

✓ REQUIRED INSTALLS

```
!pip install pmdarima statsmodels --quiet
!pip install openpyxl --quiet
```

689.1/689.1 kB 17.8 MB/s eta 0:00:00

✓ IMPORTS

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.stats.diagnostic import acorr_ljungbox

from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from pmdarima import auto_arima
from itertools import product

import os
from datetime import datetime
from google.colab import drive
```

```
# Mount Google Drive
print("Mounting Google Drive...")
drive.mount('/content/drive')

# Set display options
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")
```

```
print("All packages installed and imported successfully!")
print(f"Execution started at: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
```

Mounting Google Drive...

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

All packages installed and imported successfully!

Execution started at: 2025-11-20 17:57:56

✓ DRIVE PATHS AND FOLDER STRUCTURE

```
# Define base path - UPDATE THIS PATH TO YOUR DRIVE LOCATION
BASE_PATH = '/content/drive/MyDrive/Time_Series_Project'
```

```
# Create folder structure
```

```
FOLDERS = {
    'data_raw': f'{BASE_PATH}/data/raw',
    'data_cleaned': f'{BASE_PATH}/data/cleaned',
    'models': f'{BASE_PATH}/models',
    'results_viz': f'{BASE_PATH}/results/visualizations',
    'results_eval': f'{BASE_PATH}/results/evaluations',
    'results_forecast': f'{BASE_PATH}/results/forecasts',
    'reports': f'{BASE_PATH}/reports'
}
```

```
# Create directories
```

```
for folder_name, folder_path in FOLDERS.items():
    os.makedirs(folder_path, exist_ok=True)
    print(f"Created/Verified: {folder_name}")
```

```
print("FOLDER STRUCTURE READY")
```

```
print("Please upload the following files to the 'data/raw' folder:")
print("1. electricity_requirement_mu_2015_2025.csv")
print("2. monthly_festival_index_detailed_2015_2025.csv")
print("3. temperature_india_monthly_2015_2025.csv")
print("4. iip_yoy_2014_2025.csv")
```

```
Created/Verified: data_raw
Created/Verified: data_cleaned
Created/Verified: models
Created/Verified: results_viz
Created/Verified: results_eval
Created/Verified: results_forecast
```

Created/Verified: reports
FOLDER STRUCTURE READY
Please upload the following files to the 'data/raw' folder:
1. electricity_requirement_mu_2015_2025.csv
2. monthly_festival_index_detailed_2015_2025.csv
3. temperature_india_monthly_2015_2025.csv
4. iip_yoy_2014_2025.csv

✓ DATA LOADING AND INITIAL EXPLORATION

```
print("LOADING DATASETS")

# Load datasets
electricity_df = pd.read_csv(f"{FOLDERS['data_raw']}/electricity_requirement_mu_2015_2025.csv")
festival_df = pd.read_csv(f"{FOLDERS['data_raw']}/monthly_festival_index_detailed_2015_2025.csv")
temperature_df = pd.read_csv(f"{FOLDERS['data_raw']}/temperature_india_monthly_2015_2025.csv")
iip_df = pd.read_csv(f"{FOLDERS['data_raw']}/iip_yoy_2014_2025.csv")

print("All datasets loaded successfully!")
print(f"\nElectricity data shape: {electricity_df.shape}")
print(f"Festival data shape: {festival_df.shape}")
print(f"Temperature data shape: {temperature_df.shape}")
print(f"IIP data shape: {iip_df.shape}")
```

```
LOADING DATASETS
All datasets loaded successfully!
```

```
Electricity data shape: (131, 4)
Festival data shape: (128, 8)
Temperature data shape: (128, 4)
IIP data shape: (145, 4)
```

✓ DATA CLEANING AND PREPROCESSING

```
print("DATA CLEANING AND PREPROCESSING")
```

```
DATA CLEANING AND PREPROCESSING
```

Electricity Data

```
print("\n[1/4] Cleaning Electricity Data...")
# Remove footer rows (copyright, NaN rows)
electricity_clean = electricity_df[electricity_df['Parameter'].notna()].copy()
```

```

electricity_clean = electricity_clean[electricity_clean['Parameter'].str.contains(r'\d{4}-\d{2}', na=False)]

# Rename columns
electricity_clean.columns = ['Date', 'Requirement_MU', 'Supplied_MU', 'Shortage_Percent']

# Convert Date to datetime
electricity_clean['Date'] = pd.to_datetime(electricity_clean['Date'], format='%Y-%m')

# Extract Year and Month
electricity_clean['Year'] = electricity_clean['Date'].dt.year
electricity_clean['Month'] = electricity_clean['Date'].dt.month

# Reset index
electricity_clean = electricity_clean.reset_index(drop=True)

print(f"   Cleaned shape: {electricity_clean.shape}")
print(f"   Date range: {electricity_clean['Date'].min()} to {electricity_clean['Date'].max()}")

```

```

[1/4] Cleaning Electricity Data...
   Cleaned shape: (125, 6)
   Date range: 2015-04-01 00:00:00 to 2025-08-01 00:00:00

```

Festuvall Data

```

print("\n[2/4] Cleaning Festival Data...")

festival_clean = festival_df[['Year', 'Month', 'Monthly_Festival_Index']].copy()
print(f"   Cleaned shape: {festival_clean.shape}")

```

```

[2/4] Cleaning Festival Data...
   Cleaned shape: (128, 3)

```

Temperature-Data

```

print("\n[3/4] Cleaning Temperature Data...")

# Convert month to datetime
temperature_clean = temperature_df.copy()
temperature_clean['Date'] = pd.to_datetime(temperature_clean['month'], format='%Y-%m')
temperature_clean['Year'] = temperature_clean['Date'].dt.year
temperature_clean['Month'] = temperature_clean['Date'].dt.month

# Drop NaN rows
temperature_clean = temperature_clean.dropna(subset=['mean'])

```

```
# Select relevant columns
temperature_clean = temperature_clean[['Year', 'Month', 'mean']].copy()
temperature_clean.columns = ['Year', 'Month', 'Temperature_Mean']

print(f"    Cleaned shape: {temperature_clean.shape}")
```

```
[3/4] Cleaning Temperature Data...
    Cleaned shape: (127, 3)
```

IIP YoY% data

```
print("\n[4/4] Cleaning IIP YoY% Data...")

# Remove NaN rows
iip_clean = iip_df[iip_df['month'].notna()].copy()

# Remove extra columns
iip_clean = iip_clean[['month', 'iip_yoy']].copy()

# Correct typo 'Februrary' to 'February'
iip_clean['month'] = iip_clean['month'].str.replace('Februrary', 'February', regex=False)

# Parse the month column (format: "January, 2015")
# Use format='mixed' to handle potential inconsistencies or typos in month names like 'Februrary'
iip_clean['Date'] = pd.to_datetime(iip_clean['month'], format='mixed', dayfirst=False)
iip_clean['Year'] = iip_clean['Date'].dt.year
iip_clean['Month'] = iip_clean['Date'].dt.month

# Select relevant columns
iip_clean = iip_clean[['Year', 'Month', 'iip_yoy']].copy()
iip_clean.columns = ['Year', 'Month', 'IIP_YoY']

print(f"    Cleaned shape: {iip_clean.shape}")
```

```
[4/4] Cleaning IIP YoY% Data...
    Cleaned shape: (132, 3)
```

✓ MERGING ALL DATASETS

```
print("MERGING DATASETS")
```

```

# Start with electricity data (our target variable)
merged_df = electricity_clean[['Date', 'Year', 'Month', 'Requirement_MU']].copy()

# Merge festival data
merged_df = merged_df.merge(festival_clean, on=['Year', 'Month'], how='left')

# Merge temperature data
merged_df = merged_df.merge(temperature_clean, on=['Year', 'Month'], how='left')

# Merge IIP data
merged_df = merged_df.merge(iip_clean, on=['Year', 'Month'], how='left')

# Sort by date
merged_df = merged_df.sort_values('Date').reset_index(drop=True)

print(f"\nMerged dataset shape: {merged_df.shape}")
print(f"Date range: {merged_df['Date'].min()} to {merged_df['Date'].max()}")
print(f"\nMissing values:")
print(merged_df.isnull().sum())

# Handle missing values
if merged_df.isnull().sum().sum() > 0:
    print("\nHandling missing values...")
    # Forward fill for small gaps
    merged_df = merged_df.fillna(method='ffill').fillna(method='bfill')
    print("Missing values handled")

# Save cleaned merged dataset
merged_df.to_csv(f"{FOLDERS['data_cleaned']}/merged_data.csv", index=False)
print(f"\nMerged dataset saved to: {FOLDERS['data_cleaned']}/merged_data.csv")

print("FINAL DATASET PREVIEW")

print(merged_df.head(10))
print("\n")
print(merged_df.tail(10))

```

MERGING DATASETS

```

Merged dataset shape: (126, 7)
Date range: 2015-04-01 00:00:00 to 2025-08-01 00:00:00

```

```

Missing values:
Date          0
Year          0
Month         0
Requirement_MU 0

```

```
Monthly_Festival_Index    0
Temperature_Mean          1
IIP_YoY                  1
dtype: int64
```

```
Handling missing values...
Missing values handled
```

```
rged dataset saved to: /content/drive/MyDrive/Time_Series_Project/data/cleaned/merged_data.csv
```

```
FINAL DATASET PREVIEW
```

	Date	Year	Month	Requirement_MU	Monthly_Festival_Index	Temperature_Mean	IIP_YoY
0	2015-04-01	2015	4	89181.4	8.5	27.74	0.041
1	2015-05-01	2015	5	98315.0	4.0	30.24	0.027
2	2015-06-01	2015	6	92753.8	1.5	29.15	0.038
3	2015-07-01	2015	7	99105.2	4.5	28.64	0.042
4	2015-08-01	2015	8	99166.2	6.5	28.05	0.064
5	2015-09-01	2015	9	99019.7	14.0	27.98	0.036
6	2015-10-01	2015	10	100793.0	16.5	27.03	0.098
7	2015-11-01	2015	11	86681.0	8.5	24.09	-0.032
8	2015-12-01	2015	12	91066.8	6.0	21.13	-0.013
9	2016-01-01	2016	1	93090.0	9.0	20.45	-0.015

	Date	Year	Month	Requirement_MU	Monthly_Festival_Index	Temperature_Mean	IIP_YoY
116	2024-12-01	2024	12	129582.0	6.0	19.46	0.032
117	2025-01-01	2025	1	136363.0	9.0	19.02	0.050
118	2025-02-01	2025	2	130604.0	3.5	22.06	0.029
119	2025-03-01	2025	3	146955.0	9.0	25.52	0.030
120	2025-04-01	2025	4	148066.0	11.5	29.16	0.027
121	2025-05-01	2025	5	147948.0	4.0	29.57	0.012
122	2025-06-01	2025	6	149183.0	5.5	29.45	0.015
123	2025-07-01	2025	7	153925.0	2.5	27.93	0.048
124	2025-07-01	2025	7	153925.0	2.5	27.93	0.035
125	2025-08-01	2025	8	149795.0	13.5	27.51	0.040

✓ EXPLORATORY DATA ANALYSIS

```
print("EXPLORATORY DATA ANALYSIS")

# Basic statistics
print("\nDescriptive Statistics:")
print(merged_df.describe())

# Save statistics
stats_df = merged_df.describe()
stats_df.to_csv(f"{FOLDERS['reports']}/descriptive_statistics.csv")
```

EXPLORATORY DATA ANALYSIS

Descriptive Statistics:

	Date	Year	Month	Requirement_MU	Monthly_Festival_Index	Temperature_Mean	IIP_YoY
count	126	126.000000	126.000000	126.000000	126.000000	126.000000	126.000000
mean	2020-06-15 14:05:42.857142784	2020.000000	6.484127	115398.657143	7.468254	26.069524	0.038373
min	2015-04-01 00:00:00	2015.000000	1.000000	85030.000000	0.000000	19.020000	-0.347000
25%	2017-11-08 12:00:00	2017.000000	4.000000	101285.350000	4.000000	23.472500	0.005500
50%	2020-06-16 00:00:00	2020.000000	6.500000	110968.600000	6.750000	27.610000	0.031000
75%	2023-01-24 06:00:00	2023.000000	9.000000	128646.500000	10.375000	28.510000	0.049750
max	2025-08-01 00:00:00	2025.000000	12.000000	155346.000000	21.500000	31.080000	1.344000
std	NaN	3.059412	3.395842	18054.140070	4.645319	3.379905	0.133924

Time Series Visualization

```
print("\n[1/5] Creating time series visualizations...")

fig, axes = plt.subplots(4, 1, figsize=(15, 12))

# Target variable
axes[0].plot(merged_df['Date'], merged_df['Requirement_MU'], color='darkblue', linewidth=2)
axes[0].set_title('Electricity Requirement (MU) - April 2015 to August 2025', fontsize=14, fontweight='bold')
axes[0].set_ylabel('Requirement (MU)', fontsize=12)
axes[0].grid(True, alpha=0.3)

# Festival Index
axes[1].plot(merged_df['Date'], merged_df['Monthly_Festival_Index'], color='darkorange', linewidth=2)
axes[1].set_title('Monthly Festival Index', fontsize=14, fontweight='bold')
axes[1].set_ylabel('Festival Index', fontsize=12)
axes[1].grid(True, alpha=0.3)

# Temperature
axes[2].plot(merged_df['Date'], merged_df['Temperature_Mean'], color='darkred', linewidth=2)
axes[2].set_title('Average Temperature (°C)', fontsize=14, fontweight='bold')
axes[2].set_ylabel('Temperature (°C)', fontsize=12)
axes[2].grid(True, alpha=0.3)

# IIP YoY%
axes[3].plot(merged_df['Date'], merged_df['IIP_YoY'], color='darkgreen', linewidth=2)
axes[3].axhline(y=0, color='red', linestyle='--', alpha=0.5)
axes[3].set_title('IIP Year-over-Year Growth (%)', fontsize=14, fontweight='bold')
axes[3].set_ylabel('IIP YoY %', fontsize=12)
axes[3].set_xlabel('Date', fontsize=12)
axes[3].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/01_time_series_overview.png", dpi=300, bbox_inches='tight')
```

```
plt.close()
print("Saved: 01_time_series_overview.png")
```

```
[1/5] Creating time series visualizations...
Saved: 01_time_series_overview.png
```

Correlation

```
print("\n[2/5] Creating correlation analysis...")

correlation_vars = ['Requirement_MU', 'Monthly_Festival_Index', 'Temperature_Mean', 'IIP_YoY']
corr_matrix = merged_df[correlation_vars].corr()

plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0,
            square=True, linewidths=1, cbar_kws={"shrink": 0.8},
            fmt='.3f', vmin=-1, vmax=1)
plt.title('Correlation Matrix: Target and Exogenous Variables', fontsize=14, fontweight='bold', pad=20)
plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/02_correlation_matrix.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 02_correlation_matrix.png")

# Save correlation matrix
corr_matrix.to_csv(f"{FOLDERS['reports']}/correlation_matrix.csv")
```

```
[2/5] Creating correlation analysis...
Saved: 02_correlation_matrix.png
```

Distribution

```
print("\n[3/5] Creating distribution plots...")

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Electricity Requirement
axes[0, 0].hist(merged_df['Requirement_MU'], bins=30, color='darkblue', alpha=0.7, edgecolor='black')
axes[0, 0].set_title('Distribution: Electricity Requirement', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Requirement (MU)')
axes[0, 0].set_ylabel('Frequency')

# Festival Index
axes[0, 1].hist(merged_df['Monthly_Festival_Index'], bins=20, color='darkorange', alpha=0.7, edgecolor='black')
```

```

axes[0, 1].set_title('Distribution: Festival Index', fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Festival Index')
axes[0, 1].set_ylabel('Frequency')

# Temperature
axes[1, 0].hist(merged_df['Temperature_Mean'], bins=25, color='darkred', alpha=0.7, edgecolor='black')
axes[1, 0].set_title('Distribution: Temperature', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Temperature (°C)')
axes[1, 0].set_ylabel('Frequency')

# IIP YoY
axes[1, 1].hist(merged_df['IIP_YoY'], bins=25, color='darkgreen', alpha=0.7, edgecolor='black')
axes[1, 1].set_title('Distribution: IIP YoY%', fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('IIP YoY %')
axes[1, 1].set_ylabel('Frequency')

plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/03_distributions.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 03_distributions.png")

```

```

[3/5] Creating distribution plots...
Saved: 03_distributions.png

```

Seasonal Decomposition

```

print("\n[4/5] Performing seasonal decomposition...")

# Set Date as index for decomposition
ts_data = merged_df.set_index('Date')['Requirement_MU']

# Perform decomposition (additive)
decomposition = seasonal_decompose(ts_data, model='additive', period=12)

# Plot
fig, axes = plt.subplots(4, 1, figsize=(15, 12))

decomposition.observed.plot(ax=axes[0], color='darkblue', linewidth=2)
axes[0].set_ylabel('Observed')
axes[0].set_title('Seasonal Decomposition of Electricity Requirement', fontsize=14, fontweight='bold')
axes[0].grid(True, alpha=0.3)

decomposition.trend.plot(ax=axes[1], color='darkorange', linewidth=2)
axes[1].set_ylabel('Trend')
axes[1].grid(True, alpha=0.3)

```

```

decomposition.seasonal.plot(ax=axes[2], color='darkgreen', linewidth=2)
axes[2].set_ylabel('Seasonal')
axes[2].grid(True, alpha=0.3)

decomposition.resid.plot(ax=axes[3], color='darkred', linewidth=1)
axes[3].set_ylabel('Residual')
axes[3].set_xlabel('Date')
axes[3].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/04_seasonal_decomposition.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 04_seasonal_decomposition.png")

# Save decomposition components
decomp_df = pd.DataFrame({
    'Date': decomposition.observed.index,
    'Observed': decomposition.observed.values,
    'Trend': decomposition.trend.values,
    'Seasonal': decomposition.seasonal.values,
    'Residual': decomposition.resid.values
})
decomp_df.to_csv(f"{FOLDERS['reports']}/seasonal_decomposition.csv", index=False)

```

```

[4/5] Performing seasonal decomposition...
Saved: 04_seasonal_decomposition.png

```

Stationarity Tests

```

print("\n[5/5] Conducting stationarity tests...")

def adf_test(series, title=''):
    """Augmented Dickey-Fuller Test"""
    result = adfuller(series.dropna(), autolag='AIC')
    output = {
        'Test Statistic': result[0],
        'p-value': result[1],
        'Lags Used': result[2],
        'Observations': result[3],
        'Critical Value (1%)': result[4]['1%'],
        'Critical Value (5%)': result[4]['5%'],
        'Critical Value (10%)': result[4]['10%']
    }
    return output

```

```

def kpss_test(series, title=''):
    """KPSS Test"""
    result = kpss(series.dropna(), regression='c', nlags='auto')
    output = {
        'Test Statistic': result[0],
        'p-value': result[1],
        'Lags Used': result[2],
        'Critical Value (1%)': result[3]['1%'],
        'Critical Value (2.5%)': result[3]['2.5%'],
        'Critical Value (5%)': result[3]['5%'],
        'Critical Value (10%)': result[3]['10%']
    }
    return output

# Test original series
adf_results = adf_test(merged_df['Requirement_MU'])
kpss_results = kpss_test(merged_df['Requirement_MU'])

print("\n--- Augmented Dickey-Fuller Test (ADF) ---")
print(f"ADF Statistic: {adf_results['Test Statistic']:.6f}")
print(f"p-value: {adf_results['p-value']:.6f}")
print(f"Critical Values:")
print(f"  1%: {adf_results['Critical Value (1%)']:.3f}")
print(f"  5%: {adf_results['Critical Value (5%)']:.3f}")
print(f" 10%: {adf_results['Critical Value (10%)']:.3f}")
if adf_results['p-value'] < 0.05:
    print("Series is STATIONARY (reject H0)")
else:
    print("X Series is NON-STATIONARY (fail to reject H0)")

print("\n--- KPSS Test ---")
print(f"KPSS Statistic: {kpss_results['Test Statistic']:.6f}")
print(f"p-value: {kpss_results['p-value']:.6f}")
print(f"Critical Values:")
print(f"  1%: {kpss_results['Critical Value (1%)']:.3f}")
print(f"  5%: {kpss_results['Critical Value (5%)']:.3f}")
print(f" 10%: {kpss_results['Critical Value (10%)']:.3f}")
if kpss_results['p-value'] < 0.05:
    print("X Series is NON-STATIONARY (reject H0)")
else:
    print("Series is STATIONARY (fail to reject H0)")

# Save stationarity test results
stationarity_results = pd.DataFrame({
    'Test': ['ADF', 'KPSS'],

```

```

    'Test_Statistic': [adf_results['Test Statistic'], kpss_results['Test Statistic']],
    'p_value': [adf_results['p-value'], kpss_results['p-value']],
    'Stationary': [adf_results['p-value'] < 0.05, kpss_results['p-value'] >= 0.05]
})
stationarity_results.to_csv(f"{FOLDERS['reports']}/stationarity_tests.csv", index=False)

```

[5/5] Conducting stationarity tests...

--- Augmented Dickey-Fuller Test (ADF) ---

ADF Statistic: 0.766288

p-value: 0.991072

Critical Values:

1%: -3.490

5%: -2.887

10%: -2.581

X Series is NON-STATIONARY (fail to reject H0)

--- KPSS Test ---

KPSS Statistic: 1.682922

p-value: 0.010000

Critical Values:

1%: 0.739

5%: 0.463

10%: 0.347

X Series is NON-STATIONARY (reject H0)

/tmp/ipython-input-1284572771.py:19: InterpolationWarning: The test statistic is outside of the range of p-values available in the look-up table. The actual p-value is smaller than the p-value returned.

```
result = kpss(series.dropna(), regression='c', nlags='auto')
```

TRAIN-VALIDATION-TEST SPLIT

```
print("TRAIN-VALIDATION-TEST SPLIT")
```

```
# Total observations: 125 months (April 2015 - August 2025)
```

```
# Train: 70% = 87 months
```

```
# Validation: 15% = 19 months
```

```
# Test: 15% = 19 months
```

```
total_obs = len(merged_df)
```

```
train_size = int(total_obs * 0.70)
```

```
val_size = int(total_obs * 0.15)
```

```
test_size = total_obs - train_size - val_size
```

```
# Create splits
```

```
train_data = merged_df.iloc[:train_size].copy()
```

```

val_data = merged_df.iloc[train_size:train_size+val_size].copy()
test_data = merged_df.iloc[train_size+val_size:].copy()

print(f"\nTotal observations: {total_obs}")
print(f"\nTrain Set: {len(train_data)} months ({len(train_data)/total_obs*100:.1f}%)")
print(f"  Date range: {train_data['Date'].min()} to {train_data['Date'].max()}")
print(f"\nValidation Set: {len(val_data)} months ({len(val_data)/total_obs*100:.1f}%)")
print(f"  Date range: {val_data['Date'].min()} to {val_data['Date'].max()}")
print(f"\nTest Set: {len(test_data)} months ({len(test_data)/total_obs*100:.1f}%)")
print(f"  Date range: {test_data['Date'].min()} to {test_data['Date'].max()}")

# Save split information
split_info = pd.DataFrame({
    'Split': ['Train', 'Validation', 'Test'],
    'Start_Date': [train_data['Date'].min(), val_data['Date'].min(), test_data['Date'].min()],
    'End_Date': [train_data['Date'].max(), val_data['Date'].max(), test_data['Date'].max()],
    'N_Observations': [len(train_data), len(val_data), len(test_data)],
    'Percentage': [len(train_data)/total_obs*100, len(val_data)/total_obs*100, len(test_data)/total_obs*100]
})
split_info.to_csv(f"{FOLDERS['reports']}/data_split_info.csv", index=False)

# Visualize the split
plt.figure(figsize=(15, 6))
plt.plot(train_data['Date'], train_data['Requirement_MU'], label='Train', color='blue', linewidth=2)
plt.plot(val_data['Date'], val_data['Requirement_MU'], label='Validation', color='orange', linewidth=2)
plt.plot(test_data['Date'], test_data['Requirement_MU'], label='Test', color='green', linewidth=2)
plt.axvline(x=train_data['Date'].max(), color='red', linestyle='--', alpha=0.5, label='Train-Val Split')
plt.axvline(x=val_data['Date'].max(), color='red', linestyle='--', alpha=0.5, label='Val-Test Split')
plt.title('Train-Validation-Test Split', fontsize=14, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Electricity Requirement (MU)', fontsize=12)
plt.legend(loc='best')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/05_train_val_test_split.png", dpi=300, bbox_inches='tight')
plt.close()
print("\nSaved: 05_train_val_test_split.png")

```

TRAIN-VALIDATION-TEST SPLIT

Total observations: 126

Train Set: 88 months (69.8%)

Date range: 2015-04-01 00:00:00 to 2022-07-01 00:00:00

Validation Set: 18 months (14.3%)

Date range: 2022-08-01 00:00:00 to 2024-01-01 00:00:00

Test Set: 20 months (15.9%)
Date range: 2024-02-01 00:00:00 to 2025-08-01 00:00:00

Saved: 05_train_val_test_split.png

✓ ACF AND PACF PLOTS

```
print("ACF AND PACF ANALYSIS")

fig, axes = plt.subplots(2, 1, figsize=(15, 10))

# ACF
plot_acf(train_data['Requirement_MU'], lags=40, ax=axes[0])
axes[0].set_title('Autocorrelation Function (ACF)', fontsize=14, fontweight='bold')
axes[0].set_xlabel('Lag', fontsize=12)
axes[0].set_ylabel('ACF', fontsize=12)

# PACF
plot_pacf(train_data['Requirement_MU'], lags=40, ax=axes[1], method='ywm')
axes[1].set_title('Partial Autocorrelation Function (PACF)', fontsize=14, fontweight='bold')
axes[1].set_xlabel('Lag', fontsize=12)
axes[1].set_ylabel('PACF', fontsize=12)

plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/06_acf_pacf_plots.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 06_acf_pacf_plots.png")
```

ACF AND PACF ANALYSIS
Saved: 06_acf_pacf_plots.png

✓ EVALUATION METRICS FUNCTIONS

```
print("DEFINING EVALUATION FUNCTIONS")

def calculate_metrics(y_true, y_pred, model_name=''):
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

```

r2 = r2_score(y_true, y_pred)

# Directional Accuracy
direction_true = np.diff(y_true) > 0
direction_pred = np.diff(y_pred) > 0
directional_accuracy = np.mean(direction_true == direction_pred) * 100

metrics = {
    'Model': model_name,
    'MAE': mae,
    'RMSE': rmse,
    'MAPE': mape,
    'R2_Score': r2,
    'Directional_Accuracy': directional_accuracy
}

return metrics

def print_metrics(metrics):
    """Pretty print metrics"""
    print(f"\n{'='*60}")
    print(f"Model: {metrics['Model']}")
    print(f"{'='*60}")
    print(f"MAE (Mean Absolute Error):      {metrics['MAE']:.2f} MU")
    print(f"RMSE (Root Mean Squared Error):    {metrics['RMSE']:.2f} MU")
    print(f"MAPE (Mean Absolute % Error):      {metrics['MAPE']:.2f}%")
    print(f"R2 Score:                        {metrics['R2_Score']:.4f}")
    print(f"Directional Accuracy:              {metrics['Directional_Accuracy']:.2f}%")
    print(f"{'='*60}")

def plot_residuals(residuals, model_name, save_path):
    """Plot residual diagnostics"""
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # Residuals over time
    axes[0, 0].plot(residuals, color='darkblue', linewidth=1)
    axes[0, 0].axhline(y=0, color='red', linestyle='--')
    axes[0, 0].set_title(f'{model_name}: Residuals Over Time', fontweight='bold')
    axes[0, 0].set_xlabel('Observation')
    axes[0, 0].set_ylabel('Residuals')
    axes[0, 0].grid(True, alpha=0.3)

    # Histogram
    axes[0, 1].hist(residuals, bins=30, color='darkblue', alpha=0.7, edgecolor='black')
    axes[0, 1].set_title(f'{model_name}: Residuals Distribution', fontweight='bold')
    axes[0, 1].set_xlabel('Residuals')

```

```

axes[0, 1].set_ylabel('Frequency')
axes[0, 1].grid(True, alpha=0.3)

# Q-Q plot
from scipy import stats
stats.probplot(residuals, dist="norm", plot=axes[1, 0])
axes[1, 0].set_title(f'{model_name}: Q-Q Plot', fontweight='bold')
axes[1, 0].grid(True, alpha=0.3)

# ACF of residuals
plot_acf(residuals, lags=min(40, len(residuals)//2), ax=axes[1, 1])
axes[1, 1].set_title(f'{model_name}: ACF of Residuals', fontweight='bold')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(save_path, dpi=300, bbox_inches='tight')
plt.close()

print("Evaluation functions defined")

```

DEFINING EVALUATION FUNCTIONS
Evaluation functions defined

✓ MODEL 1 - ARIMA with Grid Search

```

print("MODEL 1: ARIMA WITH GRID SEARCH")

# Define parameter ranges for grid search
p_values = range(0, 4)
d_values = range(0, 3)
q_values = range(0, 4)

# Grid search for ARIMA
best_aic = np.inf
best_params = None
best_model = None

arma_results_list = []

total_combinations = len(p_values) * len(d_values) * len(q_values)
current_combination = 0

print(f"\nTesting {total_combinations} parameter combinations...")

for n in p_values:

```

```

for d in d_values:
    for q in q_values:
        current_combination += 1
        try:
            model = ARIMA(train_data['Requirement_MU'], order=(p, d, q))
            fitted_model = model.fit()

            # Validation predictions
            val_pred = fitted_model.forecast(steps=len(val_data))
            val_mae = mean_absolute_error(val_data['Requirement_MU'], val_pred)

            arima_results_list.append({
                'p': p, 'd': d, 'q': q,
                'AIC': fitted_model.aic,
                'BIC': fitted_model.bic,
                'Val_MAE': val_mae
            })

            # Track best model by AIC
            if fitted_model.aic < best_aic:
                best_aic = fitted_model.aic
                best_params = (p, d, q)
                best_model = fitted_model

            if current_combination % 10 == 0:
                print(f" Progress: {current_combination}/{total_combinations} combinations tested...")

        except Exception as e:
            continue

print(f"\nGrid search completed!")
print(f"\nBest ARIMA parameters: {best_params}")
print(f"Best AIC: {best_aic:.2f}")

# Save grid search results
arima_grid_df = pd.DataFrame(arima_results_list)
arima_grid_df = arima_grid_df.sort_values('AIC')
arima_grid_df.to_csv(f"{FOLDERS['reports']}/ARIMA_grid_search_results.csv", index=False)

# Display top 10 models
print("\nTop 10 ARIMA models by AIC:")
print(arima_grid_df.head(10))

# Final ARIMA model with best parameters
print(f"\nFitting final ARIMA{best_params} model on full training data...")
final_arima = ARIMA(train_data['Requirement_MU'], order=best_params)

```

```

fitted_arima = final_arima.fit()

print("\n" + str(fitted_arima.summary()))

# Predictions
arima_train_pred = fitted_arima.fittedvalues
arima_val_pred = fitted_arima.forecast(steps=len(val_data))
arima_test_pred = fitted_arima.forecast(steps=len(val_data) + len(test_data))[len(val_data):]

# Metrics on validation set
arima_val_metrics = calculate_metrics(
    val_data['Requirement_MU'].values,
    arima_val_pred.values,
    model_name='ARIMA_Validation'
)
print_metrics(arima_val_metrics)

# Metrics on test set
arima_test_metrics = calculate_metrics(
    test_data['Requirement_MU'].values,
    arima_test_pred.values,
    model_name='ARIMA_Test'
)
print_metrics(arima_test_metrics)

# Residual analysis
arima_residuals = fitted_arima.resid
plot_residuals(arima_residuals, 'ARIMA', f"{FOLDERS['results_viz']}/07_ARIMA_residuals.png")
print("\nSaved: 07_ARIMA_residuals.png")

# Save model summary
with open(f"{FOLDERS['reports']}/ARIMA_model_summary.txt", 'w') as f:
    f.write(str(fitted_arima.summary()))

print("\nARIMA model completed")

```

MODEL 1: ARIMA WITH GRID SEARCH

```

Testing 48 parameter combinations...
Progress: 10/48 combinations tested...
Progress: 20/48 combinations tested...
Progress: 30/48 combinations tested...
/usr/local/lib/python3.12/dist-packages/statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
warnings.warn("Maximum Likelihood optimization failed to ")
/usr/local/lib/python3.12/dist-packages/statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
warnings.warn("Maximum Likelihood optimization failed to ")
Progress: 40/48 combinations tested...
/usr/local/lib/python3.12/dist-packages/statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals

```

```
warnings.warn("Maximum Likelihood optimization failed to "
/usr/local/lib/python3.12/dist-packages/statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
warnings.warn("Maximum Likelihood optimization failed to "
/usr/local/lib/python3.12/dist-packages/statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
warnings.warn("Maximum Likelihood optimization failed to "
```

Grid search completed!

Best ARIMA parameters: (0, 1, 0)
Best AIC: 1795.12

Top 10 ARIMA models by AIC:

	p	d	q	AIC	BIC	Val_MAE
4	0	1	0	1795.119707	1797.585615	8761.111111
5	0	1	1	1797.066820	1801.998636	8761.111111
16	1	1	0	1797.181271	1802.113088	8760.969133
17	1	1	1	1798.369825	1805.767549	8824.418553
6	0	1	2	1798.749459	1806.147184	8773.718551
28	2	1	0	1798.806342	1806.204066	8772.221498
43	3	1	3	1799.579296	1816.840652	8510.014152
18	1	1	2	1799.609476	1809.473109	8767.174855
29	2	1	1	1800.303273	1810.166906	8806.149432
7	0	1	3	1800.375364	1810.238997	8768.295257

Fitting final ARIMA(0, 1, 0) model on full training data...

SARIMAX Results

```
=====
Dep. Variable:      Requirement_MU      No. Observations:      88
Model:              ARIMA(0, 1, 0)      Log Likelihood          -896.560
Date:               Thu, 20 Nov 2025      AIC                     1795.120
Time:               18:12:54              BIC                     1797.586
Sample:              0                    HQIC                    1796.113
                  - 88
Covariance Type:      opg
=====
              coef      std err          z      P>|z|      [0.025      0.975]
-----
sigma2          5.171e+07    6.85e+06     7.545     0.000     3.83e+07     6.51e+07
=====
Ljung-Box (L1) (Q):              5.11      Jarque-Bera (JB):              1.53
Prob(Q):                          0.02      Prob(JB):              0.47
Heteroskedasticity (H):           3.20      Skew:              0.20
Prob(H) (two-sided):              0.00      Kurtosis:              3.51
=====
```

✓ MODEL 2 - SARIMA with Grid Search

```
print("MODEL 2: SARIMA WITH GRID SEARCH")
```

```
# Define parameter ranges
p_values = range(0, 3)
```

```

d_values = range(0, 2)
q_values = range(0, 3)
P_values = range(0, 2)
D_values = range(0, 2)
Q_values = range(0, 2)
m = 12 # Monthly seasonality

best_aic_sarima = np.inf
best_params_sarima = None
best_model_sarima = None

sarima_results_list = []

total_combinations = len(p_values) * len(d_values) * len(q_values) * len(P_values) * len(D_values) * len(Q_values)
current_combination = 0

print(f"\nTesting {total_combinations} parameter combinations...")

for p in p_values:
    for d in d_values:
        for q in q_values:
            for P in P_values:
                for D in D_values:
                    for Q in Q_values:
                        current_combination += 1
                        try:
                            model = SARIMAX(
                                train_data['Requirement_MU'],
                                order=(p, d, q),
                                seasonal_order=(P, D, Q, m)
                            )
                            fitted_model = model.fit(dispatch=False, maxiter=200)

                            # Validation predictions
                            val_pred = fitted_model.forecast(steps=len(val_data))
                            val_mae = mean_absolute_error(val_data['Requirement_MU'], val_pred)

                            sarima_results_list.append({
                                'p': p, 'd': d, 'q': q,
                                'P': P, 'D': D, 'Q': Q, 'm': m,
                                'AIC': fitted_model.aic,
                                'BIC': fitted_model.bic,
                                'Val_MAE': val_mae
                            })

                            if fitted_model.aic < best_aic_sarima:

```

```

        best_aic_sarima = fitted_model.aic
        best_params_sarima = ((p, d, q), (P, D, Q, m))
        best_model_sarima = fitted_model

    if current_combination % 20 == 0:
        print(f" Progress: {current_combination}/{total_combinations} combinations tested...")

except Exception as e:
    continue

print(f"\nGrid search completed!")
print(f"\nBest SARIMA parameters:")
print(f" Order (p,d,q): {best_params_sarima[0]}")
print(f" Seasonal (P,D,Q,m): {best_params_sarima[1]}")
print(f" Best AIC: {best_aic_sarima:.2f}")

# Save grid search results
sarima_grid_df = pd.DataFrame(sarima_results_list)
sarima_grid_df = sarima_grid_df.sort_values('AIC')
sarima_grid_df.to_csv(f"{FOLDERS['reports']}/SARIMA_grid_search_results.csv", index=False)

print("\nTop 10 SARIMA models by AIC:")
print(sarima_grid_df.head(10))

# Final SARIMA model
print(f"\nFitting final SARIMA model on training data...")
final_sarima = SARIMAX(
    train_data['Requirement_MU'],
    order=best_params_sarima[0],
    seasonal_order=best_params_sarima[1]
)
fitted_sarima = final_sarima.fit(dispatch=False)

print("\n" + str(fitted_sarima.summary()))

# Predictions
sarima_train_pred = fitted_sarima.fittedvalues
sarima_val_pred = fitted_sarima.forecast(steps=len(val_data))
sarima_test_pred = fitted_sarima.forecast(steps=len(val_data) + len(test_data))[len(val_data):]

# Metrics on validation set
sarima_val_metrics = calculate_metrics(
    val_data['Requirement_MU'].values,
    sarima_val_pred.values,
    model_name='SARIMA_Validation'
)

```

```

print_metrics(sarima_val_metrics)

# Metrics on test set
sarima_test_metrics = calculate_metrics(
    test_data['Requirement_MU'].values,
    sarima_test_pred.values,
    model_name='SARIMA_Test'
)
print_metrics(sarima_test_metrics)

# Residual analysis
sarima_residuals = fitted_sarima.resid
plot_residuals(sarima_residuals, 'SARIMA', f"{FOLDERS['results_viz']}/08_SARIMA_residuals.png")
print("\nSaved: 08_SARIMA_residuals.png")

# Save model summary
with open(f"{FOLDERS['reports']}/SARIMA_model_summary.txt", 'w') as f:
    f.write(str(fitted_sarima.summary()))

print("\nSARIMA model completed")

```

135	2	1	1	1	1	1	12	1549.425095	1563.330024	4933.308105
75	1	1	0	0	1	1	12	1549.902577	1556.855041	6762.238072
35	0	1	1	0	1	1	12	1549.975320	1556.927785	6709.600667
30	0	1	0	1	1	0	12	1550.226324	1554.861300	7484.189187

```
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 3.27e+31. Standard errors may be unstable.
```

```
=====
Model: SARIMA_Validation
=====
```

```
MAE (Mean Absolute Error):      4,949.42 MU
RMSE (Root Mean Squared Error):  5,869.98 MU
MAPE (Mean Absolute % Error):    3.78%
R² Score:                       0.6927
Directional Accuracy:            82.35%
=====
```

```
=====
Model: SARIMA_Test
=====
```

```
MAE (Mean Absolute Error):      6,215.46 MU
RMSE (Root Mean Squared Error):  7,024.21 MU
MAPE (Mean Absolute % Error):    4.37%
R² Score:                       0.4162
Directional Accuracy:            73.68%
=====
```

```
Saved: 08_SARIMA_residuals.png
```

```
SARIMA_model_completed
```

✓ MODEL 3 – SARIMAX with Grid Search

```
print("MODEL 3: SARIMAX WITH EXOGENOUS VARIABLES AND GRID SEARCH")

exog_vars = ['Monthly_Festival_Index', 'Temperature_Mean', 'IIP_YoY']
train_exog = train_data[exog_vars]
val_exog = val_data[exog_vars]
test_exog = test_data[exog_vars]

print(f"Exogenous variables: {exog_vars}")
print(f"Train exog shape: {train_exog.shape}")
print(f"Val exog shape: {val_exog.shape}")
print(f"Test exog shape: {test_exog.shape}")

# Use smaller grid due to computational cost with exogenous variables
p_values = range(0, 3)
d_values = range(0, 2)
q_values = range(0, 3)
P_values = range(0, 2)
```

```

D_values = range(0, 2)
Q_values = range(0, 2)
m = 12

best_aic_sarimax = np.inf
best_params_sarimax = None
best_model_sarimax = None

sarimax_results_list = []

total_combinations = len(p_values) * len(d_values) * len(q_values) * len(P_values) * len(D_values) * len(Q_values)
current_combination = 0

print(f"\nTesting {total_combinations} parameter combinations with exogenous variables...")

for p in p_values:
    for d in d_values:
        for q in q_values:
            for P in P_values:
                for D in D_values:
                    for Q in Q_values:
                        current_combination += 1
                        try:
                            model = SARIMAX(
                                train_data['Requirement_MU'],
                                exog=train_exog,
                                order=(p, d, q),
                                seasonal_order=(P, D, Q, m)
                            )
                            fitted_model = model.fit(dispatch=False, maxiter=200)

                            # Validation predictions
                            val_pred = fitted_model.forecast(steps=len(val_data), exog=val_exog)
                            val_mae = mean_absolute_error(val_data['Requirement_MU'], val_pred)

                            sarimax_results_list.append({
                                'p': p, 'd': d, 'q': q,
                                'P': P, 'D': D, 'Q': Q, 'm': m,
                                'AIC': fitted_model.aic,
                                'BIC': fitted_model.bic,
                                'Val_MAE': val_mae
                            })

                            if fitted_model.aic < best_aic_sarimax:
                                best_aic_sarimax = fitted_model.aic
                                best_params_sarimax = ((p, d, q), (P, D, Q, m))

```

```

        best_model_sarimax = fitted_model

    if current_combination % 20 == 0:
        print(f" Progress: {current_combination}/{total_combinations} combinations tested...")

except Exception as e:
    continue

print(f"\nGrid search completed!")
print(f"\nBest SARIMAX parameters:")
print(f" Order (p,d,q): {best_params_sarimax[0]}")
print(f" Seasonal (P,D,Q,m): {best_params_sarimax[1]}")
print(f" Best AIC: {best_aic_sarimax:.2f}")

# Save grid search results
sarimax_grid_df = pd.DataFrame(sarimax_results_list)
sarimax_grid_df = sarimax_grid_df.sort_values('AIC')
sarimax_grid_df.to_csv(f"{FOLDERS['reports']}/SARIMAX_grid_search_results.csv", index=False)

print("\nTop 10 SARIMAX models by AIC:")
print(sarimax_grid_df.head(10))

# Final SARIMAX model
print(f"\nFitting final SARIMAX model on training data...")
final_sarimax = SARIMAX(
    train_data['Requirement_MU'],
    exog=train_exog,
    order=best_params_sarimax[0],
    seasonal_order=best_params_sarimax[1]
)
fitted_sarimax = final_sarimax.fit(dispatch=False)

print("\n" + str(fitted_sarimax.summary()))

# Predictions
sarimax_train_pred = fitted_sarimax.fittedvalues
sarimax_val_pred = fitted_sarimax.forecast(steps=len(val_data), exog=val_exog)
sarimax_test_pred = fitted_sarimax.forecast(steps=len(test_data), exog=test_exog)

# Metrics on validation set
sarimax_val_metrics = calculate_metrics(
    val_data['Requirement_MU'].values,
    sarimax_val_pred.values,
    model_name='SARIMAX_Validation'
)
print_metrics(sarimax_val_metrics)

```

```
# Metrics on test set
sarimax_test_metrics = calculate_metrics(
    test_data['Requirement_MU'].values,
    sarimax_test_pred.values,
    model_name='SARIMAX_Test'
)
print_metrics(sarimax_test_metrics)

# Residual analysis
sarimax_residuals = fitted_sarimax.resid
plot_residuals(sarimax_residuals, 'SARIMAX', f"{FOLDERS['results_viz']}/09_SARIMAX_residuals.png")
print("\nSaved: 09_SARIMAX_residuals.png")

# Save model summary
with open(f"{FOLDERS['reports']}/SARIMAX_model_summary.txt", 'w') as f:
    f.write(str(fitted_sarimax.summary()))

print("\nSARIMAX model completed")
```

```
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 8.47e+23. Standard errors may be unstable.
```

```
=====
Model: SARIMAX_Validation
=====
```

```
MAE (Mean Absolute Error):      4,813.69 MU
RMSE (Root Mean Squared Error):  5,702.15 MU
MAPE (Mean Absolute % Error):    3.63%
R² Score:                       0.7100
Directional Accuracy:            88.24%
=====
```

```
=====
Model: SARIMAX_Test
=====
```

```
MAE (Mean Absolute Error):      13,443.74 MU
RMSE (Root Mean Squared Error):  16,628.95 MU
MAPE (Mean Absolute % Error):    9.69%
R² Score:                       -2.2719
Directional Accuracy:            78.95%
=====
```

```
Saved: 09_SARIMAX_residuals.png
```

```
SARIMAX model completed
```

✓ COMPARING MODELS

```
print("MODEL COMPARISON")

# Compile all metrics
comparison_metrics = pd.DataFrame([
    arima_val_metrics,
    sarima_val_metrics,
    sarimax_val_metrics,
    arima_test_metrics,
    sarima_test_metrics,
    sarimax_test_metrics
])

# Save comparison
comparison_metrics.to_csv(f"{FOLDERS['results_eval']}/model_comparison_metrics.csv", index=False)

print("\nValidation Set Performance:")
print(comparison_metrics[comparison_metrics['Model'].str.contains('Validation')])

print("\nTest Set Performance:")
print(comparison_metrics[comparison_metrics['Model'].str.contains('Test')])
```

```

# Visualization: Metrics Comparison (Test Set)
test_comparison = comparison_metrics[comparison_metrics['Model'].str.contains('Test')].copy()
test_comparison['Model'] = test_comparison['Model'].str.replace('_Test', '')

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

metrics_to_plot = ['MAE', 'RMSE', 'MAPE', 'R2_Score', 'Directional_Accuracy']
colors = ['#3498db', '#e74c3c', '#2ecc71']

# MAE
axes[0, 0].bar(test_comparison['Model'], test_comparison['MAE'], color=colors)
axes[0, 0].set_title('Mean Absolute Error (MAE)', fontweight='bold')
axes[0, 0].set_ylabel('MAE (MU)')
axes[0, 0].grid(True, alpha=0.3, axis='y')

# RMSE
axes[0, 1].bar(test_comparison['Model'], test_comparison['RMSE'], color=colors)
axes[0, 1].set_title('Root Mean Squared Error (RMSE)', fontweight='bold')
axes[0, 1].set_ylabel('RMSE (MU)')
axes[0, 1].grid(True, alpha=0.3, axis='y')

# MAPE
axes[0, 2].bar(test_comparison['Model'], test_comparison['MAPE'], color=colors)
axes[0, 2].set_title('Mean Absolute Percentage Error (MAPE)', fontweight='bold')
axes[0, 2].set_ylabel('MAPE (%)')
axes[0, 2].grid(True, alpha=0.3, axis='y')

# R2 Score
axes[1, 0].bar(test_comparison['Model'], test_comparison['R2_Score'], color=colors)
axes[1, 0].set_title('R2 Score', fontweight='bold')
axes[1, 0].set_ylabel('R2 Score')
axes[1, 0].grid(True, alpha=0.3, axis='y')

# Directional Accuracy
axes[1, 1].bar(test_comparison['Model'], test_comparison['Directional_Accuracy'], color=colors)
axes[1, 1].set_title('Directional Accuracy', fontweight='bold')
axes[1, 1].set_ylabel('Accuracy (%)')
axes[1, 1].grid(True, alpha=0.3, axis='y')

# Summary Table
axes[1, 2].axis('tight')
axes[1, 2].axis('off')
table_data = []
for idx, row in test_comparison.iterrows():
    table_data.append([

```

```

        row['Model'],
        f"{row['MAE']:.2f}",
        f"{row['RMSE']:.2f}",
        f"{row['MAPE']:.2f}%",
        f"{row['R2_Score']:.4f}"
    ])
table = axes[1, 2].table(cellText=table_data,
                        collabels=['Model', 'MAE', 'RMSE', 'MAPE', 'R²'],
                        cellloc='center',
                        loc='center',
                        bbox=[0, 0, 1, 1])
table.auto_set_font_size(False)
table.set_fontsize(9)
table.scale(1, 2)

plt.suptitle('Model Performance Comparison (Test Set)', fontsize=16, fontweight='bold', y=0.98)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.savefig(f"{FOLDERS['results_eval']}/10_model_comparison.png", dpi=300, bbox_inches='tight')
plt.close()
print("\nSaved: 10_model_comparison.png")

```

MODEL COMPARISON

Validation Set Performance:

	Model	MAE	RMSE	MAPE	R2_Score	Directional_Accuracy
0	ARIMA_Validation	8761.111111	10659.005223	6.739654	-0.013402	41.176471
1	SARIMA_Validation	4949.424004	5869.984311	3.777934	0.692658	82.352941
2	SARIMAX_Validation	4813.691268	5702.146612	3.627113	0.709982	88.235294

Test Set Performance:

	Model	MAE	RMSE	MAPE	R2_Score	Directional_Accuracy
3	ARIMA_Test	15055.400000	17174.748097	10.183441	-2.490251	52.631579
4	SARIMA_Test	6215.459604	7024.210127	4.372942	0.416191	73.684211
5	SARIMAX_Test	13443.737198	16628.947568	9.693714	-2.271940	78.947368

Saved: 10_model_comparison.png

✓ ACTUALvsPREDICTED

```

print("GENERATING ACTUAL VS PREDICTED PLOTS")

# Combine all data for plotting
all_dates = pd.concat([train_data['Date'], val_data['Date'], test_data['Date']])
all_actual = pd.concat([train_data['Requirement_MU'], val_data['Requirement_MU'], test_data['Requirement_MU']])

# ARIMA predictions (need to re-forecast for complete series)

```

```

arima_all_pred = np.concatenate([
    arima_train_pred.values,
    arima_val_pred.values,
    arima_test_pred.values
])

# SARIMA predictions
sarima_all_pred = np.concatenate([
    sarima_train_pred.values,
    sarima_val_pred.values,
    sarima_test_pred.values
])

# SARIMAX predictions
sarimax_all_pred = np.concatenate([
    sarimax_train_pred.values,
    sarimax_val_pred.values,
    sarimax_test_pred.values
])

# Plot 1: Individual Model Predictions
fig, axes = plt.subplots(3, 1, figsize=(16, 12))

# ARIMA
axes[0].plot(all_dates, all_actual, label='Actual', color='black', linewidth=2, alpha=0.7)
axes[0].plot(all_dates, arima_all_pred, label='ARIMA Predicted', color='#3498db', linewidth=2, alpha=0.8)
axes[0].axvline(x=train_data['Date'].max(), color='red', linestyle='--', alpha=0.5)
axes[0].axvline(x=val_data['Date'].max(), color='red', linestyle='--', alpha=0.5)
axes[0].set_title('ARIMA: Actual vs Predicted', fontsize=14, fontweight='bold')
axes[0].set_ylabel('Electricity Requirement (MU)')
axes[0].legend(loc='best')
axes[0].grid(True, alpha=0.3)

# SARIMA
axes[1].plot(all_dates, all_actual, label='Actual', color='black', linewidth=2, alpha=0.7)
axes[1].plot(all_dates, sarima_all_pred, label='SARIMA Predicted', color='#e74c3c', linewidth=2, alpha=0.8)
axes[1].axvline(x=train_data['Date'].max(), color='red', linestyle='--', alpha=0.5)
axes[1].axvline(x=val_data['Date'].max(), color='red', linestyle='--', alpha=0.5)
axes[1].set_title('SARIMA: Actual vs Predicted', fontsize=14, fontweight='bold')
axes[1].set_ylabel('Electricity Requirement (MU)')
axes[1].legend(loc='best')
axes[1].grid(True, alpha=0.3)

# SARIMAX
axes[2].plot(all_dates, all_actual, label='Actual', color='black', linewidth=2, alpha=0.7)
axes[2].plot(all_dates, sarimax_all_pred, label='SARIMAX Predicted', color='#2ecc71', linewidth=2, alpha=0.8)

```

```

axes[2].axvline(x=train_data['Date'].max(), color='red', linestyle='--', alpha=0.5, label='Train-Val Split')
axes[2].axvline(x=val_data['Date'].max(), color='red', linestyle='--', alpha=0.5, label='Val-Test Split')
axes[2].set_title('SARIMAX: Actual vs Predicted', fontsize=14, fontweight='bold')
axes[2].set_ylabel('Electricity Requirement (MU)')
axes[2].set_xlabel('Date')
axes[2].legend(loc='best')
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/11_actual_vs_predicted_individual.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 11_actual_vs_predicted_individual.png")

# Plot 2: All Models Combined
plt.figure(figsize=(16, 8))
plt.plot(all_dates, all_actual, label='Actual', color='black', linewidth=2.5, alpha=0.8)
plt.plot(all_dates, arima_all_pred, label='ARIMA', color='#3498db', linewidth=1.5, alpha=0.7, linestyle='--')
plt.plot(all_dates, sarima_all_pred, label='SARIMA', color='#e74c3c', linewidth=1.5, alpha=0.7, linestyle='--')
plt.plot(all_dates, sarimax_all_pred, label='SARIMAX', color='#2ecc71', linewidth=1.5, alpha=0.7, linestyle='--')
plt.axvline(x=train_data['Date'].max(), color='gray', linestyle=':', alpha=0.5, label='Train-Val Split')
plt.axvline(x=val_data['Date'].max(), color='gray', linestyle=':', alpha=0.5, label='Val-Test Split')
plt.title('All Models: Actual vs Predicted Comparison', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Electricity Requirement (MU)', fontsize=12)
plt.legend(loc='best', fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/12_actual_vs_predicted_all_models.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 12_actual_vs_predicted_all_models.png")

# Plot 3: Test Set Focus
plt.figure(figsize=(14, 7))
plt.plot(test_data['Date'], test_data['Requirement_MU'], label='Actual',
         color='black', linewidth=3, marker='o', markersize=5, alpha=0.8)
plt.plot(test_data['Date'], arima_test_pred, label='ARIMA',
         color='#3498db', linewidth=2, marker='s', markersize=4, alpha=0.7)
plt.plot(test_data['Date'], sarima_test_pred, label='SARIMA',
         color='#e74c3c', linewidth=2, marker='^', markersize=4, alpha=0.7)
plt.plot(test_data['Date'], sarimax_test_pred, label='SARIMAX',
         color='#2ecc71', linewidth=2, marker='d', markersize=4, alpha=0.7)
plt.title('Test Set Performance: Actual vs Predicted (All Models)', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Electricity Requirement (MU)', fontsize=12)
plt.legend(loc='best', fontsize=11)
plt.grid(True, alpha=0.3)

```

```
plt.tight_layout()
plt.savefig(f"{FOLDERS['results_viz']}/13_test_set_comparison.png", dpi=300, bbox_inches='tight')
plt.close()
print("Saved: 13_test_set_comparison.png")
```

```
GENERATING ACTUAL VS PREDICTED PLOTS
Saved: 11_actual_vs_predicted_individual.png
Saved: 12_actual_vs_predicted_all_models.png
Saved: 13_test_set_comparison.png
```

✓ FORECASTING NEXT 6MONTHS

```
print("FUTURE FORECASTING (6 MONTHS AHEAD)")

# Generate future dates
last_date = test_data['Date'].max()
future_dates = pd.date_range(start=last_date + pd.DateOffset(months=1), periods=6, freq='MS')
future_df = pd.DataFrame({'Date': future_dates})
future_df['Year'] = future_df['Date'].dt.year
future_df['Month'] = future_df['Date'].dt.month

print(f"\nForecasting for: {future_dates[0].strftime('%Y-%m')} to {future_dates[-1].strftime('%Y-%m')}")

# Prepare future exogenous variables for SARIMAX
# Method 1: Use historical averages by month
historical_monthly_avg = merged_df.groupby('Month')[exog_vars].mean()

future_exog = pd.DataFrame()
for idx, row in future_df.iterrows():
    month = row['Month']
    future_exog = pd.concat([future_exog, historical_monthly_avg.loc[[month]]], ignore_index=True)

print("\nFuture exogenous variables (based on historical monthly averages):")
print(future_exog)

# ARIMA Future Forecast
print("\n[1/3] Generating ARIMA forecast...")
arima_future_pred = fitted_arima.forecast(steps=len(val_data) + len(test_data) + 6)
arima_future_pred = arima_future_pred[-6:]

# SARIMA Future Forecast
print("\n[2/3] Generating SARIMA forecast...")
sarima_future_pred = fitted_sarima.forecast(steps=6)

# SARIMAX Future Forecast
```

```

print("[3/3] Generating SARIMAX forecast...")
sarimax_future_pred = fitted_sarimax.forecast(steps=6, exog=future_exog)

# Create forecast dataframe
forecast_df = pd.DataFrame({
    'Date': future_dates,
    'ARIMA_Forecast': arima_future_pred.values,
    'SARIMA_Forecast': sarima_future_pred.values,
    'SARIMAX_Forecast': sarimax_future_pred.values
})

print("\nFuture Forecasts:")
print(forecast_df)

# Save forecasts
forecast_df.to_csv(f"{FOLDERS['results_forecast']}/future_forecasts_6months.csv", index=False)
print(f"\nSaved: future_forecasts_6months.csv")

# Visualize future forecasts
plt.figure(figsize=(16, 8))

# Plot historical data
plt.plot(merged_df['Date'], merged_df['Requirement_MU'],
        label='Historical Data', color='black', linewidth=2, alpha=0.7)

# Plot test set
plt.plot(test_data['Date'], test_data['Requirement_MU'],
        label='Test Set (Actual)', color='darkgray', linewidth=2.5, marker='o', markersize=5)

# Plot future forecasts
plt.plot(forecast_df['Date'], forecast_df['ARIMA_Forecast'],
        label='ARIMA Forecast', color='#3498db', linewidth=2, marker='s', markersize=6, linestyle='--')
plt.plot(forecast_df['Date'], forecast_df['SARIMA_Forecast'],
        label='SARIMA Forecast', color='#e74c3c', linewidth=2, marker='^', markersize=6, linestyle='--')
plt.plot(forecast_df['Date'], forecast_df['SARIMAX_Forecast'],
        label='SARIMAX Forecast', color='#2ecc71', linewidth=2, marker='d', markersize=6, linestyle='--')

plt.axvline(x=test_data['Date'].max(), color='red', linestyle=':', alpha=0.6, linewidth=2, label='Forecast Start')

plt.title('6-Month Future Forecasts (September 2025 - February 2026)', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=12)
plt.ylabel('Electricity Requirement (MU)', fontsize=12)
plt.legend(loc='best', fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig(f"{FOLDERS['results_forecast']}/14_future_forecasts.png", dpi=300, bbox_inches='tight')

```

```
plt.close()
print("Saved: 14_future_forecasts.png")
```

FUTURE FORECASTING (6 MONTHS AHEAD)

Forecasting for: 2025-09 to 2026-02

Future exogenous variables (based on historical monthly averages):

	Monthly_Festival_Index	Temperature_Mean	IIP_YoY
0	9.15	27.854	0.0236
1	14.95	26.708	0.0324
2	6.50	23.680	0.0274
3	6.00	20.830	0.0202
4	9.00	19.884	0.0261
5	1.75	22.060	0.0248

[1/3] Generating ARIMA forecast...

[2/3] Generating SARIMA forecast...

[3/3] Generating SARIMAX forecast...

Future Forecasts:

	Date	ARIMA_Forecast	SARIMA_Forecast	SARIMAX_Forecast
0	2025-09-01	128689.0	132288.613530	130915.809852
1	2025-10-01	128689.0	123021.603344	120859.438237
2	2025-11-01	128689.0	123158.274483	113349.658820
3	2025-12-01	128689.0	111104.345808	102562.785613
4	2026-01-01	128689.0	118853.069657	116170.619259
5	2026-02-01	128689.0	121370.343238	131048.473105

Saved: future_forecasts_6months.csv

Saved: 14_future_forecasts.png

✓ MODEL DIAGNOSTICS AND STATISTICAL TESTS

```
print("MODEL DIAGNOSTICS AND STATISTICAL TESTS")
```

```
def ljung_box_test(residuals, lags=20):
    """Perform Ljung-Box test on residuals"""
    lb_test = acorr_ljungbox(residuals, lags=lags, return_df=True)
    return lb_test
```

Ljung-Box Test for all models

```
print("\n[1/3] ARIMA Ljung-Box Test (Testing for autocorrelation in residuals):")
```

```
arma_lb = ljung_box_test(arma_residuals, lags=20)
```

```
print(arma_lb.head(10))
```

```
arma_lb.to_csv(f"{FOLDERS['reports']}/ARIMA_ljung_box_test.csv", index=False)
```

```
print("\n[2/3] SARIMA Ljung-Box Test:")
```

```

sarima_lb = ljung_box_test(sarima_residuals, lags=20)
print(sarima_lb.head(10))
sarima_lb.to_csv(f"{FOLDERS['reports']}/SARIMA_ljung_box_test.csv", index=False)

print("\n[3/3] SARIMAX Ljung-Box Test:")
sarimax_lb = ljung_box_test(sarimax_residuals, lags=20)
print(sarimax_lb.head(10))
sarimax_lb.to_csv(f"{FOLDERS['reports']}/SARIMAX_ljung_box_test.csv", index=False)

# Normality tests
from scipy.stats import shapiro, jarque_bera

def normality_tests(residuals, model_name):
    """Perform normality tests on residuals"""
    # Shapiro-Wilk test
    shapiro_stat, shapiro_p = shapiro(residuals)

    # Jarque-Bera test
    jb_stat, jb_p = jarque_bera(residuals)

    results = {
        'Model': model_name,
        'Shapiro_Statistic': shapiro_stat,
        'Shapiro_pvalue': shapiro_p,
        'JB_Statistic': jb_stat,
        'JB_pvalue': jb_p,
        'Shapiro_Normal': 'Yes' if shapiro_p > 0.05 else 'No',
        'JB_Normal': 'Yes' if jb_p > 0.05 else 'No'
    }
    return results

print("NORMALITY TESTS ON RESIDUALS")

arima_norm = normality_tests(arima_residuals, 'ARIMA')
sarima_norm = normality_tests(sarima_residuals, 'SARIMA')
sarimax_norm = normality_tests(sarimax_residuals, 'SARIMAX')

normality_df = pd.DataFrame([arima_norm, sarima_norm, sarimax_norm])
print("\n", normality_df)
normality_df.to_csv(f"{FOLDERS['reports']}/normality_tests.csv", index=False)

```

MODEL DIAGNOSTICS AND STATISTICAL TESTS

```

[1/3] ARIMA Ljung-Box Test (Testing for autocorrelation in residuals):
lb_stat lb_pvalue

```

```
1 0.060256 0.806091
2 0.204919 0.902615
3 0.310097 0.958119
4 1.025652 0.905882
5 1.092471 0.954768
6 1.670871 0.947348
7 2.076384 0.955492
8 2.089855 0.978133
9 2.134621 0.989155
10 2.153992 0.995018
```

[2/3] SARIMA Ljung-Box Test:

```
lb_stat lb_pvalue
1 0.199999 0.654722
2 0.312216 0.855467
3 0.487086 0.921719
4 0.649906 0.957360
5 0.856951 0.973260
6 0.882607 0.989674
7 1.576782 0.979532
8 1.867906 0.984797
9 1.955209 0.992156
10 1.957582 0.996655
```

[3/3] SARIMAX Ljung-Box Test:

```
lb_stat lb_pvalue
1 0.453729 0.500570
2 0.573761 0.750602
3 3.486978 0.322455
4 3.624772 0.459162
5 3.741973 0.587130
6 3.783391 0.705962
7 4.172822 0.759671
8 4.341046 0.825117
9 4.495147 0.875914
10 4.535092 0.919998
```

NORMALITY TESTS ON RESIDUALS

	Model	Shapiro_Statistic	Shapiro_pvalue	JB_Statistic	JB_pvalue	Shapiro_Normal	JB_Normal
0	ARIMA	0.641422	2.431980e-13	4023.510667	0.000000e+00	No	No
1	SARIMA	0.609942	6.108466e-14	2341.371009	0.000000e+00	No	No
2	SARIMAX	0.943595	8.030511e-04	35.051665	2.446964e-08	No	No

✓ REPORT

```
print("GENERATING COMPREHENSIVE SUMMARY REPORT")
```

```
summary_report = f"""
{'_'*80}
```

```
TIME SERIES FORECASTING PROJECT - COMPREHENSIVE SUMMARY REPORT
```

TIME SERIES FORECASTING PROJECT - COMPREHENSIVE SUMMARY REPORT

{ '_ '*80}

Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

Author: JIMS Rohini Student

Project: Electricity Demand Forecasting for India

{ '_ '*80}

1. DATASET INFORMATION

{ '_ '*80}

Target Variable: Electricity Requirement (MU)

Date Range: April 2015 to August 2025

Total Observations: {len(merged_df)}

Exogenous Variables (SARIMAX):

- Monthly Festival Index
- Temperature Mean (°C)
- IIP Year-over-Year Growth (%)

Data Split:

- Training Set: {len(train_data)} months ({len(train_data)/len(merged_df)*100:.1f}%)
Range: {train_data['Date'].min().strftime('%Y-%m')} to {train_data['Date'].max().strftime('%Y-%m')}
- Validation Set: {len(val_data)} months ({len(val_data)/len(merged_df)*100:.1f}%)
Range: {val_data['Date'].min().strftime('%Y-%m')} to {val_data['Date'].max().strftime('%Y-%m')}
- Test Set: {len(test_data)} months ({len(test_data)/len(merged_df)*100:.1f}%)
Range: {test_data['Date'].min().strftime('%Y-%m')} to {test_data['Date'].max().strftime('%Y-%m')}

{ '_ '*80}

2. STATIONARITY ANALYSIS

{ '_ '*80}

Augmented Dickey-Fuller (ADF) Test:

Test Statistic: {adf_results['Test Statistic']:.6f}

p-value: {adf_results['p-value']:.6f}

Result: {'STATIONARY' if adf_results['p-value'] < 0.05 else 'NON-STATIONARY'}

KPSS Test:

Test Statistic: {kpss_results['Test Statistic']:.6f}

p-value: {kpss_results['p-value']:.6f}

Result: {'STATIONARY' if kpss_results['p-value'] >= 0.05 else 'NON-STATIONARY'}

{ '_ '*80}

3. MODEL SPECIFICATIONS

{ '_ '*80}

MODEL 1: ARIMA

Parameters: {best_params}
AIC: {best_aic:.2f}
Parameter Selection: Grid Search

MODEL 2: SARIMA

Order (p,d,q): {best_params_sarima[0]}
Seasonal (P,D,Q,m): {best_params_sarima[1]}
AIC: {best_aic_sarima:.2f}
Parameter Selection: Grid Search

MODEL 3: SARIMAX

Order (p,d,q): {best_params_sarimax[0]}
Seasonal (P,D,Q,m): {best_params_sarimax[1]}
Exogenous Variables: 3 (Festival Index, Temperature, IIP)
AIC: {best_aic_sarimax:.2f}
Parameter Selection: Grid Search

{'_'*80}

4. VALIDATION SET PERFORMANCE

{'_'*80}

ARIMA:

MAE: {arima_val_metrics['MAE']:.2f} MU
RMSE: {arima_val_metrics['RMSE']:.2f} MU
MAPE: {arima_val_metrics['MAPE']:.2f}%
R²: {arima_val_metrics['R2_Score']:.4f}
Directional Accuracy: {arima_val_metrics['Directional_Accuracy']:.2f}%

SARIMA:

MAE: {sarima_val_metrics['MAE']:.2f} MU
RMSE: {sarima_val_metrics['RMSE']:.2f} MU
MAPE: {sarima_val_metrics['MAPE']:.2f}%
R²: {sarima_val_metrics['R2_Score']:.4f}
Directional Accuracy: {sarima_val_metrics['Directional_Accuracy']:.2f}%

SARIMAX:

MAE: {sarimax_val_metrics['MAE']:.2f} MU
RMSE: {sarimax_val_metrics['RMSE']:.2f} MU
MAPE: {sarimax_val_metrics['MAPE']:.2f}%
R²: {sarimax_val_metrics['R2_Score']:.4f}
Directional Accuracy: {sarimax_val_metrics['Directional_Accuracy']:.2f}%

{'_'*80}

5. TEST SET PERFORMANCE (FINAL EVALUATION)

{'_'*80}

ARIMA:

```
MAE: {arima_test_metrics['MAE']:, .2f} MU
RMSE: {arima_test_metrics['RMSE']:, .2f} MU
MAPE: {arima_test_metrics['MAPE']:, .2f}%
R²: {arima_test_metrics['R2_Score']:, .4f}
Directional Accuracy: {arima_test_metrics['Directional_Accuracy']:, .2f}%
```

SARIMA:

```
MAE: {sarima_test_metrics['MAE']:, .2f} MU
RMSE: {sarima_test_metrics['RMSE']:, .2f} MU
MAPE: {sarima_test_metrics['MAPE']:, .2f}%
R²: {sarima_test_metrics['R2_Score']:, .4f}
Directional Accuracy: {sarima_test_metrics['Directional_Accuracy']:, .2f}%
```

SARIMAX:

```
MAE: {sarimax_test_metrics['MAE']:, .2f} MU
RMSE: {sarimax_test_metrics['RMSE']:, .2f} MU
MAPE: {sarimax_test_metrics['MAPE']:, .2f}%
R²: {sarimax_test_metrics['R2_Score']:, .4f}
Directional Accuracy: {sarimax_test_metrics['Directional_Accuracy']:, .2f}%
```

{'_'*80}

6. BEST MODEL SELECTION

{'_'*80}

Based on Test Set Performance:

```
Best MAE: {test_comparison.loc[test_comparison['MAE'].idxmin(), 'Model']} ({test_comparison['MAE'].min():, .2f} MU)
Best RMSE: {test_comparison.loc[test_comparison['RMSE'].idxmin(), 'Model']} ({test_comparison['RMSE'].min():, .2f} MU)
Best MAPE: {test_comparison.loc[test_comparison['MAPE'].idxmin(), 'Model']} ({test_comparison['MAPE'].min():, .2f}%)
Best R²: {test_comparison.loc[test_comparison['R2_Score'].idxmax(), 'Model']} ({test_comparison['R2_Score'].max():, .4f})
Best Directional Accuracy: {test_comparison.loc[test_comparison['Directional_Accuracy'].idxmax(), 'Model']} ({test_comparison['Directional_Accuracy'].max():, .2f}%)
```

{'_'*80}

7. FUTURE FORECASTS (6 MONTHS AHEAD)

{'_'*80}

Forecast Period: September 2025 - February 2026

```
{forecast_df.to_string(index=False)}
```

{'_'*80}

8. MODEL DIAGNOSTICS

{'_'*80}

Ljung-Box Test Results (p-values < 0.05 indicate autocorrelation):

Ljung-Box test results (p-values < 0.05 indicate autocorrelation):

ARIMA: See detailed results in ARIMA_ljung_box_test.csv

SARIMA: See detailed results in SARIMA_ljung_box_test.csv

SARIMAX: See detailed results in SARIMAX_ljung_box_test.csv

Normality Tests:

ARIMA Residuals: Shapiro p-value = {arima_norm['Shapiro_pvalue']:.6f} ({'Normal' if arima_norm['Shapiro_Normal']=='Yes' else 'Non-normal'})

SARIMA Residuals: Shapiro p-value = {sarima_norm['Shapiro_pvalue']:.6f} ({'Normal' if sarima_norm['Shapiro_Normal']=='Yes' else 'Non-normal'})

SARIMAX Residuals: Shapiro p-value = {sarimax_norm['Shapiro_pvalue']:.6f} ({'Normal' if sarimax_norm['Shapiro_Normal']=='Yes' else 'Non-normal'})

{'_'*80}

9. KEY INSIGHTS

{'_'*80}

1. Seasonal Pattern: The data shows strong seasonality with period = 12 months
2. Trend: {'Upward' if decomposition.trend[-1] > decomposition.trend[0] else 'Downward'} trend observed
3. Exogenous Variables Impact: SARIMAX incorporates festival patterns, temperature, and industrial production which affect electricity demand
4. Best Performer: {test_comparison.loc[test_comparison['MAE'].idxmin()], 'Model'}} achieved lowest prediction error

{'_'*80}

10. FILES GENERATED

{'_'*80}

Data Files:

- merged_data.csv (Cleaned and merged dataset)
- future_forecasts_6months.csv (6-month ahead forecasts)

Model Reports:

- ARIMA_model_summary.txt
- SARIMA_model_summary.txt
- SARIMAX_model_summary.txt
- ARIMA_grid_search_results.csv
- SARIMA_grid_search_results.csv
- SARIMAX_grid_search_results.csv

Evaluations:

- model_comparison_metrics.csv
- descriptive_statistics.csv
- correlation_matrix.csv
- stationarity_tests.csv
- normality_tests.csv
- seasonal_decomposition.csv
- ARIMA_ljung_box_test.csv
- SARIMA_ljung_box_test.csv
- SARIMAX_ljung_box_test.csv

Visualizations:

- 01_time_series_overview.png
- 02_correlation_matrix.png
- 03_distributions.png
- 04_seasonal_decomposition.png
- 05_train_val_test_split.png
- 06_acf_pacf_plots.png
- 07_ARIMA_residuals.png
- 08_SARIMA_residuals.png
- 09_SARIMAX_residuals.png
- 10_model_comparison.png
- 11_actual_vs_predicted_individual.png
- 12_actual_vs_predicted_all_models.png
- 13_test_set_comparison.png
- 14_future_forecasts.png

```
{'_' '*80}
```

```
END OF REPORT
```

```
{'_' '*80}
```

```
"""
```

```
# Save summary report
```

```
with open(f"{FOLDERS['reports']}/COMPREHENSIVE_SUMMARY_REPORT.txt", 'w') as f:
```

```
    f.write(summary_report)
```

```
print(summary_report)
```

```
print(f"\nSaved: COMPREHENSIVE_SUMMARY_REPORT.txt")
```

- SARIMA_grid_search_results.csv
- SARIMAX_grid_search_results.csv

Evaluations:

- model_comparison_metrics.csv
- descriptive_statistics.csv
- correlation_matrix.csv
- stationarity_tests.csv
- normality_tests.csv
- seasonal_decomposition.csv
- ARIMA_ljung_box_test.csv
- SARIMA_ljung_box_test.csv
- SARIMAX_ljung_box_test.csv

Visualizations:

- 01_time_series_overview.png
- 02_correlation_matrix.png
- 03_distributions.png
- 04_seasonal_decomposition.png
- 05_train_val_test_split.png
- 06_acf_pacf_plots.png
- 07_ARIMA_residuals.png
- 08_SARIMA_residuals.png
- 09_SARIMAX_residuals.png
- 10_model_comparison.png
- 11_actual_vs_predicted_individual.png
- 12_actual_vs_predicted_all_models.png
- 13_test_set_comparison.png
- 14_future_forecasts.png

END OF REPORT

Saved: COMPREHENSIVE_SUMMARY_REPORT.txt

✓ CREATE MODEL PREDICTIONS CSV FILES

```
print("SAVING DETAILED PREDICTIONS")

# Create detailed predictions dataframe
predictions_df = pd.DataFrame({
    'Date': all_dates.values,
    'Actual': all_actual.values,
    'ARIMA_Predicted': arima_all_pred,
    'SARIMA_Predicted': sarima_all_pred,
    'SARIMAX_Predicted': sarimax_all_pred,
    'ARIMA_Error': all_actual.values - arima_all_pred,
```

```

        'SARIMA_Error': all_actual.values - sarima_all_pred,
        'SARIMAX_Error': all_actual.values - sarimax_all_pred
    })

    # Add split indicator
    predictions_df['Data_Split'] = 'Train'
    predictions_df.loc[predictions_df['Date'] >= val_data['Date'].min(), 'Data_Split'] = 'Validation'
    predictions_df.loc[predictions_df['Date'] >= test_data['Date'].min(), 'Data_Split'] = 'Test'

    predictions_df.to_csv(f"{FOLDERS['results_forecast']}/all_predictions_detailed.csv", index=False)
    print("Saved: all_predictions_detailed.csv")

```

```

SAVING DETAILED PREDICTIONS
Saved: all_predictions_detailed.csv

```

✓ FINAL SUMMARY

```

print("PROJECT EXECUTION COMPLETEDDDDD")

execution_end_time = datetime.now()
print(f"\nExecution completed at: {execution_end_time.strftime('%Y-%m-%d %H:%M:%S')}")

print("ALL FILES SAVED TO GOOGLE DRIVE")

print(f"\nBase Directory: {BASE_PATH}")
print("\nFolder Structure:")
for folder_name, folder_path in FOLDERS.items():
    print(f"    - {folder_name}: {folder_path}")

print("NEXT STEPS")

print("""
1. Review the COMPREHENSIVE_SUMMARY_REPORT.txt for detailed analysis
2. Check visualizations in results/visualizations/
3. Review model performance metrics in results/evaluations/
4. Examine future forecasts in results/forecasts/
5. Use model diagnostics for your project report

All datasets, graphs, and evaluations have been exported to your Google Drive.
""")

```

```
PROJECT EXECUTION COMPLETEDDDDD
```

Execution completed at: 2025-11-20 18:15:53

ALL FILES SAVED TO GOOGLE DRIVE

Base Directory: /content/drive/MyDrive/Time_Series_Project

Folder Structure:

- data_raw: /content/drive/MyDrive/Time_Series_Project/data/raw
- data_cleaned: /content/drive/MyDrive/Time_Series_Project/data/cleaned
- models: /content/drive/MyDrive/Time_Series_Project/models
- results_viz: /content/drive/MyDrive/Time_Series_Project/results/visualizations
- results_eval: /content/drive/MyDrive/Time_Series_Project/results/evaluations
- results_forecast: /content/drive/MyDrive/Time_Series_Project/results/forecasts
- reports: /content/drive/MyDrive/Time_Series_Project/reports

NEXT STEPS

1. Review the COMPREHENSIVE_SUMMARY_REPORT.txt for detailed analysis
2. Check visualizations in results/visualizations/
3. Review model performance metrics in results/evaluations/
4. Examine future forecasts in results/forecasts/
5. Use model diagnostics for your project report

All datasets, graphs, and evaluations have been exported to your Google Drive.