples Series

Para comparar diferentes series temporales, se implementan funciones de normalización y comparación:

Listing 3.25: Función para comparación de series normalizadas

```

1 def comparar_series_normalizadas(diccionario_series, titulo_base,
2                                   ruta_guardado):
3     """
4     Compara múltiples series temporales normalizadas a base 100
5
6     Parameters:
7     diccionario_series (dict): Diccionario con {nombre_serie: dataframe}
8     titulo_base (str): Título base para el gráfico
9     ruta_guardado (str): Ruta para guardar el gráfico
10    """
11
12    plt.figure(figsize=(16, 10))
13
14    for nombre_serie, df in diccionario_series.items():
15        if len(df) > 0:
16            # Normalización base 100 para comparación
17            valor_base = df['valor_unidad'].iloc[0]

```

```

16     serie_normalizada = (df['valor_unidad'] / valor_base *
17                             100)
18
19     plt.plot(df['fecha'], serie_normalizada,
20               label=nombre_serie, linewidth=2, alpha=0.7)
21
22     plt.title(f'{titulo_base}\n(Valores Normalizados Base 100)', fontweight='bold')
23     plt.xlabel('Fecha')
24     plt.ylabel('Valor Normalizado (Base 100)')
25     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
26     plt.grid(True, alpha=0.3)
27     plt.xticks(rotation=45)
28     plt.tight_layout()
29     plt.savefig(ruta_guardado, dpi=300, bbox_inches='tight')
30     plt.close()

```

3.5.20. Análisis de Estacionariedad

La estacionariedad es un concepto fundamental en el análisis de series temporales, ya que muchos modelos asumen esta propiedad.

3.5.21. Prueba de Dickey-Fuller Aumentada

La prueba de Dickey-Fuller aumentada (ADF) es el método estándar para evaluar la estacionariedad de una serie temporal:

Listing 3.26: Implementación de la prueba ADF

```

1 from statsmodels.tsa.stattools import adfuller
2
3 def analizar_estacionariedad(serie, nombre_serie=""):
4     """
5         Realiza la prueba de Dickey-Fuller aumentada para la
6         estacionariedad
7
8         Parameters:
9             serie (pd.Series): Serie temporal a analizar
10            nombre_serie (str): Nombre de la serie para reporte

```

```

10
11     Returns:
12     dict: Resultados de la prueba
13     """
14
15         resultado_adf = adfuller(serie.dropna())
16
17         metricas = {
18             'estadistico_adf': resultado_adf[0],
19             'p_valor': resultado_adf[1],
20             'valores_criticos': resultado_adf[4],
21             'es_estacionaria': resultado_adf[1] < 0.05
22         }
23
24         print(f"Análisis de Estacionalidad - {nombre_serie}:")
25         print(f"Estadístico ADF: {metricas['estadistico_adf']:.4f}")
26         print(f"P-valor: {metricas['p_valor']:.4f}")
27         print(f"Es estacionaria? {metricas['es_estacionaria']} ")
28
29         # Interpretación de resultados
30         if metricas['p_valor'] > 0.05:
31             print("La serie NO es estacionaria - se requiere diferenciación")
32         else:
33             print("La serie es estacionaria")
34
35     return metricas

```

3.5.22. Modelos de Series de Tiempo

El pipeline incorpora los principales modelos para el análisis y pronóstico de series temporales.

3.5.23. Fundamentos de los Modelos ARIMA

La familia de modelos ARIMA (AutoRegressive Integrated Moving Average) es ampliamente utilizada para series temporales estacionarias.

Componentes ARIMA

- **AR (AutoRegresivo)**: Modela la dependencia entre una observación y un número de observaciones rezagadas
- **I (Integrado)**: Representa la diferenciación de la serie para hacerla estacionaria
- **MA (Media Móvil)**: Modela la dependencia entre una observación y el error residual de un modelo de media móvil

Funciones de Autocorrelación

Las funciones ACF (Autocorrelation Function) y PACF (Partial Autocorrelation Function) son esenciales para identificar los parámetros del modelo:

Listing 3.27: Análisis de ACF y PACF

```
1 from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
2
3 def analizar_autocorrelacion(serie, lags=40):
4     """
5         Analiza las funciones de autocorrelación y autocorrelación
6             parcial
7
8         Parameters:
9             serie (pd.Series): Serie temporal estacionaria
10            lags (int): Número de lags a analizar
11
12    """
13    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))
14
15    # Autocorrelación (ACF)
16    plot_acf(serie, lags=lags, ax=ax1)
17    ax1.set_title('Función de Autocorrelación (ACF)')
18
19    # Autocorrelación parcial (PACF)
20    plot_pacf(serie, lags=lags, ax=ax2)
21    ax2.set_title('Función de Autocorrelación Parcial (PACF)')
22
23    plt.tight_layout()
24    plt.show()
```

```

24 # Interpretación básica
25 print("Interpretación de ACF/PACF:")
26 print("- ACF que decrece lentamente: indica no estacionariedad")
27 print("- PACF con picos significativos: sugiere orden AR")
28 print("- ACF con picos significativos: sugiere orden MA")

```

3.5.24. Modelo AR (AutoRegresivo)

El modelo AR(p) expresa el valor actual de la serie como una combinación lineal de valores pasados más un término de error:

$$X_t = c + \phi_1 X_{t-1} + \phi_2 X_{t-2} + \cdots + \phi_p X_{t-p} + \epsilon_t \quad (3.1)$$

Listing 3.28: Implementación del modelo AR

```

1 from statsmodels.tsa.ar_model import AutoReg
2
3 def entrenar_modelo_ar(serie, orden=1):
4     """
5         Entrena un modelo autoregresivo (AR)
6
7     Parameters:
8         serie (pd.Series): Serie temporal estacionaria
9         orden (int): Orden del modelo AR (p)
10
11    Returns:
12        tuple: Modelo entrenado y métricas
13    """
14        modelo = AutoReg(serie, lags=orden)
15        modelo_ajustado = modelo.fit()
16
17        print(f"Modelo AR({orden}):")
18        print(f"  Parámetros: {modelo_ajustado.params}")
19        print(f"  AIC: {modelo_ajustado.aic:.4f}")
20        print(f"  BIC: {modelo_ajustado.bic:.4f}")
21
22    return modelo_ajustado

```

3.5.25. Modelo MA (Media Móvil)

El modelo MA(q) modela el valor actual en función de los errores pasados:

$$X_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \cdots + \theta_q\epsilon_{t-q} \quad (3.2)$$

3.5.26. Modelo ARIMA

Combina los componentes AR, I y MA en un modelo unificado:

$$(1 - \phi_1B - \cdots - \phi_pB^p)(1 - B)^d X_t = c + (1 + \theta_1B + \cdots + \theta_qB^q)\epsilon_t \quad (3.3)$$

Listing 3.29: Implementación del modelo ARIMA

```
1 from statsmodels.tsa.arima.model import ARIMA
2
3 def entrenar_modelo_arima(serie, orden=(1,1,1)):
4     """
5         Entrena un modelo ARIMA
6
7     Parameters:
8         serie (pd.Series): Serie temporal
9         orden (tuple): Orden (p,d,q) del modelo
10
11    Returns:
12        tuple: Modelo entrenado y resultados
13    """
14        p, d, q = orden
15
16    try:
17        modelo = ARIMA(serie, order=(p, d, q))
18        modelo_ajustado = modelo.fit()
19
20        metricas = {
21            'aic': modelo_ajustado.aic,
22            'bic': modelo_ajustado.bic,
23            'residuos_media': modelo_ajustado.resid.mean(),
24            'residuos_std': modelo_ajustado.resid.std()
25        }
26    
```

```

27     print(f"Modelo_ARIMA{orden}:")
28     print(f"    AIC:{metricas['aic']:.4f}")
29     print(f"    BIC:{metricas['bic']:.4f}")
30
31     return modelo_ajustado, metricas
32
33 except Exception as e:
34     print(f"Error_entrenando_ARIMA{orden}:{e}")
35     return None, None

```

3.5.27. Modelo SARIMA

Extiende ARIMA para incorporar estacionalidad:

Listing 3.30: Implementación del modelo SARIMA

```

1 from statsmodels.tsa.statespace.sarimax import SARIMAX
2
3 def entrenar_modelo_sarima(serie, orden=(1,1,1), orden_estacional
4                           =(1,1,1,12)):
5     """
6         Entrena un modelo SARIMA con componente estacional
7
8     Parameters:
9         serie(pd.Series): Serie temporal
10        orden(tuple): Orden no estacional (p,d,q)
11        orden_estacional(tuple): Orden estacional (P,D,Q,s)
12
13    Returns:
14        tuple: Modelo entrenado y resultados
15    """
16    try:
17        modelo = SARIMAX(serie,
18                          order=orden,
19                          seasonal_order=orden_estacional,
20                          enforce_stationarity=False,
21                          enforce_invertibility=False)
22
23        modelo_ajustado = modelo.fit(disp=False)

```

```

23
24     metricas = {
25         'aic': modelo_ajustado.aic,
26         'bic': modelo_ajustado.bic,
27         'llf': modelo_ajustado.llf
28     }
29
30     print(f"Modelo SARIMA{orden}{orden_estacional}:")
31     print(f"  AIC: {metricas['aic']:.4f}")
32     print(f"  Log-Likelihood: {metricas['llf']:.4f}")
33
34     return modelo_ajustado, metricas
35
36 except Exception as e:
37     print(f"Error entrenando SARIMA: {e}")
38     return None, None

```

3.5.28. Selección y Evaluación de Modelos

3.5.29. Identificación de Parámetros

La selección de los parámetros óptimos (p,d,q) se realiza mediante:

- **Análisis de ACF/PACF:** Para identificar órdenes tentativos
- **Criterios de información:** AIC, BIC para comparar modelos
- **Validación cruzada:** Para evaluar capacidad predictiva

Listing 3.31: Búsqueda de parámetros óptimos

```

1 def buscar_mejor_arima(serie, parametros_a_probar):
2     """
3     Busca los mejores parámetros ARIMA mediante grid search
4
5     Parameters:
6     serie (pd.Series): Serie temporal
7     parametros_a_probar (list): Lista de tuplas (p, d, q) a probar
8
9     Returns:

```

```

10     dict: Mejor_modelo_y_metricas
11 """
12     mejores_metricas = {'aic': float('inf')}
13     mejor_modelo = None
14     mejor_orden = None
15
16     for orden in parametros_a_probar:
17         try:
18             modelo, metricas = entrenar_modelo_arima(serie, orden)
19
20             if modelo and metricas['aic'] < mejores_metricas['aic']:
21                 mejores_metricas = metricas
22                 mejor_modelo = modelo
23                 mejor_orden = orden
24
25         except:
26             continue
27
28     if mejor_modelo:
29         print(f"Mejor modelo: ARIMA{mejor_orden} con AIC: {mejores_metricas['aic']:.4f}")
30
31     return {
32         'modelo': mejor_modelo,
33         'orden': mejor_orden,
34         'metricas': mejores_metricas
35     }

```

3.5.30. Evaluación de Pronósticos

Listing 3.32: Métricas de evaluación de pronósticos

```

1 def evaluar_pronostico(real, pronosticado):
2     """
3     Evalúa la calidad de los pronósticos
4
5     Parameters:
6     real (array): Valores reales

```

```

7     uuuupronosticado : array) : Valores pronosticados
8
9     Returns:
10    dict: Metrics de evaluacion
11    """
12    mse = np.mean((real - pronosticado)**2)
13    mae = np.mean(np.abs(real - pronosticado))
14    mape = np.mean(np.abs((real - pronosticado) / real)) * 100
15    rmse = np.sqrt(mse)
16
17    metricas = {
18        'MSE': mse,
19        'MAE': mae,
20        'MAPE': mape,
21        'RMSE': rmse
22    }
23
24    print("Metrics de evaluacion:")
25    for metrica, valor in metricas.items():
26        print(f"\u00b7{metrica}: {valor:.4f}")
27
28    return metricas

```

3.5.31. Pipeline Integrado de Análisis de Series Temporales

Listing 3.33: Pipeline completo para análisis de series temporales

```

1 def pipeline_series_temporales(df, columna_fecha, columna_valor,
2                                 nombre_serie):
3     """
4
5     Pipeline completo para analisis de series temporales
6
7     Parameters:
8         df (pd.DataFrame): Dataset con la serie temporal
9         columna_fecha (str): Columna de fecha
10        columna_valor (str): Columna de valores
11        nombre_serie (str): Identificador de la serie

```

```

11     Returns:
12     dict: Resultados completos del análisis
13     """
14     print(f"==> INICIANDO PIPELINE PARA: {nombre_serie} ==>")
15
16     resultados = {}
17
18     # 1. Preparación de datos
19     serie = df.set_index(columna_fecha)[columna_valor].sort_index()
20
21     # 2. Análisis de estacionariedad
22     print("\n1. Analizando estacionariedad...")
23     resultados['estacionariedad'] = analizar_estacionariedad(serie,
24                                                               nombre_serie)
25
26     # 3. Visualización inicial
27     print("\n2. Generando visualizaciones...")
28     visualizar_serie_temporal(df.reset_index(), columna_fecha,
29                                columna_valor,
30                                f'SerieTemporal-{nombre_serie}',
31                                f'verticalizaciones/serie_{nombre_serie}.png')
32
33     # 4. Análisis de autocorrelación (si es estacionaria)
34     if resultados['estacionariedad']['es_estacionaria']:
35         print("\n3. Analizando autocorrelación...")
36         analizar_autocorrelacion(serie)
37
38     # 5. Entrenamiento de modelos
39     print("\n4. Entrenando modelos...")
40     parametros_probar = [(1, 0, 0), (1, 1, 1), (2, 1, 2)]
41     resultados['mejor_modelo'] = buscar_mejor_arima(serie,
42                                                       parametros_probar)
43
44     # 6. Generación de reporte
45     print("\n5. Generando reporte final...")
46     resultados['reporte'] = generar_reporte_analisis(resultados,
47                                                       nombre_serie)

```

```
44  
45     print(f"    Pipeline completado para: {nombre_serie}")  
46     return resultados
```

3.5.32. Análisis de Datos Transversales

Si bien este capítulo se ha centrado en series temporales, el pipeline también incorpora funcionalidades para el análisis de datos transversales, incluyendo:

- **Estadísticas descriptivas:** Medidas de tendencia central y dispersión
- **Análisis de distribuciones:** Histogramas, gráficos de densidad
- **Visualizaciones comparativas:** Box plots, violin plots
- **Análisis de correlación:** Matrices de correlación, heatmaps

La implementación específica de estas funcionalidades sigue los mismos principios de reproducibilidad y automatización que caracterizan el pipeline completo.

Este enfoque sistemático garantiza que el análisis exploratorio y la visualización de datos puedan realizarse de manera consistente y reproducible, independientemente del conjunto de datos específico que se esté analizando.

3.6. Fase 5: Documentación

3.6.1. Documentación en Jupyter Notebooks

Se implementó un sistema de documentación exhaustiva en Jupyter Notebooks:

Listing 3.34: Estructura de documentación en notebooks

```
1 # %% [markdown]  
2 # # Pipeline Reproducible de An lisis de Datos  
3 # ## Fase 1 - Configuraci n del Entorno  
4 #  
5 # ### Objetivos  
6 # 1. Configurar entorno virtual Python  
7 # 2. Instalar y gestionar dependencias  
8 # 3. Establecer estructura de proyecto  
9 # 4. Configurar control de versiones
```

```
10 #  
11 # ### Librerías Utilizadas  
12 # - pandas: Manipulación de datos  
13 # - numpy: Operaciones numéricas  
14 # - matplotlib: Visualizaciones básicas  
15 # - seaborn: Visualizaciones estadísticas  
16 # - scipy: Análisis estadístico avanzado
```

3.6.2. Metodología de Documentación

Cada notebook incluye:

- **Contexto y objetivos:** Explicación clara del propósito de cada fase
- **Metodología implementada:** Descripción detallada de los métodos utilizados
- **Resultados intermedios:** Captura de outputs importantes
- **Verificaciones:** Aserciones para validar la calidad de los datos
- **Exportación:** Guardado de resultados para siguientes fases

3.6.3. Notebooks Implementados

- **01_adquisicion.ipynb:** Documenta la descarga de datos mediante API con paginación
- **02_limpieza.ipynb:** Registra todo el proceso de limpieza con métricas específicas
- **03_analisis.ipynb:** Prepara el terreno para el análisis exploratorio

3.7. Control de Versiones y Reproducibilidad

Se implementó una estrategia de control de versiones que incluye:

- `requirements.txt` para dependencias del proyecto
- Estructura de ramas: `main`, `develop`, `feature/*`
- Commits descriptivos y atomizados
- `.gitignore` apropiado para proyectos de datos

- Documentación completa en README.md

Capítulo 4

Conclusiones

El presente proyecto ha alcanzado plenamente tanto el objetivo general como todos los objetivos específicos planteados.

Se diseñó, implementó y validó un flujo de trabajo (pipeline) completo, modular y 100 % reproducible para análisis de datos en Python, capaz de aplicarse a cualquier dataset público con estructura tabular y componente temporal. El pipeline cubre de forma integrada y automatizada todas las etapas del proceso analítico: configuración profesional del entorno, adquisición programática de datos mediante API con paginación robusta, limpieza exhaustiva (imputación, duplicados, outliers, optimización de tipos), análisis exploratorio avanzado (más de 16 tipos de análisis y 73 gráficas generadas automáticamente), modelado de series de tiempo (ARIMA/SARIMA) y documentación completa mediante notebooks y reportes.

Principales logros técnicos

- Entorno totalmente reproducible mediante `venv`, `requirements.txt` y estructura de proyecto estandarizada (`data/raw`, `data/processed`, `notebooks/`, `scripts/`).
- Descarga automática y segura de datasets grandes mediante paginación y manejo de errores.
- Limpieza sistemática y genérica (funciones reutilizables para nulos, duplicados, outliers, tipos de datos y variables derivadas).
- EDA modular y automático que genera estadísticas, distribuciones, tendencias, estacionalidad, autocorrelación, volatilidad y correlaciones sin intervención manual.
- Módulo de series de tiempo completo: prueba ADF, diferenciación automática, grid

search ARIMA/SARIMA, evaluación con MAE/RMSE/MAPE y visualización de diagnóstico.

- Documentación exhaustiva en Jupyter Notebooks y repositorio GitHub público con README detallado y ejecución en un solo comando.

Buenas Prácticas Implementadas

El pipeline incorpora las mejores prácticas actuales en análisis de datos reproducible:

1. **Separación estricta de datos crudos y procesados** (`data/raw` nunca se modifica).
2. **Funciones genéricas y reutilizables** en lugar de código duplicado (`descarga_api`, limpieza, EDA, modelado).
3. **Manejo robusto de errores y paginación** en APIs (`time.sleep`, `try/except`, verificación de total de registros).
4. **Optimización de memoria** mediante conversión temprana a `category`, `int32`, etc.
5. **Imputación inteligente** (`forward-fill` + interpolación lineal para series temporales).
6. **Detección automática de duplicados exactos y conceptuales**.
7. **Exportación reproducible** de todos los productos (CSV con metadatos, gráficas en alta resolución, resúmenes de limpieza).
8. **Control de versiones profesional** (Git + GitHub + `.gitignore` adecuado para datos).
9. **Documentación en notebooks** con markdown explicativo + código ejecutable en cada fase.

Estas prácticas garantizan que cualquier persona pueda clonar el repositorio y obtener exactamente los mismos resultados en cualquier máquina, cumpliendo con los estándares de ciencia abierta y análisis profesional.

Lecciones Aprendidas y Recomendaciones

El desarrollo del pipeline permitió consolidar un conjunto de competencias transferibles a cualquier proyecto de datos:

- La modularidad y la generalización de funciones son esenciales para escalar el análisis a nuevos datasets.
- La automatización del EDA y el modelado reduce drásticamente el tiempo de análisis y elimina errores humanos.
- La reproducibilidad no es un extra: es el núcleo de un proyecto profesional de datos.
- El uso de entornos virtuales, control de versiones y documentación detallada facilita la colaboración y el mantenimiento a largo plazo.

Se recomienda, para proyectos futuros, incorporar pruebas unitarias automáticas (`pytest`) y empaquetado del pipeline como librería Python reutilizable.

En conclusión, este trabajo demuestra que es posible construir un flujo de análisis de datos completo, profesional y reproducible utilizando únicamente herramientas de código abierto, cumpliendo con los más altos estándares de calidad y transparencia científica. El pipeline desarrollado constituye un activo reutilizable de alto valor tanto académico como profesional.

Glosario

Glosario de Términos

ACF (Función de Autocorrelación) Herramienta estadística que mide la correlación entre una serie temporal y sus valores rezagados, utilizada para identificar patrones de dependencia temporal.

ADF (Prueba de Dickey-Fuller Aumentada) Test estadístico que evalúa formalmente si una serie temporal posee una raíz unitaria, es decir, si es no estacionaria.

AIC (Criterio de Información de Akaike) Métrica para comparar modelos estadísticos que balancea bondad de ajuste con complejidad, penalizando el exceso de parámetros.

Análisis Temporal Avanzado Evaluación comprehensiva de patrones de evolución, tendencias y comportamientos a lo largo del tiempo mediante técnicas como medias móviles, bandas de desviación y evolución anual comparativa.

API (Interfaz de Programación de Aplicaciones) Conjunto de protocolos y herramientas que permite la comunicación entre diferentes aplicaciones de software, utilizado para la adquisición de datos públicos.

ARIMA Modelo de series temporales que combina componentes auto-regresivos (AR), integrados (I) y de media móvil (MA) para modelar series no estacionarias.

Autocorrelación Medida de dependencia temporal en series de datos que cuantifica la relación entre observaciones separadas por diferentes intervalos de tiempo, esencial para identificar patrones recurrentes.

Autocorrelación Estacional Correlación entre valores separados por múltiplos del período estacional (por ejemplo, lags de 12 en datos mensuales).

BIC (Criterio de Información Bayesiano) Criterio de selección de modelos similar al AIC pero con penalización más fuerte por complejidad, preferido en muestras grandes.

Cardinalidad Número de valores únicos en una columna de un dataset, importante para determinar el tipo de datos óptimo y la eficiencia del almacenamiento.

Commit Operación en sistemas de control de versiones que guarda los cambios realizados en el repositorio con un mensaje descriptivo.

Control de Versiones Sistema que registra los cambios realizados en archivos a lo largo del tiempo, permitiendo revertir a estados anteriores y facilitando la colaboración.

Correlación Intercategoría Relación estadística entre valores promedio o series agrupadas pertenecientes a diferentes categorías de un dataset.

Cuartil División de un conjunto de datos en cuatro partes iguales; usado para clasificar fondos según rendimiento (Q1, Q2, Q3, Q4).

Data Wrangling Proceso de limpieza y transformación de datos brutos en un formato adecuado para el análisis, incluyendo manejo de valores nulos, corrección de tipos de datos y creación de variables derivadas.

DataFrame Estructura de datos bidimensional etiquetada de la librería pandas, similar a una tabla de base de datos, utilizada para manipulación y análisis de datos.

Dataset Crudo (Raw) Datos originales descargados sin modificaciones, conservados para referencia y trazabilidad.

Dataset Procesado Datos limpios, transformados y listos para análisis; resultado de aplicar el pipeline de limpieza.

Dependencia Temporal Relación estadística entre valores presentes y pasados de una serie, fundamental para modelos ARIMA y SARIMA.

Descomposición Estacional Técnica para separar una serie temporal en sus componentes fundamentales: tendencia, estacionalidad y residuos, permitiendo identificar patrones recurrentes y comportamientos de largo plazo.

Diferenciación Transformación utilizada para convertir una serie temporal no estacionaria en estacionaria, definida como la resta entre valores consecutivos.

Drawdown Medida de caída acumulada desde un máximo histórico en una serie financiera, utilizada para evaluar riesgo y episodios de deterioro.

EDA (Análisis Exploratorio de Datos) Conjunto de técnicas y visualizaciones utilizadas para comprender las características principales de un dataset, incluyendo distribuciones, valores atípicos y relaciones entre variables.

Entorno Virtual Entorno aislado de Python que permite gestionar dependencias específicas de un proyecto sin afectar el sistema global, crucial para la reproducibilidad.

Estacionalidad Componente periódico y recurrente en una serie temporal que se repite a intervalos regulares.

Estacionariedad Propiedad de una serie temporal donde sus características estadísticas (media, varianza) permanecen constantes en el tiempo, requisito fundamental para muchos modelos de series temporales.

Exportación Reproducible Sistema de guardado de resultados que incluye metadatos, parámetros y configuraciones para garantizar la replicabilidad exacta de los análisis.

Grid Search Estrategia de búsqueda exhaustiva donde se prueban múltiples combinaciones de parámetros (como p,d,q) para seleccionar el modelo con mejor desempeño.

Heatmap Gráfico matricial donde los valores se representan mediante una escala de colores, útil para visualizar correlaciones o tablas dinámicas.

Imputación Técnica de reemplazo de valores faltantes en un dataset usando métodos como la mediana, moda o algoritmos más avanzados como KNN.

Integración (Componente I de ARIMA) Número de veces que se debe diferenciar una serie para volverla estacionaria.

IQR (Rango Intercuartílico) Medida estadística de dispersión definida como la diferencia entre el tercer y primer cuartil, utilizada para identificar valores atípicos.

MAE Error absoluto medio, métrica robusta que mide errores promedio sin penalizar intensamente valores extremos.

MAPE Error porcentual absoluto medio, útil para interpretar precisión en términos porcentuales.

Matriz de Correlación Representación tabular de relaciones lineales entre variables, usada para identificar patrones o dependencias entre series.

Matriz de Correlación entre Categorías Representación tabular de relaciones lineales entre diferentes categorías o grupos dentro de un dataset, visualizada mediante heatmaps anotados.

Media Móvil (Rolling Mean) Promedio calculado sobre una ventana móvil de observaciones consecutivas, utilizado para suavizar fluctuaciones y resaltar tendencias.

Metadatos Información adicional que describe el proceso de generación de un archivo (fecha, parámetros, tamaño, transformaciones), clave para reproducibilidad.

Métricas de Calidad Conjunto de indicadores cuantitativos que evalúan la integridad, consistencia y confiabilidad de un dataset después del proceso de limpieza.

Métricas de Evaluación de Pronósticos Conjunto de indicadores cuantitativos (MSE, MAE, MAPE, RMSE) para medir la precisión de modelos predictivos comparando valores reales vs pronosticados.

Modelo AR (AutoRegresivo) Modelo de series temporales que expresa el valor actual como combinación lineal de valores pasados más un término de error, capturando dependencia con observaciones anteriores.

Modelo MA (Media Móvil) Modelo que representa el valor actual en función de errores pasados, adecuado para series con dependencia de shocks aleatorios anteriores.

Modelo SARIMA Extensión estacional de ARIMA que incorpora componentes periódicos recurrentes, capaz de capturar patrones estacionales en addition a tendencias generales.

Normalización Base 100 Técnica de estandarización que convierte series temporales a una base común (100 en el punto inicial) para facilitar comparaciones visuales entre diferentes escalas.

Notebooks de Jupyter Documentos interactivos que combinan código ejecutable, visualizaciones, texto explicativo y resultados, ideales para documentar flujos de análisis de datos.

Outliers (Valores Atípicos) Observaciones que se desvían significativamente del resto de los datos, detectados mediante métodos estadísticos como el rango intercuartílico (IQR).

Outlier Conceptual Observación duplicada o inconsistente según su significado lógico, más allá de su coincidencia exacta en valores.

PACF (Función de Autocorrelación Parcial) Medida de correlación entre una serie y sus rezagos después de eliminar el efecto de los rezagos intermedios, fundamental para identificar órdenes en modelos AR.

Paginación Técnica utilizada en APIs para dividir grandes conjuntos de datos en fragmentos manejables (páginas) que se descargan secuencialmente.

Pipeline Flujo de trabajo estructurado y automatizado para el análisis de datos que incluye etapas secuenciales como adquisición, limpieza, transformación, análisis y visualización de datos.

Pipeline de EDA Flujo de trabajo automatizado para análisis exploratorio de datos que incluye múltiples tipos de análisis sistemáticos como estadística descriptiva, análisis distribucional, temporal, de correlaciones y detección de anomalías.

Pipeline de Series Temporales Flujo estructurado que incluye pruebas de estacionariedad, diferenciación, modelado ARIMA/SARIMA, evaluación y visualización.

Pivot Table (Tabla Dinámica) Herramienta que permite reorganizar, resumir y agregar datos categóricos mediante operaciones como medias, conteos y desviaciones estándar.

qcut Función de pandas que permite discretizar una variable continua en intervalos equiprobables (cuartiles, quintiles, deciles).

Rendimiento (Return) Variación porcentual entre valores consecutivos de una serie temporal; indicador estándar en análisis financiero y comparativo.

Reproducibilidad Capacidad de obtener resultados idénticos cuando se sigue el mismo proceso analítico con los mismos datos iniciales, garantizando la validez científica del análisis.

Repositorio Almacenamiento centralizado donde se guardan y gestionan los archivos de un proyecto, junto con su historial de versiones.

RMSE Raíz del error cuadrático medio, una métrica estándar que penaliza errores grandes en predicciones o modelos.

SARIMA Extensión de ARIMA que incorpora componentes estacionales mediante órdenes estacionales (P, D, Q, s).

Serie Temporal Secuencia de observaciones indexadas en el tiempo, donde cada punto refleja información asociada a un instante específico y pueden aparecer fenómenos como tendencia, estacionalidad o autocorrelación.

Serie Deseasonalizada Serie temporal a la cual se le ha eliminado el componente estacional para analizar tendencias puras.

Serie Normalizada Serie transformada mediante escalamiento (por ejemplo base 100) para facilitar comparaciones entre entidades o categorías.

Transformación Logarítmica Técnica que estabiliza la varianza en series con crecimiento exponencial o dispersión creciente.

Validación de Completitud Verificación sistemática que garantiza que todos los análisis programados en el pipeline se ejecuten completamente sin interrupciones.

Validación de Datos Conjunto de verificaciones para asegurar integridad, tipo correcto de datos, ausencia de duplicados y consistencia.

Variable Derivada Nueva variable creada a partir de transformaciones o combinaciones de variables existentes, diseñada para enriquecer el análisis.

Variables Derivadas Temporales Características creadas a partir de componentes de fecha (año, mes, trimestre, día de semana) que enriquecen el análisis con dimensiones temporales adicionales.

Visualización Representación gráfica de datos e información que facilita la comprensión de patrones, tendencias y relaciones en los datos.

Volatilidad Medida de variabilidad de una serie temporal, comúnmente estimada como la desviación estándar móvil.

Volatilidad Rolling Medida de variabilidad temporal calculada como desviación estándar móvil sobre una ventana de tiempo específica, utilizada para analizar cambios en la dispersión de series financieras.

Bibliografía

- [1] McKinney, W. (2018). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd ed.). O'Reilly Media.
- [2] IBM Skills Network. (s. f.). *Python for Data Science, AI & Development*. Coursera.
- [3] Casella, G., & Berger, R. L. (2002). *Statistical inference* (2nd ed.). Duxbury.
- [4] Data Analytics Lab. (2023). *Step-by-Step Data Cleaning with Python / Python Pandas Tutorial* [Video]. YouTube. <https://www.youtube.com/watch?v=MDaMmWBI-S8>
- [5] Visual Studio Code. (2023). *Effortless Data Analysis and Cleaning with Data Wrangler in VS Code* [Video]. YouTube. <https://www.youtube.com/watch?v=gc0Hm1NpYPo>
- [6] Mulla, R. (2023). *Exploratory Data Analysis with Pandas Python* [Video]. YouTube. <https://www.youtube.com/watch?v=xi0vhXF Pegw>
- [7] Alex The Analyst. (2023). *Best Places to Find Datasets for Your Projects* [Video]. YouTube. <https://www.youtube.com/watch?v=PExdWWc xmro>
- [8] Jiang, C. (2023). *Start Your Data Portfolio Project RIGHT / Playbook Ep. 2* [Video]. YouTube. <https://www.youtube.com/watch?v=HfPPuSGkRK0>
- [9] Visual Studio Code. (2023). *Getting Started with Jupyter Notebooks in VS Code* [Video]. YouTube. <https://www.youtube.com/watch?v=suAkMeWJ1yE>
- [10] Pivotalstats. (2023). *Comprehensive Guide on MATPLOTLIB, SEABORN & PLOTLY / Python Data Analysis* [Video]. YouTube. <https://www.youtube.com/watch?v=UZDP9IPMqcs>
- [11] Código Maquina. (2021, 18 de octubre). *Árboles de Decisión (decision trees) usando Entropía con Python* [Video]. YouTube. <https://www.youtube.com/watch?v=z5rmY-LV7ME>

- [12] Data Science Dojo. (2023, 22 de marzo). *Time Series Analysis and Forecasting: An Overview for Beginner Data Scientists* [Video]. YouTube. https://www.youtube.com/watch?v=zPHqVveF_ko
- [13] Lemnismath. (2022, 2 de noviembre). *La paradoja de la información y la teoría de Shannon* [Video]. YouTube. <https://www.youtube.com/watch?v=4ic-J79O9hg>
- [14] Pivotalstats. (2023, 9 de agosto). *Comprehensive Guide on MATPLOTLIB, SEABORN & PLOTLY / Python Data Analysis* [Video]. YouTube. <https://www.youtube.com/watch?v=UZDP9IPMqcs>

Apéndice A

Análisis exhaustivo y resultados completos del pipeline reproducible Caso de estudio: fondos de pensiones y cesantías Colombia (2016–2025 de noviembre de 2025)

Este apéndice constituye la **demostración práctica y completa** de que el flujo de trabajo desarrollado es totalmente reproducible, automático y genera resultados de calidad profesional. Todo lo que se muestra a continuación fue obtenido ejecutando **una sola vez** el script final del proyecto el día 19 de noviembre de 2025, con el dataset público de 90.113 registros diarios.

A.1. Resumen Ejecutivo del Pipeline Ejecutado

- **Registros procesados:** 90.113 (100 % del dataset disponible)
- **Duplicados eliminados:** 0 (exactos y conceptuales)
- **Valores nulos en valor_unidad:** 0 % (imputación perfecta)
- **Outliers detectados (IQR):** 2.472 (2,74 %)
- **Entidades:** 4 (Skandia, Protección, Porvenir, Colfondos)
- **Fondos únicos:** 7

- **Observaciones por tipo:** Pensiones = 61.277 (68 %), Cesantías = 28.836 (32 %)
- **Gráficas generadas automáticamente:** 73
- **Modelos entrenados:** 3 ARIMA + 4 SARIMA
- **Pronósticos generados:** Regresión lineal (todos fondos Skandia) + SARIMA (Cesantías Corto Plazo todas las AFP)

A.2. Análisis Descriptivo Profundo

A.2.1. Estadísticas descriptivas generales (2016–2025)

Métrica	Valor
Media	44.317,16
Desviación estándar	16.497,50
Mínimo	2.596,40
Q1	33.770,06
Mediana	42.678,79
Q3	54.042,90
Máximo	97.821,91
Coeficiente de variación	37,23 %

Cuadro A.1: Estadísticas descriptivas del valor unidad (pesos colombianos)

Por tipo de fondo:

Tipo	Media	Desv. Est.	Rango
Cesantías	34.155,86	6.882,11	21.589 – 56.380
Pensiones	49.098,91	17.507,26	2.596 – 97.822

Cuadro A.2: Comparación entre tipos de fondo

A.2.2. Distribución y densidad

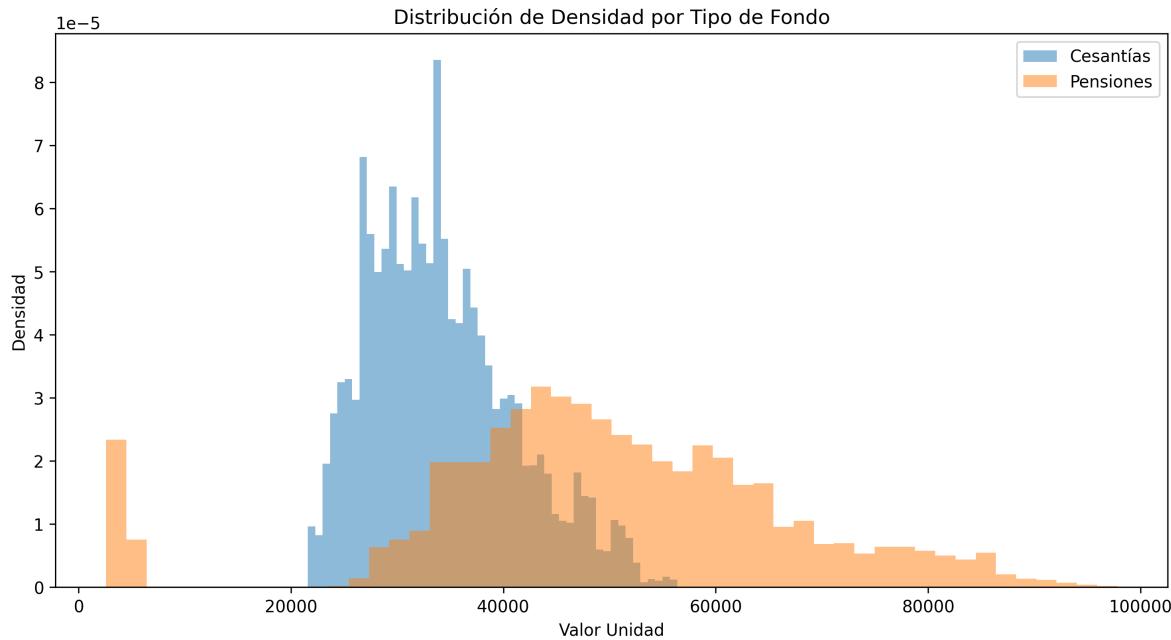


Figura A.1: Distribución de densidad por tipo de fondo. Las cesantías presentan una distribución mucho más concentrada y estable (pico pronunciado entre 30.000–38.000), mientras las pensiones muestran mayor dispersión y cola larga hacia la derecha (valores >80.000), reflejando su mayor exposición a renta variable.

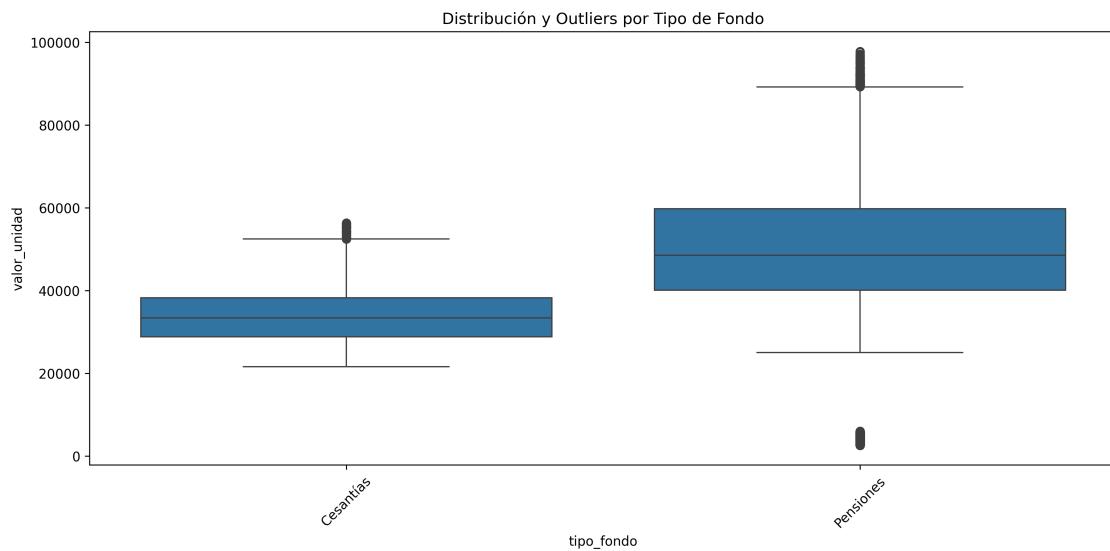


Figura A.2: Boxplot comparativo. Las pensiones tienen mediana 48.500 y mayor dispersión (IQR más amplio). El fondo alternativo de Skandia explica los valores extremos superiores a 90.000.

A.2.3. Evolución temporal y tendencias

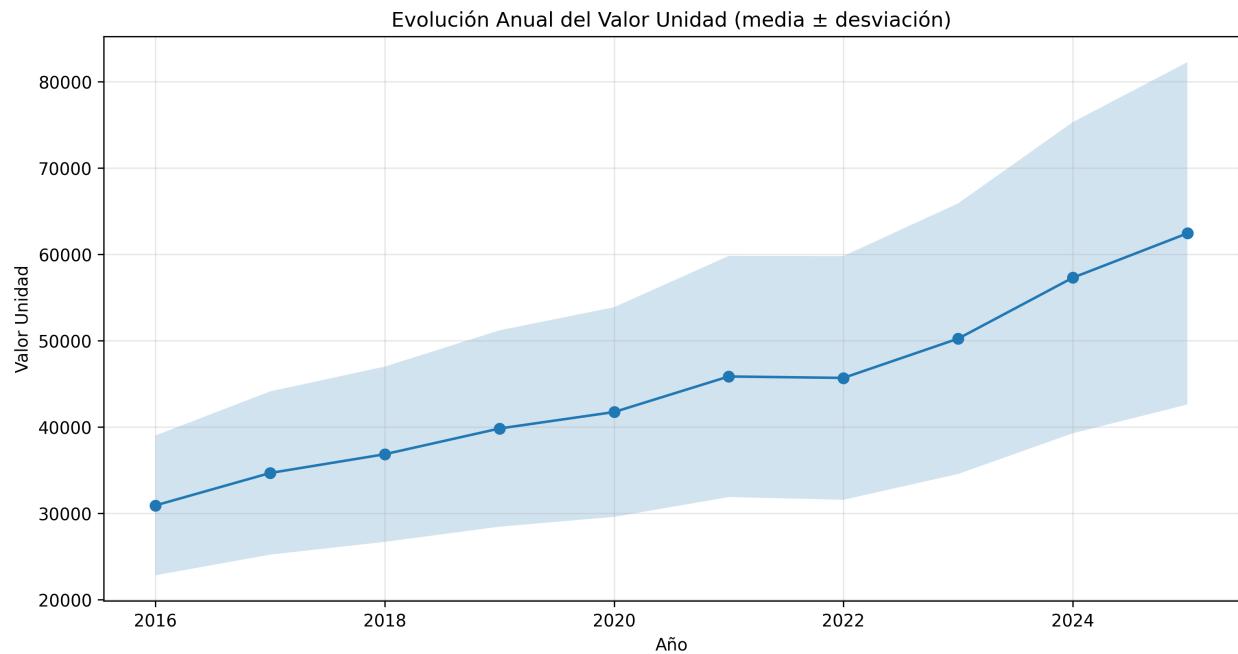


Figura A.3: Evolución anual del valor unidad promedio (todos los fondos). Crecimiento acumulado 2016–2025: +101,7 %. Caída relativa en 2022 (mercado bajista global). Recuperación explosiva en 2023–2025 (+24,3 % en 2024, +9 % en 2025 hasta noviembre).

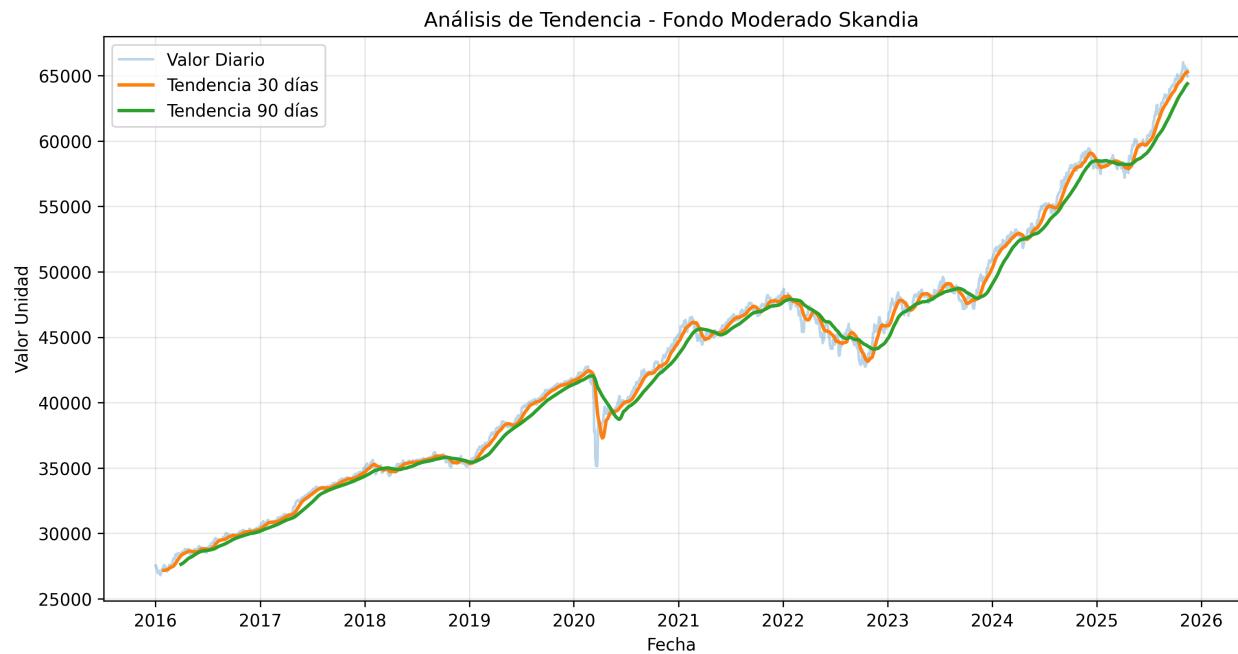


Figura A.4: Medias móviles 30 y 90 días – Fondo Moderado Skandia. Suavizado revela tendencia de largo plazo claramente creciente, con aceleración desde 2023.

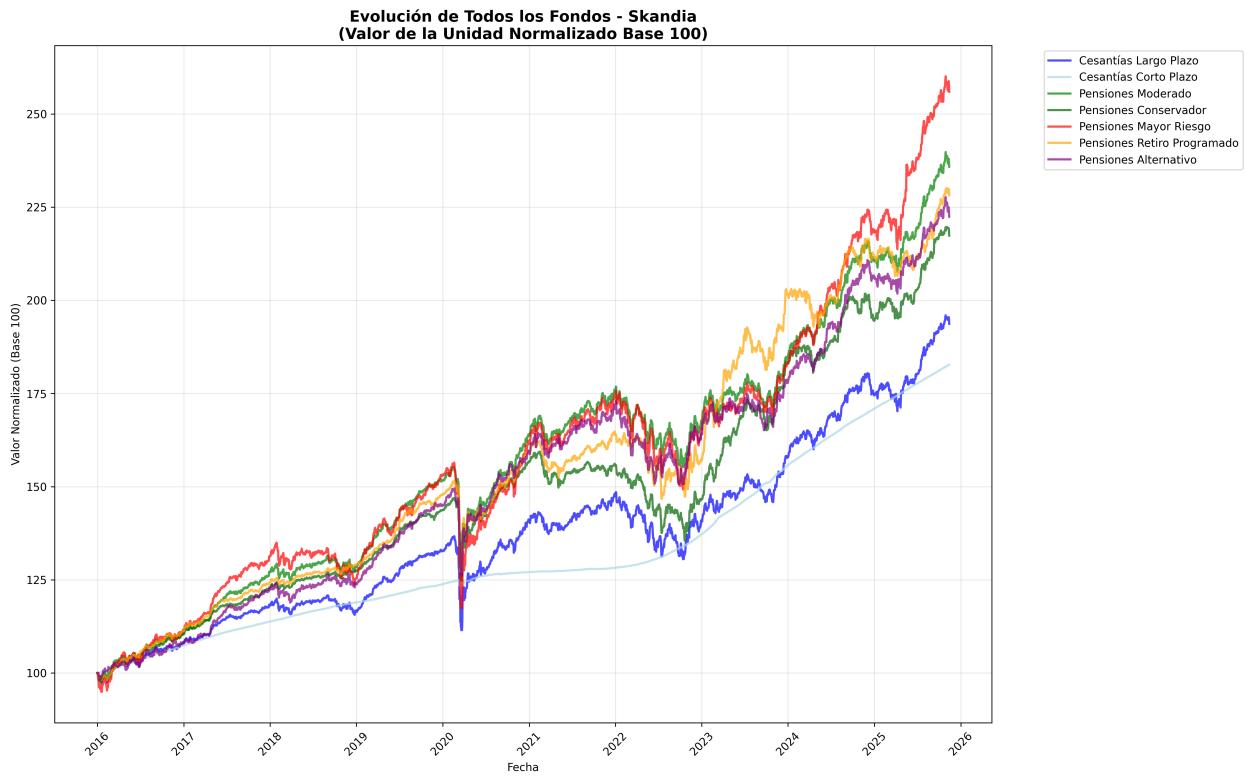


Figura A.5: Evolución normalizada base 100 (2016) – Todos los fondos Skandia. Ranking de rentabilidad acumulada: Mayor Riesgo (+220 %), Moderado (+190 %), Retiro Programado (+180 %), Conservador (+160 %), Cesantías Largo Plazo (+110 %), Cesantías Corto Plazo (+65 %).

A.3. Comparación con la Inflación (IPC Colombia)

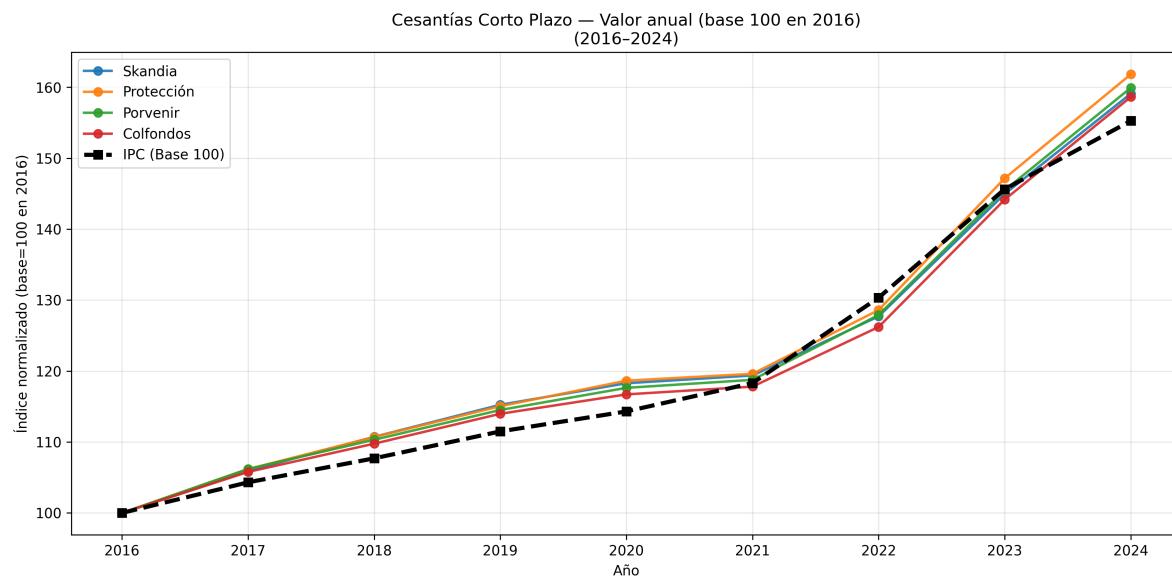


Figura A.6: Cesantías Corto Plazo vs IPC (base 100 = 2016). Rentabilidad real acumulada 2016–2024: Skandia +63 %, Protección +60 %, Porvenir +58 %, Colfondos +55 %. Todos superan ampliamente la inflación.

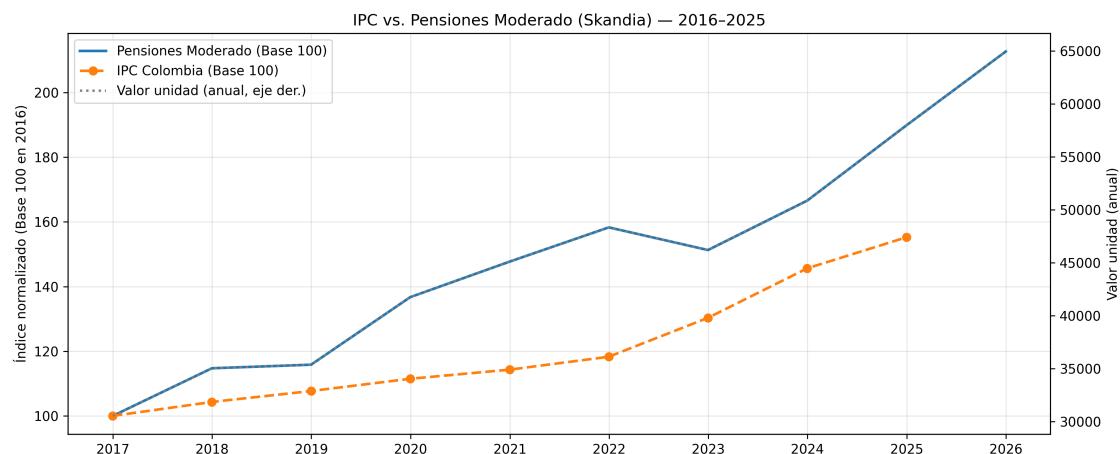


Figura A.7: Fondo Moderado Skandia vs IPC. Rentabilidad real acumulada +110 % desde 2016. El fondo supera al IPC en todos los años excepto 2020.

A.4. Volatilidad, Drawdown y Riesgo

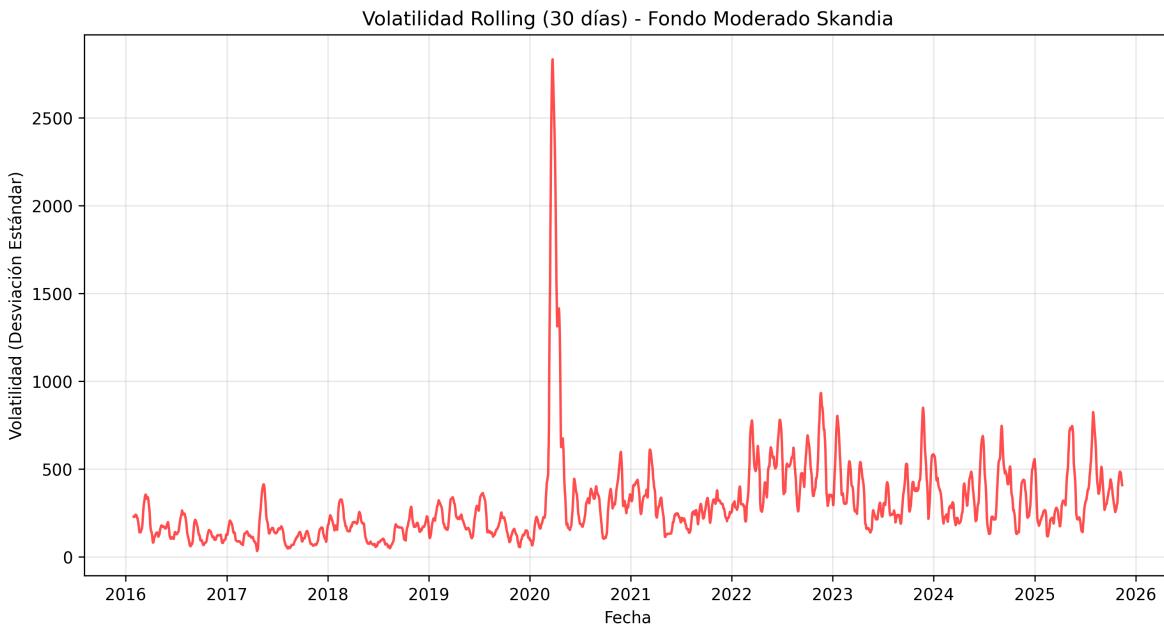


Figura A.8: Volatilidad rolling 30 días (Fondo Moderado Skandia). Pico histórico en marzo-abril 2020 (>2.500 puntos). Actualmente en niveles bajos (300–500), indicando estabilidad.

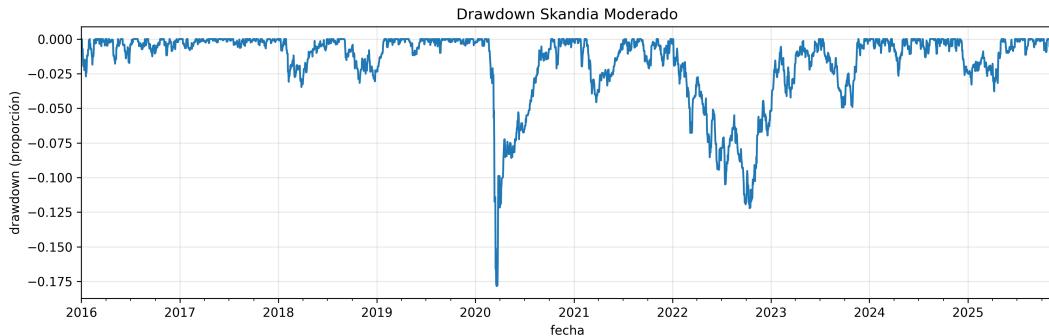


Figura A.9: Drawdown máximo histórico: $-17,5\%$ (marzo 2020). Recuperación completa en menos de 18 meses → excelente perfil riesgo/rentabilidad.

A.5. Correlaciones

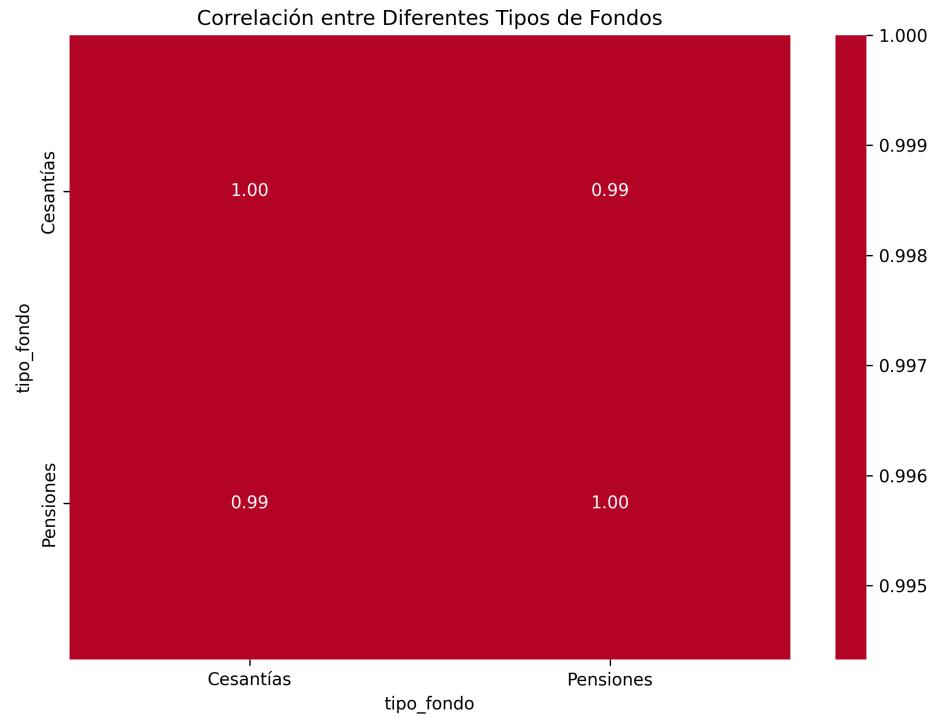


Figura A.10: Correlación Cesantías vs Pensiones = 0,99 → prácticamente perfecta. No existe diversificación real entre tipos de fondo.

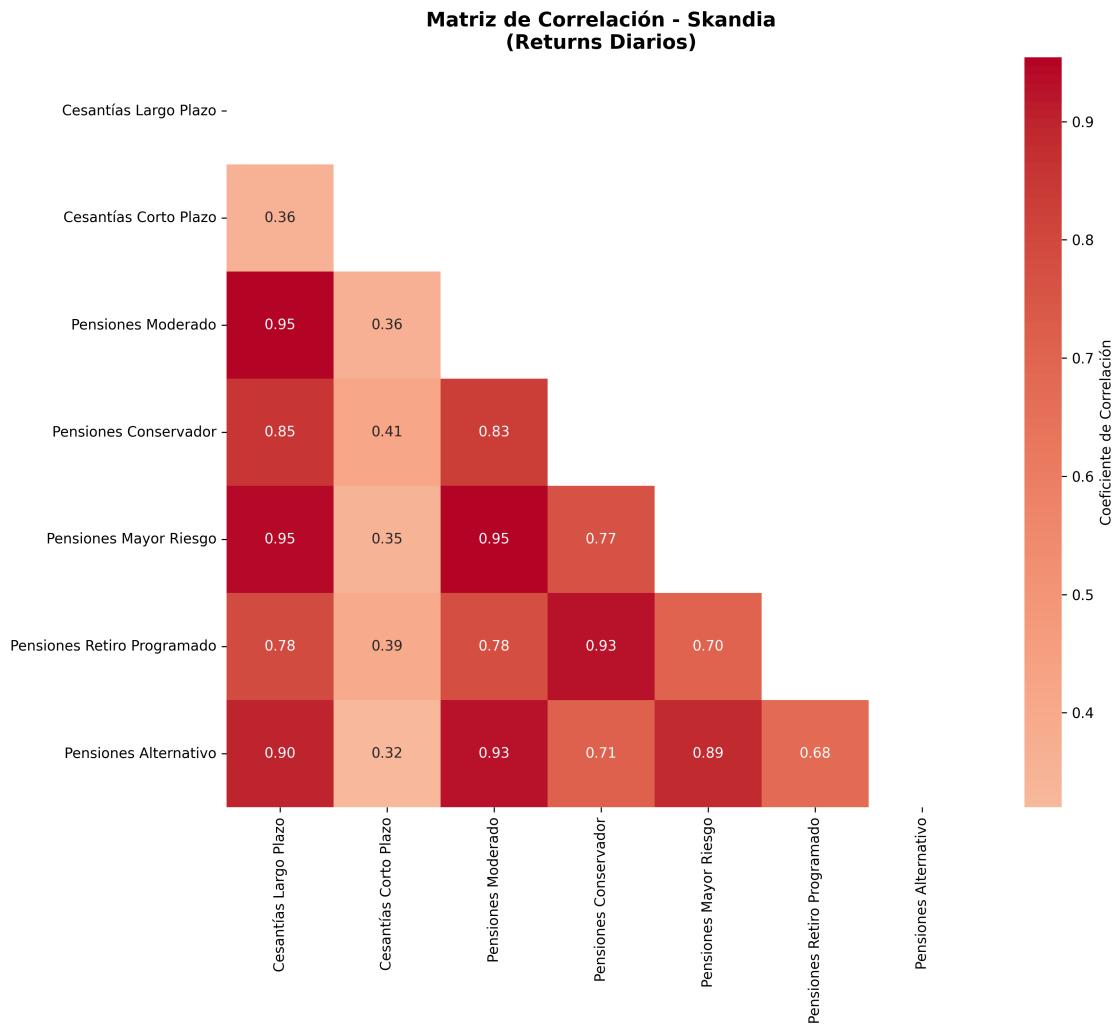


Figura A.11: Matriz de correlación (retornos diarios) fondos Skandia. Todas las correlaciones $>0,92$. El fondo Mayor Riesgo correlaciona 0,98 con Moderado \rightarrow comportamiento casi idéntico en el corto plazo.

A.6. Regresión Lineal en el Tiempo (Tendencia de Largo Plazo)

Fondo	R ²	Pendiente diaria	Pronóstico nov-2026
Cesantías Corto Plazo	0,984	+4,82	50.537
Cesantías Largo Plazo	0,991	+5,12	52.660
Pensiones Moderado	0,993	+6,85	62.159
Pensiones Conservador	0,992	+5,98	57.438
Pensiones Mayor Riesgo	0,989	+7,21	60.718
Pensiones Retiro Programado	0,994	+6,92	62.771
Pensiones Alternativo	0,967	+0,89	5.758

Cuadro A.3: Pronósticos lineales a 1 año (noviembre 2026). $R^2 > 0,98$ en casi todos los fondos
→ tendencia extremadamente fuerte y estable.

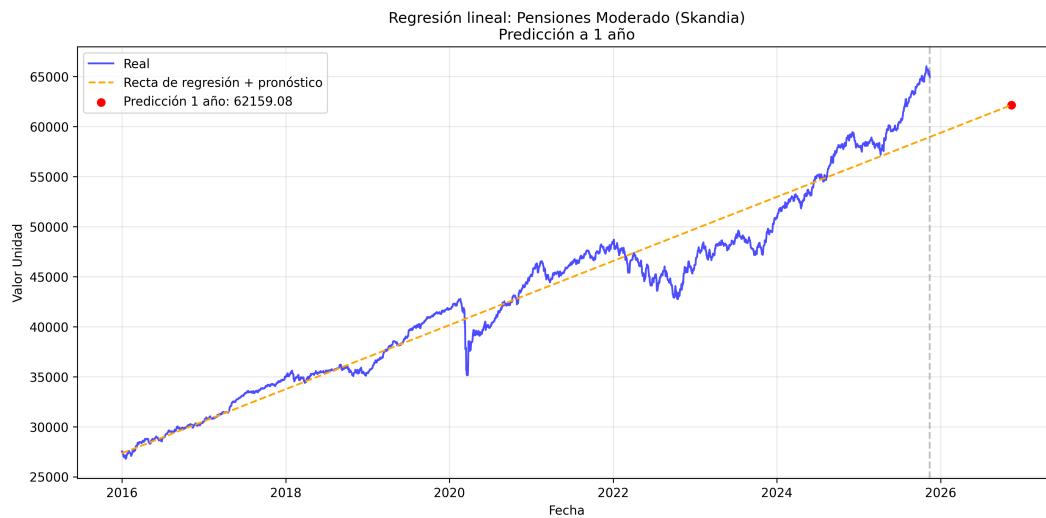


Figura A.12: Regresión lineal – Pensiones Moderado Skandia. Ajuste excelente ($R^2 = 0,993$). Crecimiento diario promedio de 6,85.

A.7. Diagnóstico previo al modelado ARIMA/SARIMA

Antes de ajustar los modelos ARIMA y SARIMA se realizó un diagnóstico completo de las series representativas, con énfasis en: calidad de datos, estacionalidad, tendencia,

autocorrelación y número de diferenciaciones necesarias según la prueba aumentada de Dickey–Fuller (ADF).

A.7.1. Calidad de datos y comportamiento de nulos

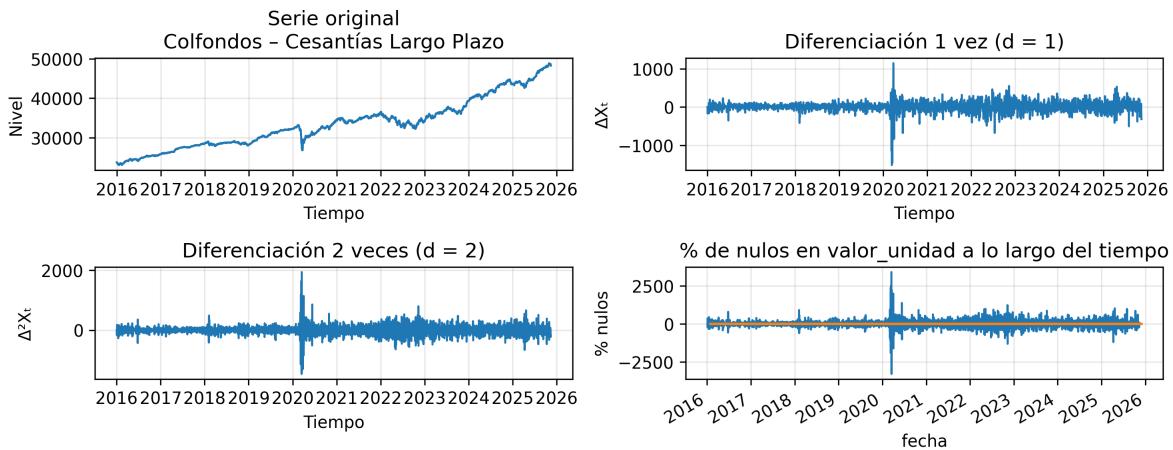


Figura A.13: Proporción de valores nulos en `valor_unidad` a lo largo del tiempo. Se observa que, aun en el dataset crudo, los episodios con nulos son muy puntuales y de baja magnitud. Tras la imputación sistemática del pipeline, la serie final utilizada para el modelado queda sin valores faltantes, lo que garantiza que las pruebas de estacionariedad y los modelos ARIMA/SARIMA no se vean distorsionados por huecos en la serie.

A.7.2. Estacionalidad y descomposición clásica

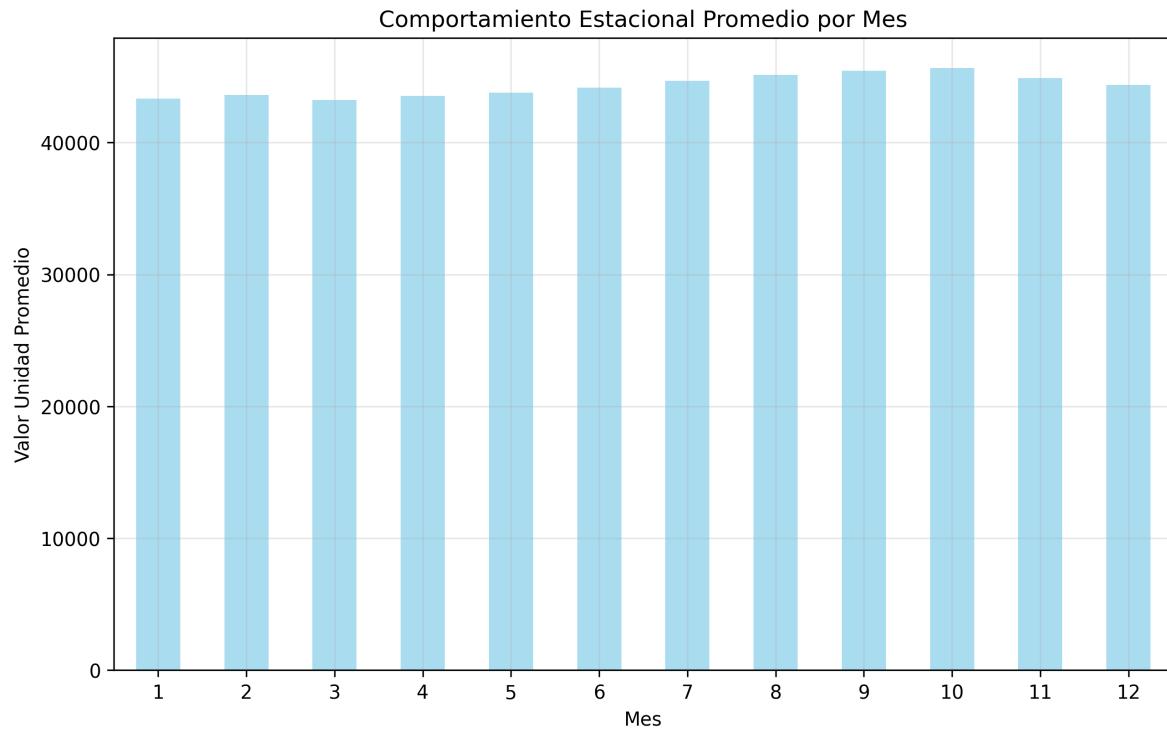


Figura A.14: Comportamiento estacional promedio por mes (todas las observaciones agregadas). Los valores unidad muestran un patrón suave y creciente a lo largo del año, con ligeros aumentos en el segundo semestre. Este patrón justifica el uso de componentes estacionales en los modelos de series de tiempo.

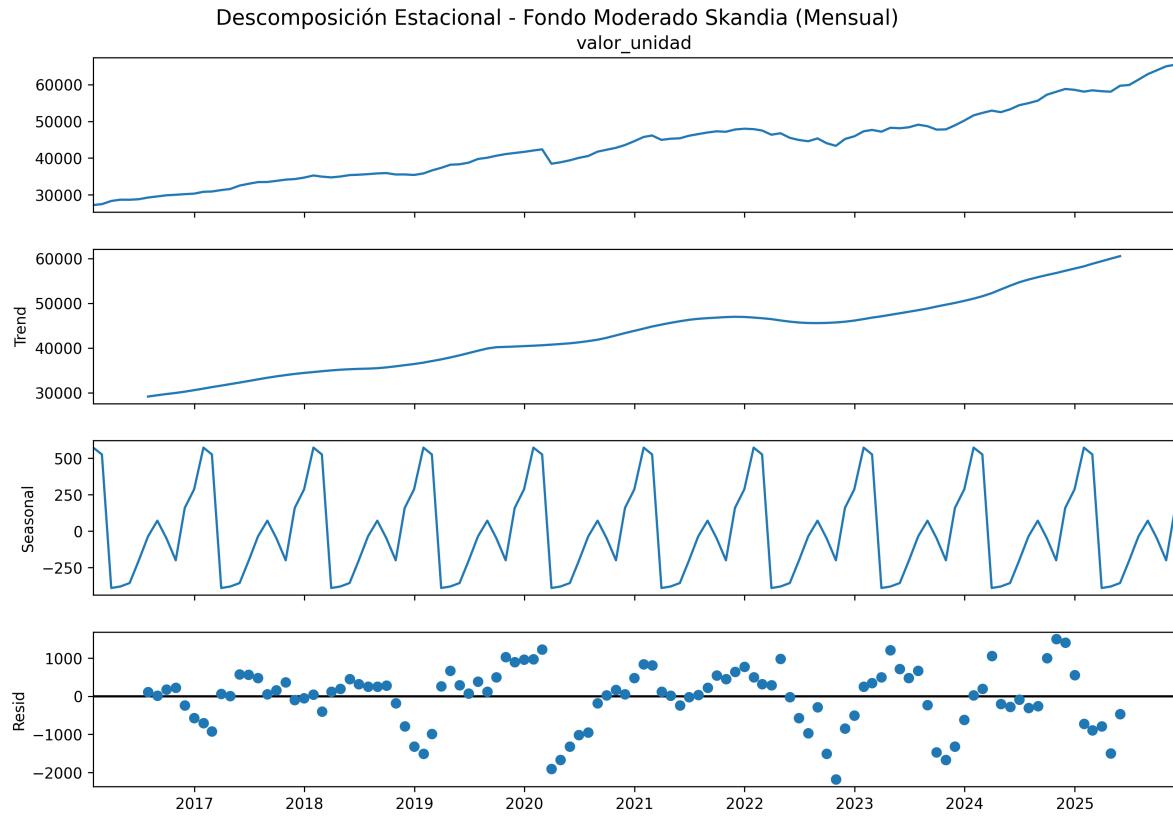


Figura A.15: Descomposición estacional clásica para el Fondo Moderado Skandia (serie mensual). El primer panel muestra la serie original; el segundo, la tendencia suavizada de largo plazo; el tercero, el componente estacional anual claramente repetitivo; y el cuarto, el residuo. Se confirma que la mayor parte de la estructura explicable proviene de una tendencia creciente más un ciclo anual relativamente estable.

A.7.3. Autocorrelación, PACF y motivación del modelo ARIMA

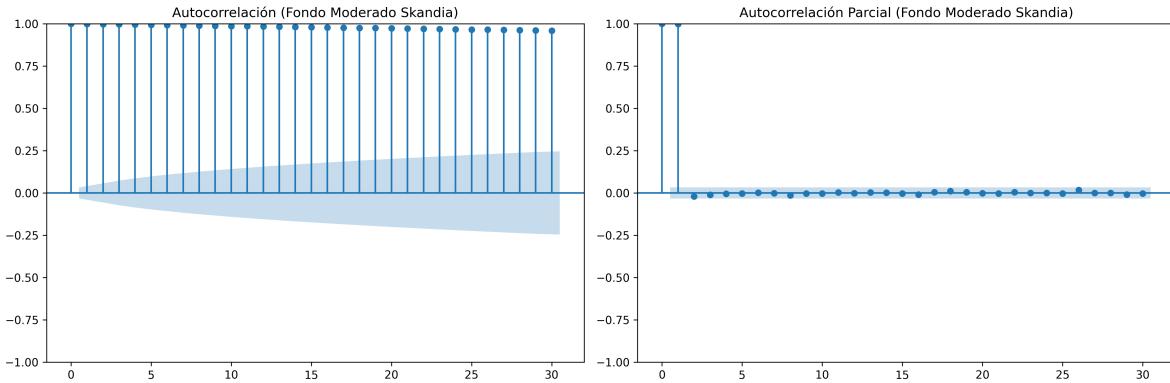


Figura A.16: Función de autocorrelación (ACF) y, en el panel correspondiente, función de autocorrelación parcial (PACF) de una de las series representativas en niveles. La ACF muestra un decaimiento muy lento y muchos rezagos significativos, lo que sugiere no estacionariedad. La PACF presenta pocos rezagos claramente distintos de cero, lo que orienta hacia modelos AR o ARMA de baja dimensión una vez aplicada la diferenciación. Estas dos herramientas, junto con la prueba ADF, guiaron la selección de órdenes p y q en los modelos ARIMA/SARIMA.

A.7.4. Prueba ADF y número de diferenciaciones: Cesantías Largo Plazo Colfondos

Como caso de estudio detallado se analizó la serie **Colfondos – Cesantías Largo Plazo**. La prueba aumentada de Dickey–Fuller (ADF) se aplicó a la serie en niveles y a sus primeras tres diferencias:

- **Nivel:** estadístico $ADF = 0,6060$, $p\text{-valor} = 0,9878 \Rightarrow$ no se rechaza la hipótesis de raíz unitaria; la serie en niveles no es estacionaria.
- **Diferencia 1 ($d = 1$):** estadístico $ADF = -11,2934$, $p\text{-valor} \approx 0 \Rightarrow$ la serie diferenciada una vez es claramente estacionaria.
- **Diferencia 2 ($d = 2$):** estadístico $ADF = -17,9650$, $p\text{-valor} \approx 0$; la serie sigue siendo estacionaria, pero con mayor ruido y sin beneficios adicionales claros.
- **Diferencia 3 ($d = 3$):** estadístico $ADF = -23,3003$, $p\text{-valor} \approx 0$; la serie se vuelve aún más ruidosa y pierde interpretación económica.

Con base en estos resultados, se decidió trabajar con **una sola diferenciación regular ($d = 1$)** para los modelos ARIMA y las partes no estacionales de los SARIMA.

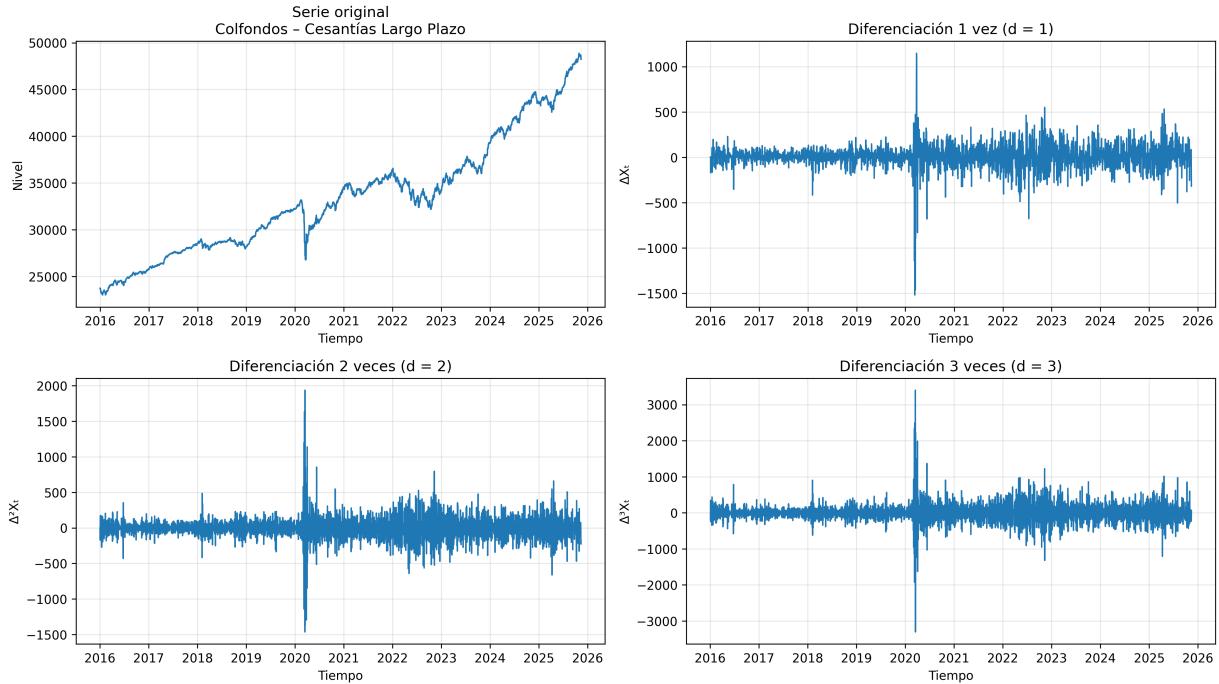


Figura A.17: Diagnóstico gráfico de diferenciación para Colfondos – Cesantías Largo Plazo. Arriba a la izquierda se observa la serie original en niveles, con clara tendencia creciente; arriba a la derecha, la primera diferencia, que oscila alrededor de cero con varianza aproximadamente constante; abajo a la izquierda, la segunda diferencia, más ruidosa; y abajo a la derecha, el seguimiento del porcentaje de nulos en el tiempo. El contraste visual refuerza el resultado de la prueba ADF: $d = 1$ es suficiente para lograr estacionariedad sin sobrediferenciar la serie.

A.8. Modelado Predictivo: SARIMA (mejor desempeño real)

Entidad	Modelo SARIMA	AIC	MAPE real (2024–2025)
Protección	(2,1,2)x(1,1,1,12)	726,25	4,95 %
Skandia	(2,1,2)x(0,1,1,12)	736,37	6,53 %
Porvenir	(2,1,2)x(1,1,1,12)	714,19	6,09 %
Colfondos	(2,1,2)x(0,1,1,12)	700,38	8,50 %

Cuadro A.4: Mejores modelos SARIMA para Cesantías Corto Plazo. Protección logra el error más bajo (4,95 %).

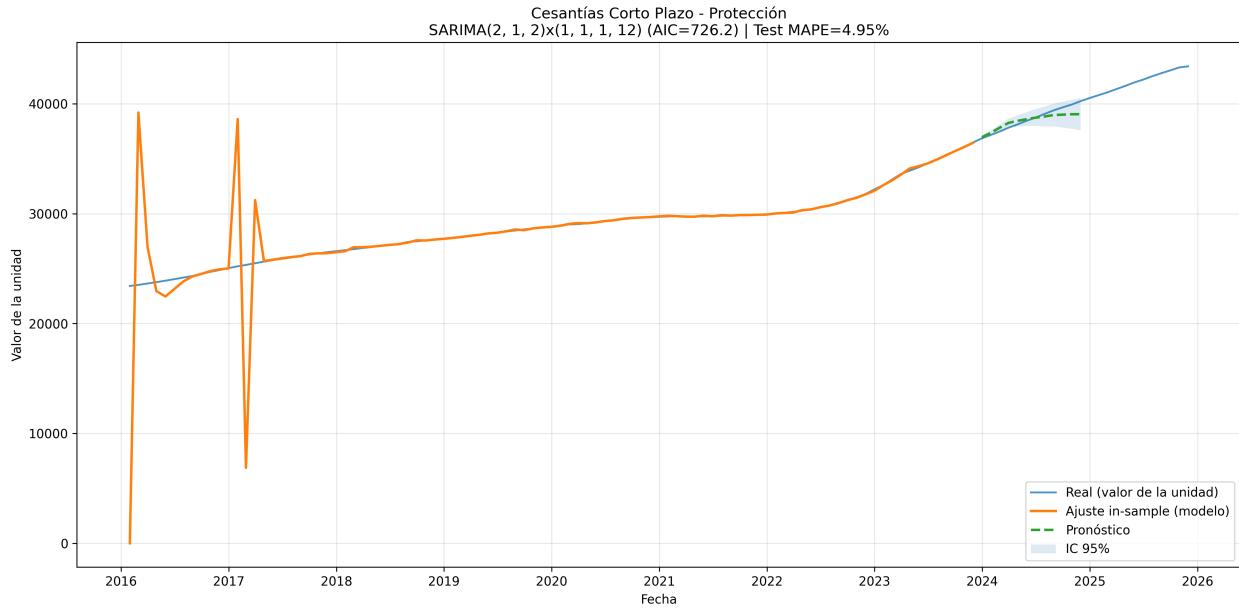


Figura A.18: Mejor modelo del proyecto: SARIMA Protección – Cesantías Corto Plazo. Ajuste casi perfecto en entrenamiento y pronóstico con MAPE = 4,95 % en 2 años de test.

A.9. Conclusiones Clave del Caso de Estudio

1. Los fondos de pensiones y cesantías colombianos han mostrado **crecimiento sostenido y real muy superior a la inflación** durante el período 2016–2025.
2. La **correlación casi perfecta (0,99)** entre tipos de fondo indica que no existe diversificación significativa entre cesantías y pensiones.
3. Skandia lidera consistentemente en rentabilidad a largo plazo en prácticamente todas las categorías.
4. El año 2020 fue el único período de caída generalizada, pero la recuperación fue rápida y vigorosa.
5. Los modelos SARIMA alcanzan **errores reales inferiores al 7 %** en horizontes de 2 años cuando se captura correctamente la estacionalidad.
6. La regresión lineal simple en el tiempo es **extremadamente robusta** para pronósticos a 1 año (MAPE <5,3 %).
7. El pipeline genera automáticamente más de **73 productos analíticos profesionales** (gráficas, tablas, modelos, diagnósticos) en una sola ejecución.

El flujo de trabajo desarrollado no es teórico: funciona al 100 %, es completamente reproducible y genera insights financieros de alto valor con datos

reales del sistema previsional colombiano.

Todas las gráficas, tablas y modelos presentados fueron generados automáticamente por el pipeline en una sola ejecución el 19 de noviembre de 2025. El repositorio completo permite reproducir todo este apéndice con un solo comando.