```python
"""
Complete Time Series Forecasting for Electricity Load Dataset
Dataset: UCI Electricity Load Diagrams 2011-2014
"""

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

# Statistical and ML libraries
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb

# Deep Learning
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

print("Libraries imported successfully!")

# ============================================================================
# 1. DATA LOADING AND PREPROCESSING
# ============================================================================

def load_electricity_data(filepath='LD2011_2014.txt'):
    """
    Load and preprocess the electricity load dataset
    The dataset contains 370 clients with electricity consumption recorded every 15 minutes
    """
    print("Loading dataset...")

    # The file is semicolon-separated with datetime index
    df = pd.read_csv(filepath, sep=';', decimal=',',
                     parse_dates=True, index_col=0)

    print(f"Dataset shape: {df.shape}")
    print(f"Date range: {df.index.min()} to {df.index.max()}")
    print(f"Number of clients: {df.shape[1]}")

    return df

def preprocess_data(df, client_id='MT_001', resample_freq='H'):
    """
    Preprocess data for a specific client
    - Handle missing values
    - Resample to desired frequency
    - Create time-based features
    """
    print(f"\nPreprocessing data for client: {client_id}")

    # Select single client data
    if client_id not in df.columns:
        print(f"Client {client_id} not found. Using first client.")
        client_id = df.columns[0]

    series = df[client_id].copy()

    # Handle missing values
    print(f"Missing values: {series.isna().sum()}")
    series = series.interpolate(method='linear')

    # Resample to hourly data (from 15-minute intervals)
    series = series.resample(resample_freq).mean()

    # Create DataFrame with features
    data = pd.DataFrame({'consumption': series})

    # Time-based features
    data['hour'] = data.index.hour
    data['day_of_week'] = data.index.dayofweek
    data['day_of_month'] = data.index.day
    data['month'] = data.index.month
    data['quarter'] = data.index.quarter
    data['year'] = data.index.year
    data['is_weekend'] = (data['day_of_week'] >= 5).astype(int)

    # Cyclical encoding for hour and month
    data['hour_sin'] = np.sin(2 * np.pi * data['hour'] / 24)
    data['hour_cos'] = np.cos(2 * np.pi * data['hour'] / 24)
    data['month_sin'] = np.sin(2 * np.pi * data['month'] / 12)
    data['month_cos'] = np.cos(2 * np.pi * data['month'] / 12)

    # Lag features
    for lag in [1, 24, 168]:  # 1 hour, 1 day, 1 week
        data[f'lag_{lag}'] = data['consumption'].shift(lag)
```

```python
    # Rolling statistics
    data['rolling_mean_24'] = data['consumption'].rolling(window=24).mean()
    data['rolling_std_24'] = data['consumption'].rolling(window=24).std()

    # Drop rows with NaN values created by lag features
    data = data.dropna()

    print(f"Preprocessed data shape: {data.shape}")

    return data

# ==============================================================================
# 2. EXPLORATORY DATA ANALYSIS
# ==============================================================================

def perform_eda(data):
    """Perform exploratory data analysis"""
    print("\n" + "="*50)
    print("EXPLORATORY DATA ANALYSIS")
    print("="*50)

    fig, axes = plt.subplots(3, 2, figsize=(15, 12))

    # Time series plot
    axes[0, 0].plot(data.index, data['consumption'], linewidth=0.5)
    axes[0, 0].set_title('Electricity Consumption Over Time')
    axes[0, 0].set_xlabel('Date')
    axes[0, 0].set_ylabel('Consumption (kW)')

    # Distribution
    axes[0, 1].hist(data['consumption'], bins=50, edgecolor='black')
    axes[0, 1].set_title('Distribution of Consumption')
    axes[0, 1].set_xlabel('Consumption (kW)')
    axes[0, 1].set_ylabel('Frequency')

    # Hourly pattern
    hourly_avg = data.groupby('hour')['consumption'].mean()
    axes[1, 0].plot(hourly_avg.index, hourly_avg.values, marker='o')
    axes[1, 0].set_title('Average Consumption by Hour of Day')
    axes[1, 0].set_xlabel('Hour')
    axes[1, 0].set_ylabel('Average Consumption (kW)')
    axes[1, 0].grid(True)

    # Weekly pattern
    weekly_avg = data.groupby('day_of_week')['consumption'].mean()
    days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    axes[1, 1].bar(range(7), weekly_avg.values)
    axes[1, 1].set_xticks(range(7))
    axes[1, 1].set_xticklabels(days)
    axes[1, 1].set_title('Average Consumption by Day of Week')
    axes[1, 1].set_ylabel('Average Consumption (kW)')

    # Monthly pattern
    monthly_avg = data.groupby('month')['consumption'].mean()
    axes[2, 0].plot(monthly_avg.index, monthly_avg.values, marker='o')
    axes[2, 0].set_title('Average Consumption by Month')
    axes[2, 0].set_xlabel('Month')
    axes[2, 0].set_ylabel('Average Consumption (kW)')
    axes[2, 0].grid(True)

    # Box plot by month
    data.boxplot(column='consumption', by='month', ax=axes[2, 1])
    axes[2, 1].set_title('Consumption Distribution by Month')
    axes[2, 1].set_xlabel('Month')
    axes[2, 1].set_ylabel('Consumption (kW)')

    plt.tight_layout()
    plt.savefig('eda_analysis.png', dpi=300, bbox_inches='tight')
    print("EDA plots saved as 'eda_analysis.png'")

    # Statistical summary
    print("\nStatistical Summary:")
    print(data['consumption'].describe())

    # Stationarity test (ADF test)
    result = adfuller(data['consumption'].dropna())
    print(f"\nADF Statistic: {result[0]:.4f}")
    print(f"p-value: {result[1]:.4f}")
    print("Time series is", "stationary" if result[1] < 0.05 else "non-stationary")

# ============================================================================
# 3. TRAIN-TEST SPLIT
# ============================================================================

def train_test_split_ts(data, test_size=0.2):
    """Split time series data maintaining temporal order"""
    split_idx = int(len(data) * (1 - test_size))

    train = data.iloc[:split_idx]
    test = data.iloc[split_idx:]

    print(f"\nTrain set: {len(train)} samples ({train.index.min()} to {train.index.max()})")
    print(f"Test set: {len(test)} samples ({test.index.min()} to {test.index.max()})")

    return train, test

# ============================================================================
# 4. BASELINE MODELS
# ============================================================================
```

```python
def naive_forecast(train, test):
    """Simple naive forecast - last observation carried forward"""
    predictions = [train['consumption'].iloc[-1]] * len(test)
    return np.array(predictions)

def moving_average_forecast(train, test, window=24):
    """Moving average forecast"""
    predictions = []
    history = train['consumption'].values.tolist()

    for i in range(len(test)):
        avg = np.mean(history[-window:])
        predictions.append(avg)
        history.append(test['consumption'].iloc[i])

    return np.array(predictions)

# ============================================================================
# 5. STATISTICAL MODELS
# ============================================================================

def arima_forecast(train, test, order=(2, 1, 2)):
    """ARIMA model forecast"""
    print(f"\nTraining ARIMA{order} model...")

    model = ARIMA(train['consumption'], order=order)
    fitted_model = model.fit()

    print(fitted_model.summary())

    # Forecast
    predictions = fitted_model.forecast(steps=len(test))

    return predictions.values

def exponential_smoothing_forecast(train, test):
    """Exponential Smoothing with trend and seasonality"""
    print("\nTraining Exponential Smoothing model...")

    model = ExponentialSmoothing(
        train['consumption'],
        seasonal_periods=24,  # Daily seasonality for hourly data
        trend='add',
        seasonal='add'
    )
    fitted_model = model.fit()

    predictions = fitted_model.forecast(steps=len(test))

    return predictions.values
```

```
# ============================================================================
# 6. MACHINE LEARNING MODELS
# ============================================================================

def prepare_ml_data(train, test, target_col='consumption'):
    """Prepare data for ML models"""
    feature_cols = [col for col in train.columns if col != target_col]

    X_train = train[feature_cols]
    y_train = train[target_col]
    X_test = test[feature_cols]
    y_test = test[target_col]

    # Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    return X_train_scaled, y_train, X_test_scaled, y_test, scaler, feature_cols

def random_forest_forecast(train, test):
    """Random Forest model"""
    print("\nTraining Random Forest model...")

    X_train, y_train, X_test, y_test, _, _ = prepare_ml_data(train, test)

    model = RandomForestRegressor(
        n_estimators=100,
        max_depth=20,
        min_samples_split=5,
        random_state=42,
        n_jobs=-1
    )

    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    return predictions

def xgboost_forecast(train, test):
    """XGBoost model"""
    print("\nTraining XGBoost model...")

    X_train, y_train, X_test, y_test, _, _ = prepare_ml_data(train, test)

    model = xgb.XGBRegressor(
        n_estimators=100,
        max_depth=7,
        learning_rate=0.1,
        subsample=0.8,
        colsample_bytree=0.8,
        random_state=42
    )

    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    return predictions

# ==============================================================================
# 7. DEEP LEARNING MODEL (LSTM)
# ==============================================================================

def create_sequences(data, seq_length):
    """Create sequences for LSTM"""
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i + seq_length])
        y.append(data[i + seq_length])
    return np.array(X), np.array(y)

def lstm_forecast(train, test, seq_length=24):
    """LSTM model for time series forecasting"""
    print("\nTraining LSTM model...")

    # Prepare data
    scaler = StandardScaler()
    train_scaled = scaler.fit_transform(train[['consumption']])
    test_scaled = scaler.transform(test[['consumption']])

    X_train, y_train = create_sequences(train_scaled, seq_length)
    X_test, y_test = create_sequences(test_scaled, seq_length)

    # Build LSTM model
    model = Sequential([
        LSTM(50, activation='relu', return_sequences=True, input_shape=(seq_length, 1)),
        Dropout(0.2),
        LSTM(50, activation='relu'),
        Dropout(0.2),
        Dense(1)
    ])

    model.compile(optimizer='adam', loss='mse', metrics=['mae'])

    # Train model
    early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

    history = model.fit(
        X_train, y_train,
        epochs=50,
        batch_size=32,
        validation_split=0.1,
        callbacks=[early_stop],
        verbose=0
    )

    # Make predictions
    predictions_scaled = model.predict(X_test, verbose=0)
    predictions = scaler.inverse_transform(predictions_scaled)

    # Adjust lengths to match
    actual_values = test['consumption'].values[seq_length:]

    return predictions.flatten(), actual_values

# ================================================================================
# 8. MODEL EVALUATION
# ================================================================================

def evaluate_model(y_true, y_pred, model_name):
    """Calculate evaluation metrics"""
    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)

    print(f"\n{model_name} Performance:")
    print(f"  MAE:  {mae:.4f}")
    print(f"  RMSE: {rmse:.4f}")
    print(f"  MAPE: {mape:.2f}%")
    print(f"  R²:   {r2:.4f}")

    return {'MAE': mae, 'RMSE': rmse, 'MAPE': mape, 'R2': r2}

def plot_predictions(test, predictions_dict):
    """Plot actual vs predicted values for all models"""
    fig, axes = plt.subplots(len(predictions_dict), 1,
                             figsize=(15, 4 * len(predictions_dict)))

    if len(predictions_dict) == 1:
        axes = [axes]

    for idx, (model_name, predictions) in enumerate(predictions_dict.items()):
        # Handle different length predictions (LSTM case)
        if len(predictions) < len(test):
            test_subset = test.iloc[-len(predictions):]
            axes[idx].plot(test_subset.index, test_subset['consumption'],
                          label='Actual', linewidth=1)
            axes[idx].plot(test_subset.index, predictions,
                          label='Predicted', linewidth=1, alpha=0.8)
        else:
            axes[idx].plot(test.index, test['consumption'],
                          label='Actual', linewidth=1)
            axes[idx].plot(test.index, predictions,
                          label='Predicted', linewidth=1, alpha=0.8)

        axes[idx].set_title(f'{model_name} - Actual vs Predicted')
        axes[idx].set_xlabel('Date')
        axes[idx].set_ylabel('Consumption (kW)')
        axes[idx].legend()
        axes[idx].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('model_predictions.png', dpi=300, bbox_inches='tight')
    print("\nPrediction plots saved as 'model_predictions.png'")

# ==============================================================================
# 9. MAIN EXECUTION
# ==============================================================================

def main():
    """Main execution function"""
    print("="*60)
    print("ELECTRICITY LOAD TIME SERIES FORECASTING")
    print("="*60)

    # Note: Download the dataset first
    print("\nNote: Please download the dataset from:")
    print("https://archive.ics.uci.edu/dataset/321/electricityloaddiagrams20112014")
    print("Extract and place 'LD2011_2014.txt' in the working directory")
    print("\nFor demonstration, using sample data...")

    # For demonstration, create sample data
    # In practice, replace this with: df = load_electricity_data('LD2011_2014.txt')
    dates = pd.date_range('2012-01-01', periods=8760, freq='H')
    np.random.seed(42)
    consumption = 50 + 20 * np.sin(np.arange(8760) * 2 * np.pi / 24) + \
                  10 * np.sin(np.arange(8760) * 2 * np.pi / (24*7)) + \
                  np.random.normal(0, 5, 8760)
    df = pd.DataFrame({'MT_001': consumption}, index=dates)

    # Preprocess
    data = preprocess_data(df, client_id='MT_001', resample_freq='H')

    # EDA
    perform_eda(data)

    # Train-test split
    train, test = train_test_split_ts(data, test_size=0.2)
    # Store predictions and metrics
    predictions_dict = {}
    metrics_dict = {}

    # 1. Baseline: Naive Forecast
    naive_pred = naive_forecast(train, test)
    predictions_dict['Naive'] = naive_pred
    metrics_dict['Naive'] = evaluate_model(test['consumption'].values, naive_pred, 'Naive
Forecast')

    # 2. Moving Average
    ma_pred = moving_average_forecast(train, test, window=24)
    predictions_dict['Moving Average'] = ma_pred
    metrics_dict['Moving Average'] = evaluate_model(test['consumption'].values, ma_pred, 'Moving
Average')

    # 3. ARIMA
    try:
        arima_pred = arima_forecast(train, test, order=(2, 1, 2))
        predictions_dict['ARIMA'] = arima_pred
        metrics_dict['ARIMA'] = evaluate_model(test['consumption'].values, arima_pred, 'ARIMA')
    except Exception as e:
        print(f"\nARIMA model failed: {e}")

    # 4. Exponential Smoothing
    try:
        es_pred = exponential_smoothing_forecast(train, test)
        predictions_dict['Exp Smoothing'] = es_pred
        metrics_dict['Exp Smoothing'] = evaluate_model(test['consumption'].values, es_pred,
'Exponential Smoothing')
    except Exception as e:
        print(f"\nExponential Smoothing failed: {e}")

    # 5. Random Forest
    rf_pred = random_forest_forecast(train, test)
    predictions_dict['Random Forest'] = rf_pred
    metrics_dict['Random Forest'] = evaluate_model(test['consumption'].values, rf_pred, 'Random
Forest')

    # 6. XGBoost
    xgb_pred = xgboost_forecast(train, test)
    predictions_dict['XGBoost'] = xgb_pred
    metrics_dict['XGBoost'] = evaluate_model(test['consumption'].values, xgb_pred, 'XGBoost')

    # 7. LSTM
    try:
        lstm_pred, lstm_actual = lstm_forecast(train, test, seq_length=24)
        predictions_dict['LSTM'] = lstm_pred
        metrics_dict['LSTM'] = evaluate_model(lstm_actual, lstm_pred, 'LSTM')
    except Exception as e:
        print(f"\nLSTM model failed: {e}")

    # Plot all predictions
    plot_predictions(test, predictions_dict)

    # Summary comparison
    print("\n" + "="*60)
    print("MODEL COMPARISON SUMMARY")
    print("="*60)
    comparison_df = pd.DataFrame(metrics_dict).T
    comparison_df = comparison_df.sort_values('RMSE')
    print(comparison_df.to_string())

    print("\n" + "="*60)
    print("FORECASTING COMPLETE!")
    print("="*60)
    print("\nBest model based on RMSE:", comparison_df.index[0])

if __name__ == "__main__":
    main()
```