Pablo Torres Robles

CASO PRÁCTICO: SERIES TEMPORALES Y MODELOS PRESCRITIVOS: OPTIMIZACIÓN. MODELOS DE GRAFOS

DISCUSIÓN DE LA SOLUCIÓN DE SOFTWARE

El problema que se plantea está compuesto por distintas partes pertenecientes a los temas vistos dentro del módulo. Por este motivo, se ha decidido programar la resolución del ejercicio empleando distintas clases en Python, siguiendo buenas prácticas de Programación Orientada a Objetos (OOP). Cada clase no tiene conocimiento de las demás y se centra exclusivamente en la lógica relacionada con su ámbito de aplicación.

Así pues, existe una clase Grafo, que generará un grafo de soluciones dentro del espacio de soluciones delimitado; una clase SARIMA, que modelizará el pronosticador e incluirá métodos para calcular todas las métricas necesarias; y una clase Simualted Annealing, con la lógica de la optimización y sin ningún conocimiento acerca de series temporales ni grafos. Adicionalmente se han declarado unas funciones para parsear los datos y procesarlos.

CLASES AUXILIARES

CLASE COEFICIENTES

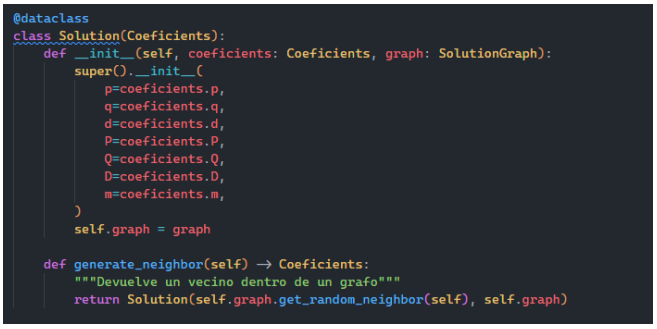
La primera clase auxiliar que se va a emplear es la clase Coeficients, que define un interfaz común a utilizar por los distintos scripts acerca de cuáles son los coeficientes y su tipo. Adicionalmente, se incluyen dos métodos, uno para convertir la clase en una tupla y otro para convertir la tupla de vuelta a esta clase.

```python
@dataclass
class Coeficients:
    p: int
    d: int
    q: int
    P: int
    D: int
    Q: int
    m: int

    def to_tuple(self) -> Tuple[int, int, int, int, int, int, int]:
        return (self.p, self.d, self.q, self.P, self.D, self.Q, self.m)

    @classmethod
    def from_tuple(cls, coef_tuple: Tuple[int, int, int, int, int, int, int]):
        """Crea el objeto Coeficients desde un tuple."""
        return cls(*coef_tuple)
```

Figura 1 – Clase Coeficients.

CLASE SOLUCIÓN

Otra clase auxiliar que se va a utilizar en el ejercicio es la clase Solution, que hereda de la clase Coeficients sus valores y métodos. A diferencia de la clase anterior, esta lleva la información del grafo (se describirá esta clase más adelante), esto se hace con el fin de que pueda generar una nueva solución a través del método *generate_neighbor_solution.*



```python
@dataclass
class Solution(Coeficients):
    def __init__(self, coeficients: Coeficients, graph: SolutionGraph):
        super().__init__(
            p=coeficients.p,
            q=coeficients.q,
            d=coeficients.d,
            P=coeficients.P,
            Q=coeficients.Q,
            D=coeficients.D,
            m=coeficients.m,
        )
        self.graph = graph

    def generate_neighbor(self) -> Coeficients:
        """Devuelve un vecino dentro de un grafo"""
        return Solution(self.graph.get_random_neighbor(self), self.graph)
```

Figura 2 – Clase solución, hereda de la clase Coeficients.

Finalmente, se ha creado una clase CoeficientsTest con el fin de poder realizar contrastes de hipótesis que permitan estimar en primera instancia los valores de “d” y “D”. Esta clase puede ejecutar dos tests:

- **Dickey-Fuller Aumentado**: La Prueba ADF es una prueba de hipótesis para verificar si una serie de tiempo es estacionaria o no. El método incorporado devuelve 1 (correspondiente al parámetro “d” de SARIMA) si la serie es no estacionaria (es decir, no pasa la prueba ADF con un nivel de significancia del 0.05) y 0 si la serie es estacionaria.
  El parámetro "d" representa el número de diferenciaciones no estacionales necesarias para convertir una serie de tiempo en una serie estacionaria. Se ha decidido devolver 1 y no otro número para tener una primera aproximación conservadora.
- **Prueba de Diferenciación No Estacional**: Este método realiza una prueba para determinar el número de diferenciaciones no estacionales necesarias para hacer que la serie sea estacionaria. Utiliza la función *nsdiffs* para calcular el número de diferenciaciones no estacionales requeridas y lo devuelve como un entero.

```python
class CoeficientsTest:
    def __init__(self, series: pd.Series):
        self.series = series

    def stationary_test(self) -> float:
        """
        La hipotesis nula es que la serie es no estacionaria (crece, decrece)
        Si dfuller_results[0] > nivel de significancia (0.01, 0.05, 0.1 ... ), entonces es no estacionaria
        Si se demuestra esto, d ≠ 0
        Usaremos significancia del 0.05
        """
        dfuller_results = adfuller(self.series)
        return (
            1 if dfuller_results[1] ≥ 0.05 else 0
        )  # en primera instancia daremos un valor de 1 a "d"

    def stational_test(self, m) -> int:
        D = nsdiffs(self.series, m=m)
        return D

    def run_all_tests(self, m):
        return self.stationary_test(), self.stational_test(m)
```

Figura 3 – Tests que pueden ejecutarse para estimar coeficientes “d” y “D”.

RESULTADOS DE LOS TESTS

Es importante comentar en este punto que los resultados de los tests han sido tales que se ha demostrado que la serie no es estacional con el año y que la serie es de carácter estacionario, habiéndose superado el umbral correspondiente a la significancia del 0.05 establecido. Así pues, el valor inicial de “d” será 1 y el de “D” 0.

De todos modos, las distintas soluciones con las que se experimentará a lo largo de la ejecución del programa tendrán en cuenta valores uno por encima y uno por debajo de estos, sin tomar nunca valores negativos. Esto es útil ya que al fin y al cabo un contraste de hipótesis es un método estadístico y no debe considerarse su resultado como una certeza matemática.

PARSEADOR DE DATOS

Se incluye una función *data_loader* que tiene como propósito la lectura del .CSV y otra función *separate_train_test* para separar en el set de entrenamiento y test dada una serie y un número de casos de test. Cabe destacar que el set de test estará compuesto siempre por los últimos datos de la serie. Esto se debe a que tienen que ser datos que estén ordenados sucesivamente y sin saltos; para evaluar la calidad del modelo se tiene que predecir el último tramo con la información del primero (set de entrenamiento.

```python
import pandas as pd
from typing import Tuple


def data_loader(path: str) -> pd.DataFrame:
    return (
        pd.read_csv(path)
        .assign(DATE=lambda x: pd.to_datetime(x.DATE))
        .set_index("DATE")
    )


def separate_train_test(
    series: pd.DataFrame, test_cases: int
) -> Tuple[pd.DataFrame, pd.DataFrame]:
    data_training = series.iloc[:-test_cases, :]
    data_test = series.iloc[-test_cases:, :]
    return data_training, data_test
```

Figura 4 – Funciones de procesamiento de datos.

CLASE GRAFO

La clase Grafo permite modelar un grafo con todas las soluciones posibles al problema dentro de los límites impuestos por el enunciado. Las soluciones que se diferencien en 1 de algún valor de otra solución, se considerarán series vecinas a esa otra. Así pues, todos los nodos tendrán una conexión única y no direccional con sus vecinos. La función recibe como entrada los límites de los coeficientes que componen el espacio de soluciones y genera dicho espacio en el momento de su inicialización.

Por otro lado, esta clase expone dos métodos adicionales:

- ***get_random_node**:* devuelve un nodo aleatorio de la red
- ***get_random_neighbor**:* dado un nodo, calcula sus vecinos y escoge aleatoriamente uno de ellos. Este método va a ser necesario para que la clase SimmulatedAnnealing pueda iterar con nuevas soluciones.

```python
class SolutionGraph:
    def __init__(self, max_coefs: Coeficients, m: int):
        self.max_coefs = max_coefs
        self.m = m
        self.graph = self.__generate_graph__()

    def __generate_graph__(self) -> Graph:
        G = nx.Graph()
        for p in range(self.max_coefs.p):
            for q in range(self.max_coefs.q):
                for d in range(
                    max(self.max_coefs.d - 1, 0), self.max_coefs.d + 2
                ):  # Solo queremos uno por arriba y abajo (sin negativos)
                    for P in range(self.max_coefs.P):
                        for Q in range(self.max_coefs.Q):
                            for D in range(
                                max(self.max_coefs.D - 1, 0),
                                self.max_coefs.D
                                + 2,  # Solo queremos uno por arriba y abajo (sin negativos)
                            ):
                                # Cada nodo se representa como una tupla de parámetros
                                node = (p, d, q, P, D, Q, self.m)
                                G.add_node(node)
        for node in G.nodes():
            for i in range(len(node)):
                for diff in [
                    -1,
                    1,
                ]:  # va a sumar y restar uno a cada parámetro para calcular quienes son los vecinos
                    new_params = list(node)
                    new_params[i] += diff
                    neighbor = tuple(new_params)
                    if neighbor in G.nodes() and not G.has_edge(node, neighbor):
                        G.add_edge(node, neighbor)

        return G

    def get_random_node(self) -> Coeficients:
        """Devuelve un nodo aleatorio de la red"""
        random_node = random.choice(list(self.graph.nodes()))
        return Coeficients.from_tuple(random_node)

    def get_random_neighbor(self, node: Coeficients) -> Coeficients:
        """Devuelve un vecino aleatorio de la red a un nodo de entrada"""
        neighbors = list(self.graph.neighbors(node.to_tuple()))
        if neighbors:
            random_neighbor = random.choice(neighbors)
            return Coeficients.from_tuple(random_neighbor)
        else:
            # Si el nodo no tiene vecinos, se devuelve el propio nodo como resultado
            return node
```

Figura 5 – Clase Grafo que modela el espacio de soluciones.

En lo que respecta al espacio de soluciones, es importante tener en cuenta que solo existe un valor en la dimensión m, que es 12 (aunque la clase Grafo permite introducir uno distinto como argumento). Esto se debe a que este parámetro sirve para indicar cuál es el intervalo de estacionalidad en un modelo, y ya se sabe de antemano que es de 12 meses (un año). Por lo tanto, este parámetro permanece fijo y no se va a optimizar.

CLASE SARIMA

Se ha construido esta clase como abstracción de la clase SARIMAX del paquete statsmodels. En nuestra implementación se reciben los coeficientes como argumentos y una serie temporal a la que ajustarse. Una vez se ha definido un objeto SARIMAModel, pueden utilizarse una serie de métodos que calcularán distintas métricas relacionadas con el pronosticador.

- ***get_rmse**:* Este método calcula la raíz del error cuadrático medio (RMSE) entre las observaciones reales (un conjunto de prueba) y los valores pronosticados por el modelo SARIMA. El RMSE es una medida de cuán bien se ajusta el modelo a los datos de prueba.
- ***get_res_corr_penalization***: Este método calcula una penalización basada en la autocorrelación de los residuos del modelo SARIMA. Utiliza el test de Ljung-Box para evaluar la autocorrelación en diferentes retrasos (lags). Si encuentra que la autocorrelación es significativa (p-value < threshold) en al menos uno de los lags, aplica una penalización al costo total. Se usa un threshold de 0.05 por defecto.
- ***get_test_cost**:* Este método combina el RMSE y la penalización de correlación de residuos para calcular un costo total para el modelo en función de su desempeño en el conjunto de prueba. El costo se utiliza para evaluar y comparar diferentes modelos SARIMA. Un costo más bajo indica un mejor ajuste del modelo a los datos de prueba.

```python
class SARIMAModel:
    def __init__(self, series: pd.Series, coeficients: Coeficients):
        self.series = series
        self.coeficients = coeficients

    def fit_model(self):
        model = SARIMAX(
            self.series,
            order=(self.coeficients.p, self.coeficients.d, self.coeficients.q),
            seasonal_order=(
                self.coeficients.P,
                self.coeficients.D,
                self.coeficients.Q,
                self.coeficients.m,
            ),
        )
        return model.fit()

    def __get_rmse__(self, test_series: pd.Series, forecasted_values) -> float:
        rmse = np.sqrt(mean_squared_error(test_series, forecasted_values))
        return float(rmse)

    def __get_res_corr_penalization__(
        self,
        model_results,
        lags: int,
        penalization: float = 10,
        threshold: float = 0.05,
    ) -> float:
        test_result = acorr_ljungbox(model_results.resid, lags)  # type: ignore
        # El valor [1] del resultado del test nos indica la significancia estadística de la autocorrelación en cada retraso
        p_values = test_result["lb_pvalue"]
        return penalization if any(p_value < threshold for p_value in p_values) else 0

    def get_test_cost(self, test_series: pd.Series):
        """
        El coste es la suma del rmse sobre el conjunto de test y una penalizacion
        para las soluciones donde los residuos esteén correlacionados
        """
        model_results = self.fit_model()
        forecast = model_results.get_forecast(steps=len(test_series))  # type: ignore
        forecasted_values = forecast.predicted_mean

        rmse = self.__get_rmse__(test_series, forecasted_values)
        cost = self.__get_res_corr_penalization__(model_results, 6, 10, 0.05)
        return rmse + cost
```

Figura 6 – Clase SARIMAModel, abstracción de la clase SARIMAX de statsmodels.

Cabe destacar que SARIMAX permite incluir variables exógenas en el modelo, de ahí la “X”, sin embargo, no se hará uso de esta posibilidad, al no existir estas variables en nuestro problema. De este modo, aunque el nombre de la clase de statsmodels sea SARIMAX, aquí se emplea como si fuera SARIMA.

DIFICULTAD DE IMPLEMENTACIÓN DE UN MECANISMO DE ROLLING FORESCAST

El "Rolling Forecast" es una técnica que implica reentrenar el modelo de series temporales con cada nueva predicción y luego usar el modelo actualizado para predecir el siguiente valor en la secuencia temporal. Esto se repite a lo largo de todo el conjunto de prueba. El beneficio de esta técnica es que el modelo puede "aprender" de los nuevos datos a medida que están disponibles y, en teoría, mejorar su precisión a lo largo del tiempo.

Sin embargo, esta técnica puede ser computacionalmente costosa, ya que implica repetidos reentrenamientos del modelo. Esto puede no ser práctico en situaciones donde los recursos computacionales son limitados.

En lugar de utilizar el RMSE con Rolling Forecast, se ha optado por utilizar una métrica sustituta, que es el RMSE sin Rolling Forecast. Esto significa que se evalúa el rendimiento del modelo utilizando el RMSE convencional, que se calcula sobre todo el conjunto de prueba sin reentrenar el modelo en cada iteración. Se asume que si un modelo tiene un RMSE más bajo en esta métrica, también tendrá un mejor rendimiento en la métrica Rolling Forecast si fuera posible el permitirse usarla.

CLASE SIMULATED ANNEALING

Esta es la clase relacionada con la lógica de la optimización, que se ha desarrollado para ser agnóstica completamente al problema a optimizar.

```python
class SimulatedAnnealing:
    def __init__(self, initial_solution, temperature, cooling_rate, max_iterations):
        self.current_solution = initial_solution
        self.best_solution = initial_solution
        self.temperature = temperature
        self.cooling_rate = cooling_rate
        self.max_iterations = max_iterations
        self.solution_history: List[SolutionHistoryEntry] = []

    def __get_acceptance_probability__(self, old_cost, new_cost, temperature):
        if new_cost < old_cost:
            return 1.0
        return math.exp((old_cost - new_cost) / temperature)

    def __update_history__(self, new_entry: SolutionHistoryEntry):
        self.solution_history.append(new_entry)

    def __get_cost_from_cache__(self, solution):
        for entry in self.solution_history:
            if entry.solution == solution:
                return entry.cost
```

Figura 7 – Clase SimulatedAnnealing, junto con sus métodos privados.

```python
def iteration(self, cost_function):
    # Generar un nuevo vecino solución
    neighbor_solution = self.current_solution.generate_neighbor()
    # Calcular el coste de la solución actual y la del vecino, intentar siempre mirar el cache
    current_cost = self.__get_cost_from_cache__(
        self.current_solution
    ) or cost_function(self.current_solution)

    neighbor_cost = self.__get_cost_from_cache__(
        neighbor_solution
    ) or cost_function(neighbor_solution)

    # Si la solución del vecino es mejor o se acepta con cierta probabilidad, actualizar la solución actual
    if (
        neighbor_cost < current_cost
        or random.random()
        < self.__get_acceptance_probability__(
            current_cost, neighbor_cost, self.temperature
        )
    ):
        self.current_solution = neighbor_solution

    # Actualizar la mejor solución con la actual si la actual es la mejor
    self.__update_history__(SolutionHistoryEntry(neighbor_solution, neighbor_cost))
    if neighbor_cost < cost_function(self.best_solution):
        self.best_solution = neighbor_solution
    # Reducir temperatura
    self.temperature *= self.cooling_rate

def optimize(self, cost_function):
    iteration = 0
    while iteration < self.max_iterations:
        self.iteration(cost_function)
        iteration += 1

    return self.best_solution

def get_history(self) -> List[SolutionHistoryEntry]:
    return self.solution_history
```

Figura 8 – Métodos públicos de la clase SimulatedAnnealing.

- ***get_acceptance_probability***: Este método calcula la probabilidad de aceptar una nueva solución basada en la diferencia entre el costo de la solución actual y la nueva solución, así como la temperatura actual. Si la nueva solución es mejor (costo más bajo), la probabilidad de aceptación es 1 (100%). Si la nueva solución es peor, la probabilidad se calcula usando una función exponencial que depende de la diferencia de costos y la temperatura actual.
- ***update_history***: Este método agrega una nueva entrada de historial a la lista solution_history. La entrada de historial contiene información sobre la solución y su costo en una iteración específica.
- ***get_cost_from_cache***: Este método busca en la caché (historial de soluciones) para ver si ya se ha calculado el costo de una solución determinada en iteraciones anteriores. Si la solución se encuentra en la caché, devuelve el costo almacenado; de lo contrario, devuelve None. La utilidad de este cache reside en que acelera mucho la ejecución del programa, ya que algunos nodos pueden ser revisitados para viajar a regiones distintas del espacio de soluciones, y gracias al cache, no hay que recalcular su costo.
- ***iteration***: Este método representa una iteración del algoritmo de recocido simulado. En cada iteración, se genera una solución vecina, se calcula el costo actual y el costo de la solución vecina, y se decide si se acepta o no la solución vecina basándose en la probabilidad de aceptación. Luego, se actualiza la solución actual, la mejor solución y el historial.
- ***optimize**:* Este método ejecuta el algoritmo de recocido simulado durante un número máximo de iteraciones (max_iterations) con el objetivo de encontrar la mejor solución posible dadas las restricciones y la función de costo (cost_function). Devuelve la mejor solución encontrada.

- ***get_history***: Este método devuelve una lista de todas las soluciones y sus costos registrados en el historial durante la ejecución del algoritmo. Esto permite realizar un seguimiento de cómo evolucionaron las soluciones a lo largo del tiempo. Muy útil en el momento del debugging.

PROGRAMA PRINCIPAL

El programa principal utiliza las clases hasta ahora descritas para resolver el problema planteado.

INICIALIZACIÓN DEL PROGRAMA

Primero, se definen varias constantes y parámetros clave que influirán en el proceso de optimización, basándose en la información del enunciado. El valor de COOLING_RATE se ha seleccionado después de un proceso iterativo para encontrar un parámetro que permita una exploración de soluciones más agresiva al principio, y que se vaya volviendo más conservador a la velocidad adecuada.

Después de cargar y dividir los datos en conjuntos de entrenamiento y prueba, se realizan pruebas de estacionalidad en los datos de entrenamiento utilizando la clase CoeficientsTest. Estas pruebas determinan los valores de d y D. También se establece un límite para los coeficientes del modelo SARIMA utilizando la clase Coeficients. El siguiente paso es la construcción de un grafo de soluciones posibles utilizando la clase SolutionGraph.

SELECCIÓN DE LA SOLUCIÓN INICIAL

Para declarar una solución inicial, se define un objeto de la clase Solution. El valor inicial que se utiliza en este código se ha calculado externamente en un notebook haciendo uso de la centralidad.

Para ello se importa la función *closeness_centrality* del paquete networkx. El resultado de *closeness_centrality* es un diccionario donde cada clave es un nodo del grafo y el valor correspondiente es su medida de centralidad de cercanía. Se puede usar esta medida para identificar los nodos que están más cerca de todos los demás nodos en el grafo, es decir, nodos que tienen un acceso fácil al resto de nodos.

Esta solución inicial actúa como punto de partida para el algoritmo de recocido simulado, y es óptima ya que su proximidad al resto de nodos ayudará a que las soluciones no se queden estancadas en los bordes del espacio de soluciones.

Cabe destacar, que se han impuesto algunos límites a los valores que la solución inicial puede tomar, ya que se desea encontrar soluciones iniciales con valores pequeños y que cumplan con los valores esperados de “d” y de “D”.

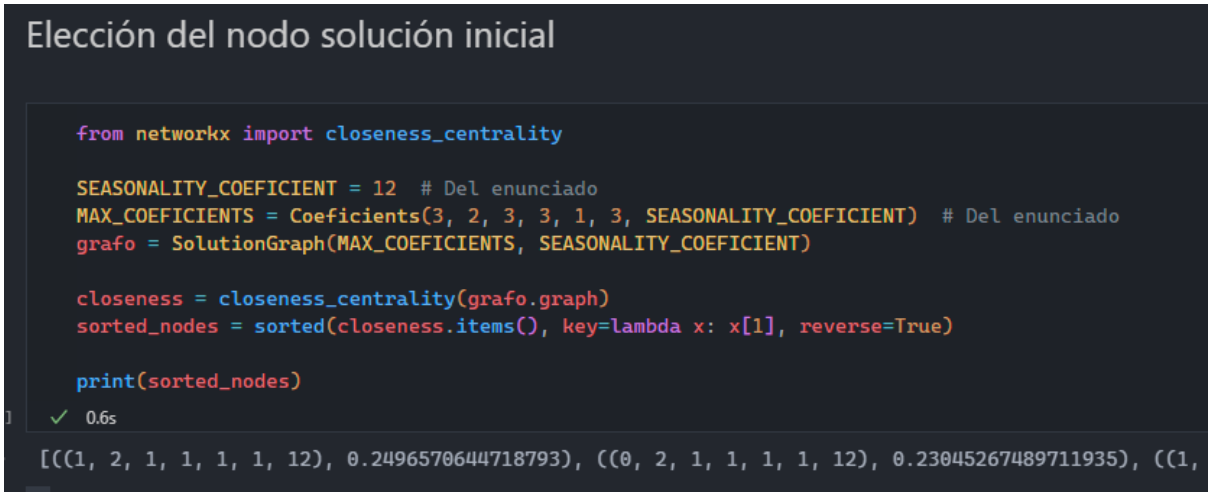


Figura 9 – Celda en Jupyter Notebook para encontrar soluciones iniciales óptimas.

Así pues, se escoge la solución (1 ,2, 1, 1, 1, 1, 12) como inicial.

EJECUCIÓN DEL RECOCIDO SIMULADO

Continuando con la ejecución del programa principal, se define una función *cost_function* que evalúa el costo de una solución dada. Como se ha dicho antes, el costo se calcula como la raíz del error cuadrático medio (RMSE) entre las predicciones del modelo SARIMA y los datos de prueba, con una penalización adicional si los residuos del modelo están correlacionados.

La optimización se realiza ejecutando el algoritmo de recocido simulado con nuestra implementación en la clase SimmulatedAnnealing y con la función de costo *cost_function*. Durante cada iteración, se genera una solución vecina y se evalúa su costo. Si la solución vecina es mejor o se acepta con cierta probabilidad, se actualiza la solución actual. También se mantiene un historial de soluciones y costos a lo largo de las iteraciones.

Finalmente, el script imprime la mejor solución encontrada por el algoritmo de recocido simulado y muestra el historial completo de soluciones y costos a lo largo del proceso de optimización.

```python
DATA_LOCATION = "./Caso_Practico/Electric_Production.csv"
INITIAL_TEMPERATURE = 10  # Del enunciado 10
COOLING_RATE = 0.95
MAX_ITERATIONS = 30  # Del enunciado 30
SEASONALITY_COEFICIENT = 12  # Del enunciado 12
LIMIT_COEFICIENT = 8
N_TEST_CASES = 50  # Del enunciado 50

if __name__ == "__main__":
    df = data_loader(DATA_LOCATION)
    df_train, df_test = separate_train_test(df, N_TEST_CASES)

    d, D = CoeficientsTest(df_train.IPG2211A2N).run_all_tests(SEASONALITY_COEFICIENT)

    limit_coeficients = Coeficients(
        LIMIT_COEFICIENT,
        d,
        LIMIT_COEFICIENT,
        LIMIT_COEFICIENT,
        D,
        LIMIT_COEFICIENT,
        SEASONALITY_COEFICIENT,
    )

    graph = SolutionGraph(limit_coeficients, SEASONALITY_COEFICIENT)

    initial_solution = Solution(
        Coeficients(1, 2, 1, 1, 1, 1, SEASONALITY_COEFICIENT), graph
    )

    sa = SimulatedAnnealing(
        initial_solution, INITIAL_TEMPERATURE, COOLING_RATE, MAX_ITERATIONS
    )

    def cost_function(solution: Solution) -> float:
        print(solution)
        current_model = SARIMAModel(df_train.IPG2211A2N, solution)
        return current_model.get_test_cost(df_test.IPG2211A2N)

    best_solution = sa.optimize(cost_function)
    print("Best solution found:", best_solution)
    print_solution_history(sa.get_history())
```

Figura 10 - Programa principal de la aplicación.

PRESENTACIÓN DE RESULTADOS

Tras la ejecución del programa se ha obtenido como solución óptima del método de recocido simulado la compuesta por los siguientes parámetros:

p	d	q	P	D	Q	m
1	1	1	0	1	4	12

RMSE = 3.418

Estos parámetros son aquellos que han obtenido el RMSE más alto en el set de entrenamiento, tras realizar la ejecución del recocido en múltiples ocasiones. Se ha demostrado a través de la ejecución del test de Ljung–Box, y su posterior aplicación de una penalización, que esta solución no está afectada por una correlación de sus residuos. Como la penalización impuesta es de 10, y el coste de esta solución es de 3.418, se considera que se superó el test.

Es interesante comprobar como el valor de “D” que el algoritmo de optimización ha encontrado como óptimo no es 0, como indicaba el test de estacionalidad, sino 1. A decir verdad, si se observa la gráfica de la serie

temporal parece evidente que todos los años hay un máximo en enero y otro a mitad de año, con dos mínimos en medio, por lo que no era de extrañar que un valor de “D” distinto de cero fuera mejor solución en la práctica.

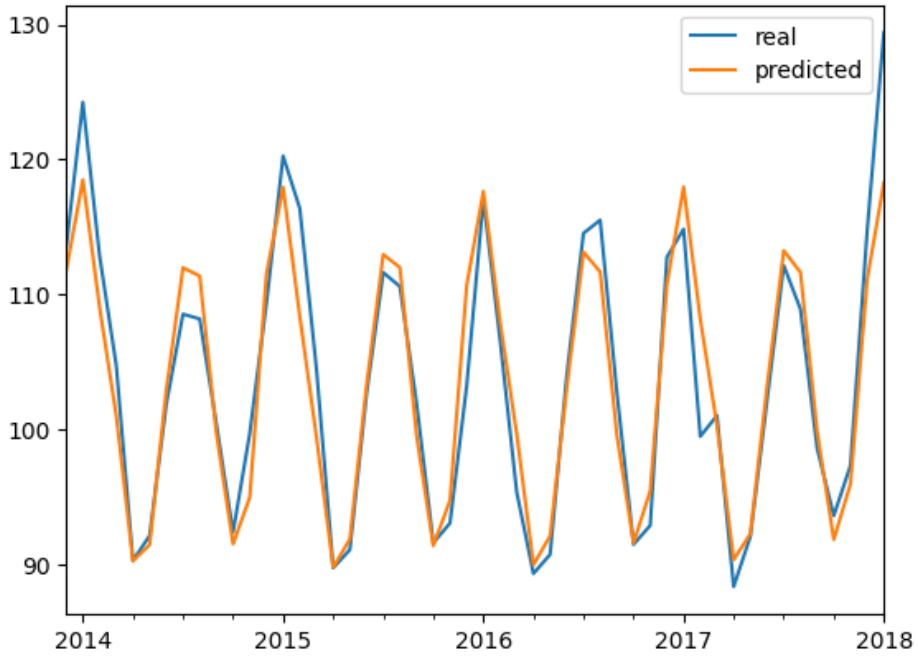


Figura 11 - Resultados del modelo prediciendo los últimos 50 valores de la serie.