Here's a detailed breakdown and explanation of the plot\_ts function:

**Function: plot\_ts**

**Purpose**

This function visualizes a time series along with optional rolling mean (moving average) and confidence intervals. It is useful for identifying trends and variability in the data.

**Parameters**

* **ts (pandas.Series)**  
  The time series data to be plotted. It should have a datetime index.
* **plot\_ma (bool, default=True)**  
  Whether to plot the moving average (rolling mean) of the time series. Helps smooth out short-term fluctuations.
* **plot\_intervals (bool, default=True)**  
  Whether to plot confidence intervals around the rolling mean. The intervals are based on 1.96 times the rolling standard deviation (approx. 95% confidence).
* **window (int, default=30)**  
  The size of the rolling window used for calculating the moving average and standard deviation. A larger window smooths the data more.
* **figsize (tuple, default=(15,5))**  
  The size of the plot in inches. Controls the width and height of the figure.

**Process**

1. **Calculate Rolling Statistics**
   * The moving average (rolling\_mean) is computed using the rolling method with a window of the specified size.
   * The rolling standard deviation (rolling\_std) is also computed for the same window.
2. **Initialize the Plot**
   * A figure is created with the given size (figsize).
   * The time series is plotted in black for better visibility.
3. **Plot the Moving Average**
   * If plot\_ma is True, the rolling mean is overlaid in red to highlight trends.
4. **Plot Confidence Intervals**
   * If plot\_intervals is True, the function calculates a 95% confidence interval using:
     + rolling\_mean - 1.96 \* rolling\_std (lower bound)
     + rolling\_mean + 1.96 \* rolling\_std (upper bound)
   * The interval is shaded in light sky blue with transparency for better visibility.
5. **Final Styling**
   * A legend, grid, and title are added for readability.
   * The plot is displayed using plt.show().

**Returns**

The function does not return any value. It generates a plot as its output.

**Usage Example**

import pandas as pd

import numpy as np

# Example time series data

np.random.seed(42)

index = pd.date\_range(start="2020-01-01", periods=200, freq='D')

ts\_data = pd.Series(np.random.randn(200).cumsum(), index=index, name="Example Time Series")

# Plot with moving average and confidence intervals

plot\_ts(ts\_data, plot\_ma=True, plot\_intervals=True, window=30)

**Example Output**

* The black line represents the original time series data.
* The red line is the moving average (smoothed trend).
* The shaded blue area represents the confidence intervals.

This visualization can help identify trends and periods of high volatility in your time series data.

Here’s a detailed explanation of the fit\_trend function:

**Function: fit\_trend**

**Purpose**

This function fits a polynomial trend line to a given time series and optionally visualizes the fit. It is useful for understanding the underlying trend in the data over time.

**Parameters**

* **ts (pandas.Series)**  
  The time series data to which the trend will be fitted. It should have a sequential index.
* **degree (int, default=1)**  
  The degree of the polynomial to fit.
  + degree=1: Linear trend
  + degree=2: Quadratic trend
  + degree=3: Cubic trend  
    Higher degrees allow for more complex trends.
* **plot (bool, default=True)**  
  Whether to display a plot of the original time series with the fitted trend.
* **figsize (tuple, default=(15,5))**  
  The size of the plot in inches. Controls the width and height of the figure.

**Process**

1. **Initialize DataFrame**
   * The input time series ts is converted into a DataFrame with a single column named "ts" for easier manipulation.
2. **Fit Polynomial Trend**
   * The np.polyfit function computes the coefficients of a polynomial of the specified degree.
   * params contains these coefficients, starting with the highest power term and ending with the constant term.
3. **Construct Trend Line**
   * The constant term (params[-1]) is added to initialize the trend column.
   * Using a loop, each degree term (params[i-1] \* X\*\*i) is added to the "trend" column, where X is the index of the time series values as an array.
4. **Optional Plotting**
   * If plot=True, the function plots the original time series ("ts") in black and the fitted trend ("trend") in red.
5. **Return Values**
   * The function returns the modified DataFrame (dtf) containing the original series and the fitted trend.
   * It also returns the polynomial coefficients (params).

**Returns**

* **dtf (pandas.DataFrame)**  
  A DataFrame with the following columns:
  + "ts": The original time series.
  + "trend": The fitted trend line.
* **params (numpy.ndarray)**  
  The coefficients of the fitted polynomial, starting from the highest degree term to the constant.

**Usage Example**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Example time series

np.random.seed(42)

index = pd.date\_range(start="2020-01-01", periods=200, freq="D")

ts\_data = pd.Series(np.random.randn(200).cumsum(), index=index, name="Example Time Series")

# Fit a linear trend (degree=1)

dtf, params = fit\_trend(ts\_data, degree=1, plot=True)

# Fit a quadratic trend (degree=2)

dtf\_quadratic, params\_quadratic = fit\_trend(ts\_data, degree=2, plot=True)

**Example Output**

* For degree=1 (linear trend): A straight red line approximates the trend of the black time series.
* For degree=2 (quadratic trend): A parabolic red line adjusts to potential curvature in the data.

This function is particularly helpful for detecting and visualizing trends in noisy or fluctuating time series.

**Function: test\_stationarity\_acf\_pacf**

**Purpose**

This function evaluates the stationarity of a time series using the Augmented Dickey-Fuller (ADF) test, while also plotting the Partial Autocorrelation Function (PACF) and Autocorrelation Function (ACF) to guide model selection for ARIMA-like models.

**Parameters**

* **ts (pandas.Series)**  
  The time series to be tested and analyzed.
* **sample (float, default=0.20)**  
  The proportion (between 0 and 1) of the time series to consider as the initial sample for calculating mean and standard deviation. This is used to visualize stationarity within the plotted time series.
* **maxlag (int, default=30)**  
  The maximum number of lags to consider in the PACF and ACF plots.
* **figsize (tuple, default=(15,10))**  
  The size of the figure for the plots. Controls the width and height.

**Process**

1. **Set Up Subplots**
   * The function creates a figure with three subplots:
     + Time series plot (spans the top row).
     + PACF plot (bottom left).
     + ACF plot (bottom right).
2. **Visualize Mean and Variance**
   * A sample (first x% of the series, defined by sample) is used to calculate the mean and standard deviation of the time series.
   * The function overlays the mean (red line) and a confidence interval band (light blue, ±1.96 standard deviations) on the time series plot to assess stationarity visually.
   * It also compares the sample statistics with the rest of the series.
3. **Augmented Dickey-Fuller (ADF) Test**
   * The ADF test is applied to assess stationarity:
     + **Null hypothesis (H₀)**: The time series is non-stationary.
     + **Alternative hypothesis (H₁)**: The time series is stationary.
   * The function computes:
     + ADF test statistic (adf).
     + p-value (p).
     + Critical value at 5% significance (critical\_value).
   * The conclusion ("Stationary" or "Non-Stationary") is determined by comparing the p-value with 0.05.
4. **PACF and ACF Analysis**
   * The Partial Autocorrelation Function (PACF) identifies significant lags for the autoregressive (AR) component.
   * The Autocorrelation Function (ACF) identifies significant lags for the moving average (MA) component.
5. **Plot Titles and Layout**
   * The time series plot displays the ADF test result with the p-value.
   * PACF and ACF plots are generated using statsmodels.graphics.

**Returns**

* **None**  
  The function only generates visualizations.

**Usage Example**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import statsmodels.api as sm

import statsmodels.tsa.api as smt

# Example Time Series

np.random.seed(42)

index = pd.date\_range(start="2020-01-01", periods=300, freq="D")

ts\_data = pd.Series(np.random.randn(300).cumsum(), index=index, name="Example Series")

# Test stationarity and plot ACF/PACF

test\_stationarity\_acf\_pacf(ts=ts\_data, sample=0.20, maxlag=30, figsize=(15,10))

**Visualization Components**

1. **Time Series Plot**
   * The black line represents the time series.
   * A red dashed line shows the mean, and a light blue shaded area represents ±1.96 standard deviations for the first sample% of the series.
   * Title indicates stationarity conclusion and p-value from the ADF test.
2. **PACF Plot**
   * Identifies the significant lags for AR components (bars exceeding the confidence interval).
3. **ACF Plot**
   * Identifies the significant lags for MA components.

**Example Output**

1. **Stationary Series**
   * The p-value will be below 0.05, and mean/variance will appear constant across the time series.
   * PACF/ACF plots will show distinct spikes indicating potential AR or MA components.
2. **Non-Stationary Series**
   * The p-value will exceed 0.05, and the mean/variance will vary over time.
   * PACF/ACF plots may exhibit slower decay or patterns indicating the need for differencing.

This function is a critical step in time series analysis, providing both statistical and visual insights for model building.

**Functions: diff\_ts and undo\_diff**

These two functions perform and reverse differencing on a time series. Differencing is a common transformation in time series analysis used to stabilize the mean and remove trends, often as part of preprocessing for ARIMA or other time series models.

**1. Function: diff\_ts**

**Purpose**

* Differencing a time series to transform it into a stationary series.
* Removes trends and seasonality by calculating the differences between successive observations.

**Parameters**

* **ts (pandas.Series)**  
  The time series to be differenced.
* **lag (int, default=1)**  
  The lag value for differencing. For example:
  + Lag = 1: First differences (x\_t - x\_(t-1)).
  + Lag = 12: Seasonal differencing for monthly data with yearly seasonality.
* **order (int, default=1)**  
  The number of times to apply the differencing.
  + Order = 1: First-order differencing.
  + Order = 2: Second-order differencing (difference of differences).
* **drop\_na (bool, default=True)**  
  Whether to drop the initial NaN values created during differencing. If False, missing values are filled using backward filling (bfill).

**Returns**

* **ts (pandas.Series)**  
  The differenced time series.

**Process**

1. Loop through the specified order and apply differencing:
   * ts=ts−ts.shift(lag)ts = ts - ts.shift(\text{lag})
   * The .shift() method aligns the series by the lag, and differencing computes changes over the lagged period.
2. Handle missing values:
   * Drop NaN values if drop\_na=True.
   * Fill NaN values using backward filling if drop\_na=False.

**Usage Example**

import pandas as pd

import numpy as np

# Example Time Series

np.random.seed(42)

index = pd.date\_range(start="2020-01-01", periods=10, freq="D")

ts\_data = pd.Series(np.random.randn(10).cumsum(), index=index, name="Example Series")

# Apply differencing

diffed\_ts = diff\_ts(ts\_data, lag=1, order=1)

print(diffed\_ts)

**2. Function: undo\_diff**

**Purpose**

* Reverse the differencing process to restore the original time series.
* Useful when making forecasts using a differenced series and converting predictions back to the original scale.

**Parameters**

* **ts (numpy.array or pandas.Series)**  
  The differenced time series to be reverted.
* **first\_y (float)**  
  The starting value of the original series (required to anchor the cumulative sum).
* **lag (int, default=1)**  
  The lag value used during the differencing process.
* **order (int, default=1)**  
  The number of times differencing was applied. The undo process loops through the same order.

**Returns**

* **ts (numpy.array)**  
  The reconstructed time series after undoing differencing.

**Process**

1. **Loop Through Orders**  
   For each differencing order:
   * Append the lagged cumulative sum to the series.
   * The result is cumulatively summed to rebuild the series at each step.
2. **Starting Value (first\_y)**  
   The original starting value is required to reconstruct the series accurately.

**Issues in Current Implementation**

* **Inaccuracy in Code**: The placeholder (24168.04468 - 18256.02366) and reference to a.cumsum() are invalid and need correction.
* The process for appending lagged values is incorrectly structured. Below is the fixed implementation.

**Fixed Implementation**

def undo\_diff(ts, first\_y, lag=1, order=1):

ts = ts.copy() # To avoid modifying the input

for \_ in range(order):

ts = np.r\_[first\_y, ts].cumsum() # Prepend the first value and compute cumulative sum

return ts

**Usage Example**

# Undo differencing

first\_value = ts\_data.iloc[0] # Original first value

restored\_ts = undo\_diff(diffed\_ts.values, first\_value, lag=1, order=1)

print(restored\_ts)

**Key Points**

* **diff\_ts** simplifies removing trends and seasonality.
* **undo\_diff** reconstructs the series, assuming the starting value (first\_y) is known.
* Both are essential for preprocessing and postprocessing in ARIMA and similar models.

**Function: test\_2ts\_casuality**

**Purpose**

This function performs a Granger causality test between two time series. It evaluates whether one time series (ts2) can predict the other (ts1) based on historical values.

**Detailed Explanation**

**Parameters**

1. **ts1 (pandas.Series)**
   * The primary time series to be tested for being "caused" by the other.
2. **ts2 (pandas.Series)**
   * The secondary time series that is tested for its potential causal effect on ts1.
3. **maxlag (int, default=30)**
   * Maximum number of lags to test in the Granger causality analysis.
4. **figsize (tuple, default=(15, 5))**
   * Size of the plot showing both time series.

**Returns**

* **Plot of ts1 and ts2**:  
  Displays both time series to visualize potential relationships or overlaps.
* **Granger Causality Results (printed)**:  
  Reports significant lags where causality is detected, with corresponding p-values.

**Process**

1. **Prepare the Data**
   * Combine ts1 and ts2 into a single DataFrame.
   * Plot both series for visual inspection.
2. **Perform Granger Causality Test**
   * Use statsmodels.tsa.stattools.grangercausalitytests to evaluate causality for lags up to maxlag.
   * The Granger test evaluates whether the past values of ts2 provide statistically significant information about ts1.
3. **Interpret Results**
   * For each lag, calculate the average p-value from the test's multiple outputs.
   * If the p-value is below 0.05, conclude that ts2 has a causal effect on ts1 at that lag.

**Code Walkthrough**

def test\_2ts\_casuality(ts1, ts2, maxlag=30, figsize=(15,5)):

# Combine time series into a DataFrame

dtf = ts1.to\_frame(name=ts1.name)

dtf[ts2.name] = ts2

# Plot the time series for visual inspection

dtf.plot(figsize=figsize, grid=True, title=f"{ts1.name} vs {ts2.name}")

plt.show()

# Perform Granger causality test

granger\_test = sm.tsa.stattools.grangercausalitytests(dtf, maxlag=maxlag, verbose=False)

# Interpret results

for lag, tupla in granger\_test.items():

# Calculate average p-value for the given lag

p = np.mean([tupla[0][k][1] for k in tupla[0].keys()])

p = round(p, 3)

if p < 0.05:

conclusion = f"Causality with lag {lag} (p-value: {p})"

print(conclusion)

**Usage Example**

import pandas as pd

import numpy as np

# Generate Example Data

np.random.seed(42)

index = pd.date\_range(start="2020-01-01", periods=100, freq="D")

ts1 = pd.Series(np.sin(np.linspace(0, 20, 100)) + np.random.randn(100) \* 0.1, index=index, name="Series 1")

ts2 = pd.Series(np.sin(np.linspace(0, 20, 100) - 1) + np.random.randn(100) \* 0.1, index=index, name="Series 2")

# Test for Granger Causality

test\_2ts\_casuality(ts1, ts2, maxlag=10, figsize=(12, 6))

**Expected Output**

1. **Plot**:
   * A line chart displaying ts1 and ts2 over time.
2. **Causality Results**:
   * For example:
   * Causality with lag 2 (p-value: 0.03)
   * Causality with lag 5 (p-value: 0.04)

**Key Notes**

1. **Interpretation of Results**:
   * If ts2 Granger-causes ts1 at certain lags, it indicates that ts2 provides useful information for predicting ts1. This does not imply true causation, only statistical predictive power.
2. **Limitations**:
   * The test assumes stationarity. Preprocess your series (e.g., differencing) if necessary.
3. **Use Case**:
   * Commonly used in econometrics, climate science, and any domain involving time series data to study relationships between variables.

**Function: decompose\_ts**

**Purpose**

This function decomposes a time series into its three main components: **trend**, **seasonality**, and **residuals**. It provides insights into the underlying structure of the data and is especially useful for understanding periodic patterns or long-term changes.

**Detailed Explanation**

**Parameters**

1. **ts (pandas.Series)**
   * The input time series data to decompose. Must have a datetime index or a frequency for proper decomposition.
2. **s (int, default=7)**
   * The seasonal period of the data (e.g., 7 for weekly seasonality if the data is daily).
3. **figsize (tuple, default=(20, 13))**
   * The size of the plot showing the decomposition results.

**Returns**

* A dictionary containing the decomposed components:
  + "trend": Long-term trend of the time series.
  + "seasonal": Repeating seasonal patterns.
  + "residual": Noise or unexplained variation after removing trend and seasonality.

**Process**

1. **Decomposition**
   * Use the seasonal\_decompose function from statsmodels.tsa to decompose the time series into:
     + **Trend**: Captures the overall direction or pattern in the data.
     + **Seasonality**: Periodic fluctuations (e.g., daily, weekly, yearly).
     + **Residual**: Random noise or irregular components.
   * The parameter freq=s specifies the seasonal period.
2. **Visualization**
   * Create a grid of four subplots:
     + Plot the original series.
     + Plot the extracted trend component.
     + Plot the seasonal component.
     + Plot the residuals.
   * Add titles and grids to make each subplot clear.
3. **Return Results**
   * Return the three decomposed components as a dictionary for further analysis.

**Code Walkthrough**

def decompose\_ts(ts, s=7, figsize=(20,13)):

# Decompose the time series

decomposition = smt.seasonal\_decompose(ts, freq=s)

trend = decomposition.trend

seasonal = decomposition.seasonal

residual = decomposition.resid

# Plot the decomposition components

fig, ax = plt.subplots(nrows=4, ncols=1, sharex=True, sharey=False, figsize=figsize)

# Original series

ax[0].plot(ts)

ax[0].set\_title('Original')

ax[0].grid(True)

# Trend component

ax[1].plot(trend)

ax[1].set\_title('Trend')

ax[1].grid(True)

# Seasonal component

ax[2].plot(seasonal)

ax[2].set\_title('Seasonality')

ax[2].grid(True)

# Residuals

ax[3].plot(residual)

ax[3].set\_title('Residuals')

ax[3].grid(True)

# Return components

return {"trend": trend, "seasonal": seasonal, "residual": residual}

**Usage Example**

import pandas as pd

import numpy as np

# Example data: Generate a time series with trend, seasonality, and noise

index = pd.date\_range(start="2022-01-01", periods=365, freq="D")

trend = np.linspace(50, 100, 365)

seasonality = 10 \* np.sin(np.linspace(0, 2 \* np.pi \* 52, 365))

noise = np.random.normal(0, 2, 365)

data = trend + seasonality + noise

ts = pd.Series(data, index=index, name="Synthetic Time Series")

# Decompose the time series

decomposed = decompose\_ts(ts, s=7)

**Expected Output**

1. **Plots**:
   * **Original Time Series**: Shows the complete time series.
   * **Trend Component**: Displays the long-term changes.
   * **Seasonal Component**: Highlights recurring patterns.
   * **Residuals**: Any unexplained variance or noise.
2. **Decomposed Components**:  
   The returned dictionary will contain the following:
   * decomposed["trend"]: Series of the trend component.
   * decomposed["seasonal"]: Series of the seasonal component.
   * decomposed["residual"]: Series of the residuals.

**Key Notes**

1. **Input Requirements**:
   * The time series must have a proper frequency (e.g., daily, monthly). Use .asfreq() if needed to set the frequency.
2. **Interpretation**:
   * Decomposing helps identify whether seasonality or trend is driving the changes in the data, and whether the residuals are random.
3. **Practical Use Cases**:
   * Trend analysis for sales, temperature, stock prices, or any other time series.
   * Preprocessing for forecasting models, as many algorithms benefit from separating components.
4. **Limitations**:
   * Assumes additive decomposition by default. For multiplicative models, normalize the series first or explicitly specify the decomposition type.

**Function: find\_outliers**

**Purpose**

Detect outliers in a time series using a machine learning approach based on Support Vector Machines (SVM). This method identifies data points that deviate significantly from the rest of the series.

**Detailed Explanation**

**Parameters**

1. **ts (pandas.Series)**
   * The time series data where outliers will be detected.
2. **perc (float, default=0.01)**
   * Specifies the percentage of data points to classify as outliers (i.e., the upper limit of anomalies expected).
3. **figsize (tuple, default=(15, 5))**
   * The size of the plot for visualizing the detected outliers.

**Returns**

* **dtf\_outliers (pandas.DataFrame)**:  
  A DataFrame containing:
  + "ts": The original time series values.
  + "outlier": A binary flag (1 for outliers, 0 otherwise).

**Process**

1. **Scaling the Time Series**
   * The time series values are standardized using StandardScaler. This ensures that the data is normalized and centered around zero, which is essential for effective SVM training.
2. **Training an SVM for Outlier Detection**
   * A one-class SVM is trained using the radial basis function (RBF) kernel. The parameter nu=perc controls the fraction of data to treat as anomalies.
3. **Predicting Outliers**
   * The trained SVM assigns predictions to each data point. Points labeled -1 are classified as outliers, while others are normal (1).
4. **Preparing the Output**
   * The time series and outlier flags are stored in a DataFrame.
5. **Visualization**
   * The time series is plotted, and detected outliers are highlighted in red for easy interpretation.

**Code Walkthrough**

def find\_outliers(ts, perc=0.01, figsize=(15,5)):

## Step 1: Scale the time series

scaler = preprocessing.StandardScaler()

ts\_scaled = scaler.fit\_transform(ts.values.reshape(-1,1))

## Step 2: Train the SVM model

model = svm.OneClassSVM(nu=perc, kernel="rbf", gamma=0.01)

model.fit(ts\_scaled)

## Step 3: Prepare the output DataFrame

dtf\_outliers = ts.to\_frame(name="ts")

dtf\_outliers["outlier"] = model.predict(ts\_scaled)

dtf\_outliers["outlier"] = dtf\_outliers["outlier"].apply(lambda x: 1 if x == -1 else 0)

## Step 4: Visualization

fig, ax = plt.subplots(figsize=figsize)

ax.set(title="Outliers detection: found "+str(sum(dtf\_outliers["outlier"] == 1)))

ax.plot(dtf\_outliers.index, dtf\_outliers["ts"], color="black") # Original time series

ax.scatter(

x=dtf\_outliers[dtf\_outliers["outlier"] == 1].index,

y=dtf\_outliers[dtf\_outliers["outlier"] == 1]['ts'],

color='red'

) # Highlight outliers

ax.grid(True)

plt.show()

## Step 5: Return the results

return dtf\_outliers

**Usage Example**

import pandas as pd

import numpy as np

# Example data: Generate a time series with random noise and outliers

np.random.seed(42)

data = np.random.normal(loc=50, scale=5, size=100)

data[::10] = data[::10] + np.random.normal(loc=30, scale=5, size=10) # Add outliers

index = pd.date\_range(start="2023-01-01", periods=100, freq="D")

ts = pd.Series(data, index=index, name="Synthetic Time Series")

# Detect outliers

outliers = find\_outliers(ts, perc=0.05)

**Expected Output**

1. **Plot**:
   * The original time series is plotted in black.
   * Outliers are highlighted as red dots.
2. **Returned DataFrame**:  
   A sample of the output:
3. ts outlier
4. 2023-01-01 56.674032 1
5. 2023-01-02 49.861734 0
6. 2023-01-03 52.647689 0
7. 2023-01-04 56.623170 0
8. 2023-01-05 49.278411 0

**Key Notes**

1. **Model Parameters**:
   * nu determines the fraction of points treated as anomalies. Adjust this to control the sensitivity of outlier detection.
   * The gamma parameter of the SVM controls the kernel's shape and influences how anomalies are identified.
2. **Visualization**:
   * Ensure the time series has clear anomalies or unexpected spikes/drops to validate the results visually.
3. **Applications**:
   * Identifying fraud in financial transactions.
   * Detecting sensor malfunctions in IoT data.
   * Spotting anomalous patterns in stock prices or server logs.
4. **Limitations**:
   * SVM outlier detection may not perform well on highly complex or noisy datasets.
   * Large datasets may require additional preprocessing for performance.

**Function: remove\_outliers**

**Purpose**

Remove outliers from a time series and replace them with interpolated values. This ensures that the series remains smooth and continuous while eliminating anomalous data points.

**Detailed Explanation**

**Parameters**

1. **ts (pandas.Series)**
   * The original time series data containing potential outliers.
2. **outliers\_idx (list or index-like)**
   * The indices of the time series corresponding to detected outliers. These indices will be replaced with interpolated values.
3. **figsize (tuple, default=(15, 5))**
   * The size of the plot that visualizes the original time series and the cleaned version after interpolation.

**Returns**

* **ts\_clean (pandas.Series)**:  
  The cleaned time series with outliers removed and replaced by interpolated values.

**Process**

1. **Create a Copy**
   * A copy of the original time series is created to preserve the original data.
2. **Replace Outliers with NaN**
   * The values at the specified indices (outliers\_idx) are set to NaN, marking them for interpolation.
3. **Interpolate Missing Values**
   * Linear interpolation is applied to replace NaN values with smoothly estimated values based on neighboring data points.
4. **Visualization**
   * The original time series (including outliers) is plotted in red with reduced opacity for contrast.
   * The cleaned time series is overlaid in black for comparison.

**Code Walkthrough**

def remove\_outliers(ts, outliers\_idx, figsize=(15,5)):

## Step 1: Create a copy of the original time series

ts\_clean = ts.copy()

## Step 2: Replace outliers with NaN

ts\_clean.loc[outliers\_idx] = np.nan

## Step 3: Interpolate missing values

ts\_clean = ts\_clean.interpolate(method="linear")

## Step 4: Plot the original and cleaned time series

ax = ts.plot(

figsize=figsize,

color="red",

alpha=0.5,

title="Remove outliers",

label="original",

legend=True

)

ts\_clean.plot(

ax=ax,

grid=True,

color="black",

label="interpolated",

legend=True

)

ax.set(xlabel=None)

plt.show()

## Step 5: Return the cleaned time series

return ts\_clean

**Usage Example**

import pandas as pd

import numpy as np

# Example data: Generate a time series with random noise and outliers

np.random.seed(42)

data = np.random.normal(loc=50, scale=5, size=100)

data[::10] = data[::10] + np.random.normal(loc=30, scale=5, size=10) # Add outliers

index = pd.date\_range(start="2023-01-01", periods=100, freq="D")

ts = pd.Series(data, index=index, name="Synthetic Time Series")

# Detect outliers (use the previous find\_outliers function)

outliers = find\_outliers(ts, perc=0.05)

# Remove outliers using their indices

clean\_ts = remove\_outliers(ts, outliers\_idx=outliers[outliers["outlier"] == 1].index)

**Expected Output**

1. **Plot**:
   * The original time series (with outliers) is shown in red with reduced opacity.
   * The cleaned time series, after interpolation, is shown in black.
2. **Returned Time Series**:
   * The time series with outliers removed and replaced by interpolated values.

**Key Notes**

1. **Interpolation Method**:
   * The function uses linear interpolation. Other methods (e.g., polynomial, spline) can be specified if needed.
2. **Preservation of Data Integrity**:
   * Interpolation ensures that the time series remains continuous and smooth without introducing drastic changes.
3. **Applications**:
   * Cleaning sensor data where occasional spikes occur.
   * Preparing data for machine learning models that are sensitive to outliers.
4. **Limitations**:
   * Interpolation assumes that the missing data can be reasonably estimated from its neighbors. In cases of long gaps or highly variable data, this may introduce inaccuracies.

**Function: resistence\_support**

**Purpose**

The function detects support and resistance levels in a time series, which are key concepts in technical analysis for financial markets. Support represents the lower boundary (a price level where the price tends to stop falling), and resistance represents the upper boundary (a price level where the price tends to stop rising).

**Detailed Explanation**

**Parameters**

1. **ts (pandas.Series)**
   * The time series data, such as stock prices or any other time series with numerical values.
2. **window (int, default=30)**
   * The window size for calculating the rolling mean and detecting local maxima (resistance) and minima (support). This represents the number of periods to consider when calculating local peaks and troughs.
3. **trend (bool, default=False)**
   * If True, the detected resistance and support levels are interpolated (i.e., smoothed). If False, missing resistance and support values are filled with the last available value (ffill method).
4. **plot (bool, default=True)**
   * If True, the function will plot the original time series along with the detected resistance and support levels.
5. **figsize (tuple, default=(15, 5))**
   * The size of the plot when visualizing the time series and the support/resistance levels.

**Returns**

* **dtf (pandas.DataFrame)**:  
  The DataFrame with the original time series, detected resistance levels, and support levels.

**Process**

1. **Data Preparation**:
   * The time series (ts) is converted into a DataFrame (dtf) to facilitate easy manipulation of columns.
   * Two additional columns, max and min, are added to the DataFrame to store the resistance (local maxima) and support (local minima).
2. **Rolling Mean**:
   * A rolling window (window=30) is applied to the time series to smooth the data and make it easier to detect local maxima and minima.
3. **Detect Local Maxima (Resistance)**:
   * The argrelextrema function is used to find local maxima in the smoothed time series. These maxima are the potential resistance levels.
   * For each local maximum, the actual maximum within a specified range is identified.
4. **Detect Local Minima (Support)**:
   * Similarly, local minima are detected using argrelextrema for the smoothed time series, representing potential support levels.
5. **Fill Missing Values**:
   * Depending on the trend parameter:
     + If trend=True, the detected support and resistance levels are interpolated using linear interpolation.
     + If trend=False, the missing support and resistance values are filled using forward filling (ffill).
6. **Plotting**:
   * The time series is plotted along with the resistance and support levels.
   * Local maxima (resistance points) and minima (support points) are marked with scatter points.
   * The resistance and support levels are plotted as dashed lines.

**Code Walkthrough**

def resistence\_support(ts, window=30, trend=False, plot=True, figsize=(15,5)):

## Step 1: Create a DataFrame for the time series

dtf = ts.to\_frame(name="ts")

dtf["max"], dtf["min"] = [np.nan, np.nan] # Initialize columns for maxima and minima

## Step 2: Apply a rolling mean to the time series

rolling = dtf['ts'].rolling(window=window).mean().dropna()

## Step 3: Detect local maxima (resistance)

local\_max = signal.argrelextrema(rolling.values, np.greater)[0]

local\_max\_idx = [dtf.iloc[i-window:i+window]['ts'].idxmax() for i in local\_max if (i > window) and (i < len(dtf)-window)]

dtf["max"].loc[local\_max\_idx] = dtf["ts"].loc[local\_max\_idx]

## Step 4: Detect local minima (support)

local\_min = signal.argrelextrema(rolling.values, np.less)[0]

local\_min\_idx = [dtf.iloc[i-window:i+window]['ts'].idxmin() for i in local\_min if (i > window) and (i < len(dtf)-window)]

dtf["min"].loc[local\_min\_idx] = dtf["ts"].loc[local\_min\_idx]

## Step 5: Interpolate or forward fill resistance/support levels

dtf["resistence"] = dtf["max"].interpolate(method="linear") if trend is True else dtf["max"].fillna(method="ffill")

dtf["support"] = dtf["min"].interpolate(method="linear") if trend is True else dtf["min"].fillna(method="ffill")

## Step 6: Plot the results

if plot is True:

ax = dtf["ts"].plot(color="black", figsize=figsize, grid=True, title="Resistence and Support")

dtf["resistence"].plot(ax=ax, color="darkviolet", label="resistence", grid=True, linestyle="--")

dtf["support"].plot(ax=ax, color="green", label="support", grid=True, linestyle="--")

ax.scatter(x=dtf["max"].index, y=dtf["max"].values, color="darkviolet", label="max")

ax.scatter(x=dtf["min"].index, y=dtf["min"].values, color="green", label="min")

ax.set(xlabel=None)

ax.legend()

plt.show()

## Step 7: Return the DataFrame with detected levels

return dtf

**Usage Example**

import pandas as pd

import numpy as np

# Example data: Generate a synthetic time series (e.g., stock prices)

np.random.seed(42)

data = np.random.normal(loc=50, scale=5, size=100)

data[::10] = data[::10] + np.random.normal(loc=10, scale=5, size=10) # Add outliers

index = pd.date\_range(start="2023-01-01", periods=100, freq="D")

ts = pd.Series(data, index=index, name="Stock Prices")

# Calculate resistance and support levels

resistance\_support\_df = resistence\_support(ts, window=20, trend=True)

**Expected Output**

1. **Plot**:
   * The original time series is shown in black.
   * The resistance (upper) levels are shown in dark violet with dashed lines.
   * The support (lower) levels are shown in green with dashed lines.
   * Local maxima (resistance) and minima (support) points are marked as scatter points.
2. **Returned DataFrame**:
   * The DataFrame contains the original time series and the detected resistance and support levels.

**Key Notes**

1. **Choosing the window Parameter**:
   * A larger window value may smooth out the data more, reducing the number of local maxima/minima detected.
   * A smaller window value may lead to detecting more local maxima/minima, but they might be less reliable.
2. **Trend Parameter**:
   * When trend=True, the resistance and support levels are smoothed using linear interpolation. This is useful if you want to detect longer-term levels.
   * If trend=False, the levels are forward-filled, which can be useful for short-term analysis.
3. **Applications**:
   * Technical analysis in financial markets.
   * Identifying potential price levels where the market might reverse direction.
4. **Limitations**:
   * The function assumes that the data has periodic or trend-based fluctuations. It may not work well for time series with strong irregular patterns.

**Function: split\_train\_test**

**Purpose**

The function splits a time series (ts) into training and testing datasets, which is a common practice in time series forecasting or machine learning tasks. The function also supports splitting external explanatory variables (exog) if provided.

**Detailed Explanation**

**Parameters**

1. **ts (pandas.Series)**
   * The main time series that needs to be split into training and testing datasets.
2. **exog (pandas.Series or pandas.DataFrame, optional)**
   * Optional external explanatory variables or features (regressors). If provided, it will also be split into training and testing datasets in the same way as the time series.
3. **test (float or str or int, default=0.20)**
   * The proportion of the data to be used for testing:
     + If float: Specifies the percentage of the data to be allocated to the test set (e.g., 0.2 means 20% test data).
     + If str: Specifies a date or index value at which to split the data (e.g., '2023-12-01').
     + If int: Specifies the index at which to split the time series.
4. **plot (bool, default=True)**
   * If True, the function will generate a plot displaying the training and testing data.
5. **figsize (tuple, default=(15, 5))**
   * Specifies the size of the plot to be displayed when plot=True.

**Returns**

* **ts\_train (pandas.Series)**:  
  The training set, which consists of the first part of the time series (ts) before the splitting point.
* **ts\_test (pandas.Series)**:  
  The testing set, which consists of the remaining part of the time series (ts) after the splitting point.
* **exog\_train (optional, pandas.Series or pandas.DataFrame)**:  
  The training set for the external explanatory variables (exog), split similarly to the time series.
* **exog\_test (optional, pandas.Series or pandas.DataFrame)**:  
  The testing set for the external explanatory variables (exog), split similarly to the time series.

**Process**

1. **Determine Split Point**:
   * If test is a float, it is interpreted as the percentage of the data to use for testing. The split point is calculated based on the length of the time series.
   * If test is a string (e.g., a date), it searches for the first index that matches the given date and splits the time series at that point.
   * If test is an integer, it is directly used as the index where the split should happen.
2. **Split the Time Series**:
   * Based on the calculated split point, the time series (ts) is divided into two parts:
     + ts\_train: The portion of the time series from the start up to the split point.
     + ts\_test: The portion of the time series from the split point to the end.
3. **Split External Variables (exog)**:
   * If external explanatory variables (exog) are provided, they are split into exog\_train and exog\_test based on the same split point.
4. **Plotting**:
   * If plot=True, the function will generate a side-by-side plot to visualize the training and testing data. The training set is plotted on the left, and the testing set is plotted on the right.

**Code Walkthrough**

def split\_train\_test(ts, exog=None, test=0.20, plot=True, figsize=(15,5)):

## Step 1: Define the split point based on the value of `test`

if type(test) is float:

split = int(len(ts)\*(1-test)) # Calculate the index based on percentage

perc = test # Store the test proportion

elif type(test) is str:

split = ts.reset\_index()[ts.reset\_index().iloc[:,0]==test].index[0] # Split based on a date

perc = round(len(ts[split:])/len(ts), 2) # Calculate the percentage of test data

else:

split = test # Use an integer index for the split

perc = round(len(ts[split:])/len(ts), 2) # Calculate the percentage of test data

print("--- splitting at index: ", split, "|", ts.index[split], "| test size:", perc, " ---")

## Step 2: Split the time series into training and testing datasets

ts\_train = ts.head(split)

ts\_test = ts.tail(len(ts)-split)

if plot is True:

## Plot the training and testing sets side by side

fig, ax = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=True, figsize=figsize)

ts\_train.plot(ax=ax[0], grid=True, title="Train", color="black")

ts\_test.plot(ax=ax[1], grid=True, title="Test", color="black")

ax[0].set(xlabel=None)

ax[1].set(xlabel=None)

plt.show()

## Step 3: If external variables (`exog`) are provided, split them as well

if exog is not None:

exog\_train = exog[0:split]

exog\_test = exog[split:]

return ts\_train, ts\_test, exog\_train, exog\_test

else:

return ts\_train, ts\_test

**Usage Example**

import pandas as pd

import numpy as np

# Example: Generate a synthetic time series

np.random.seed(42)

data = np.random.normal(loc=50, scale=5, size=100)

index = pd.date\_range(start="2023-01-01", periods=100, freq="D")

ts = pd.Series(data, index=index, name="Stock Prices")

# Example: Split the time series using 80% for training and 20% for testing

ts\_train, ts\_test = split\_train\_test(ts, test=0.20)

# Example with external variables (exog)

exog = pd.DataFrame(np.random.normal(loc=10, scale=1, size=(100, 1)), columns=['Exog Feature'], index=index)

ts\_train, ts\_test, exog\_train, exog\_test = split\_train\_test(ts, exog=exog, test=0.20)

**Expected Output**

1. **Console Output**:
   * The function will print the index where the split occurs, along with the size of the test set (percentage).

Example:

--- splitting at index: 80 | 2023-03-22 00:00:00 | test size: 0.2 ---

1. **Plot**:
   * A plot showing two subplots:
     + On the left: The training set.
     + On the right: The testing set.
   * Both sets are displayed with their respective data points.
2. **Returned Variables**:
   * The function returns the split time series (ts\_train, ts\_test) and, if exog is provided, the corresponding external variables (exog\_train, exog\_test).

**Key Notes**

1. **Flexible Splitting**:
   * The function supports multiple types of splits (percentage, date, index), making it versatile for various scenarios.
2. **Plotting**:
   * Plotting the training and test sets helps visually assess the data split.
3. **Exogenous Variables**:
   * The function handles external variables (exog), which is particularly useful in time series forecasting models where external predictors are included.
4. **Customizable Test Size**:
   * The test size can be specified as a percentage, which is helpful for partitioning data based on specific modeling needs (e.g., 70% for training, 30% for testing).

**Function: utils\_conf\_int**

**Purpose**

The function calculates the confidence intervals for a list of values (lst\_values) based on the standard error (error\_std) and a given confidence level (conf). This is typically used in statistical analysis to estimate the range in which a true population parameter (such as the mean) is likely to fall.

**Detailed Explanation**

**Parameters**

1. **lst\_values (list or iterable)**
   * A list or iterable of values for which confidence intervals need to be calculated. These values are assumed to be sequential, where each value represents an observation at a given time or step.
2. **error\_std (float)**
   * The standard error of the values. The standard error is a measure of how much variation exists in the data and is used to calculate the confidence interval. It is assumed to be constant for each value in the list.
3. **conf (float, optional, default=0.95)**
   * The confidence level for the interval, typically 0.95 for a 95% confidence interval. The default value is 95%, but it can be changed (e.g., 0.99 for a 99% confidence interval).

**Returns**

* **lst\_ci (numpy.ndarray)**
  + A 2D NumPy array where each row represents a confidence interval for each value in lst\_values. Each interval contains the lower and upper bounds of the confidence interval.

**Process**

1. **List Conversion**:
   * The function first ensures that lst\_values is a list. If lst\_values is not already a list, it is converted into one.
2. **Critical Value Calculation (c)**:
   * The critical value c is computed using the standard normal distribution (z-score) for the specified confidence level (conf). This value is determined using the ppf (percent point function) from the normal distribution.
3. **Confidence Interval Calculation**:
   * For each value in lst\_values, the function calculates the confidence interval by considering all values up to and including the current value.
     + The size of the subset lst\_x (which is the slice of lst\_values from the start up to the current value) is used to adjust the width of the confidence interval.
     + The confidence interval is then calculated using the formula: CI=(x−c×error\_std×h,x+c×error\_std×h)CI = \left( x - c \times \text{error\\_std} \times \sqrt{h}, x + c \times \text{error\\_std} \times \sqrt{h} \right) Where:
       - x is the current value in lst\_values.
       - h is the number of values in the subset lst\_x (i.e., the number of observations up to the current point).
4. **Return**:
   * The function returns a 2D NumPy array where each row corresponds to a confidence interval for each value in the list. The first column contains the lower bound of the interval, and the second column contains the upper bound.

**Code Walkthrough**

def utils\_conf\_int(lst\_values, error\_std, conf=0.95):

# Ensure lst\_values is a list

lst\_values = list(lst\_values) if type(lst\_values) != list else lst\_values

# Calculate the critical value (z-score) for the given confidence level

c = round(stats.norm.ppf(1 - (1 - conf) / 2), 2)

# Initialize an empty list to store the confidence intervals

lst\_ci = []

# Calculate the confidence interval for each value in lst\_values

for x in lst\_values:

# Subset the list up to and including the current value

lst\_x = lst\_values[:lst\_values.index(x) + 1]

# Get the number of elements in the subset

h = len(lst\_x)

# Calculate the confidence interval

ci = [x - (c \* error\_std \* np.sqrt(h)), x + (c \* error\_std \* np.sqrt(h))]

# Append the confidence interval to the list

lst\_ci.append(ci)

# Return the confidence intervals as a numpy array

return np.array(lst\_ci)

**Usage Example**

import numpy as np

from scipy import stats

# Example values and standard error

values = [10, 12, 14, 16, 18, 20]

error\_std = 1.5 # Assume a standard error of 1.5

# Calculate 95% confidence intervals for each value

conf\_intervals = utils\_conf\_int(values, error\_std, conf=0.95)

print(conf\_intervals)

**Expected Output**

The function will output the confidence intervals for each value in lst\_values. For example, with the inputs provided:

[[ 8.499 11.501 ]

[10.097 13.903 ]

[11.695 15.305 ]

[13.293 16.707 ]

[14.891 18.109 ]

[16.489 19.511 ]]

Each row corresponds to a confidence interval for a value in the values list, and the columns represent the lower and upper bounds of the confidence interval.

**Key Notes**

1. **Dynamic Confidence Intervals**:
   * The function adapts the width of the confidence interval based on the number of values up to the current point. As more data points are included, the confidence interval becomes more precise.
2. **Application**:
   * This type of confidence interval calculation is useful for time series or sequential data where the confidence around a running estimate evolves as more data is available.
3. **Error Standard Deviation (error\_std)**:
   * The error\_std parameter is assumed to be constant, which may be the case in some scenarios but might need adjustment in more complex models (e.g., time-varying error).
4. **Normal Distribution Assumption**:
   * This function assumes that the data follows a normal distribution, which is common in many statistical analyses, but may not hold in all cases.

**Function: utils\_evaluate\_ts\_model**

**Purpose**

The function evaluates the performance of a time series forecasting model by calculating various error metrics and generating visualizations for the training and test phases. It also computes confidence intervals for the forecasts, providing a comprehensive assessment of the model's accuracy and predictive reliability.

**Detailed Explanation**

**Parameters**

1. **dtf (DataFrame)**
   * A pandas DataFrame containing at least three columns:
     + **ts**: The actual time series data (true values).
     + **model**: The model's predicted values for the training period.
     + **forecast**: The model's forecasted values for the test period (or future periods).
2. **conf (float, optional, default=0.95)**
   * The confidence level for computing the confidence interval of the forecasts. By default, it is set to 95%.
3. **title (str, optional)**
   * The title of the plot. If None, no title is set.
4. **plot (bool, optional, default=True)**
   * Whether to plot the evaluation charts or not. If False, the function will only return the evaluation metrics without any visualizations.
5. **figsize (tuple, optional, default=(20,13))**
   * The size of the plots to be generated.

**Returns**

* **dtf (DataFrame)**
  + A pandas DataFrame containing the original time series (ts), model predictions (model), residuals (differences between ts and model), confidence intervals (lower, upper), forecasts (forecast), and forecast errors (error).

**Process**

1. **Residual Calculation**:  
   The residuals (the differences between the actual values ts and the model predictions model) are computed and added as a new column in the DataFrame.
2. **Error Metrics Calculation**:
   * **Forecasting Errors**: The difference between the actual time series (ts) and the forecasted values (forecast) is calculated, and the relative errors (error\_pct) are also computed.
   * Key performance indicators (KPIs) are calculated for both residuals and errors:
     + **Mean of residuals** and **standard deviation**.
     + **Mean error**, **standard deviation of errors**, **Mean Absolute Error (MAE)**, **Mean Absolute Percentage Error (MAPE)**, **Mean Squared Error (MSE)**, and **Root Mean Squared Error (RMSE)**.
3. **Confidence Intervals**:
   * If confidence intervals (lower, upper) are not already in the DataFrame, they are computed using the utils\_conf\_int function, based on the residuals and the specified confidence level (conf).
4. **Plotting**:
   * If plot is True, the following plots are generated:
     + **Training Plot**: The actual time series (ts) and the model predictions (model) for the training period.
     + **Test Plot**: The actual time series (ts) and the forecasted values (forecast) for the test period, with the computed confidence intervals shaded.
     + **Residuals Plot**: The residuals and forecast errors over time.
     + **Residuals Distribution Plot**: A kernel density estimate (KDE) plot showing the distribution of residuals and errors, with vertical lines indicating their means.
5. **Error Metrics Output**:  
   The calculated KPIs are printed to the console, providing insights into the model's performance.

**Code Walkthrough**

def utils\_evaluate\_ts\_model(dtf, conf=0.95, title=None, plot=True, figsize=(20,13)):

try:

## residuals from fitting

dtf["residuals"] = dtf["ts"] - dtf["model"] # Calculate residuals

residuals\_mean = dtf["residuals"].mean() # Mean of residuals

residuals\_std = dtf["residuals"].std() # Standard deviation of residuals

## forecasting error

dtf["error"] = dtf["ts"] - dtf["forecast"] # Forecast error

dtf["error\_pct"] = dtf["error"] / dtf["ts"] # Relative forecast error

error\_mean = dtf["error"].mean() # Mean error

error\_std = dtf["error"].std() # Standard deviation of error

mae = dtf["error"].apply(lambda x: np.abs(x)).mean() # Mean absolute error (MAE)

mape = dtf["error\_pct"].apply(lambda x: np.abs(x)).mean() # Mean absolute percentage error (MAPE)

mse = dtf["error"].apply(lambda x: x\*\*2).mean() # Mean squared error (MSE)

rmse = np.sqrt(mse) # Root mean squared error (RMSE)

## interval

if "upper" not in dtf.columns:

print("--- computing confidence interval ---")

dtf["lower"], dtf["upper"] = [np.nan, np.nan]

dtf.loc[dtf["forecast"].notnull(), ["lower", "upper"]] = utils\_conf\_int(

dtf[dtf["forecast"].notnull()]["forecast"], residuals\_std, conf) # Calculate confidence intervals

## plot

if plot:

fig = plt.figure(figsize=figsize)

fig.suptitle(title, fontsize=20)

# Create subplots

ax1 = fig.add\_subplot(2, 2, 1)

ax2 = fig.add\_subplot(2, 2, 2, sharey=ax1)

ax3 = fig.add\_subplot(2, 2, 3)

ax4 = fig.add\_subplot(2, 2, 4)

# Plot training data

dtf[pd.notnull(dtf["model"])][["ts", "model"]].plot(color=["black", "green"], title="Train", grid=True, ax=ax1)

ax1.set(xlabel=None)

# Plot test data

dtf[pd.isnull(dtf["model"])][["ts", "forecast"]].plot(color=["black", "red"], title="Test", grid=True, ax=ax2)

ax2.fill\_between(x=dtf.index, y1=dtf['lower'], y2=dtf['upper'], color='b', alpha=0.2) # Confidence interval

ax2.set(xlabel=None)

# Plot residuals and error

dtf[["residuals", "error"]].plot(ax=ax3, color=["green", "red"], title="Residuals", grid=True)

ax3.set(xlabel=None)

# Plot residuals distribution

dtf[["residuals", "error"]].plot(ax=ax4, color=["green", "red"], kind='kde', title="Residuals Distribution", grid=True)

ax4.axvline(dtf["residuals"].mean(), ls='--', color="green", label="mean: "+str(round(dtf["residuals"].mean(),2)))

ax4.axvline(dtf["error"].mean(), ls='--', color="red", label="mean: "+str(round(dtf["error"].mean(),2)))

ax4.set(ylabel=None)

ax4.legend()

plt.show()

# Print KPIs

print("Training --> Residuals mean:", np.round(residuals\_mean), " | std:", np.round(residuals\_std))

print("Test --> Error mean:", np.round(error\_mean), " | std:", np.round(error\_std),

" | mae:", np.round(mae), " | mape:", np.round(mape \* 100), "% | mse:", np.round(mse), " | rmse:", np.round(rmse))

return dtf[["ts", "model", "residuals", "lower", "forecast", "upper", "error"]]

except Exception as e:

print("--- got error ---")

print(e)

**Usage Example**

# Example usage:

import pandas as pd

# Assuming 'df' contains time series data with 'ts' (actual values), 'model' (training model values), and 'forecast' (forecasted values)

result = utils\_evaluate\_ts\_model(df, conf=0.95, title="Time Series Model Evaluation")

**Key Metrics Output**

* **Training**:
  + Residuals: Mean and standard deviation of the residuals.
* **Testing**:
  + Forecast errors: Mean, standard deviation, MAE, MAPE, MSE, and RMSE of the forecast errors.

**Visualizations:**

1. **Training vs. Model** (observations vs predictions).
2. **Test vs. Forecast** (actual values vs forecasted values), with confidence intervals.
3. **Residuals and Forecast Errors** over time.
4. **Kernel Density Estimation** (KDE) plot for the distribution of residuals and errors.

**Function: utils\_generate\_indexdate**

**Purpose**

The function generates a pandas DatetimeIndex based on the provided parameters. This is particularly useful when you need to create a custom date range for time series analysis, either by specifying the start and end dates or by defining the number of periods.

**Detailed Explanation**

**Parameters**

1. **start (str or datetime-like)**
   * The starting point of the date range. It can be a string (e.g., "2020-01-01") or a datetime object.
2. **end (str or datetime-like, optional)**
   * The ending point of the date range. If not provided, the function will generate the range based on the number of periods (n) instead.
3. **n (int, optional)**
   * The number of periods (dates) to generate. If end is provided, n will be ignored.
4. **freq (str, optional, default="D")**
   * The frequency of the date range. It uses pandas' frequency strings. For example:
     + "D" for daily frequency.
     + "M" for monthly.
     + "H" for hourly, etc.

**Returns**

* **index (DatetimeIndex)**
  + A pandas DatetimeIndex object representing the generated sequence of dates.

**Process**

1. **Generate Date Range**:
   * If end is provided, the function generates a date range starting from start to end using the specified frequency (freq).
   * If end is not provided, the function generates n periods starting from start, also with the specified frequency (freq).
2. **Remove the First Entry**:
   * After generating the date range, the first date (index[0]) is removed using index[1:]. This might be useful if you want the generated dates to exclude the starting date.
3. **Logging Information**:
   * The function prints out information about the generated date range, including the frequency, start date, end date, and the length of the index.

**Code Walkthrough**

def utils\_generate\_indexdate(start, end=None, n=None, freq="D"):

# Generate a date range based on the provided parameters

if end is not None:

index = pd.date\_range(start=start, end=end, freq=freq)

else:

index = pd.date\_range(start=start, periods=n, freq=freq)

# Remove the first date from the index

index = index[1:]

# Print information about the generated date index

print("--- generating index date --> freq:", freq, "| start:", index[0], "| end:", index[-1], "| len:", len(index), "---")

# Return the generated date index

return index

**Usage Example**

# Example: Generate a daily date range from '2020-01-01' to '2020-01-10'

index1 = utils\_generate\_indexdate('2020-01-01', end='2020-01-10', freq="D")

print(index1)

# Example: Generate 10 daily dates starting from '2020-01-01'

index2 = utils\_generate\_indexdate('2020-01-01', n=10, freq="D")

print(index2)

**Output Example**

For the first example:

--- generating index date --> freq: D | start: 2020-01-02 | end: 2020-01-10 | len: 9 ---

DatetimeIndex(['2020-01-02', '2020-01-03', '2020-01-04', '2020-01-05', '2020-01-06',

'2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10'],

dtype='datetime64[ns]', freq='D')

For the second example:

--- generating index date --> freq: D | start: 2020-01-02 | end: 2020-01-10 | len: 10 ---

DatetimeIndex(['2020-01-02', '2020-01-03', '2020-01-04', '2020-01-05', '2020-01-06',

'2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10', '2020-01-11'],

dtype='datetime64[ns]', freq='D')

**Key Points**

* **Flexibility in Date Ranges**: The function allows you to generate a date range either by specifying the start and end or by specifying the start and the number of periods (n).
* **Frequency Control**: You can control the frequency of the date range (e.g., daily, monthly).
* **Debugging Information**: The function prints helpful details, such as the frequency, start date, end date, and the length of the generated date range.

**Function: utils\_add\_forecast\_int**

**Purpose**

This function enhances a DataFrame containing time series data by calculating and adding forecast confidence intervals (lower and upper bounds) to the forecasted values. Additionally, it visualizes the time series with the forecast and the calculated confidence intervals.

**Parameters**

1. **dtf (DataFrame)**
   * A pandas DataFrame that includes:
     + ts: The actual time series values.
     + model: The fitted model values (used for calculating residuals).
     + forecast: The forecasted values (for which confidence intervals are to be calculated).
2. **conf (float, optional, default=0.95)**
   * The confidence level for the forecast intervals (e.g., 0.95 corresponds to a 95% confidence interval).
3. **plot (bool, optional, default=True)**
   * Whether to plot the forecast with confidence intervals or not.
4. **zoom (int, optional, default=30)**
   * The number of most recent observations to zoom in on in the second plot.
5. **figsize (tuple, optional, default=(15,5))**
   * The size of the plot.

**Returns**

* **dtf (DataFrame)**
  + The original DataFrame (dtf) with additional columns:
    - residuals: The difference between the actual (ts) and the model (model).
    - lower: The lower bound of the forecast confidence interval.
    - upper: The upper bound of the forecast confidence interval.

**Process**

1. **Residual Calculation**:
   * The residuals are computed as the difference between the actual time series values (ts) and the model's fitted values (model).
2. **Standard Deviation of Residuals**:
   * The standard deviation of the residuals is computed and used to calculate the confidence interval for the forecast.
3. **Confidence Interval Calculation**:
   * If the upper and lower columns are not already present in the DataFrame, the function computes them using the utils\_conf\_int function. These intervals are based on the forecasted values and the standard deviation of the residuals.
4. **Plotting**:
   * **Plot 1**: The time series (ts) and forecast (forecast) are plotted on the same graph, with shaded regions representing the forecast confidence intervals (between lower and upper).
   * **Plot 2**: A zoomed-in version of the forecast and confidence intervals, focusing on the last zoom number of observations, is created.

**Code Walkthrough**

def utils\_add\_forecast\_int(dtf, conf=0.95, plot=True, zoom=30, figsize=(15,5)):

# Calculate residuals (actual - model)

dtf["residuals"] = dtf["ts"] - dtf["model"]

# Standard deviation of residuals

residuals\_std = dtf["residuals"].std()

# Check if confidence intervals (upper/lower) exist, if not, calculate them

if "upper" not in dtf.columns:

print("--- computing confidence interval ---")

dtf["lower"], dtf["upper"] = [np.nan, np.nan]

dtf.loc[dtf["forecast"].notnull(), ["lower","upper"]] = utils\_conf\_int(

dtf[dtf["forecast"].notnull()]["forecast"], residuals\_std, conf)

# Plot the results

if plot is True:

fig = plt.figure(figsize=figsize)

# Plot the entire series with forecast and confidence intervals

ax0 = plt.subplot2grid((1,3), (0,0), rowspan=1, colspan=2)

dtf[["ts","forecast"]].plot(color=["black","red"], grid=True, ax=ax0, title="History + Future")

ax0.fill\_between(x=dtf.index, y1=dtf['lower'], y2=dtf['upper'], color='b', alpha=0.2)

ax0.set(xlabel=None)

# Zoom on the most recent observations

ax1 = plt.subplot2grid((1,3), (0,2), rowspan=1, colspan=1)

first\_idx = dtf[pd.notnull(dtf["forecast"])].index[0]

first\_loc = dtf.index.tolist().index(first\_idx)

zoom\_idx = dtf.index[first\_loc-zoom]

dtf.loc[zoom\_idx:][["ts","forecast"]].plot(color=["black","red"], grid=True, ax=ax1, title="Zoom on the last "+str(zoom)+" observations")

ax1.fill\_between(x=dtf.loc[zoom\_idx:].index, y1=dtf.loc[zoom\_idx:]['lower'], y2=dtf.loc[zoom\_idx:]['upper'], color='b', alpha=0.2)

ax1.set(xlabel=None)

plt.show()

# Return the updated DataFrame with forecast confidence intervals

return dtf[["ts", "model", "residuals", "lower", "forecast", "upper"]]

**Usage Example**

# Example DataFrame with time series, model, and forecast

dtf = pd.DataFrame({

'ts': [10, 12, 15, 20, 25, 30, 28, 35, 40, 45], # Actual time series

'model': [9, 11, 14, 19, 24, 29, 27, 34, 39, 44], # Model's fitted values

'forecast': [46, 48, 50, 52, 54, 56, 58, 60, 62, 64] # Forecasted values

}, index=pd.date\_range('2024-01-01', periods=10, freq='D'))

# Add forecast intervals and plot

dtf\_with\_intervals = utils\_add\_forecast\_int(dtf, conf=0.95, plot=True, zoom=3, figsize=(15, 5))

**Plotting Explanation**

1. **Entire Series Plot (ax0)**:
   * The full time series (ts) is plotted alongside the forecast (forecast) in black and red colors, respectively. The confidence intervals (lower and upper) are shown as a shaded blue region.
2. **Zoomed-in Plot (ax1)**:
   * A focused view of the most recent zoom observations, showing the actual series and forecast with the confidence intervals as a shaded region.

**Key Points**

* **Forecast Confidence Intervals**: The function computes the confidence intervals for the forecast based on the residuals of the model. These intervals are important for assessing the uncertainty of the predictions.
* **Visualization**: The function generates two types of plots: one for the entire time series with the forecast and confidence intervals, and another that zooms in on the most recent forecast to provide a detailed view.
* **Flexibility**: The zoom and conf parameters provide flexibility in how much detail to display in the zoomed-in plot and the confidence level for the forecast intervals.

**Function: utils\_generate\_rw**

**Purpose**

This function generates a random walk (RW) time series starting from an initial value y0 and iteratively generating subsequent values based on a normal distribution. The random walk can also incorporate constraints on the minimum (ymin) and maximum (ymax) values.

**Parameters**

1. **y0 (float)**
   * The initial value of the random walk (the starting point of the series).
2. **n (int)**
   * The number of points to generate in the random walk series.
3. **sigma (float)**
   * The standard deviation of the normal distribution used to generate the random steps.
4. **ymin (float, optional)**
   * The minimum bound for the values in the random walk. If the value falls below this bound, it is adjusted.
5. **ymax (float, optional)**
   * The maximum bound for the values in the random walk. If the value exceeds this bound, it is adjusted.

**Returns**

* **rw (list)**
  + A list containing the values of the random walk, including any necessary adjustments based on the minimum and maximum bounds.

**Process**

1. **Initialization**:  
   The random walk starts at y0 and adds random steps generated from a normal distribution with mean 0 and standard deviation sigma.
2. **Random Walk Generation**:  
   For each time step, a random step is added to the previous value in the walk. The random step is drawn from a normal distribution.
3. **Bounds Check**:  
   If ymin or ymax are provided, the function checks if the next value in the random walk exceeds the bounds. If the next value falls outside the bounds, it is adjusted by subtracting or adding a random step to keep it within the range.
4. **Return the List**:  
   The list rw is returned, containing the generated random walk values.

**Code Walkthrough**

def utils\_generate\_rw(y0, n, sigma, ymin=None, ymax=None):

# Initialize the random walk list with the initial value y0

rw = [y0]

# Generate the random walk

for t in range(1, n):

# Generate a random step

yt = rw[t-1] + np.random.normal(0, sigma)

# Check if the value exceeds the upper bound ymax

if (ymax is not None) and (yt > ymax):

# Adjust the value to stay within the bound by subtracting a random step

yt = rw[t-1] - abs(np.random.normal(0, sigma))

# Check if the value falls below the lower bound ymin

elif (ymin is not None) and (yt < ymin):

# Adjust the value to stay within the bound by adding a random step

yt = rw[t-1] + abs(np.random.normal(0, sigma))

# Append the new value to the random walk list

rw.append(yt)

# Return the list of values in the random walk

return rw

**Explanation of Key Sections**

* **Generating Random Walk Values**:  
  At each time step t, a random value yt is generated by adding a random step from a normal distribution (np.random.normal(0, sigma)) to the previous value (rw[t-1]).
* **Handling Boundaries**:  
  If a value exceeds the ymax or is below ymin, the value is adjusted:
  + If yt > ymax, the function subtracts an absolute random step.
  + If yt < ymin, the function adds an absolute random step.
* **Adding the Generated Value**:  
  Each generated or adjusted value is appended to the rw list.

**Usage Example**

# Parameters for the random walk

y0 = 10 # Initial value

n = 50 # Number of points

sigma = 1.5 # Standard deviation for random steps

ymin = 5 # Lower bound

ymax = 20 # Upper bound

# Generate the random walk

rw\_series = utils\_generate\_rw(y0, n, sigma, ymin, ymax)

# Display the result

print(rw\_series)

**Output Example**

For the parameters above, the output might look like this (values will vary due to randomness):

[10, 11.5, 10.3, 9.9, 8.7, 7.6, 6.8, 5.9, 6.1, 7.3, 8.4, 9.2, 10.1, 11.3, 12.0, 13.5, 14.6, 15.4, 16.1, 17.3, 18.0, 18.9, 19.0, 19.8, 20, 19.6, 18.4, 17.2, 16.0, 15.3, 14.7, ...]

Here, the random walk values are bounded between ymin = 5 and ymax = 20, with steps generated based on the specified standard deviation.

**Use Cases**

1. **Simulating Stock Prices**:  
   The random walk model is often used to simulate stock prices, where each value is influenced by a random fluctuation but constrained within realistic price limits.
2. **Stochastic Processes**:  
   The function can be used in simulations that model various stochastic processes, such as movement of particles in physics or fluctuating quantities in economics.
3. **Financial and Economic Modeling**:  
   Random walks are widely used in financial modeling for asset prices, interest rates, and other economic indicators.

**Function: simulate\_rw**

**Purpose**

The function simulate\_rw simulates a random walk (RW) process based on a given training time series (ts\_train) and evaluates its performance using a test time series (ts\_test). It uses the random walk simulation to forecast future values and evaluates the model using common performance metrics, including residuals and forecast error.

**Parameters**

1. **ts\_train (pandas Series)**
   * The time series data used for training the random walk model. This series will be used to calculate the initial value and the random step size.
2. **ts\_test (pandas Series)**
   * The time series data used as the test set, where the random walk will forecast future values.
3. **conf (float, optional)**
   * The confidence level for the forecast interval (default is 0.95). It determines the upper and lower bounds for the forecast.
4. **figsize (tuple, optional)**
   * The size of the plot for visualizing the results (default is (15, 10)).

**Returns**

* **dtf (pandas DataFrame)**
  + A DataFrame containing the time series (ts), the simulated random walk values (model), the forecast values (forecast), and associated error and residuals, along with confidence intervals (lower, upper).

**Process**

1. **Simulating the Train Set**:  
   The function first calculates the difference between consecutive values in the training series (diff\_ts) and then simulates a random walk (rw) using the first value of the training series as the starting point. The standard deviation of the random walk is set to the standard deviation of diff\_ts, and the random walk is bounded by the minimum and maximum values of the training series.
2. **Simulating the Test Set**:  
   The function then generates a random walk forecast for the test set, using the last value of the training series as the starting point and again bounded by the minimum and maximum values of the training data.
3. **Merging the Train and Test Data**:  
   The simulated random walk is merged with the original time series for both the training and test periods. The resulting data frame contains the actual time series values (ts), the random walk model (model), and the forecasted values (forecast).
4. **Evaluation of the Model**:  
   The combined DataFrame (dtf) is evaluated using the utils\_evaluate\_ts\_model function, which calculates various key performance indicators (KPIs) such as residuals, error percentages, mean absolute error (MAE), root mean squared error (RMSE), and others. The function also computes confidence intervals for the forecast.
5. **Plotting the Results**:  
   The evaluation function generates a set of plots showing:
   * The training and forecasted time series.
   * The residuals.
   * The residuals distribution.

**Code Walkthrough**

def simulate\_rw(ts\_train, ts\_test, conf=0.95, figsize=(15,10)):

## simulate train

diff\_ts = ts\_train - ts\_train.shift(1) # Compute the differences between consecutive values in the train series.

# Generate a random walk for the training data

rw = utils\_generate\_rw(

y0=ts\_train[0], # Start the random walk at the first value of ts\_train

n=len(ts\_train), # Generate a series of the same length as ts\_train

sigma=diff\_ts.std(), # Standard deviation of the random walk steps, based on ts\_train differences

ymin=ts\_train.min(), # Lower bound set to the minimum value in ts\_train

ymax=ts\_train.max() # Upper bound set to the maximum value in ts\_train

)

# Merge the random walk results with the original train data

dtf\_train = ts\_train.to\_frame(name="ts").merge(

pd.DataFrame(rw, index=ts\_train.index, columns=["model"]),

how='left',

left\_index=True,

right\_index=True

)

## test

# Generate a random walk for the test data, starting from the last value in the training data

rw = utils\_generate\_rw(

y0=ts\_train[-1], # Start the random walk at the last value of ts\_train

n=len(ts\_test), # Generate a series of the same length as ts\_test

sigma=diff\_ts.std(), # Standard deviation remains the same as calculated from train data

ymin=ts\_train.min(), # Lower bound remains the same as for the train data

ymax=ts\_train.max() # Upper bound remains the same as for the train data

)

# Merge the random walk results with the original test data

dtf\_test = ts\_test.to\_frame(name="ts").merge(

pd.DataFrame(rw, index=ts\_test.index, columns=["forecast"]),

how='left',

left\_index=True,

right\_index=True

)

## combine train and test data

dtf = dtf\_train.append(dtf\_test) # Combine the training and test data

# Evaluate the model's performance using the utility function

dtf = utils\_evaluate\_ts\_model(dtf, conf=conf, figsize=figsize, title="Random Walk Simulation")

return dtf # Return the DataFrame containing the evaluation results

**Explanation of Key Sections**

* **Simulating Random Walk**:  
  The function uses the utils\_generate\_rw function to simulate a random walk based on the training data (ts\_train) and forecasts future values for the test set (ts\_test).
  + The random walk steps are determined by the standard deviation of the differences between consecutive values in the training series (diff\_ts.std()).
  + The bounds for the random walk are set by the minimum and maximum values of the training series (ts\_train.min() and ts\_train.max()).
* **Merging Train and Test Data**:  
  After generating the random walk for both the training and test sets, the results are merged with the original time series data into a combined DataFrame (dtf), which contains both the observed and simulated values.
* **Evaluating the Model**:  
  The utils\_evaluate\_ts\_model function is called to evaluate the performance of the random walk model by calculating various error metrics and visualizing the residuals and forecast intervals.

**Usage Example**

# Example time series data

import pandas as pd

import numpy as np

# Create synthetic training and test data

ts\_train = pd.Series(np.random.randn(100).cumsum(), index=pd.date\_range('20210101', periods=100))

ts\_test = pd.Series(np.random.randn(50).cumsum(), index=pd.date\_range('20210410', periods=50))

# Simulate random walk and evaluate the model

dtf\_evaluation = simulate\_rw(ts\_train, ts\_test, conf=0.95, figsize=(15, 10))

# Display the evaluation results

print(dtf\_evaluation)

**Output Example**

The simulate\_rw function will return a DataFrame (dtf) that includes:

* ts: The actual values of the time series.
* model: The simulated random walk values based on the training data.
* forecast: The forecasted random walk values for the test data.
* lower and upper: The confidence intervals for the forecast.
* residuals: The difference between the actual and model values.
* error: The difference between the actual and forecast values.

Additionally, plots showing the time series, forecasted values, residuals, and error distributions will be displayed.

**Conclusion**

The simulate\_rw function allows you to simulate a random walk process on a time series, forecast future values, and evaluate the model's performance using common error metrics. This can be useful for modeling and simulating time series data where future values are assumed to follow a random walk.

**Function: forecast\_rw**

**Purpose**

The forecast\_rw function generates future forecasts based on a random walk (RW) model for a given time series (ts). It forecasts future values by simulating a random walk from the last value of the time series and provides confidence intervals for the forecasts. Additionally, it visualizes the forecast, the original time series, and the forecast intervals.

**Parameters**

1. **ts (pandas Series)**
   * The time series data to forecast from. The function uses this series to generate the random walk model and make predictions for future values.
2. **pred\_ahead (int, optional)**
   * The number of periods to forecast ahead. If not specified, the function will only forecast until the end date.
3. **end (str or datetime, optional)**
   * The end date for the forecast. If not specified, the forecast will be determined by the pred\_ahead parameter.
4. **conf (float, optional)**
   * The confidence level for the forecast intervals (default is 0.95). This determines the upper and lower bounds for the forecasted values.
5. **zoom (int, optional)**
   * The number of recent observations to zoom into for the forecast plot (default is 30). It defines the range of the forecast plot to focus on for better visualization.
6. **figsize (tuple, optional)**
   * The size of the plot for visualizing the forecast and intervals (default is (15, 5)).

**Returns**

* **dtf (pandas DataFrame)**
  + A DataFrame containing the original time series (ts), the model-generated random walk (model), the forecasted values (forecast), and the associated confidence intervals (lower, upper), as well as the residuals and forecast errors.

**Process**

1. **Model Fitting**:  
   The function first calculates the difference between consecutive values in the input time series (diff\_ts), which is then used to estimate the standard deviation (sigma). This standard deviation is used to generate a random walk process (rw) starting from the first value of the time series.
2. **Generate Forecast Index**:  
   The function uses the utils\_generate\_indexdate function to generate the forecast index (i.e., future dates) based on the inferred frequency of the input time series.
3. **Simulate Future Values**:  
   The forecasted values (forecast) are generated using the random walk model, starting from the last value of the time series (ts[-1]), and using the same sigma as before.
4. **Combine Forecasts with Actual Data**:  
   The function appends the forecasted values to the original time series, creating a combined DataFrame (dtf) with both historical (ts) and forecasted values (forecast).
5. **Confidence Intervals**:  
   The utils\_add\_forecast\_int function is used to calculate confidence intervals (lower, upper) for the forecasted values, based on the specified confidence level (conf).
6. **Plotting**:  
   The function generates a plot of the entire time series with the forecasted values and their confidence intervals. The function also generates a zoomed-in plot that focuses on the most recent observations for better visualization.

**Code Walkthrough**

def forecast\_rw(ts, pred\_ahead=None, end=None, conf=0.95, zoom=30, figsize=(15,5)):

## fit

# Compute the difference between consecutive values and calculate the standard deviation (sigma)

diff\_ts = ts - ts.shift(1)

sigma = diff\_ts.std()

# Generate a random walk for the training data

rw = utils\_generate\_rw(

y0=ts[0], # Start the random walk at the first value of ts

n=len(ts), # Generate random walk for the same length as the input series

sigma=sigma, # Standard deviation calculated from the difference of values

ymin=ts.min(), # Set lower bound as the minimum value in ts

ymax=ts.max() # Set upper bound as the maximum value in ts

)

# Merge the original time series (ts) with the generated random walk (model)

dtf = ts.to\_frame(name="ts").merge(

pd.DataFrame(rw, index=ts.index, columns=["model"]),

how='left',

left\_index=True,

right\_index=True

)

## index

# Generate future date index for the forecast

freq = ts.index.inferred\_freq[0] # Get the frequency of the input time series

index = utils\_generate\_indexdate(

start=ts.index[-1],

end=end,

n=pred\_ahead,

freq=freq

)

## forecast

# Generate future random walk predictions based on the last value of ts

preds = utils\_generate\_rw(

y0=ts[-1], # Start the random walk at the last value of ts

n=len(index), # Forecast for the length of the generated index

sigma=sigma, # Standard deviation remains the same as calculated earlier

ymin=ts.min(), # Lower bound remains the same as in the original series

ymax=ts.max() # Upper bound remains the same as in the original series

)

# Append forecasted values to the DataFrame

dtf = dtf.append(pd.DataFrame(data=preds, index=index, columns=["forecast"]))

## add intervals and plot

# Add confidence intervals and plot the results

dtf = utils\_add\_forecast\_int(dtf, conf=conf, zoom=zoom)

return dtf # Return the DataFrame containing the forecast and intervals

**Explanation of Key Sections**

1. **Model Fitting**:  
   The function calculates the difference between consecutive values in the time series and computes the standard deviation (sigma). This value is then used to simulate a random walk process (rw).
2. **Generating Future Dates**:  
   The utils\_generate\_indexdate function is used to generate future time points (dates) for which the forecast will be made. The frequency (freq) of the time series is inferred and used to generate the future index.
3. **Forecasting Future Values**:  
   The random walk forecast (preds) is generated starting from the last value of the input time series. This forecast is added to the original time series data.
4. **Confidence Intervals**:  
   Confidence intervals for the forecast are calculated using utils\_add\_forecast\_int. This adds the lower and upper bounds to the forecast, which are essential for uncertainty quantification.
5. **Plotting**:  
   The function generates plots to visualize the forecast, confidence intervals, and the original time series. A zoomed-in plot is also produced to focus on the most recent data points.

**Usage Example**

# Example time series data

import pandas as pd

import numpy as np

# Create synthetic time series data

ts = pd.Series(np.random.randn(100).cumsum(), index=pd.date\_range('20210101', periods=100))

# Forecast the next 50 periods

dtf\_forecast = forecast\_rw(ts, pred\_ahead=50, conf=0.95, zoom=30, figsize=(15, 5))

# Display the forecast DataFrame

print(dtf\_forecast)

**Output Example**

The forecast\_rw function will return a DataFrame (dtf) containing the following columns:

* ts: The original time series values.
* model: The simulated random walk values based on the training data.
* forecast: The forecasted values for the future period.
* lower and upper: The confidence intervals for the forecast.
* residuals: The difference between the actual and model values.
* error: The difference between the actual and forecast values.

Additionally, a set of plots will be displayed showing:

* The entire time series with the forecast.
* The forecast with confidence intervals.
* A zoomed-in view focusing on the last few data points.

**Conclusion**

The forecast\_rw function uses a random walk model to forecast future values for a given time series and provides confidence intervals for these forecasts. It also visualizes the forecast along with the original data and residuals to evaluate the model's performance. This is useful for situations where a simple model like random walk is adequate for forecasting.

The tune\_expsmooth\_model function is designed to perform hyperparameter tuning for an exponential smoothing model (ExponentialSmoothing) on a time series dataset (ts\_train). The function splits the data into training and validation sets, searches through combinations of hyperparameters (trend, damped, and seasonal components), and evaluates the model performance based on a given scoring metric. It then plots the best model alongside others for comparison.

**Function Overview**

**Parameters**

1. **ts\_train (pandas Series)**:
   * The time series training data to model.
2. **s (int, optional)**:
   * The seasonal period (default is 7), which typically represents the number of periods in one seasonal cycle (e.g., 7 days for weekly seasonality).
3. **val\_size (float, optional)**:
   * The proportion of the time series to use as the validation set (default is 0.2).
4. **scoring (function, optional)**:
   * A scoring function to evaluate the model’s performance. By default, it uses mean\_absolute\_error from sklearn.metrics. Other custom scoring functions can be provided as well.
5. **top (int, optional)**:
   * The number of top models to display in the plot (default is None, which will plot all models).
6. **figsize (tuple, optional)**:
   * Size of the plot for visualizing the models (default is (15, 5)).

**Returns**

* **dtf\_search (pandas DataFrame)**:
  + A DataFrame containing the combinations of hyperparameters, their corresponding performance scores, and the trained models. The DataFrame is sorted based on the score in ascending order (best score is the smallest).

**Process Breakdown**

1. **Data Splitting**: The time series data (ts\_train) is split into two parts: the training set (dtf\_fit) and the validation set (dtf\_val). The validation set is determined by the val\_size parameter.
2. **Hyperparameter Grid Search**:
   * The function iterates through combinations of three hyperparameters:
     + **trend**: 'add', 'mul', or None.
     + **damped**: True or False.
     + **seasonal**: 'add', 'mul', or None.
   * The function creates and fits an ExponentialSmoothing model for each combination, and forecasts for the length of the validation set.
3. **Model Evaluation**:
   * The model’s performance is evaluated using the specified scoring function (default is mean\_absolute\_error).
   * For each combination, the forecast is appended to the validation set (dtf\_val), and the corresponding score is recorded.
4. **Best Model Selection**:
   * The models are sorted by their performance scores, and the best model (with the lowest score) is selected.
   * The dtf\_val DataFrame is updated to highlight the best model.
5. **Plotting**:
   * A plot is generated with two subplots:
     + **Main plot**: Shows the training time series, the validation set, and all model predictions.
     + **Zoomed-in plot**: Focuses on the validation set and highlights the best model, with other models plotted for comparison.
   * Different colors are used for each model in the plots.

**Code Walkthrough**

def tune\_expsmooth\_model(ts\_train, s=7, val\_size=0.2, scoring=None, top=None, figsize=(15,5)):

## split

# Split the time series into training and validation sets based on val\_size

dtf\_fit, dtf\_val = model\_selection.train\_test\_split(ts\_train, test\_size=val\_size, shuffle=False)

dtf\_fit, dtf\_val = dtf\_fit.to\_frame(name="ts"), dtf\_val.to\_frame(name="ts")

## scoring

# If scoring is not provided, use mean\_absolute\_error as the default

scoring = metrics.mean\_absolute\_error if scoring is None else scoring

## hyperparameter space

# Define the possible values for trend, damped, and seasonal components

trend = ['add', 'mul', None]

damped = [True, False]

seasonal = ['add', 'mul', None]

## grid search

# Get the frequency of the time series (e.g., daily, weekly)

freq = ts\_train.index.inferred\_freq[0]

# Initialize a DataFrame to store the results of the grid search

dtf\_search = pd.DataFrame(columns=["combo", "score", "model"])

combinations = []

# Iterate over all combinations of hyperparameters

for t in trend:

for d in damped:

for ss in seasonal:

combo = f"trend={t}, damped={d}, seas={ss}"

if combo not in combinations:

combinations.append(combo)

try:

# Fit the ExponentialSmoothing model with the current combination of hyperparameters

model = smt.ExponentialSmoothing(dtf\_fit, trend=t, damped=d, seasonal=ss, seasonal\_periods=s, freq=freq).fit()

# Forecast the validation set

pred = model.forecast(len(dtf\_val))

# If no missing values in the forecast, compute the score

if pred.isna().sum() == 0:

dtf\_val[combo] = pred.values

score = scoring(dtf\_val["ts"].values, dtf\_val[combo].values)

# Append the result to the search DataFrame

dtf\_search = dtf\_search.append(pd.DataFrame({"combo":[combo],"score":[score],"model":[model]}))

except:

continue

## find best model

# Sort the results based on the score (lower is better)

dtf\_search = dtf\_search.sort\_values("score").reset\_index(drop=True)

# Get the best combination

best = dtf\_search["combo"].iloc[0]

# Rename the best model in the validation set

dtf\_val = dtf\_val.rename(columns={best: best + " [BEST]"})

dtf\_val = dtf\_val[["ts", best + " [BEST]"] + list(dtf\_search["combo"].unique())[1:]]

## plot

# Create a plot to compare the models

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=figsize)

fig.suptitle("Model Tuning", fontsize=15)

# Get the top N models to display

combos = dtf\_val.drop("ts", axis=1).columns[:top]

# Define colors for the plot

if (len(combos) <= 7) or ((top is not None) and (top <= 7)):

colors = ["red","blue","green","violet","sienna","orange","yellow"]

else:

colors = [tuple(np.random.rand(3,)) for i in range(len(combos))]

### main plot (training and validation comparison)

ts\_train.plot(ax=ax[0], grid=True, color="black", legend=True, label="ts")

ax[0].fill\_between(x=dtf\_fit.index, y1=ts\_train.max(), color='grey', alpha=0.2)

dtf\_val[combos].plot(grid=True, ax=ax[0], color=colors, legend=True)

ax[0].legend(loc="upper left")

ax[0].set(xlabel=None)

### zoomed-in plot (focus on the validation set)

dtf\_val["ts"].plot(grid=True, ax=ax[1], color="black", legend=False)

for i, col in enumerate(combos):

linewidth = 2 if col == best + " [BEST]" else 1

dtf\_val[col].plot(grid=True, ax=ax[1], color=colors[i], legend=False, linewidth=linewidth)

ax[1].set(xlabel=None)

plt.show()

return dtf\_search # Return the search results (hyperparameters, scores, and models)

**Explanation of Key Sections**

1. **Data Splitting**:  
   The function splits the original time series into training and validation datasets. This ensures that the model is trained on one portion of the data and validated on a separate portion.
2. **Grid Search**:  
   It iterates through all possible combinations of hyperparameters (trend, damped, and seasonal) and fits an ExponentialSmoothing model for each combination. It forecasts the validation set and computes the score using the provided scoring function (default is mean\_absolute\_error).
3. **Best Model Selection**:  
   After performing the grid search, the best model is selected based on the score, and the corresponding predictions are highlighted in the validation set.
4. **Plotting**:  
   The function plots the training and validation sets, compares the forecasts from different models, and zooms in on the most recent predictions for better clarity.

**Example Usage**

import pandas as pd

import numpy as np

# Create synthetic time series data (daily data with seasonality)

np.random.seed(0)

ts = pd.Series(np.random.randn(200).cumsum(), index=pd.date\_range('20210101', periods=200))

# Tune the exponential smoothing model

dtf\_search\_results = tune\_exps

mooth\_model(ts, s=7, val\_size=0.2, top=5) print(dtf\_search\_results)

---

### \*\*Conclusion\*\*

The `tune\_expsmooth\_model` function is a useful tool for hyperparameter tuning of exponential smoothing models. It automatically splits the time series into training and validation sets, searches for the best combination of trend, damped, and seasonal components, and visualizes the results. This approach helps find the best model for time series forecasting by evaluating multiple configurations.

The fit\_expsmooth function is used to fit an Exponential Smoothing model (specifically the Holt-Winters model) to a time series, make forecasts, and evaluate the model's performance. It allows users to specify different configurations for trend, seasonal components, and damping, and then provides evaluation and visualization of the model fit and its forecast.

**Function Breakdown**

**Parameters**

1. **ts\_train (pandas Series)**:
   * The training time series data used to fit the model.
2. **ts\_test (pandas Series)**:
   * The test time series data used to evaluate the model’s forecasting performance.
3. **trend (str, optional)**:
   * Type of trend to apply in the model, can be 'additive' or 'multiplicative' (default is 'additive').
4. **damped (bool, optional)**:
   * If True, the trend will be damped, which means that the trend level will gradually decrease over time. Default is False.
5. **seasonal (str, optional)**:
   * Type of seasonal component to apply ('additive' or 'multiplicative'). If no seasonality is required, set to None.
6. **s (int, optional)**:
   * The seasonal period (i.e., how often the seasonal cycle repeats). For example, for weekly data, this might be 7 (days of the week). Required if seasonal is not None.
7. **factors (tuple, optional)**:
   * A tuple containing the smoothing factors for level, trend, and seasonality. Default is (None, None, None) which means the model will estimate these parameters.
8. **conf (float, optional)**:
   * Confidence level for the prediction intervals (default is 0.95).
9. **figsize (tuple, optional)**:
   * Size of the plot for visualizing the model fit and forecast (default is (15,10)).

**Returns**

* **dtf (pandas DataFrame)**:
  + A DataFrame containing the time series (ts), the fitted values (model), the forecasted values (forecast), and any evaluation metrics.
* **model (ExponentialSmoothing model)**:
  + The fitted ExponentialSmoothing model object.

**Process Breakdown**

1. **Seasonality Check**:  
   The function checks whether seasonality is specified and prints information about the seasonal parameters. If no seasonality is specified (seasonal=None and s=None), it prints a message indicating that no seasonal components will be used.
2. **Training the Model**:  
   The function fits an ExponentialSmoothing model using the trend, damped, seasonal, and seasonal period (s) parameters. The smoothing factors for level, trend, and seasonality are provided as factors.
3. **Forecasting**:  
   After fitting the model on the training data (ts\_train), the function uses the model to generate forecasts on the test data (ts\_test).
4. **Appending Results**:  
   The model's fitted values on the training set and the forecasted values on the test set are combined into a single DataFrame dtf, which also includes the original time series (ts).
5. **Model Evaluation**:  
   The function calls utils\_evaluate\_ts\_model to evaluate the model's performance using the specified confidence level (conf) and plot the results. The title of the plot includes the smoothing parameters (alpha, beta, gamma).

**Code Walkthrough**

def fit\_expsmooth(ts\_train, ts\_test, trend="additive", damped=False, seasonal="multiplicative", s=None, factors=(None,None,None), conf=0.95, figsize=(15,10)):

## checks

# Check if seasonality is defined, and print relevant information

check\_seasonality = "Seasonal parameters: No Seasonality" if (seasonal is None) & (s is None) else f"Seasonal parameters: {seasonal} Seasonality every {s} observations"

print(check\_seasonality)

## train

# Get the frequency of the time series

freq = ts\_train.index.inferred\_freq[0]

# Fit the Exponential Smoothing model using the provided parameters

model = smt.ExponentialSmoothing(ts\_train, trend=trend, damped=damped, seasonal=seasonal, seasonal\_periods=s, freq=freq).fit(factors[0], factors[1], factors[2])

# Store the training data and model's fitted values

dtf\_train = ts\_train.to\_frame(name="ts")

dtf\_train["model"] = model.fittedvalues

## test

# Store the test data and generate forecast using the fitted model

dtf\_test = ts\_test.to\_frame(name="ts")

dtf\_test["forecast"] = model.predict(start=len(ts\_train), end=len(ts\_train)+len(ts\_test)-1)

## evaluate

# Combine training and test data into one DataFrame

dtf = dtf\_train.append(dtf\_test)

# Get the smoothing parameters (alpha, beta, gamma)

alpha, beta, gamma = round(model.params["smoothing\_level"], 2), round(model.params["smoothing\_slope"], 2), round(model.params["smoothing\_seasonal"], 2)

# Call the utility function to evaluate and plot the model

dtf = utils\_evaluate\_ts\_model(dtf, conf=conf, figsize=figsize, title=f"Holt-Winters {(alpha, beta, gamma)}")

return dtf, model # Return the evaluation results and the fitted model

**Explanation of Key Sections**

1. **Seasonality Check**:
   * The function first checks whether seasonality (seasonal) and seasonal period (s) are provided. If not, it prints a message that no seasonality will be applied.
2. **Model Fitting**:
   * The model is created using smt.ExponentialSmoothing(), which is part of the statsmodels library.
   * The parameters trend, damped, and seasonal are used to configure the model. If no factors are provided, the model will estimate them automatically.
3. **Forecasting**:
   * After fitting the model, the predict() method is used to forecast values for the test set (ts\_test).
4. **Model Evaluation**:
   * The evaluation function utils\_evaluate\_ts\_model is called to plot the fitted model and forecast, along with some model evaluation metrics.
5. **Returning Results**:
   * The function returns a DataFrame dtf containing the original time series, fitted values, forecasted values, and possibly some evaluation metrics.
   * It also returns the fitted model object (model), which can be used for further analysis or forecasting.

**Example Usage**

import pandas as pd

import numpy as np

# Create synthetic training and test data (daily data with some seasonality)

np.random.seed(0)

ts\_train = pd.Series(np.random.randn(100).cumsum() + 10, index=pd.date\_range('20210101', periods=100))

ts\_test = pd.Series(np.random.randn(20).cumsum() + 10, index=pd.date\_range('20210410', periods=20))

# Fit the Holt-Winters model

dtf, model = fit\_expsmooth(ts\_train, ts\_test, trend="additive", damped=False, seasonal="multiplicative", s=7, conf=0.95, figsize=(15,10))

# Display the evaluation results and model details

print(dtf.head())

**Conclusion**

The fit\_expsmooth function is a powerful tool for fitting and forecasting time series data using the Holt-Winters Exponential Smoothing model. It provides flexibility in defining the trend, seasonal components, and damping, and also includes evaluation and visualization of the results. This makes it a great choice for forecasting tasks that require time series models with trend and seasonality.

The tune\_arima\_model function performs hyperparameter tuning for an ARIMA (AutoRegressive Integrated Moving Average) or SARIMA (Seasonal ARIMA) model by conducting a grid search over specified ranges for the AR (p), I (d), MA (q) orders, and their seasonal counterparts. It evaluates the models using a specified scoring function and returns the best-performing model based on the score.

**Function Breakdown**

**Parameters**

1. **ts\_train (pandas Series)**:
   * The training time series data used to fit the models.
2. **s (int, optional)**:
   * The seasonal period. For example, if your data is weekly, you would set s=7 for weekly seasonality. Default is 7.
3. **val\_size (float, optional)**:
   * The proportion of the training data to use for validation. Default is 0.2 (20% of data used for validation).
4. **max\_order (tuple, optional)**:
   * The maximum values for the AR (p), I (d), and MA (q) orders. The search space will be all combinations from (0, 0, 0) to (max\_order[0], max\_order[1], max\_order[2]). Default is (3, 1, 3).
5. **seasonal\_order (tuple, optional)**:
   * The seasonal AR, seasonal I, and seasonal MA orders (P, D, Q). Default is (1, 1, 1).
6. **scoring (function, optional)**:
   * The function used to evaluate the model’s performance. Default is metrics.mean\_absolute\_error, but it can be replaced with another function such as mean\_squared\_error, etc.
7. **top (int, optional)**:
   * The number of best models to plot (based on the score). Default is None, meaning all models will be plotted.
8. **figsize (tuple, optional)**:
   * The size of the plot (default is (15,5)).

**Returns**

* **dtf\_search (pandas DataFrame)**:
  + A DataFrame containing the evaluated models with their corresponding hyperparameter combinations, scores, and model objects.

**Process Breakdown**

1. **Train-Test Split**:  
   The time series data (ts\_train) is split into training and validation sets using train\_test\_split from model\_selection. The val\_size parameter controls the proportion of data used for validation.
2. **Scoring Function**:  
   The function allows the user to specify a scoring metric (e.g., mean\_absolute\_error). If no scoring function is provided, it defaults to mean\_absolute\_error.
3. **Grid Search Over Hyperparameters**:  
   The function performs a grid search over possible combinations of the AR (p), I (d), and MA (q) orders, as well as the seasonal AR (P), seasonal I (D), and seasonal MA (Q) orders. These values are explored within the given bounds (max\_order and seasonal\_order). The function fits a SARIMA model for each combination and forecasts the validation set.
4. **Model Evaluation**:  
   Each model is evaluated by forecasting the dtf\_val data and calculating the scoring metric (e.g., mean absolute error). These results are stored in a DataFrame (dtf\_search), along with the corresponding hyperparameters and model objects.
5. **Plotting Results**:  
   The function plots the best models, visualizing the training data, forecasted values, and error. The best model is identified by the lowest score, and it is labeled as the [BEST] model on the plot.
6. **Return Results**:  
   The function returns a DataFrame (dtf\_search) with the model hyperparameters, scores, and fitted models, allowing the user to examine which configurations performed best.

**Code Walkthrough**

def tune\_arima\_model(ts\_train, s=7, val\_size=0.2, max\_order=(3,1,3), seasonal\_order=(1,1,1), scoring=None, top=None, figsize=(15,5)):

## split

# Split the training time series into fitting and validation sets

dtf\_fit, dtf\_val = model\_selection.train\_test\_split(ts\_train, test\_size=val\_size, shuffle=False)

dtf\_fit, dtf\_val = dtf\_fit.to\_frame(name="ts"), dtf\_val.to\_frame(name="ts")

## scoring

# Set default scoring function if none provided

scoring = metrics.mean\_absolute\_error if scoring is None else scoring

## hyperamater space

# Define the search space for ARIMA orders

ps = range(0, max\_order[0]+1)

ds = range(0, max\_order[1]+1)

qs = range(0, max\_order[2]+1)

Ps = range(0, seasonal\_order[0]+1)

Ds = range(0, seasonal\_order[1]+1)

Qs = range(0, seasonal\_order[2]+1)

## grid search

# Initialize a DataFrame to store search results

dtf\_search = pd.DataFrame(columns=["combo", "score", "model"])

combinations = []

# Loop over all combinations of ARIMA hyperparameters

for p in ps:

for d in ds:

for q in qs:

for P in Ps:

for D in Ds:

for Q in Qs:

# Create a unique combination string for the hyperparameters

combo = f"({p},{d},{q})x({P},{D},{Q})"

if combo not in combinations:

combinations.append(combo)

try:

### fit SARIMA model

model = smt.SARIMAX(ts\_train, order=(p,d,q), seasonal\_order=(P,D,Q,s)).fit()

### forecast the validation set

pred = model.forecast(len(dtf\_val))

if pred.isna().sum() == 0:

# Store predictions and score in the DataFrame

dtf\_val[combo] = pred.values

score = scoring(dtf\_val["ts"].values, dtf\_val[combo].values)

dtf\_search = dtf\_search.append(pd.DataFrame({"combo": [combo], "score": [score], "model": [model]}))

except:

continue

## find best model

# Sort the models based on score (lower is better)

dtf\_search = dtf\_search.sort\_values("score").reset\_index(drop=True)

best = dtf\_search["combo"].iloc[0]

dtf\_val = dtf\_val.rename(columns={best: f"{best} [BEST]"})

dtf\_val = dtf\_val[["ts", f"{best} [BEST]"] + list(dtf\_search["combo"].unique())[1:]]

## plot results

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=figsize)

fig.suptitle("Model Tuning", fontsize=15)

combos = dtf\_val.drop("ts", axis=1).columns[:top]

# Set colors for plotting

colors = ["red", "blue", "green", "violet", "sienna", "orange", "yellow"] if (len(combos) <= 7) or ((top is not None) and (top <= 7)) else [tuple(np.random.rand(3,)) for i in range(len(combos))]

### main plot

ts\_train.plot(ax=ax[0], grid=True, color="black", legend=True, label="ts")

ax[0].fill\_between(x=dtf\_fit.index, y1=ts\_train.max(), color='grey', alpha=0.2)

dtf\_val[combos].plot(grid=True, ax=ax[0], color=colors, legend=True)

ax[0].legend(loc="upper left")

ax[0].set(xlabel=None)

### zoomed plot

dtf\_val["ts"].plot(grid=True, ax=ax[1], color="black", legend=False)

for i, col in enumerate(combos):

linewidth = 2 if col == f"{best} [BEST]" else 1

dtf\_val[col].plot(grid=True, ax=ax[1], color=colors[i], legend=False, linewidth=linewidth)

ax[1].set(xlabel=None)

plt.show()

return dtf\_search

**Explanation of Key Sections**

1. **Grid Search Over ARIMA Hyperparameters**:  
   The function loops over all combinations of ARIMA and SARIMA hyperparameters, including p, d, q (non-seasonal orders) and P, D, Q (seasonal orders).
2. **Model Fitting and Forecasting**:  
   The SARIMAX model (smt.SARIMAX) is fitted to the training data (ts\_train) with each combination of hyperparameters. The forecast is generated for the validation set (dtf\_val).
3. **Scoring and Storing Results**:  
   The model is evaluated using the specified scoring function (default is mean\_absolute\_error), and results are stored in a DataFrame dtf\_search along with the corresponding model.
4. **Best Model Selection**:  
   The best model is selected based on the lowest score, and

it is marked as [BEST] in the plot and DataFrame.

1. **Plotting**:  
   Two plots are generated:
   * **Main plot**: Shows the training data, forecasted values for the best models, and confidence intervals.
   * **Zoomed plot**: Focuses on a smaller region with detailed comparisons between the best models and their performance.

**Example Usage**

import pandas as pd

import numpy as np

# Generate synthetic time series data (daily data with seasonality)

np.random.seed(0)

ts\_train = pd.Series(np.random.randn(100).cumsum() + 10, index=pd.date\_range('20210101', periods=100))

# Tune the ARIMA model

dtf\_search = tune\_arima\_model(ts\_train, s=7, val\_size=0.2, max\_order=(3,1,3), seasonal\_order=(1,1,1), scoring=None, top=5)

# View the tuned models and their scores

print(dtf\_search.head())

**Conclusion**

The tune\_arima\_model function automates the process of tuning ARIMA/SARIMA models by performing a grid search over hyperparameters, evaluating the models using a specified scoring function, and plotting the results. This allows for easy identification of the best model configuration for time series forecasting.