Time Series

ARIMA, SARIMA, Exponential Smoothing
Business applications
Math behind

Step-by-step process:
preparation
evaluation
fine-tuning

Julia Lenc

Analytics Journey

Business Analytics (BA)

1. Intro: Business and Revenue models. KPIs
2. Business models translated into analytics
3. Techniques: Descriptive, Diagnostic, Predictive, Prescriptive

Diagnostic Techniques

1. Inference: hypotheses testing
2. Unsupervised Learning: clustering, dimensionality reduction, anomalies

Predictive Techniques

1. Supervised learning: overview
2. Preparation: data pre-processing
3. Foundations: model choice and evaluation
4. Regression: linear and non-linear
5. Classification: logistic regression, Naive Bayes, k-NNs
6. Time series: ARIMA, SARIMA, Exponential Smoothing (this presentation!)
7. Advanced: Decision Trees, SVM
8. Ensemble: bagging, boosting, stacking
9. Neural Networks: FFNN, CNN, RNN, Transformers

Prescriptive Techniques

1. Optimization: Linear, Non-linear and Dynamic programming
2. Simulation: Monte Carlo, Discreet Events, System Dynamics
3. Probabilistic Sequence: Markov Chains, Markov Decision Processes
4. Reinforcement Learning: Q-Learning, Deep RL, Policy Gradient

Julia Lenc

What is Time Series Forecasting?

Definition

Time Series is a **regression** model designed to predict future values using patterns observed in past data over time. It focuses on data with a **time dimension** - every observation depends on when it occurred.

The key model for:

1. Supply chain: short-term demand forecasting, inventory.
2. Website traffic: predicting daily visitors.
3. Energy usage: forecasting electricity demand.

Julia Lenc

Time Series vs Factor-Based Regression

Time dependency matters

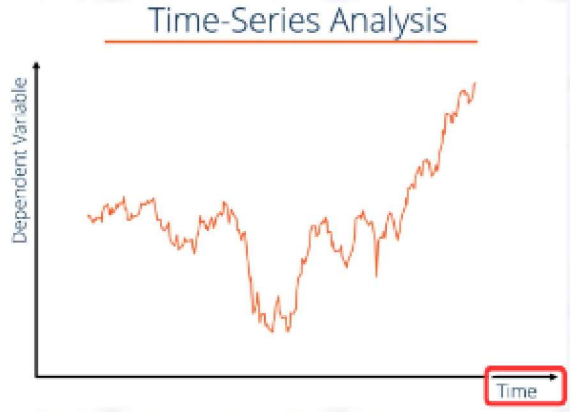
- In time series, order is critical ("yesterday affects today").
- In purely factor-based regression, order doesn't matter.

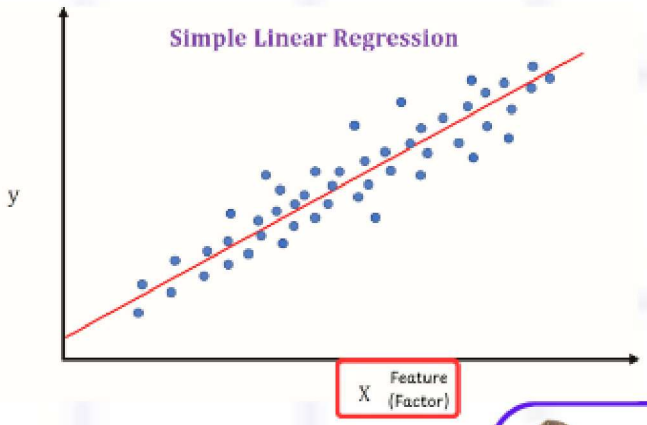
Autocorrelation

- Time series assumes past values influence future values
- Factor-based regression uses independent variables (a must!)

Special components

- Time series has trends, seasonality, and random noise.
- Factor-based regression doesn't directly account for sequences.

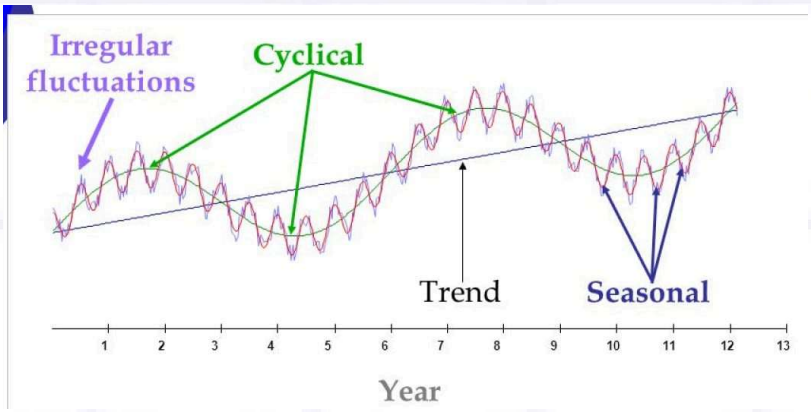




Julia Lenc

4 components of TS model

Time Series Model Components



- **Trend**: the direction in which the time series data changes over time.
- **Cycle**: long-term, non-fixed fluctuations often driven by economic or market cycles.
- **Seasonality:** regular, repeating patterns or cycles in the data (e.g., weekly, monthly, or yearly sales spikes).
- **Irregular fluctuations (noise):** random, unpredictable variability in the original time series that remains after decomposing it into trend, seasonality and cyclic components.

Visual: Peerdh


Julia Lenc

Core Models

ARIMA (AutoRegressive Integrated Moving Average)

- Captures relationships between lags of a time series and removes noise through stationarity and differencing.
- For **univariate** datasets with **no strong seasonality** but a clear **trend.**

SARIMA (Seasonal ARIMA)

- Extends ARIMA to account for seasonal patterns (e.g., weekly).
- For **univariate** datasets where **seasonality** and **trend** are present.

Exponential Smoothing

- Focuses on level, trend and seasonality by weighting **more recent** observations heavily.
- For datasets with **reliable seasonal or trend** components, especially when smoother, faster forecasts are needed.

Business Applications

ARIMA (AutoRegressive Integrated Moving Average)

For datasets **without significant seasonal** patterns, focusing on capturing trends and short-term dependencies.

- Financial forecasting (exchange rates, stock prices)
- Demand forecasting (production, supply chain)
- Traffic and transportation management

SARIMA (Seasonal ARIMA)

Best suited for datasets exhibiting strong **seasonality**, effectively modeling recurring seasonal cycles alongside trends.

- Hospitality and tourism
- Energy consumption
- Retail sales

Exponential Smoothing

For simpler, fast-moving forecasts where **recent data** heavily influences predictions, and robust trends or seasonality are present but need **smoothing** rather than deep statistical analysis. **Short-term**.

- Short-term demand planning
- Resources allocation

Source: Geeks for Geeks

Mathematical explanation

ARIMA: the foundation

$\text{ARIMA}(p, d, q)$ ARIMA is a function with 3 parameters

1. **AR (AutoRegressive)**: relationship between past values and the current value

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \epsilon_t$$

ϕ is a coefficient multiplying the lagged values of Y

p defines the number of past lags. If p=2 : $Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \epsilon_t$

2. **I (Integrated)**: the process of making time series stationary by **differencing**.

A stationary series has a constant mean and variance over time, making it easier to model. Differencing is a part of pre-processing, not ARIMA formula.

d shifts ARIMA's focus from absolute values to changes in values

if d=0, Y_t will be predicted if d=1, $Y_t - Y_{t-1}$ will be predicted

if d=2, $(Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2})$ will be predicted

3. **Moving average component (MA)**: use of past error terms to forecast.

$$Y_t = \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \epsilon_t$$

θ is a coefficient multiplying the lagged forecast errors

q defines the number of past lags. If q=2 : $Y_t = \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \epsilon_t$

Mathematical explanation

ARIMA: *determining p, d, q*

Partial Autocorrelation Function (PACF)

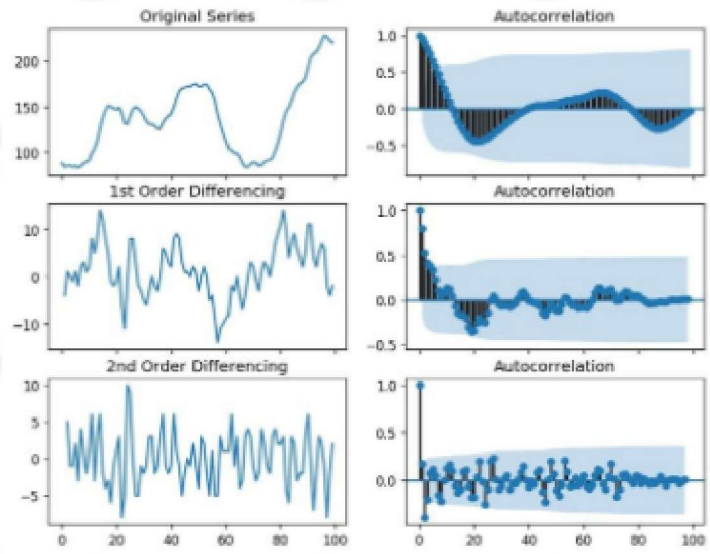
For AR models, PACF measures the correlation between Yt and Yt−k (lag k), while removing the influence of intermediate lags. A sharp cut-off of correlation certain lag **indicates p**.

Differenting

Stabilizing mean and variance over time by predicting not Y_t , but $Y_t - Y_{t-1}$ (d=1), $(Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2})$ (d=2), etc.

Augmented Dickey-Fuller (ADF) test.

Stabilization of mean and variance **indicate d**.



Autocorrelation Function (ACF)

For MA models, ACF measures correlation between a time series and its lagged version. ACF typically shows a sharp cut-off after a certain lag (**indicates q**) and zero correlation for later lags.

Mathematical explanation

ARIMA: the entire formula

$\text{ARIMA}(p, d, q)$ ARIMA is a function with 3 parameters

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \epsilon_t$$

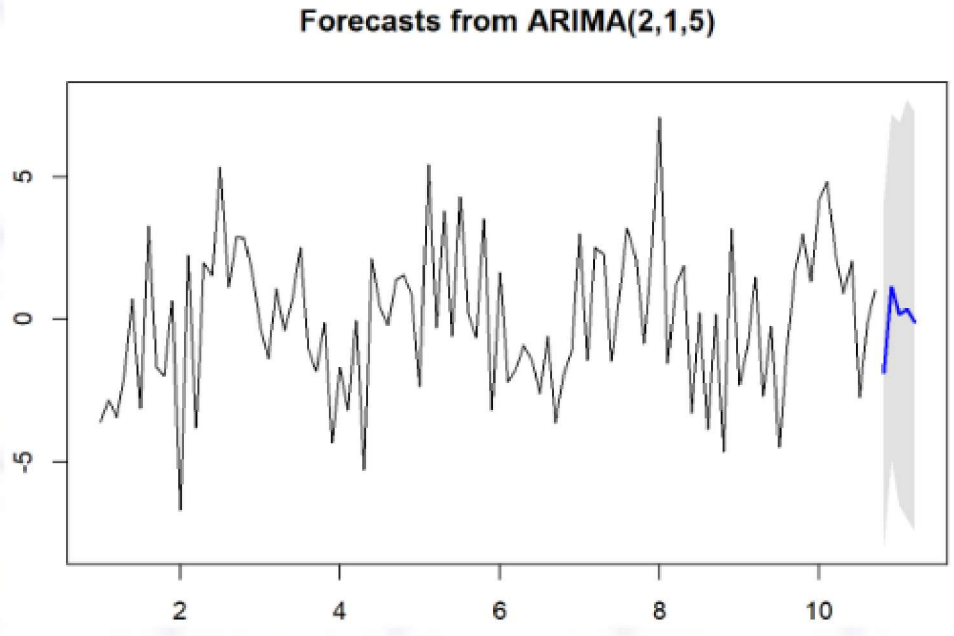
Yt : differenced version of your time series data after **d** transformations.

c : constant (optional) term, representing the mean of the series (if included).

ϕ : coefficients of the **AR process**. Related to past values. Defined on basis of **p**.

θ : coefficients of the **MA process**. Related to past errors. Defined on basis of **q**.

ϵt : error/residual at time t : the part that's unexplained by AR + MA.



p=2: AR depends on two last lags

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \epsilon_t$$

d=1: the time series has been differenced once and $Y_t - Y_{t-1}$ will be predicted

q=5: MA uses five past errors


Julia Lenc

Mathematical explanation

SARIMA: seasonal ARIMA

- ARIMA assumes stationarity in data by removing linear trends.
- SARIMA extends ARIMA by explicitly modeling **seasonal patterns**, which ARIMA cannot handle well with just linear differencing.

#	Feature	ARIMA	SARIMA
1	**Full Name**	AutoRegressive Integrated Moving Average	Seasonal AutoRegressive Integrated Moving Average
2	**Architecture**	Statistical, Linear	Statistical, Linear
3	**Nonlinearity**	No	No
4	**Seasonality**	No	Yes
5	**Model Parameters**	p, d, q (AR, I, MA)	p, d, q (AR, I, MA), P, D, Q, m (seasonal)
6	**Data Requirements**	Small to Moderate	Small to Moderate
7	**Training Time**	Fast	Fast
8	**Stationarity**	Requires Stationary (or Differenced)	Requires Stationary (or Differenced)
9	**Interpretability**	High	Moderate
10	**Multivariate Support**	No	No
11	**Best For**	Univariate Time Series with Trends	Univariate Time Series with Seasonality
12	**Use Cases**	Stock Market, Economy	Monthly Sales, Temperature, Energy Demand

Table from: Farzad Nobar. Time Series - ARIMA vs. SARIMA

Julia Lenc

Mathematical explanation

Explaining SARIMA through comparison with ARIMA

1. We know that **ARIMA** is a function with **3 parameters** $\mathrm{ARIMA}(p,d,q)$

2. SARIMA is a function with **7 parameters** $SARIMA(p,d,q)(P,D,Q,s)$

p, d, q : same as in ARIMA

- p : Order of the Auto-Regressive (AR) term - lags of the series.
- d : Degree of differencing to make the data stationary.
- q : Order of the Moving Average (MA) term - lags of the error terms.

P, D, Q : seasonal terms

- **P :** number of significant seasonal AR terms.

  E.g., PACF spikes at lags 12, 24... for s=12.
- **D :** seasonal differencing removes repeated patterns of length s

  E.g., Yt−Yt−12 for s=12.
- **Q :** seasonal MA order models residual seasonality (e.g., ACF spikes)

Similarly to ARIMA, use PACF to calculate P, ADF test for D and ACF for Q.

s: length of the season

s=12: seasonal frequency for monthly data (1 cycle = 1 year = 12 months)

s=4: seasonal frequency for quarterly data (1 cycle = 1 year = 4 quarters)

See next slide to understand full SARIMA formula

Julia Lenc

Mathematical explanation

Explaining SARIMA through comparison with ARIMA

1. This is **generic (expanded) ARIMA** formula

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \epsilon_t$$

2. This is **compact (non-expanded) ARIMA** formula

$$\phi(B)(1-B)^d Y_t = \theta(B)\epsilon_t$$

ϕ(B) : Non-seasonal Auto-Regressive **(AR)** structure.

(1−B)d : Non-seasonal **differencing** for trend removal.

θ(B) : Non-seasonal Moving Average **(MA)** structure.

Yt : Target variable being forecasted.

ϵt : Random error (or residual).

3. This is **compact SARIMA** formula

$$\phi(B)\Phi(B^s)(1-B)^d(1-B^s)^D Y_t = \theta(B)\Theta(B^s)\epsilon_t$$

ϕ(B) : non-seasonal Auto-Regressive **(AR)** structure. Like in **ARIMA**

Φ(Bs) : seasonal Auto-Regressive **(SAR)** structure. SARIMA-specific

(1−B)d : non-seasonal **differencing**. Like in **ARIMA**

(1−Bs)D : seasonal **differencing**. SARIMA-specific

θ(B) : non-seasonal Moving Average **(MA)** structure. Like in **ARIMA**

Θ(Bs) : seasonal Moving Average **(SMA)** structure. SARIMA-specific

Yt : target variable being forecasted.

ϵt : random error (or residual).

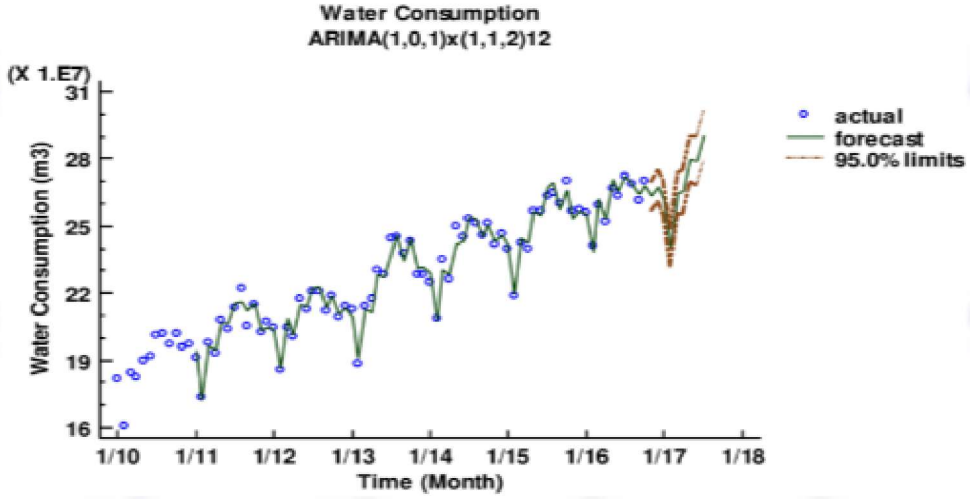
See next slide to understand SARIMA through via example

Mathematical explanation

Decomposing SARIMA

We plug the **arguments**. SARIMA(p, d, q) (P,D,Q,s) → SARIMA (1, 0, 1) (1, 1, 2, 12)

$$(1-\Phi_1 B^{12})(1-\phi_1 B)(1-B^{12})Y_t = (1+\Theta_1 B^{12}+\Theta_2 B^{24})(1+\theta_1 B)\epsilon_t$$



$(1-\phi_1 B)$ and $(1+\theta_1 B)$ **non-seasonal** components. They show that water consumption is growing.

$(1-\phi_1 B)$ explains how the current month's water consumption depends on the **previous month**

$(1+\theta_1 B)$ adjusts for random shocks or errors in water consumption from the **previous month**

$(1-\Phi_1 B^{12})$, $(1-B^{12})$ and $(1+\Theta_1 B^{12}+\Theta_2 B^{24})$

Seasonal components. They show summer peaks (esp. July) and winter drops

$(1-\Phi_1 B^{12})$ explains how this year's water consumption at the same time of year (lag 12 months) depends on **last year's** water consumption during the **same month**

$(1-B^{12})$ removes the **annual seasonality**

$\Theta_1 B^{12}$ models how random shocks or **unusual events** in water consumption

$\Theta_2 B^{24}$ at seasonal intervals (12 and 24 months) influence the current value

See next slide to for another method: Exponential Smoothing

Mathematical explanation

Exponential Smoothing

A time series forecasting method that gives exponentially decreasing weights to older observations, emphasizing recent data.

Simple Exponential Smoothing (SES)

$$S_t = \alpha Y_t + (1 - \alpha) S_{t-1}$$

St: smoothed value (forecast) at time t .

Yt: actual observation at time t .

St−1: smoothed value from the previous time period.

α (0 < α ≤ 1): smoothing factor (controls how much weight is given to recent data:

- Close to 1 → more weight on recent data (reacts quickly to changes).
- Close to 0 → more weight on historical trends (smoother forecasts).

Double Exponential Smoothing (Holt): captures trends

$$S_t = \alpha Y_t + (1 - \alpha)(S_{t-1} + T_{t-1})$$
$$T_t = \gamma(S_t - S_{t-1}) + (1 - \gamma) T_{t-1}$$

St: Smoothed value.

Tt : Trend component.

γ : Trend smoothing factor.

Triple Exponential Smoothing (Holt-Winters): captures seasonality

$$S_t = \alpha \frac{Y_t}{C_{t-p}} + (1 - \alpha)(S_{t-1} + T_{t-1})$$

Ct−p : Seasonal component for period p

Julia Lenc

The process: Stage 1

Understand and formulate the business question as:

Specific: Define the problem clearly

- What is the business metric? (e.g., water consumption, sales, demand)
- What do we want to predict? (e.g., next month, next year?)

Measurable: Determine evaluation criteria (e.g., MSE).

Achievable: Ensure data and resources match the modeling goals.

Relevant: Validate that the forecast will drive action (e.g., demand forecasting → resource planning, inventory)

Time-bound: Identify stakeholders and key deadlines for delivery.

Julia Lenc

The process: Stage 2

Prepare the data:

1. **Data Cleansing** (duplicates, missing values, outliers, formatting). See how.

2. **Transformation**: make data stationary.

Time series may have trends or seasonality that need to be corrected to ensure stationarity (i.e., constant mean and variance over time):

Detrending: remove trends using differencing: $y_t' = y_t - y_{t-1}$

Deseasonalizing: remove repeating seasonal patterns (if considering SARIMA).

Check stationarity using statistical tests like: Augmented Dickey-Fuller (ADF), KPSS test.

Timing note: stationarity adjustments are often applied later (while building the model). If unsure, this can be performed after exploratory analysis (EDA).

3. **Add Lag Features/Time-Based Features**:

Add **lag features** (prior values of the series, e.g., y_{t-1} , y_{t-12}) to capture historical influence.

Add **seasonality indicators** (e.g., month number, day of the week).

Create e**xogenous variables** if they influence y_t : holidays, promotions, weather data, etc.

4. **Scaling and Normalization**:

Several models work better with scaled data. Use **MinMaxScaler** or **StandardScaler** to normalize values.

5. **Feature Selection (lagged variables)**:

For exogenous variables (in SARIMAX, Seasonal Exponential Smoothing with inputs)

you need to select only the most relevant predictors.

Julia Lenc

Stage 2

```python
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from statsmodels.tsa.stattools import adfuller  # For stationarity tests

# Step 0: Load the raw data
# Assume "my_raw_data.csv" has columns: "date" (datetime) and "value" (numeric time series data)
df = pd.read_csv("my_raw_data.csv")
df['date'] = pd.to_datetime(df['date'])  # Ensure the date column is in datetime format
df = df.sort_values('date')  # Sort data by time
df.set_index('date', inplace=True)  # Set the date column as the index

# Check the data
print(df.head())  # Quick preview of the data

# Step 1: Data Cleansing is in another presentation, so skipping here...

# Step 2: Transformation (Make Data Stationary) - Relevant for ARIMA/SARIMA
# Check stationarity using the Augmented Dickey-Fuller (ADF) test
result = adfuller(df['value'])
print(f"ADF Test Statistic: {result[0]}")
print(f"P-value: {result[1]}")
if result[1] > 0.05:
    print("Data is not stationary. Applying differencing...")
    df['value_diff_1'] = df['value'] - df['value'].shift(1)  # First-order differencing
    df.dropna(inplace=True)  # Drop NaN values caused by differencing
    # Repeat stationarity test if needed

# Note: Deseasonalizing can be done here if seasonal patterns are present (using differencing or seasonal decompo
ion)

# Step 3: Add Lag Features and Time-Based Features
# Add lag features (prior values of the series)
df['lag_1'] = df['value'].shift(1)  # Lag of 1 month
df['lag_12'] = df['value'].shift(12)  # Lag of 1 year (for monthly data)
# Drop NaN values from lagging
df.dropna(inplace=True)

# Add time-based features (if applicable)
df['month'] = df.index.month  # Extract month (e.g., for seasonal analysis)
df['day_of_week'] = df.index.dayofweek  # Day of week (e.g., Monday=0, Sunday=6)

# Step 4: Scaling and Normalization - Relevant for machine learning models
scaler = MinMaxScaler(feature_range=(0, 1))  # Scale values between 0 and 1
df_scaled = df[['value', 'lag_1', 'lag_12']].copy()  # Keep numeric columns for scaling
df_scaled = pd.DataFrame(scaler.fit_transform(df_scaled), index=df.index, columns=df_scaled.columns)
print(df_scaled.head())  # Check scaled values

# Note: StandardScaler can also be used when normalizing for models like LSTMs or ensuring Gaussian distribution.

# Step 5: Feature Selection - Relevant for models with exogenous variables
# (e.g., SARIMAX or Seasonal Exponential Smoothing with inputs)
# Select lagged variables and exogenous features as needed
# Ensure the exogenous features correlate with the target variable ('value')!

# Summary:
print("Preprocessing complete! Data is ready for modeling.")
# Your final dataframe (df or df_scaled) now includes lagged features, stationary transformations, and scaled va
s.

# Save the preprocessed data for later use
df.to_csv("my_preprocessed_data.csv")
```

Julia Lenc

The process: Stage 3

Validate assumptions using EDA. Choose the model

1. Key Assumptions for ARIMA, SARIMA and Exponential Smoothing

Linearity: visualize residuals/scatter plots. Past values have a linear relationship with the forecast.

If violated → consider more complex models, e.g., LSTM.

Stationarity (ARIMA, SARIMA) :ADF/KPSS tests. Mean, variance and autocorrelation are constant over time.

If violated → Use differencing/deseasonalizing.

Autocorrelation: ACF/PACF plots. The series depends on its lags.

No autocorrelation → simpler models may outperform.

Homoscedasticity: residual plots. Constant variance of errors over time.

If violated → Apply log/Box-Cox transformations or move to ARCH/GARCH.

2. Define Time Dependencies

Trend: overall upward/downward movement (linear/exponential).

Seasonality: cyclical, predictable patterns (e.g., monthly, yearly).

Noise: random fluctuations or shocks.

3. Choose initial model

Linear, stationary data → **ARIMA** (the simplest and the most interpretable)

Linear, stationary data with seasonality → **SARIMA**

Linear, non-stationary data → **Exponential Smoothing**

Non-linear data, complex dynamics → Neural Networks (e.g., **LSTM**)

Volatility (heteroscedasticity) → **ARCH/GARCH** (good for high volatility financial series)

Stage 3.1

```python
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Step 1: Import preprocessed data
df = pd.read_csv("my_preprocessed_data.csv", index_col='date', parse_dates=True)  # Import the file
print(df.head())  # Check the first few rows

# Step 2: Key Assumption Validation

## 2.1 Check Linearity (Visualize residuals or scatter plots if available)
plt.figure(figsize=(10, 5))
df['value'].plot(title="Time Series Plot (Checking for Linearity)")
plt.show()

# If residual plots (or explicit transformations) suggest non-linear patterns, consider Neural Networks
s).

## 2.2 Check Stationarity Using ADF Test (ARIMA/SARIMA require stationarity)
print("\nChecking Stationarity...")
adf_result = adfuller(df['value'])
print(f"ADF Test Statistic: {adf_result[0]}")
print(f"P-value: {adf_result[1]}")
if adf_result[1] > 0.05:
    print(" → Non-stationary! Consider differencing or deseasonalizing.")
else:
    print(" → Stationary! Suitable for ARIMA/SARIMA.")

## 2.3 Check for Autocorrelation (ACF and PACF plots)
plot_acf(df['value'], title="ACF Plot (Autocorrelation)")
plot_pacf(df['value'], title="PACF Plot (Partial Autocorrelation)")
plt.show()

# If no observable patterns in autocorrelation, simpler models (or non-linear models) may be preferable.

## 2.4 Check Homoscedasticity (Residual plots for constant variance)
# Assuming residuals are calculated post-model fitting (or use current values to proxy). Example:
residuals = df['value'] - df['value'].mean()  # Simulate residuals here
plt.figure(figsize=(10, 5))
plt.plot(residuals, label="Residuals")
plt.title("Residual Plot (Check for Homoscedasticity)")
plt.axhline(0, color='red', linestyle='--')
plt.legend()
plt.show()

# If variance increases over time → Apply log transformations or move to ARCH/GARCH models.
```

Julia Lenc

Stage 3.2

This is continuation of the code from previous slide.

It will not work on its own (no reference file)!

```
# Step 3: Define Time Dependencies and Choose a Model

## 3.1 Check for Trend
df['value'].plot(title="Trend Detection", figsize=(10, 5))
plt.show()

## 3.2 Observe Seasonality
# Use month/year groupings or seasonal decomposition (seasonality strength)
df['month'] = df.index.month  # Extract month
monthly_mean = df.groupby('month')['value'].mean()
monthly_mean.plot(kind='bar', title="Seasonality: Mean Monthly Values", figsize=(10, 5))
plt.show()

## 3.3 Define the Model Based on EDA Results
print("\nModel Recommendations Based on EDA:")
if adf_result[1] <= 0.05:  # Stationary
    print(" → ARIMA (Stationary, non-seasonal)")
    print(" → SARIMA if seasonality exists.")
if not residuals.var() or adf_result[1] > 0.05:  # Non-stationary
    print(" → Exponential Smoothing (handles trends/seasonality without stationarity).")
if not residuals.var() or significant_non_linearity_detected:  # Example flag
    print(" → Neural Networks (e.g., LSTM) for non-linearity.")
if "heteroscedasticity_detected":  # Example condition
    print(" → ARCH/GARCH for high volatility.")

---

# Save the recommendations based on EDA
# You can also output summarized findings into a CSV/Markdown table for quick reference
```

Julia Lenc

The process: Stage 4

Estimate the parameters (arguments)

1. Non-seasonal parameters (p, d, q)

p : number of autoregressive (AR) terms. Shows how many past observations influence the current observation.

Identified using PACF (Partial Autocorrelation Function): **number of significant lags in PACF plot → p**.

d : number of times the data is differenced for stationarity. Achieves stationarity by differencing:

d=0 : Data is stationary. d=1 : First differencing removes trend.

d=2 : Second differencing removes quadratic trends.

Validate with **ADF test → use minimum differencing for stationarity**.

q : number of moving average (MA) terms. Captures the dependency of current observation on past residuals.

Identified using ACF (Autocorrelation Function): **number of significant lags in ACF plot → q**.

2. Seasonal parameters (P, D, Q, s)

P : seasonal autoregressive (SAR) terms. **Seasonal patterns in PACF plot at lag multiples of s → P**.

D : seasonal differencing order to remove seasonal trends. Seasonal data is stationary after **differencing by s → D=1** (usually enough).

Q : seasonal moving average (SMA) terms. **Seasonal patterns in ACF plot at lag multiples of s → Q**.

s : easonsal period, based on data frequency of seasonability. s=12 (monthly data), s=4 (quarterly data), s=7 (daily data for weekly seasonality, e.g. peaks on weekends).

Julia Lenc

Stage 4.1

```python
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Step 1: Import preprocessed data
df = pd.read_csv("my_preprocessed_data.csv", index_col='date', parse_dates=True)  # Import the file
print(df.head())  # View the first few rows

# Extract the target time series column (renamed 'value' for simplicity)
series = df['value']

# Step 2: Non-seasonal parameters (p, d, q)

## 2.1 Estimate 'd': Differencing to make the series stationary
# Run the Augmented Dickey-Fuller (ADF) test
print("\nChecking stationarity (ADF Test):")
adf_result = adfuller(series)
print(f"ADF Test Statistic: {adf_result[0]}")
print(f"P-value: {adf_result[1]}")

if adf_result[1] > 0.05:
    print(" → Non-stationary. Apply differencing (d=1).")
    differenced_series = series.diff().dropna()  # First differencing
else:
    print(" → Stationary. d=0.")
    differenced_series = series  # No differencing needed

## 2.2 Estimate 'p': Check PACF for autoregressive terms
print("\nEstimating 'p' (AR terms) via PACF plot:")
plot_pacf(differenced_series, title="PACF Plot", lags=20)
plt.show()

## 2.3 Estimate 'q': Check ACF for moving average terms
print("\nEstimating 'q' (MA terms) via ACF plot:")
plot_acf(differenced_series, title="ACF Plot", lags=20)
plt.show()
```

Julia Lenc

Stage 4.2

This is continuation of the code from previous slide.

It will not work on its own (no reference file)!

```
# Step 3: Seasonal parameters (P, D, Q, s)

## 3.1 Determine seasonality period (s)
# Known periodicity based on domain knowledge
s = 12  # Assuming monthly data
print(f"\nDetermining seasonality: s={s} (e.g., monthly data).")

## 3.2 Estimate 'D': Seasonal differencing for stationarity
# Seasonal difference only if trends repeats every 's' periods
seasonally_differenced = series.diff(s).dropna()

adf_seasonal = adfuller(seasonally_differenced)
print(f"\nSeasonal ADF Test (after differencing by {s} periods):")
print(f"ADF Test Statistic: {adf_seasonal[0]}")
print(f"P-value: {adf_seasonal[1]}")
if adf_seasonal[1] > 0.05:
    print(" → Seasonal pattern still present. D may need adjustment.")
else:
    print(" → Seasonal stationarity achieved. D=1.")

## 3.3 Estimate 'P' and 'Q': Seasonal PACF and ACF
print("\nEstimating seasonal parameters (P, Q):")
print("PACF → P (Seasonal AR), ACF → Q (Seasonal MA)")
plot_pacf(seasonally_differenced, lags=40, title="Seasonal PACF Plot")
plot_acf(seasonally_differenced, lags=40, title="Seasonal ACF Plot")
plt.show()

# Step 4: Summarize parameter selection
print("\nSummary of Parameters:")
print(f"Non-seasonal (p, d, q): Based on PACF/ACF and transformations.")
print(f"Seasonal (P, D, Q, s): P, D, Q from seasonal plots; s={s} (frequency).")

# Note: For final values, you can automate grid search using pmdarima or fine-tune.
```

Julia Lenc

The process: Stage 5

Run the initial model and evaluate

Evaluation metrics

yt = actual value at time t y^t = predicted value at time t n = time points

MAE (Mean Absolute Error): $\mathrm{MAE} = \frac{1}{n}\sum_{t=1}^{n}|y_t - \hat{y}_t|$ rage absolute difference between the predicted and actual values. The most interpretable metric. Use when errors are acceptable as long as they cancel each other. Example: house prices.

MSE (Mean Squared Error): $\mathrm{MSE} = \frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2$:rage of the squared difference between the actual and predicted values. Penalizes for large errors. Use when relationship is deterministic, precision is critical, larger errors significantly impact conclusions. Example: R&D, engineering.

RMSE (Root Mean Squared Error): $\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}$ MSE, but expressed in the same units as the target variable. Use when larger errors must be penalized because they lead to system collapse. Example: supply chain, electricity demand forecasting.

DO NOT USE MAPE (Mean Average Percentage Error)!

If your data contains zeros or very small values (consider WMAPE or other alternatives).


Julia Lenc

Stage 5.1

```python
# This script demonstrates basic time series model evaluation using Python
# For real projects, choose only one model based on your data and research!
# Here we show ARIMA, SARIMA, and Exponential Smoothing for learning only.

import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error

# 1. Load your preprocessed time series data from a CSV file
df = pd.read_csv("my_preprocessed_data.csv", index_col=0, parse_dates=True)

# 2. Assume the time series is in column 'y'. Adjust as necessary.
y = df['y']

# 3. Split data into train and test (e.g., last 20% for testing)
train_size = int(len(y) * 0.8)
train, test = y.iloc[:train_size], y.iloc[train_size:]

# 4. Initialize and fit models (choose only ONE in practice, here for demo)
# a) ARIMA example
arima_model = ARIMA(train, order=(1,1,1)).fit()
arima_pred_train = arima_model.predict(start=train.index[0], end=train.index[-1])
arima_pred_test = arima_model.predict(start=test.index[0], end=test.index[-1])

# b) SARIMA example
sarima_model = SARIMAX(train, order=(1,1,1), seasonal_order=(1,1,1,12)).fit(disp=False)
sarima_pred_train = sarima_model.predict(start=train.index[0], end=train.index[-1])
sarima_pred_test = sarima_model.predict(start=test.index[0], end=test.index[-1])

# c) Exponential Smoothing example
es_model = ExponentialSmoothing(train, seasonal='add', seasonal_periods=12).fit()
es_pred_train = es_model.fittedvalues
es_pred_test = es_model.forecast(len(test))
```

Julia Lenc

Stage 5.2

This is continuation of the code from previous slide.

It will not work on its own (no reference file)!

```
# 5. Evaluate using MSE (Mean Squared Error)
def print_mse_results(name, actual_train, pred_train, actual_test, pred_test):
    mse_train = mean_squared_error(actual_train, pred_train)
    mse_test = mean_squared_error(actual_test, pred_test)
    print(f"\n{name} Model:")
    print(f"- Train MSE: {mse_train:.2f}")
    print(f"- Test MSE: {mse_test:.2f}")
    # Model should not overfit: test MSE should not be much higher than train MSE
    print("-" * 40)

print_mse_results("ARIMA", train, arima_pred_train, test, arima_pred_test)
print_mse_results("SARIMA", train, sarima_pred_train, test, sarima_pred_test)
print_mse_results("Exp. Smoothing", train, es_pred_train, test, es_pred_test)

# 6. Interpreting the results (see terminal output)
# If train MSE is good (meets your criteria) and test MSE is similar (no big increase),
# the model is likely generalizing well (no overfitting).
# If test MSE grows much larger than train MSE, revisit your model choice/parameters/data.
```

Julia Lenc

The process: Stage 6.1

If the model overfits or underfits

1. Check residuals for...

White Noise: residuals should look like random noise (no systematic patterns, no clear trends or seasonality)

No Autocorrelation: use autocorrelation plots (ACF, PACF of residuals). If residuals are autocorrelated, important structure is still unmodeled.

Constant Variance: no "funnel" shapes (heteroscedasticity) in residuals over time.

Normality: for some models, we want residuals to be roughly normal (bell-shaped). This usually matters more for confidence intervals than for forecast accuracy.

No Outliers or Structural Breaks: big spikes may indicate outliers, missed level shifts, or regime changes.

If residuals are not random, return to preprocessing and check:

- Time series stationarity (try differencing, transformation).
- Seasonality patterns (may need to include in model).
- Missing values and interpolation.
- Presence of structural breaks or regime shifts.

Julia Lenc

Stage 6.1

```python
# We ran ARIMA and the model doesn't meet evaluation criteria (MSE).
# This is a generic example for both overfitting and underfitting.
# As a first check, we will assess residuals.

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import scipy.stats as stats

# Load the time series data
df = pd.read_csv("my_preprocessed_data.csv", index_col=0, parse_dates=True)
y = df['y']
train_size = int(len(y) * 0.8)
train = y.iloc[:train_size]

# Fit ARIMA model (example, parameters should be selected carefully in practice)
model = ARIMA(train, order=(1,1,1)).fit()

# Get residuals (difference between actual and predicted)
residuals = model.resid

# 1. Visualize residuals for randomness (White Noise)
plt.figure(figsize=(10,4))
plt.plot(residuals)
plt.title('Residuals over Time')
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.axhline(0, color='red', linestyle='--')
plt.show()

# 2. Check for autocorrelation in residuals (should be like white noise)
fig, ax = plt.subplots(1, 2, figsize=(12,4))
plot_acf(residuals, lags=30, ax=ax[0])
plot_pacf(residuals, lags=30, ax=ax[1])
ax[0].set_title('ACF of Residuals')
ax[1].set_title('PACF of Residuals')
plt.tight_layout()
plt.show()
```

Julia Lenc

Stage 6.1

This is continuation of the code from previous slide.

It will not work on its own (no reference file)!

```
# 3. Check for constant variance ("funnel" means non-constant!)
plt.figure(figsize=(10,4))
plt.scatter(train.index, residuals)
plt.title('Checking for Constant Variance')
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.axhline(0, color='red', linestyle='--')
plt.show()

# 4. Check if residuals are roughly normal (bell-shaped)
sns.histplot(residuals, kde=True)
plt.title('Histogram of Residuals (Normality Check)')
plt.xlabel('Residual')
plt.ylabel('Frequency')
plt.show()

# 5. (Optional) Q-Q plot for normality check
stats.probplot(residuals, dist="norm", plot=plt)
plt.title("Q-Q plot of residuals")
plt.show()

# 6. Check for outliers or structural breaks (look for big spikes)
plt.figure(figsize=(10,4))
plt.plot(residuals)
plt.title('Outlier/Structural Break Check')
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.axhline(0, color='red', linestyle='--')
plt.show()

# If residuals are not random, return to preprocessing.
# If residuals are random or there are no errors in preprocessing and assumptions,
# consider next steps (see next slide).
```

Julia Lenc

The process: Stage 6.2

Residuals are not an issue, no preprocessing errors, but..

1. The model overfits

- Simplify the model (reduce parameters, use more regularization)
- Use more training data if possible
- Remove unnecessary features or lags
- Use cross-validation (e.g., walk-forward validation)

2. The model underfits

- Try a more complex model (higher order, add seasonality, more features)
- Add more exogenous variables (calendar events, weather, etc.)
- Transform features or use longer lag windows

Stage 6.2

Overfitting remedies

```python
import pandas as pd
from sklearn.linear_model import Lasso, Ridge
from sklearn.model_selection import train_test_split, TimeSeriesSplit
from statsmodels.tsa.arima.model import ARIMA

# Load data (with possible extra features)
df = pd.read_csv("my_data_with_features.csv", index_col=0, parse_dates=True)
y = df['y']
# Example: Multiple features for the ML model, not just the time series itself
X = df.drop(columns=['y'])

# Split train/test (time series safe!)
train_size = int(len(y) * 0.8)
X_train, X_test = X.iloc[:train_size], X.iloc[train_size:]
y_train, y_test = y.iloc[:train_size], y.iloc[train_size:]

# -- 1. Model simplification: Use lower-order ARIMA
model_simpler = ARIMA(y_train, order=(0,1,1)).fit()
print("Simpler ARIMA AIC:", model_simpler.aic)

# -- 2. Train with more data if possible
# Here we artificially extend train window
train_larger = int(len(y) * 0.9)
y_train_larger = y.iloc[:train_larger]
model_moredata = ARIMA(y_train_larger, order=(1,1,1)).fit()
print('More data ARIMA AIC:', model_moredata.aic)

# -- 3. Regularization (on ML example: Lasso/L1 and Ridge/L2)
# Only for demonstration, since ARIMA doesn't directly do this
lasso = Lasso(alpha=0.05)
lasso.fit(X_train, y_train)
print('Lasso (L1) train score:', lasso.score(X_train, y_train))
print('Lasso (L1) test score:', lasso.score(X_test, y_test))

ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
print('Ridge (L2) train score:', ridge.score(X_train, y_train))
print('Ridge (L2) test score:', ridge.score(X_test, y_test))

# -- 4. Remove excessive/redundant features (manually for demo)
# Suppose 'redundant_feature' is highly correlated with another variable
if 'redundant_feature' in X.columns:
    X_reduced = X.drop(columns=['redundant_feature'])
else:
    X_reduced = X.copy() # or select a different feature to remove

X_train_red, X_test_red = X_reduced.iloc[:train_size], X_reduced.iloc[train_size:]
lasso.fit(X_train_red, y_train)
print("Lasso after feature removal (L1) test score:", lasso.score(X_test_red, y_test))

# -- 5. Cross-validation (walk-forward) for robust evaluation
tscv = TimeSeriesSplit(n_splits=5)
test_scores = []
for train_idx, test_idx in tscv.split(X_train_red):
    lasso.fit(X_train_red.iloc[train_idx], y_train.iloc[train_idx])
    score = lasso.score(X_train_red.iloc[test_idx], y_train.iloc[test_idx])
    test_scores.append(score)
print("Walk-forward validation Lasso test scores:", test_scores)
```

Julia Lenc

Stage 6.2

Underfitting remedies

```python
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load data (with possible new or engineered features)
df = pd.read_csv("my_data_with_features.csv", index_col=0, parse_dates=True)
y = df['y']

# Optional: load extra exogenous features, like holidays, weather, etc
if 'exog1' in df.columns:
    exog = df[['exog1', 'exog2']]  # List your exogenous features
else:
    exog = None

# Add lagged features (demonstration for ML models)
df['y_lag1'] = df['y'].shift(1)
df['y_lag2'] = df['y'].shift(2)

# Add rolling mean as feature (feature transformation)
df['rolling_mean_3'] = df['y'].rolling(window=3).mean()

# Train/test split (skip first 2 rows because of lags)
df = df.dropna()
train_size = int(len(df) * 0.8)
train, test = df.iloc[:train_size], df.iloc[train_size:]
y_train, y_test = train['y'], test['y']

# -- 1. Higher-order ARIMA: Increase p, d, or q (add complexity)
model_complex = ARIMA(y_train, order=(3,1,3)).fit()
print("Complex ARIMA AIC:", model_complex.aic)

# -- 2. Add exogenous variables (if available)
if exog is not None:
    exog_train, exog_test = exog.iloc[:train_size], exog.iloc[train_size:]
    model_exog = ARIMA(y_train, exog=exog_train, order=(1,1,1)).fit()
    print("Model with exogenous features AIC:", model_exog.aic)

# -- 3. Use lagged features and transformations for ML model
features = ['y_lag1', 'y_lag2', 'rolling_mean_3']
X_train, X_test = train[features], test[features]
# ML model with more features/complexity
linreg = LinearRegression().fit(X_train, y_train)
y_pred = linreg.predict(X_test)
print('ML with new features/test MSE:', mean_squared_error(y_test, y_pred))

# -- 4. Try model with seasonality (for SARIMA)
try:
    from statsmodels.tsa.statespace.sarimax import SARIMAX
    model_seasonal = SARIMAX(y_train, order=(1,1,1), seasonal_order=(1,1,1,12)).fit(disp=False)
    print("SARIMA (with seasonality) AIC:", model_seasonal.aic)
except ImportError:
    print("statsmodels SARIMAX not available for seasonality demo.")
```

Julia Lenc

The process: Stage 6.3

'Remedies' did not work. Last try before escalating to neural networks - hyperparameter fine-tuning

Three approaches

- **Grid search** – trying *all* combinations in a parameter grid
- **Random search** – sampling parameters randomly
- **Automated tools** – like Optuna, Scikit-Optimize

Let's prepare the file for optimization:

```
# "We ran ARIMA. Our model does not meet evaluation criteria.
# No issues with residuals and data preprocessing.
# Overfitting (underfitting) remedies did not help.
# Last try before escalating to neural networks - hyperparameter optimization."

import pandas as pd
import numpy as np
import itertools
import random
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# --- Load your preprocessed data
df = pd.read_csv("my_preprocessed_data.csv", index_col=0, parse_dates=True)
y = df['y']

# --- Train/test split (time series safe)
train_size = int(len(y) * 0.8)
y_train, y_test = y.iloc[:train_size], y.iloc[train_size:]
```

Julia Lenc

The process: Stage 6.3

This is continuation of of the code.

It will not work without set up (see previous slide)!

```
# --- Grid Search for best ARIMA(p,d,q)
p = d = q = range(0, 3)
pdq = list(itertools.product(p, d, q))

best_aic = np.inf
best_order = None
best_model = None

print('Starting ARIMA grid search...')

for order in pdq:
    try:
        model = ARIMA(y_train, order=order)
        results = model.fit()
        if results.aic < best_aic:
            best_aic = results.aic
            best_order = order
            best_model = results
        print(f"Tested ARIMA{order}, AIC={results.aic:.2f}")
    except Exception as e:
        # It's common for certain (p,d,q) to fail (e.g., non-stationary or non-invertible)
        print(f"ARIMA{order} failed. {e}")

print('\nBest ARIMA order (grid search):', best_order, 'with AIC:', best_aic)
# Evaluate best grid search model on test set
y_pred = best_model.forecast(steps=len(y_test))
print('Test MSE (grid search):', mean_squared_error(y_test, y_pred))

# --- Random Search for best ARIMA(p,d,q)
search_space = [(random.randint(0, 4), random.randint(0, 2), random.randint(0, 4)) for _ in range(15)]
best_aic_rand = np.inf
best_order_rand = None
best_model_rand = None

print('\nStarting ARIMA random search...')
for order in search_space:
    try:
        model = ARIMA(y_train, order=order)
        results = model.fit()
        if results.aic < best_aic_rand:
            best_aic_rand = results.aic
            best_order_rand = order
            best_model_rand = results
        print(f"Tested ARIMA{order}, AIC={results.aic:.2f}")
    except Exception as e:
        print(f"ARIMA{order} failed. {e}")

print('\nBest ARIMA order (random search):', best_order_rand, 'with AIC:', best_aic_rand)
# Evaluate best random search model on test set
y_pred_rand = best_model_rand.forecast(steps=len(y_test))
print('Test MSE (random search):', mean_squared_error(y_test, y_pred_rand))

# --- Recommendations
# If neither hyperparameter search yields a satisfactory model,
# consider escalating to deep learning models (e.g., LSTM).
```

Julia Lenc