Documentación de Modelos ARIMA y SARIMA para Análisis de Series Temporales

Diseño de Sistemas Embebidos

Universidad Autónoma de Tamaulipas

Facultad de Ingeniería

Equipo:

Juan Julián Paniagua Rico - a2213332303
Isaac Sayeg Posadas Perez - a2213332197
Jorge Roberto García Azzua - a2221335006

19 de mayo de 2025

Capítulo 1

Introducción

1.1. Descripción General

Este documento proporciona una documentación detallada de los modelos ARIMA (AutoRegressive Integrated Moving Average) y SARIMA (Seasonal AutoRegressive Integrated Moving Average) implementados para el análisis y predicción de series temporales. Estos modelos son ampliamente utilizados en estadística y aprendizaje automático para modelar y predecir datos de series temporales.

Los modelos ARIMA y SARIMA son técnicas estadísticas que utilizan observaciones pasadas para predecir valores futuros en una serie temporal. ARIMA se utiliza para series temporales no estacionales, mientras que SARIMA extiende ARIMA para manejar patrones estacionales en los datos.

1.2. Características Principales

Las implementaciones de ARIMA y SARIMA ofrecen las siguientes características principales:

- Ajuste (fitting) de modelos a datos de series temporales
- Predicción de valores futuros
- Evaluación del rendimiento del modelo
- Diagnóstico visual mediante gráficos
- Búsqueda en malla para optimización de hiperparámetros
- Exportación de resultados a archivos CSV
- Descomposición de series temporales (solo en SARIMA)

Esta documentación incluye explicaciones detalladas de las clases, métodos y funcionalidades implementadas, así como ejemplos de uso y recomendaciones para la aplicación efectiva de estos modelos.

Capítulo 2

Modelo ARIMA

2.1. Descripción General

ARIMA (AutoRegressive Integrated Moving Average) es un modelo estadístico utilizado para analizar y predecir datos de series temporales. El modelo ARIMA combina tres componentes:

- **AR (AutoRegressive):** Utiliza la relación entre una observación y un número específico de observaciones retrasadas.
- **I (Integrated):** Aplica diferenciación para hacer que la serie temporal sea estacionaria.
- **MA (Moving Average):** Utiliza la dependencia entre una observación y un error residual de un modelo de media móvil aplicado a observaciones anteriores.

La implementación de ARIMA en este proyecto se realiza a través de la clase `ArimaModel`, que proporciona una interfaz completa para trabajar con modelos ARIMA.

2.2. Clase ArimaModel

La clase `ArimaModel` encapsula toda la funcionalidad necesaria para implementar y gestionar modelos ARIMA para el análisis y predicción de series temporales.

2.2.1. Constructor

```
def __init__(self, data=None, order=(1, 0, 0)):
    """
    Inicializa un modelo ARIMA con los par metros especificados.

    Args:
        data (array-like, optional): Serie temporal para entrenar el
   modelo.
        order (tuple, optional): Orden del modelo ARIMA (p,d,q).
            p: T rminos autorregresivos
```

```
            d: Diferenciaci n necesaria para hacer la serie estacionaria
            q: T rminos de media m vil
    """
    self.data = data
    self.order = order
    self.model = None
    self.fitted_model = None
    self.predictions = None
    self.residuals = None
    self.results_manager = None
```

El constructor inicializa un modelo ARIMA con los datos y el orden especificados. El orden es una tupla (p, d, q) donde:

- p: Número de términos autorregresivos
- d: Grado de diferenciación
- q: Número de términos de media móvil

2.2.2. Método fit

```
def fit(self, data=None):
    """
    Entrena el modelo ARIMA con los datos proporcionados.

    Args:
        data (array-like, optional): Serie temporal para entrenar el
    modelo.
            Si no se proporciona, se utilizan los datos del constructor.

    Returns:
        self: La instancia del modelo entrenado.
    """
    if data is not None:
        self.data = data

    if self.data is None:
        raise ValueError("No se han proporcionado datos para entrenar el
    modelo")

    self.model = ARIMA(self.data, order=self.order)
    self.fitted_model = self.model.fit()
    self.residuals = self.fitted_model.resid

    return self
```

Este método entrena el modelo ARIMA con los datos proporcionados. Si no se proporcionan datos, utiliza los datos pasados al constructor. Devuelve la instancia del modelo para permitir el encadenamiento de métodos.

2.2.3. Método predict

```
def predict(self, steps=1):
    """
    Realiza predicciones con el modelo ARIMA entrenado.

    Args:
        steps (int, optional): N mero de pasos a predecir. Por defecto es
    1.

    Returns:
        array: Predicciones del modelo.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de realizar
   predicciones")

    # Realizar predicci n
    self.predictions = self.fitted_model.forecast(steps=steps)
    return self.predictions
```

Este método realiza predicciones con el modelo ARIMA entrenado. El parámetro `steps` especifica el número de pasos futuros a predecir.

2.2.4. Método evaluate

```
def evaluate(self):
    """
    Eval a el rendimiento del modelo calculando m tricas comunes.

    Returns:
        dict: Diccionario con las m tricas de evaluaci n.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de evaluarlo"
   )

    # Calcular predicciones en muestra
    in_sample_predictions = self.fitted_model.fittedvalues

    # Calcular errores
    residuals = self.data - in_sample_predictions
    mse = np.mean(residuals ** 2)
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(residuals))

    # Crear diccionario de resultados
    metrics = {
        'AIC': self.fitted_model.aic,
        'BIC': self.fitted_model.bic,
        'MSE': mse,
        'RMSE': rmse,
        'MAE': mae
```

```
    }

    return metrics
```

Este método evalúa el rendimiento del modelo calculando métricas comunes como AIC (Criterio de Información de Akaike), BIC (Criterio de Información Bayesiano), MSE (Error Cuadrático Medio), RMSE (Raíz del Error Cuadrático Medio) y MAE (Error Absoluto Medio).

2.2.5. Método plot_diagnostics

```
def plot_diagnostics(self, figsize=(12, 8)):
    """
    Genera gr ficos de diagn stico para el modelo ARIMA.

    Args:
        figsize (tuple, optional): Tama o de la figura.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de generar diagn sticos")

    fig, axes = plt.subplots(2, 2, figsize=figsize)

    # Gr fico de la serie original y las predicciones
    axes[0, 0].plot(self.data, label='Observado')
    axes[0, 0].plot(self.fitted_model.fittedvalues, color='red', label='Predicciones')
    axes[0, 0].set_title('Valores observados vs predicciones')
    axes[0, 0].legend()

    # Gr fico de residuos
    axes[0, 1].plot(self.residuals)
    axes[0, 1].set_title('Residuos')
    axes[0, 1].axhline(y=0, color='red', linestyle='--')

    # ACF de residuos
    plot_acf(self.residuals, ax=axes[1, 0], lags=20)
    axes[1, 0].set_title('ACF de residuos')

    # PACF de residuos
    plot_pacf(self.residuals, ax=axes[1, 1], lags=20)
    axes[1, 1].set_title('PACF de residuos')

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

Este método genera gráficos de diagnóstico para el modelo ARIMA, incluyendo:

- Valores observados vs predicciones
- Residuos

- Función de Autocorrelación (ACF) de residuos
- Función de Autocorrelación Parcial (PACF) de residuos

2.2.6. Método plot_forecast

```
def plot_forecast(self, steps=10, alpha=0.05, figsize=(10, 6)):
    """
    Visualiza las predicciones futuras con intervalos de confianza.

    Args:
        steps (int, optional): N mero de pasos a predecir. Por defecto es 10.
        alpha (float, optional): Nivel de significancia para los intervalos de confianza.
        figsize (tuple, optional): Tama o de la figura.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de visualizar predicciones")

    # Realizar predicci n
    forecast_result = self.fitted_model.get_forecast(steps=steps)
    forecast_index = np.arange(len(self.data), len(self.data) + steps)

    # Obtener predicciones e intervalos de confianza
    forecast_mean = forecast_result.predicted_mean
    conf_int = forecast_result.conf_int(alpha=alpha)

    # Crear figura
    plt.figure(figsize=figsize)

    # Graficar datos hist ricos
    plt.plot(np.arange(len(self.data)), self.data, label='Observado')

    # Graficar predicciones
    plt.plot(forecast_index, forecast_mean, color='red', label='Predicci n')

    # Graficar intervalos de confianza
    plt.fill_between(forecast_index,
                     conf_int.iloc[:, 0],
                     conf_int.iloc[:, 1],
                     color='pink', alpha=0.3, label=f'Intervalo de confianza {(1 - alpha) * 100}%')

    plt.title('Pron stico ARIMA')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Este método visualiza las predicciones futuras con intervalos de confianza. Muestra los datos históricos, las predicciones y los intervalos de confianza en un gráfico.

2.2.7. Método grid_search

```
def grid_search(self, data=None, p_range=(0, 2), d_range=(0, 2), q_range=(0, 2)):
    """
    Realiza una b squeda en malla para encontrar los mejores hiperpar metros (p,d,q).

    Args:
        data (array-like, optional): Serie temporal. Si no se proporciona, se usan los datos del constructor.
        p_range (tuple, optional): Rango de valores para p.
        d_range (tuple, optional): Rango de valores para d.
        q_range (tuple, optional): Rango de valores para q.

    Returns:
        tuple: La mejor configuraci n (p,d,q) encontrada.
    """
    if data is not None:
        self.data = data

    if self.data is None:
        raise ValueError("No se han proporcionado datos para la b squeda en malla")

    # Inicializar ResultsManager para guardar resultados
    results = []
    self.results_manager = ResultsManager(results)

    best_aic = float('inf')
    best_order = None

    # Iterar sobre todas las combinaciones
    for p in range(p_range[0], p_range[1] + 1):
        for d in range(d_range[0], d_range[1] + 1):
            for q in range(q_range[0], q_range[1] + 1):
                try:
                    # Crear y ajustar modelo
                    model = ARIMA(self.data, order=(p, d, q))
                    model_fit = model.fit()

                    # Guardar resultados
                    result = ResultManager(
                        va=model_fit.aic,
                        vo=best_aic,
                        iteracion=p * 100 + d * 10 + q,
                        modelo=f"ARIMA({p},{d},{q})"
                    )
                    self.results_manager.guardar_dato(result)

                    # Actualizar el mejor modelo encontrado
                    if model_fit.aic < best_aic:
                        best_aic = model_fit.aic
                        best_order = (p, d, q)
```

```
                    print(f"ARIMA({p},{d},{q}) - AIC: {model_fit.aic}")

                except Exception as e:
                    print(f"Error en ARIMA({p},{d},{q}): {str(e)}")

    # Actualizar los par metros del modelo con la mejor configuraci n
    self.order = best_order

    # Entrenar el modelo con los mejores par metros
    self.fit()

    return best_order
```

Este método realiza una búsqueda en malla para encontrar los mejores hiperparámetros (p, d, q) para el modelo ARIMA. Prueba diferentes combinaciones de parámetros y selecciona la que produce el menor valor de AIC.

2.2.8. Método to_csv

```
def to_csv(self, filepath):
    """
    Guarda los resultados de la b squeda en malla en un archivo CSV.

    Args:
        filepath (str): Ruta donde guardar el archivo CSV.
    """
    if self.results_manager:
        self.results_manager.to_csv(filepath)
    else:
        print("No hay resultados para guardar")
```

Este método guarda los resultados de la búsqueda en malla en un archivo CSV.

2.2.9. Método summary

```
def summary(self):
    """
    Devuelve un resumen del modelo ARIMA.

    Returns:
        str: Resumen del modelo.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de generar el
    resumen")

    return self.fitted_model.summary()
```

Este método devuelve un resumen del modelo ARIMA, incluyendo estadísticas y parámetros estimados.

Capítulo 3

Modelo SARIMA

3.1. Descripción General

SARIMA (Seasonal AutoRegressive Integrated Moving Average) es una extensión del modelo ARIMA que incorpora componentes estacionales. Además de los componentes AR, I y MA, SARIMA incluye sus equivalentes estacionales:

- **SAR (Seasonal AutoRegressive):** Componente autorregresivo estacional.
- **SI (Seasonal Integrated):** Diferenciación estacional.
- **SMA (Seasonal Moving Average):** Componente de media móvil estacional.

La implementación de SARIMA en este proyecto se realiza a través de la clase `SarimaModel`, que proporciona una interfaz completa para trabajar con modelos SARIMA.

3.2. Clase SarimaModel

La clase `SarimaModel` encapsula toda la funcionalidad necesaria para implementar y gestionar modelos SARIMA para el análisis y predicción de series temporales con patrones estacionales.

3.2.1. Constructor

```
def __init__(self, data=None, order=(1, 0, 0), seasonal_order=(0, 0, 0, 0)
    ):
     """
     Inicializa un modelo SARIMA con los par metros especificados.

     Args:
         data (array-like, optional): Serie temporal para entrenar el
    modelo.
         order (tuple, optional): Orden del modelo ARIMA (p,d,q).
             p: T rminos autorregresivos
```

```
            d: Diferenciaci n necesaria para hacer la serie estacionaria
            q: T rminos de media m vil
        seasonal_order (tuple, optional): Orden estacional (P,D,Q,s).
            P: T rminos autorregresivos estacionales
            D: Diferenciaci n estacional
            Q: T rminos de media m vil estacionales
            s: Per odo estacional
    """
    self.data = data
    self.order = order
    self.seasonal_order = seasonal_order
    self.model = None
    self.fitted_model = None
    self.predictions = None
    self.residuals = None
    self.results_manager = None
```

El constructor inicializa un modelo SARIMA con los datos, el orden y el orden estacional especificados. El orden es una tupla (p, d, q) y el orden estacional es una tupla (P, D, Q, s) donde:

- P: Número de términos autorregresivos estacionales
- D: Grado de diferenciación estacional
- Q: Número de términos de media móvil estacional
- s: Período estacional

3.2.2. Método fit

```
def fit(self, data=None, exog=None):
    """
    Entrena el modelo SARIMA con los datos proporcionados.

    Args:
        data (array-like, optional): Serie temporal para entrenar el
    modelo.
            Si no se proporciona, se utilizan los datos del constructor.
        exog (array-like, optional): Variables ex genas para la
    regresi n.

    Returns:
        self: La instancia del modelo entrenado.
    """
    if data is not None:
        self.data = data

    if self.data is None:
        raise ValueError("No se han proporcionado datos para entrenar el
    modelo")

    self.model = SARIMAX(
```

```
        self.data,
        exog=exog,
        order=self.order,
        seasonal_order=self.seasonal_order,
        enforce_stationarity=False,
        enforce_invertibility=False
    )

    self.fitted_model = self.model.fit(disp=False)
    self.residuals = self.fitted_model.resid

    return self
```

Este método entrena el modelo SARIMA con los datos proporcionados. Si no se proporcionan datos, utiliza los datos pasados al constructor. También permite incluir variables exógenas para la regresión. Devuelve la instancia del modelo para permitir el encadenamiento de métodos.

3.2.3. Método predict

```
def predict(self, steps=1, exog=None):
    """
    Realiza predicciones con el modelo SARIMA entrenado.

    Args:
        steps (int, optional): N mero de pasos a predecir. Por defecto es 1.
        exog (array-like, optional): Variables ex genas para la predicci n.

    Returns:
        array: Predicciones del modelo.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de realizar predicciones")

    # Realizar predicci n
    self.predictions = self.fitted_model.forecast(steps=steps, exog=exog)
    return self.predictions
```

Este método realiza predicciones con el modelo SARIMA entrenado. El parámetro `steps` especifica el número de pasos futuros a predecir. También permite incluir variables exógenas para la predicción.

3.2.4. Método evaluate

```
def evaluate(self):
    """
    Eval a el rendimiento del modelo calculando m tricas comunes.

```

```
    Returns:
        dict: Diccionario con las m tricas de evaluaci n.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de evaluarlo")

    # Calcular predicciones en muestra
    in_sample_predictions = self.fitted_model.fittedvalues

    # Calcular errores
    residuals = self.data - in_sample_predictions
    mse = np.mean(residuals ** 2)
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(residuals))

    # Crear diccionario de resultados
    metrics = {
        'AIC': self.fitted_model.aic,
        'BIC': self.fitted_model.bic,
        'MSE': mse,
        'RMSE': rmse,
        'MAE': mae
    }

    return metrics
```

Este método evalúa el rendimiento del modelo calculando métricas comunes como AIC, BIC, MSE, RMSE y MAE.

3.2.5. Método plot_diagnostics

```
def plot_diagnostics(self, figsize=(16, 12)):
    """
    Genera gr ficos de diagn stico para el modelo SARIMA.

    Args:
        figsize (tuple, optional): Tama o de la figura.
    """
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de generar diagn sticos")

    fig, axes = plt.subplots(3, 2, figsize=figsize)

    # Gr fico de la serie original y las predicciones
    axes[0, 0].plot(self.data, label='Observado')
    axes[0, 0].plot(self.fitted_model.fittedvalues, color='red', label='Predicciones')
    axes[0, 0].set_title('Valores observados vs predicciones')
    axes[0, 0].legend()

    # Gr fico de residuos
```

```
    axes[0, 1].plot(self.residuals)
    axes[0, 1].set_title('Residuos')
    axes[0, 1].axhline(y=0, color='red', linestyle='--')

    # ACF de residuos
    plot_acf(self.residuals, ax=axes[1, 0], lags=40)
    axes[1, 0].set_title('ACF de residuos')

    # PACF de residuos
    plot_pacf(self.residuals, ax=axes[1, 1], lags=40)
    axes[1, 1].set_title('PACF de residuos')

    # Histograma de residuos
    axes[2, 0].hist(self.residuals, bins=25)
    axes[2, 0].set_title('Histograma de residuos')

    # QQ plot de residuos
    from scipy import stats
    stats.probplot(self.residuals, dist="norm", plot=axes[2, 1])
    axes[2, 1].set_title('QQ plot de residuos')

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

Este método genera gráficos de diagnóstico para el modelo SARIMA, incluyendo:

- Valores observados vs predicciones
- Residuos
- Función de Autocorrelación (ACF) de residuos
- Función de Autocorrelación Parcial (PACF) de residuos
- Histograma de residuos
- QQ plot de residuos

3.2.6. Método plot_forecast

```
def plot_forecast(self, steps=10, alpha=0.05, figsize=(12, 6), exog=None):
    """
    Visualiza las predicciones futuras con intervalos de confianza.

    Args:
        steps (int, optional): N mero de pasos a predecir. Por defecto es
    10.
        alpha (float, optional): Nivel de significancia para los
   intervalos de confianza.
        figsize (tuple, optional): Tama o de la figura.
        exog (array-like, optional): Variables ex genas para la
   predicci n.
    """
```

```
    if self.fitted_model is None:
        raise ValueError("El modelo debe ser entrenado antes de visualizar predicciones")

    # Realizar predicci n
    forecast_result = self.fitted_model.get_forecast(steps=steps, exog=exog)
    forecast_index = np.arange(len(self.data), len(self.data) + steps)

    # Obtener predicciones e intervalos de confianza
    forecast_mean = forecast_result.predicted_mean
    conf_int = forecast_result.conf_int(alpha=alpha)

    # Crear figura
    plt.figure(figsize=figsize)

    # Graficar datos hist ricos
    plt.plot(np.arange(len(self.data)), self.data, label='Observado')

    # Graficar predicciones
    plt.plot(forecast_index, forecast_mean, color='red', label='Predicci n')

    # Graficar intervalos de confianza
    plt.fill_between(forecast_index,
                     conf_int.iloc[:, 0],
                     conf_int.iloc[:, 1],
                     color='pink', alpha=0.3, label=f'Intervalo de confianza {(1 - alpha) * 100}%')

    plt.title('Pron stico SARIMA')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Este método visualiza las predicciones futuras con intervalos de confianza. Muestra los datos históricos, las predicciones y los intervalos de confianza en un gráfico. A diferencia del método equivalente en ARIMA, este método también permite incluir variables exógenas para la predicción.

3.2.7. Método grid_search

```
def grid_search(self, data=None, p_range=(0, 2), d_range=(0, 1), q_range=(0, 2),
                P_range=(0, 1), D_range=(0, 1), Q_range=(0, 1), s_values=[12]):
    """
    Realiza una b squeda en malla para encontrar los mejores hiperpar metros.

    Args:
        data (array-like, optional): Serie temporal. Si no se proporciona, se usan los datos del constructor.
```

```
        p_range (tuple): Rango de valores para p (componente AR).
        d_range (tuple): Rango de valores para d (componente I).
        q_range (tuple): Rango de valores para q (componente MA).
        P_range (tuple): Rango de valores para P (componente AR estacional).
        D_range (tuple): Rango de valores para D (componente I estacional).
        Q_range (tuple): Rango de valores para Q (componente MA estacional).
        s_values (list): Lista de valores estacionales a probar.

    Returns:
        tuple: La mejor configuraci n (p,d,q)(P,D,Q,s) encontrada.
    """
    if data is not None:
        self.data = data

    if self.data is None:
        raise ValueError("No se han proporcionado datos para la b squeda en malla")

    # Inicializar ResultsManager para guardar resultados
    results = []
    self.results_manager = ResultsManager(results)

    best_aic = float('inf')
    best_params = None

    # Contador para iteraci n
    iteration_counter = 0

    # Iterar sobre todas las combinaciones
    for p in range(p_range[0], p_range[1] + 1):
        for d in range(d_range[0], d_range[1] + 1):
            for q in range(q_range[0], q_range[1] + 1):
                for P in range(P_range[0], P_range[1] + 1):
                    for D in range(D_range[0], D_range[1] + 1):
                        for Q in range(Q_range[0], Q_range[1] + 1):
                            for s in s_values:
                                iteration_counter += 1

                                # Validar combinaci n
                                if p == 0 and q == 0 and P == 0 and Q == 0:
                                    continue

                                # Mostrar progreso
                                print(f"Evaluando SARIMA({p},{d},{q})({P},{D},{Q},{s})")

                                try:
                                    # Crear y ajustar modelo
                                    model = SARIMAX(
                                        self.data,
```

```
                                        order=(p, d, q),
                                        seasonal_order=(P, D, Q, s),
                                        enforce_stationarity=False,
                                        enforce_invertibility=False
                                    )
                                    model_fit = model.fit(disp=False)

                                    # Guardar resultados
                                    result = ResultManager(
                                        va=model_fit.aic,
                                        vo=best_aic,
                                        iteracion=iteration_counter,
                                        modelo=f"SARIMA({p},{d},{q})({P},{
    D},{Q},{s})"
                                    )
                                    self.results_manager.guardar_dato(
    result)

                                    # Actualizar el mejor modelo
    encontrado
                                    if model_fit.aic < best_aic:
                                        best_aic = model_fit.aic
                                        best_params = (p, d, q, P, D, Q, s
    )

                                    print(f"SARIMA({p},{d},{q})({P},{D},{Q
    },{s}) - AIC: {model_fit.aic}")

                                except Exception as e:
                                    print(f"Error en SARIMA({p},{d},{q})({
    P},{D},{Q},{s}): {str(e)}")

    # Actualizar los par metros del modelo con la mejor configuraci n
    if best_params:
        p, d, q, P, D, Q, s = best_params
        self.order = (p, d, q)
        self.seasonal_order = (P, D, Q, s)

        # Entrenar el modelo con los mejores par metros
        self.fit()

    return best_params
```

Este método realiza una búsqueda en malla para encontrar los mejores hiperparámetros para el modelo SARIMA. A diferencia del método equivalente en ARIMA, este método también busca los mejores valores para los componentes estacionales (P, D, Q, s). Prueba diferentes combinaciones de parámetros y selecciona la que produce el menor valor de AIC.

3.2.8. Método to_csv

```
def to_csv(self, filepath):
    """
    Guarda los resultados de la b squeda en malla en un archivo CSV.
```

```
4
5       Args:
6           filepath (str): Ruta donde guardar el archivo CSV.
7       """
8       if self.results_manager:
9           self.results_manager.to_csv(filepath)
10      else:
11          print("No hay resultados para guardar")
```

Este método guarda los resultados de la búsqueda en malla en un archivo CSV.

3.2.9. Método summary

```
1   def summary(self):
2       """
3       Devuelve un resumen del modelo SARIMA.
4
5       Returns:
6           str: Resumen del modelo.
7       """
8       if self.fitted_model is None:
9           raise ValueError("El modelo debe ser entrenado antes de generar el
        resumen")
10
11      return self.fitted_model.summary()
```

Este método devuelve un resumen del modelo SARIMA, incluyendo estadísticas y parámetros estimados.

3.2.10. Método decompose

```
1   def decompose(self, type='additive'):
2       """
3       Descompone la serie temporal en sus componentes de tendencia,
4       estacionalidad y residuos.
5
6       Args:
7           type (str): Tipo de descomposici n ('additive' o 'multiplicative
        ').
8
9       Returns:
10          object: Resultado de la descomposici n.
11      """
12      from statsmodels.tsa.seasonal import seasonal_decompose
13
14      if self.data is None:
15          raise ValueError("No hay datos para descomponer")
16
17      # Crear un  ndice  para la descomposici n
18      index = pd.date_range(start='2000-01-01', periods=len(self.data), freq
        ='D')
19      series = pd.Series(self.data, index=index)
```

```
20
21      # Descomponer la serie
22      decomposition = seasonal_decompose(series, model=type, period=self.
        seasonal_order[3])
23
24      # Graficar los resultados
25      fig, axes = plt.subplots(4, 1, figsize=(12, 10), sharex=True)
26
27      axes[0].plot(decomposition.observed)
28      axes[0].set_title('Serie Original')
29
30      axes[1].plot(decomposition.trend)
31      axes[1].set_title('Tendencia')
32
33      axes[2].plot(decomposition.seasonal)
34      axes[2].set_title('Estacionalidad')
35
36      axes[3].plot(decomposition.resid)
37      axes[3].set_title('Residuos')
38
39      plt.tight_layout()
40      plt.show()
41
42      return decomposition
```

Este método descompone la serie temporal en sus componentes de tendencia, estacionalidad y residuos. Utiliza la función `seasonal_decompose` de statsmodels para realizar la descomposición y visualiza los resultados en un gráfico. Este método es exclusivo de la clase `SarimaModel` y no está presente en la clase `ArimaModel`.

Capítulo 4

Ejemplos de Uso

4.1. Ejemplo de Uso de ARIMA

A continuación se muestra un ejemplo básico de cómo utilizar la clase `ArimaModel` para analizar y predecir una serie temporal:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from arima import ArimaModel

# Cargar datos (ejemplo con datos sint ticos)
np.random.seed(42)
data = np.cumsum(np.random.normal(0, 1, 100))  # Serie temporal sint tica

# Crear y entrenar modelo ARIMA
model = ArimaModel(data=data, order=(1, 1, 1))
model.fit()

# Evaluar el modelo
metrics = model.evaluate()
print("M tricas de evaluaci n:")
for key, value in metrics.items():
    print(f"{key}: {value}")

# Realizar predicciones
predictions = model.predict(steps=10)
print("\nPredicciones:")
print(predictions)

# Visualizar diagn sticos
model.plot_diagnostics()

# Visualizar pron stico
model.plot_forecast(steps=10)

# Buscar mejores par metros
best_order = model.grid_search(p_range=(0, 2), d_range=(0, 2), q_range=(0,
    2))
```

```
print(f"\nMejor orden encontrado: {best_order}")

# Guardar resultados
model.to_csv("arima_results.csv")
```

4.2. Ejemplo de Uso de SARIMA

A continuación se muestra un ejemplo básico de cómo utilizar la clase `SarimaModel` para analizar y predecir una serie temporal con componentes estacionales:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sarima import SarimaModel

# Cargar datos (ejemplo con datos sint ticos con estacionalidad)
np.random.seed(42)
t = np.arange(100)
trend = 0.1 * t
seasonal = 5 * np.sin(2 * np.pi * t / 12)  # Estacionalidad con per odo 12
noise = np.random.normal(0, 1, 100)
data = trend + seasonal + noise  # Serie temporal sint tica con estacionalidad

# Crear y entrenar modelo SARIMA
model = SarimaModel(data=data, order=(1, 0, 1), seasonal_order=(1, 0, 1, 12))
model.fit()

# Evaluar el modelo
metrics = model.evaluate()
print("M tricas de evaluaci n:")
for key, value in metrics.items():
    print(f"{key}: {value}")

# Realizar predicciones
predictions = model.predict(steps=24)  # Predecir 2 per odos estacionales
print("\nPredicciones:")
print(predictions)

# Visualizar diagn sticos
model.plot_diagnostics()

# Visualizar pron stico
model.plot_forecast(steps=24)

# Descomponer la serie
decomposition = model.decompose(type='additive')

# Buscar mejores par metros (b squeda limitada para el ejemplo)
best_params = model.grid_search(
```

```
    p_range=(0, 1), d_range=(0, 1), q_range=(0, 1),
    P_range=(0, 1), D_range=(0, 1), Q_range=(0, 1),
    s_values=[12]
)
print(f"\nMejores par metros encontrados: {best_params}")

# Guardar resultados
model.to_csv("sarima_results.csv")
```

Capítulo 5

Comparación entre ARIMA y SARIMA

5.1. Similitudes

Los modelos ARIMA y SARIMA comparten varias similitudes:

- Ambos son modelos estadísticos para el análisis y predicción de series temporales.
- Ambos utilizan componentes autorregresivos (AR) y de media móvil (MA).
- Ambos pueden aplicar diferenciación para hacer que la serie sea estacionaria.
- Ambos proporcionan métodos para ajuste, predicción, evaluación y diagnóstico.
- Ambos utilizan criterios como AIC y BIC para evaluar la calidad del modelo.

5.2. Diferencias

Las principales diferencias entre ARIMA y SARIMA son:

- SARIMA incluye componentes estacionales (SAR, SI, SMA) que ARIMA no tiene.
- SARIMA es más adecuado para series temporales con patrones estacionales recurrentes.
- SARIMA tiene más parámetros para ajustar (p, d, q, P, D, Q, s).
- SARIMA incluye un método adicional (`decompose`) para descomponer la serie en tendencia, estacionalidad y residuos.
- La búsqueda en malla para SARIMA es más compleja y computacionalmente intensiva debido al mayor número de parámetros.

5.3. Cuándo Usar Cada Modelo

- **Use ARIMA cuando:**
  - La serie temporal no muestra patrones estacionales claros.
  - Se necesita un modelo más simple con menos parámetros.
  - El tiempo de cómputo es una preocupación.
- **Use SARIMA cuando:**
  - La serie temporal muestra patrones estacionales claros.
  - Se necesita capturar y modelar la estacionalidad explícitamente.
  - Se requiere descomponer la serie en sus componentes.
  - La precisión en la predicción de patrones estacionales es importante.

Capítulo 6

Conclusiones

6.1. Resumen

En este documento, hemos proporcionado una documentación detallada de las implementaciones de los modelos ARIMA y SARIMA para el análisis y predicción de series temporales. Estas implementaciones ofrecen una interfaz completa y fácil de usar para trabajar con estos modelos estadísticos.

Las clases `ArimaModel` y `SarimaModel` encapsulan toda la funcionalidad necesaria para ajustar modelos, realizar predicciones, evaluar el rendimiento, visualizar diagnósticos y resultados, optimizar hiperparámetros y exportar resultados.

6.2. Aplicaciones Prácticas

Los modelos ARIMA y SARIMA tienen numerosas aplicaciones prácticas en diversos campos:

- **Finanzas:** Predicción de precios de acciones, tasas de interés y otros indicadores financieros.
- **Economía:** Pronóstico de indicadores económicos como PIB, inflación y desempleo.
- **Meteorología:** Predicción de temperaturas, precipitaciones y otros fenómenos climáticos.
- **Ventas:** Pronóstico de ventas con patrones estacionales.
- **Energía:** Predicción de consumo de energía y demanda eléctrica.
- **Transporte:** Análisis de patrones de tráfico y demanda de transporte.
- **Salud:** Predicción de incidencia de enfermedades estacionales.

6.3. Limitaciones y Consideraciones

A pesar de su utilidad, es importante tener en cuenta algunas limitaciones y consideraciones al utilizar estos modelos:

- Los modelos ARIMA y SARIMA asumen que los patrones históricos continuarán en el futuro.
- La precisión de las predicciones tiende a disminuir a medida que el horizonte de predicción aumenta.
- La selección adecuada de los parámetros (p, d, q, P, D, Q, s) es crucial para el rendimiento del modelo.
- Estos modelos pueden no ser adecuados para series temporales con cambios estructurales o no linealidades complejas.
- La búsqueda en malla para optimizar hiperparámetros puede ser computacionalmente intensiva, especialmente para SARIMA.

6.4. Trabajo Futuro

Algunas posibles mejoras y extensiones para estas implementaciones incluyen:

- Implementación de validación cruzada para series temporales.
- Integración con otros modelos de series temporales como Prophet o modelos de aprendizaje profundo.
- Optimización del rendimiento para conjuntos de datos grandes.
- Implementación de métodos para manejar valores faltantes y outliers.
- Desarrollo de interfaces gráficas para facilitar el uso por parte de usuarios no técnicos.

En resumen, las implementaciones de ARIMA y SARIMA presentadas en este documento proporcionan herramientas poderosas y flexibles para el análisis y predicción de series temporales, con aplicaciones en una amplia variedad de campos.