DDU-framework

January 27, 2026

[5]:
```
%load_ext autoreload
%autoreload 2
```

[7]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso, LassoCV
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler
from datetime import datetime, timedelta
from IPython.display import display, Math
import re
from arch import arch_model
from arch.univariate import ZeroMean, GARCH, Normal
from scipy.optimize import minimize
from scipy.stats import norm
import pickle
from statsmodels.graphics.tsaplots import plot_acf
import scipy.stats as stats
```

1 Functions

[9]:
```
def plot_smoothed_residuals(date_strings, res, win, season):

    # Calculate evenly spaced indices
    indices = np.linspace(0, len(date_strings) - 1, num_labels_to_show,␣
 ↪dtype=int)
    selected_dates = date_strings[indices]

    # Take absolute value of residuals and apply rolling window smoothing␣
 ↪algorithm
    y_res = abs(res).rolling(window=win).mean()

    # Plot smoothed residuals
    plt.plot(date_strings, y_res)
```

```python
    # Apply sampled ticks and rotated labels
    ax = plt.gca()
    ax.set_xticks(selected_dates)
    ax.set_xticklabels(selected_dates, rotation=-45)
    #ax.set_ylim([0, 1.75])

    plt.title(season + " Season Smoothed Residuals")
    plt.show()

    return y_res
```

[10]:
```python
def garch_exog_loglik(params, y, x):
    """
    Log-likelihood for GARCH(1,1) with exogenous variable in variance equation
    """
    omega, alpha, beta, gamma = params
    n = len(y)

    # Initialize
    sigma2 = np.var(y)  # Initial variance
    loglik = 0

    for t in range(1, n):
        # GARCH variance equation with exogenous variable
        sigma2 = omega + alpha * y[t-1]**2 + beta * sigma2 + gamma * x[t]

        # Ensure positive variance
        sigma2 = max(sigma2, 1e-8)

        # Log-likelihood contribution (assuming normal errors)
        loglik += -0.5 * (np.log(2 * np.pi) + np.log(sigma2) + y[t]**2 / sigma2)

    return -loglik  # Return negative for minimization
```

[11]:
```python
def fit_garch_exog(y, x):
    """
    Fit GARCH(1,1) with exogenous variable in variance equation
    """
    # Initial parameter guesses
    init_params = [np.var(y) * 0.01, 0.1, 0.8, 0.01]

    # Parameter bounds and constraints
    bounds = [(1e-8, None),     # omega > 0
              (0, 1),           # 0 <= alpha < 1
              (0, 1),           # 0 <= beta < 1
              (None, None)]     # gamma can be any value
```

```python
    constraints = [{'type': 'ineq',
                    'fun': lambda params: 0.9999 - params[1] - params[2]}]  #␣
↪alpha + beta < 1

    # Explicit convergence criteria
    options = {
        'ftol': 1e-9,      # Function tolerance
        'gtol': 1e-6,      # Gradient tolerance
        'xtol': 1e-6,      # Parameter tolerance
        'maxiter': 1000,   # Maximum iterations
        'disp': True,      # Display convergence info
        'eps': 1.4901161193847656e-08  # Step size for gradient approximation
    }

    # Store initial values for comparison
    initial_params = np.array(init_params)
    initial_func_val = garch_exog_loglik(initial_params, y, x)

    # Optimize
    result = minimize(garch_exog_loglik, init_params,
                      args=(y, x), method='SLSQP',
                      bounds=bounds, constraints=constraints,
                      options=options)

    # Calculate convergence metrics
    final_grad_norm = np.linalg.norm(result.jac) if hasattr(result, 'jac') else␣
↪None
    final_func_val = result.fun
    final_params = result.x

    # Calculate relative changes (approximate since we don't have␣
↪second-to-last values)
    # These are approximations - the actual algorithm tracks␣
↪iteration-to-iteration changes
    param_change_norm = np.linalg.norm(final_params - initial_params)
    func_change = abs(final_func_val - initial_func_val)
    relative_func_change = func_change / abs(initial_func_val) if␣
↪initial_func_val != 0 else func_change

    # Print detailed convergence information
    print(f"=== Convergence Analysis ===")
    print(f"Convergence achieved: {result.success}")
    print(f"Termination message: {result.message}")
    print(f"Number of iterations: {result.nit}")
    print(f"Function evaluations: {result.nfev}")
    print(f"")
    print(f"=== Tolerance Checks ===")
```

```
    print(f"Final function value: {final_func_val}")
    print(f"Function tolerance: {options['ftol']}")
    print(f"Relative function change (total): {relative_func_change:.2e}")
    print(f"")
    print(f"Final gradient norm: {final_grad_norm}")
    print(f"Gradient tolerance: {options['gtol']}")
    print(f"Gradient criterion met: {final_grad_norm < options['gtol'] if␣
↪final_grad_norm else 'Unknown'}")
    print(f"")
    print(f"Parameter change norm (total): {param_change_norm:.2e}")
    print(f"Parameter tolerance: {options['xtol']}")
    print(f"")
    print(f"Final parameters: {final_params}")

    return result
```

[12]:
```
def forecast_volatility(result, y_train, x_test):
    """
    Simple volatility forecasting for GARCH-X model

    Parameters:
    -----------
    result : fitted model result from fit_garch_exog
    y_train : training data (your y.values)
    x_test : test set exogenous variables (upstream inflow)

    Returns:
    --------
    volatility_forecast : array of forecasted volatilities
    """
    omega, alpha, beta, gamma = result.x

    # Convert to numpy array to avoid pandas indexing warnings
    x_test = np.array(x_test)

    # Get the last variance from training data
    sigma2 = np.var(y_train)  # Initialize
    for t in range(1, len(y_train)):
        sigma2 = omega + alpha * y_train[t-1]**2 + beta * sigma2 + gamma * 0  #␣
↪No x in training

    # Forecast volatilities
    forecasts = []
    last_y = y_train[-1]

    for t in range(len(x_test)):
        if t == 0:
```

```
            # First forecast uses last training observation
            sigma2 = omega + alpha * last_y**2 + beta * sigma2 + gamma *␣
↪x_test[t]
        else:
            # Use previous forecast as input (E[y^2] = sigma^2)
            sigma2 = omega + (alpha + beta) * sigma2 + gamma * x_test[t]

        forecasts.append(np.sqrt(sigma2))  # Convert to volatility

    return np.array(forecasts)
```

[18]:
```
def create_confidence_bands(y_test, y_pred_test, volatility_forecast,␣
↪confidence_level):
    """
    Create confidence intervals around predictions using GARCH volatility␣
↪forecasts

    Parameters:
    -----------
    y_test : array-like
        Actual test values (normalized residuals)
    y_pred_test : array-like
        Point predictions for test set
    volatility_forecast : array-like
        GARCH volatility forecasts
    confidence_level : float
        Confidence level (default 0.95 for 95% CI)

    Returns:
    --------
    dict : Dictionary with upper and lower bounds
    """
    from scipy.stats import norm

    # Calculate z-score for confidence level
    alpha = 1 - confidence_level
    z_score = norm.ppf(1 - alpha/2)  # For 95% CI, z   1.96

    # Create confidence bands around predictions
    lower_bound = y_pred_test - z_score * volatility_forecast
    upper_bound = y_pred_test + z_score * volatility_forecast

    return {
        'lower_bound': lower_bound,
        'upper_bound': upper_bound,
        'z_score': z_score
    }
```

```python
[20]: def plot_forecast_with_uncertainty(y_test, y_pred, volatility_forecast):
    """
    Plot forecast with uncertainty bands in standard format

    Parameters:
    -----------
    y_test : pandas Series or array
        Actual test values
    y_pred : pandas Series or array
        Point predictions for test set
    volatility_forecast : array
        GARCH volatility forecasts

    Returns:
    --------
    volatility_forecast : array (for consistency with your example)
    """
    # Create 95% confidence intervals
    upper = y_pred + 1.96 * volatility_forecast
    lower = y_pred - 1.96 * volatility_forecast

    # Plot
    plt.figure(figsize=(12, 2))
    plt.plot(y_test.index, y_test, label="Actual", color='blue')
    plt.plot(y_test.index, y_pred, label="Predicted", color='red')
    plt.fill_between(y_test.index, lower, upper, color='gray', alpha=0.3,␣
↪label="95% CI")
    plt.xticks(rotation=-45)
    plt.legend()
    plt.title("Forecast with GARCH-X Uncertainty")
    plt.show()

    return volatility_forecast
```

```python
[22]: def rescale(z, mean, std):
    return z*std + mean
```

2 Methods

3 Data Processing

3.1 Load Training and Test Data

```python
[28]: # Load pickle file
with open("wet_ols_model.pkl", "rb") as f:
    wet_save_data = pickle.load(f)
```

```
# Unpack
wet_splits = wet_save_data["data"]
wet_y_pred   = wet_save_data["y_pred"]
wet_residuals   = wet_save_data["residuals"]
```

[30]:
```
# Load pickle file
with open("dry_ols_model.pkl", "rb") as f:
    dry_save_data = pickle.load(f)

# Unpack
dry_splits = dry_save_data["data"]
dry_y_pred   = dry_save_data["y_pred"]
dry_residuals   = dry_save_data["residuals"]
```

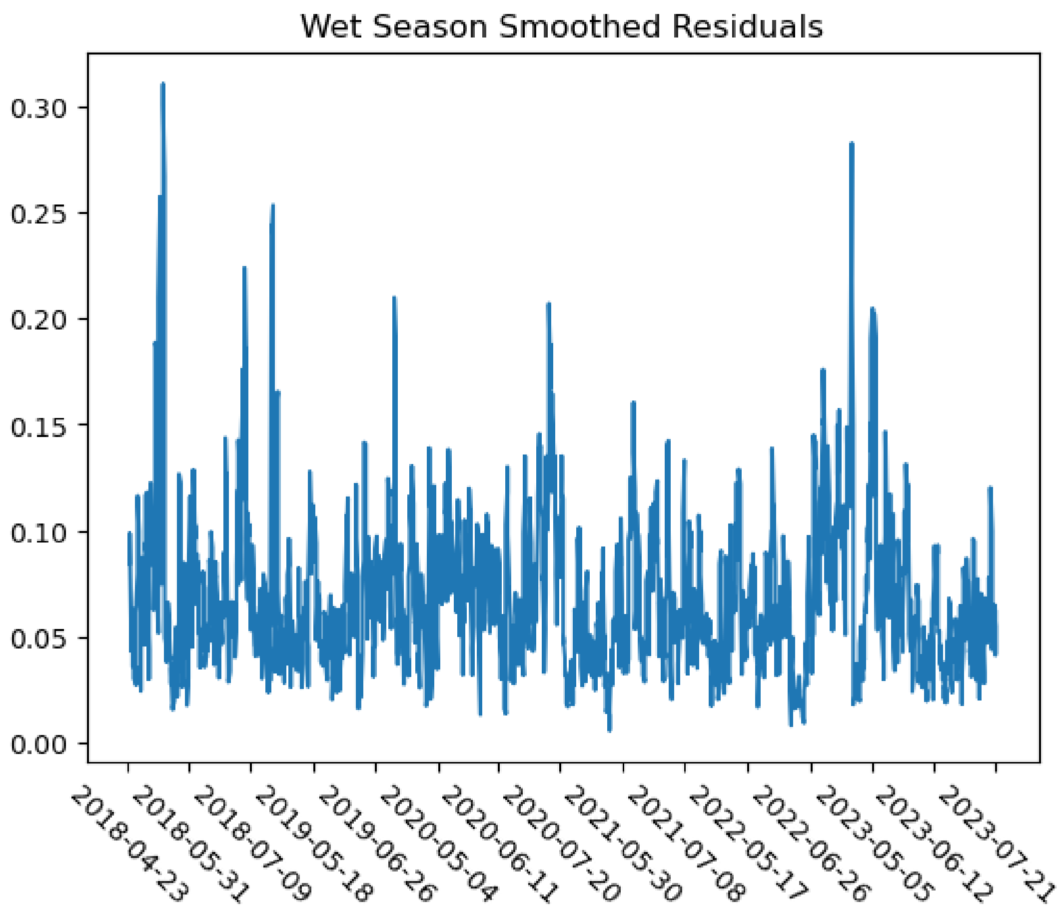
4 Wet Season DDU Model

4.1 Calculate Date Indices

[34]:
```
# Extract training dates
wet_date_strings = wet_splits["wet_X_train"].index.strftime('%Y-%m-%d').
↪to_numpy()

# Set how many x-axis labels you want
num_labels_to_show = 15
```

4.2 Pre-Processing

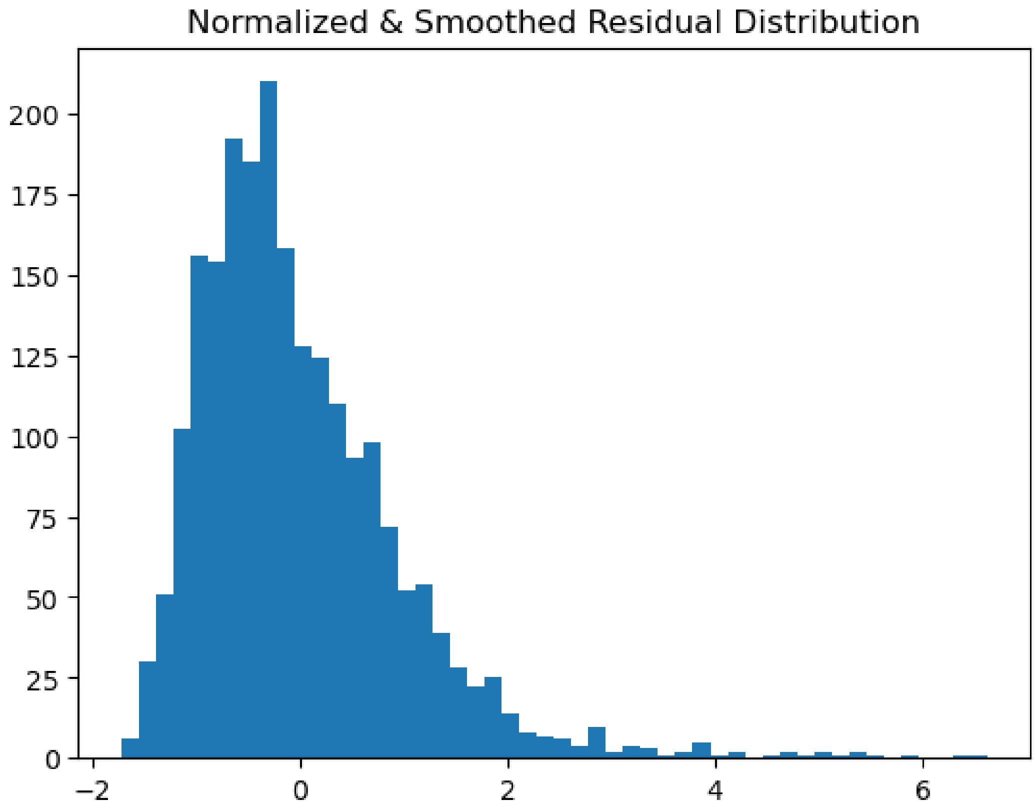
[37]:
```
# 1. Take absolute value of residuals

# 2. Apply a smoothing algorithm

wet_win = 5
wet_y_res = plot_smoothed_residuals(wet_date_strings, wet_residuals, wet_win,␣
↪"Wet")
```



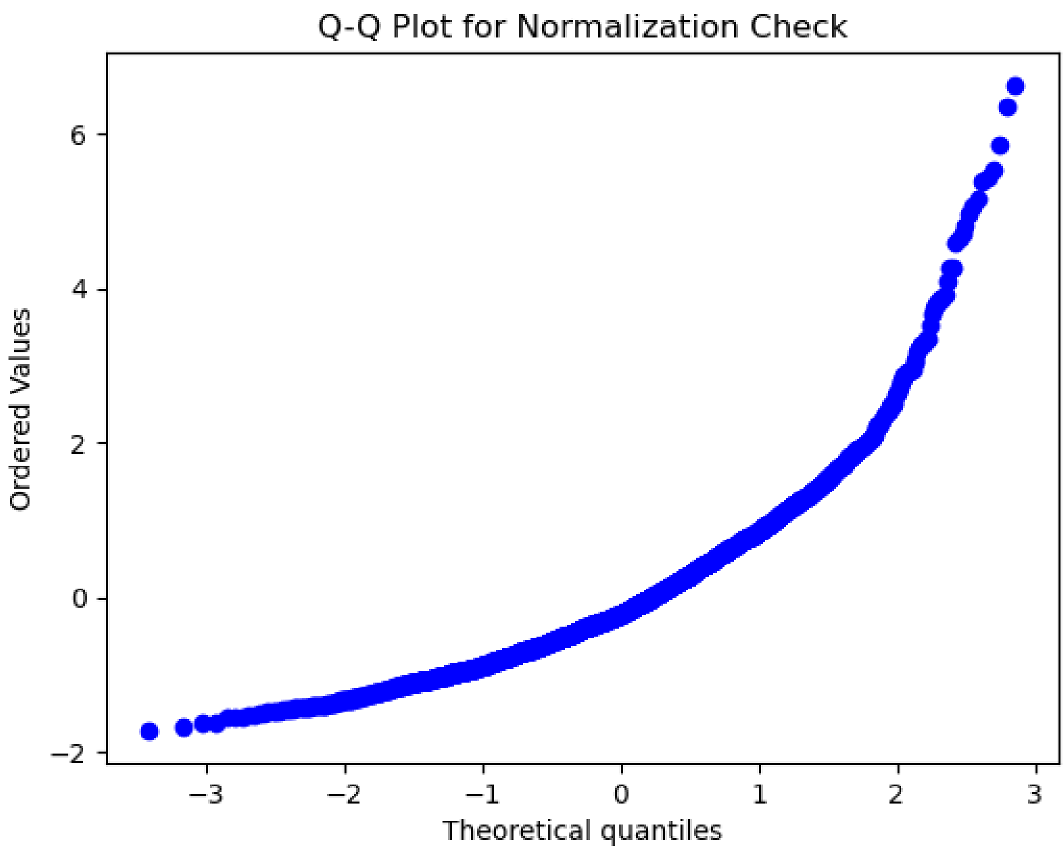
[39]:
```
# 3. Normalize residuals
print(f"Before normalization - Mean: {wet_y_res.mean():.6f}, Std: {wet_y_res.
 ↪std():.6f}")

wet_y_res_norm = (wet_y_res - wet_y_res.mean()) / wet_y_res.std()
print(f"After normalization - Mean: {wet_y_res_norm.mean():.6f}, Std:␣
 ↪{wet_y_res_norm.std():.6f}")
```

```
Before normalization - Mean: 0.068806, Std: 0.036420
After normalization - Mean: 0.000000, Std: 1.000000
```

[41]:
```
# 4. Visualize transformed residuals

plt.hist(wet_y_res_norm, bins=50)
plt.title('Normalized & Smoothed Residual Distribution')
plt.show()
```



[43]:
```
# 5. Recheck QQ Plots

stats.probplot(wet_y_res_norm, dist="norm", plot=plt)
plt.title("Q-Q Plot for Normalization Check")
plt.show()
```



4.3 Train ARCH-X Model

[46]:
```
garch_x_col = 'outflow_m3hr_Lag_1'
```

[48]:
```
# Drop any NaNs from lag alignment
garch_df = pd.concat([wet_splits["wet_X_train"][garch_x_col], wet_y_res_norm],␣
↪axis=1).dropna()

X = garch_df[garch_x_col] # exogenous predictor (outflow lag 1)
y = garch_df[0]           # smoothed and normalized residuals
```

[50]:
```
# Apply to your data

result = fit_garch_exog(y.values, X.values)

print("-------------------------------")
if result.success:
    omega, alpha, beta, gamma = result.x
    print("GARCH(1,1) with Exogenous Variable Results:")
    print(f"omega (constant): {omega:.6f}")
```

```python
    print(f"alpha (ARCH): {alpha:.6f}")
    print(f"beta (GARCH): {beta:.6f}")
    print(f"gamma (exogenous): {gamma:.6f}")
    print(f"Log-likelihood: {-result.fun:.2f}")
else:
    print("Optimization failed:", result.message)
```

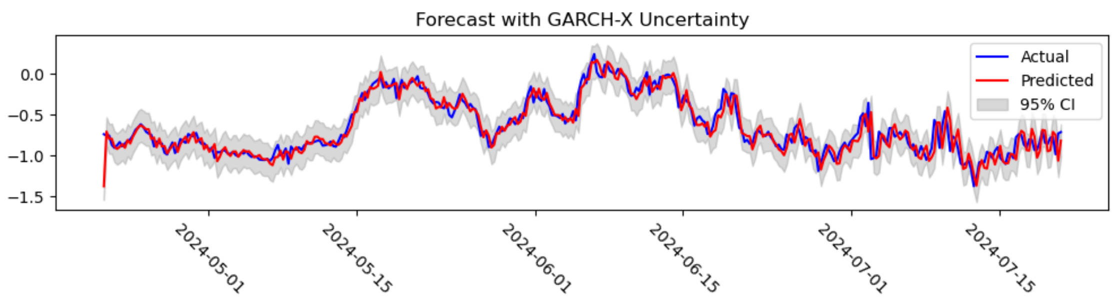
```
/var/folders/w1/xfz4_86j0zg0zz10g620bhz00000gn/T/ipykernel_74815/3796071886.py:3
2: OptimizeWarning: Unknown solver options: gtol, xtol
  result = minimize(garch_exog_loglik, init_params,
/opt/anaconda3/lib/python3.12/site-packages/scipy/optimize/_slsqp_py.py:437:
RuntimeWarning: Values in x were outside bounds during a minimize step, clipping
to bounds
  fx = wrapped_fun(x)
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 2416.296665515165
            Iterations: 24
            Function evaluations: 137
            Gradient evaluations: 24
=== Convergence Analysis ===
Convergence achieved: True
Termination message: Optimization terminated successfully
Number of iterations: 24
Function evaluations: 137

=== Tolerance Checks ===
Final function value: 2416.296665515165
Function tolerance: 1e-09
Relative function change (total): 1.76e-01

Final gradient norm: 32.826995880051065
Gradient tolerance: 1e-06
Gradient criterion met: False

Parameter change norm (total): 1.08e+00
Parameter tolerance: 1e-06

Final parameters: [0.17158209 0.81454127 0.         0.00832174]
------------------------------
GARCH(1,1) with Exogenous Variable Results:
omega (constant): 0.171582
alpha (ARCH): 0.814541
beta (GARCH): 0.000000
gamma (exogenous): 0.008322
Log-likelihood: -2416.30
```

4.3.1 Forecast Volatility on Test Set

[53]:
```
x_test = wet_splits["wet_X_test"][garch_x_col]
volatility_forecast = forecast_volatility(result, y.values, x_test)
```

[55]:
```
y_pred_test = wet_y_pred
y_test = wet_splits['wet_y_test']

# Create confidence bands
confidence_level = 0.95
ci_bands = create_confidence_bands(y_test, y_pred_test, volatility_forecast,␣
↪confidence_level)

# Plot everything together
rescaled_vol = rescale(volatility_forecast, wet_y_res.mean(), wet_y_res.std())
cond_vol = plot_forecast_with_uncertainty(y_test, y_pred_test, rescaled_vol)
```



4.3.2 Min/Max Estimated Std Analysis

[58]:
```
# Raw Variance range on testing dataset

volatility_forecast.min(), volatility_forecast.max()
```

[58]: (0.4406710835041836, 0.9650427165790462)

[60]:
```
# Rescaled variance range on testing dataset

rescale(volatility_forecast.min(), wet_y_res.mean(), wet_y_res.std()),␣
↪rescale(volatility_forecast.max(), wet_y_res.mean(), wet_y_res.std())
```

[60]: (0.08485540928014904, 0.10395310766759766)

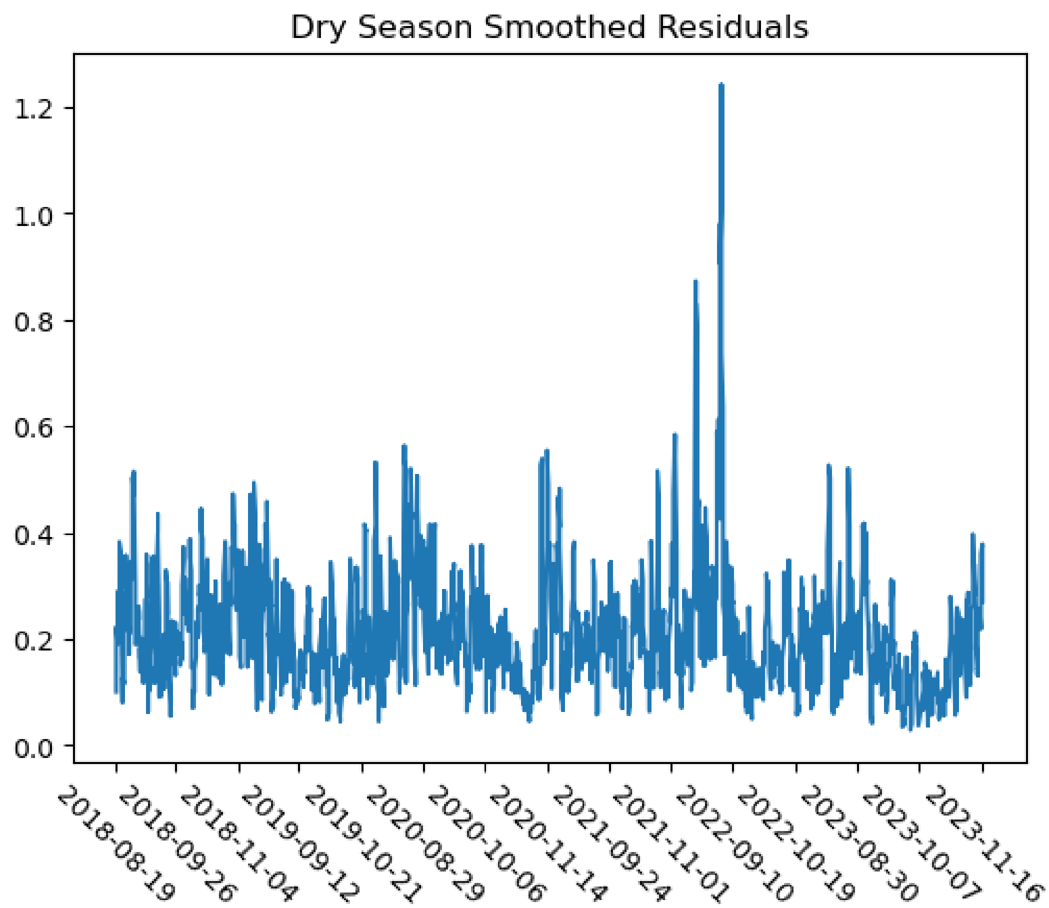
5 Dry Season DDU Model

5.1 Calculate Date Indices

[64]:
```
# Extract training dates
dry_date_strings = dry_splits["dry_X_train"].index.strftime('%Y-%m-%d').
↪to_numpy()
```

5.2 Visualize Residuals

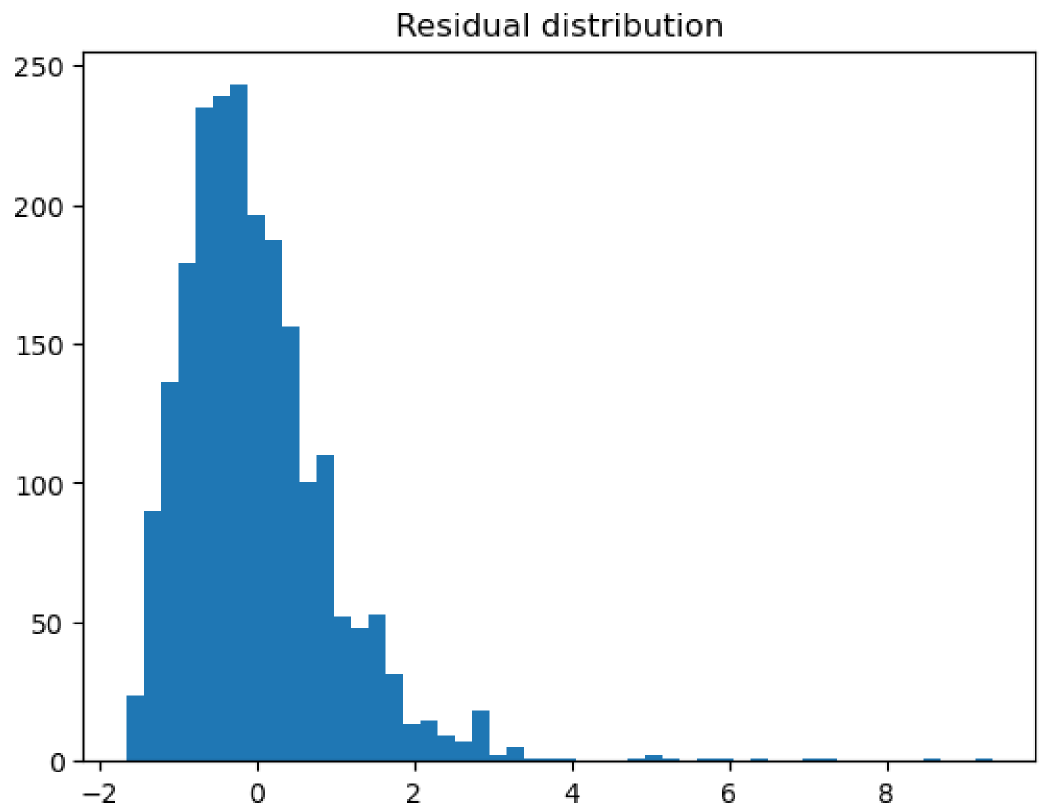
[67]:
```
# 1. Take absolute value of residuals

# 2. Apply a smoothing algorithm

dry_win = 5

dry_y_res = plot_smoothed_residuals(dry_date_strings, dry_residuals, dry_win,␣
↪"Dry")
```



[69]:
```python
# 3. Normalize residuals
print(f"Before normalization - Mean: {dry_y_res.mean():.6f}, Std: {dry_y_res.
    ↪std():.6f}")

dry_y_res_norm = (dry_y_res - dry_y_res.mean()) / dry_y_res.std()
print(f"After normalization - Mean: {dry_y_res_norm.mean():.6f}, Std:␣
    ↪{dry_y_res_norm.std():.6f}")
```

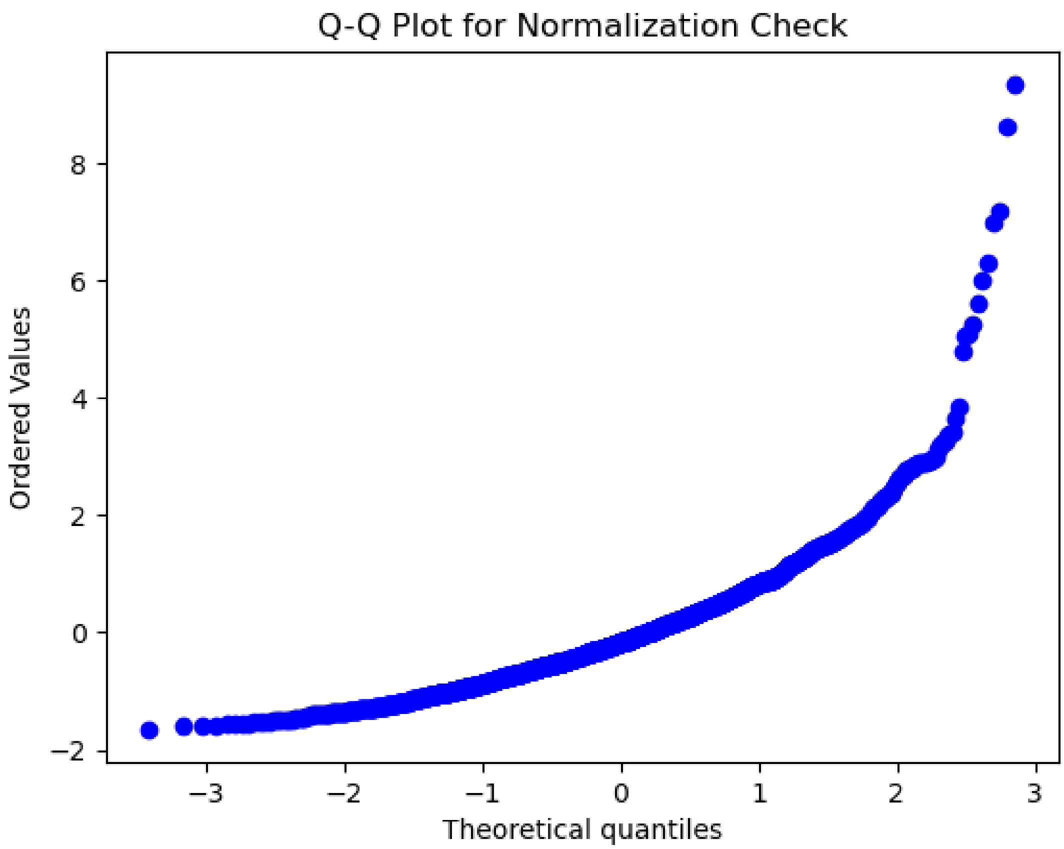
```
Before normalization - Mean: 0.210965, Std: 0.110301
After normalization - Mean: -0.000000, Std: 1.000000
```

[71]:
```python
# 4. Visualize transformed residuals

plt.hist(dry_y_res_norm, bins=50)
plt.title('Residual distribution')
plt.show()
```



[73]:
```python
# 5. Recheck QQ Plots

stats.probplot(dry_y_res_norm, dist="norm", plot=plt)
plt.title("Q-Q Plot for Normalization Check")
```

```
plt.show()
```



5.3 Train GARCH Model

[76]:
```
# Drop any NaNs from lag alignment
garch_df = pd.concat([dry_splits["dry_X_train"][garch_x_col], dry_y_res_norm],␣
↪axis=1).dropna()

X = garch_df[garch_x_col] # exogenous predictor (outflow lag 1)
y = garch_df[0]           # smoothed and normalized residuals
```

[78]:
```
# Apply to your data
# result = fit_garch_exog(y.values, X.values)
result = fit_garch_exog(y.values, X.values)

if result.success:
    omega, alpha, beta, gamma = result.x
    print("GARCH(1,1) with Exogenous Variable Results:")
    print(f"omega (constant): {omega:.6f}")
    print(f"alpha (ARCH): {alpha:.6f}")
```

```python
    print(f"beta (GARCH): {beta:.6f}")
    print(f"gamma (exogenous): {gamma:.6f}")
    print(f"Log-likelihood: {-result.fun:.2f}")
else:
    print("Optimization failed:", result.message)
```

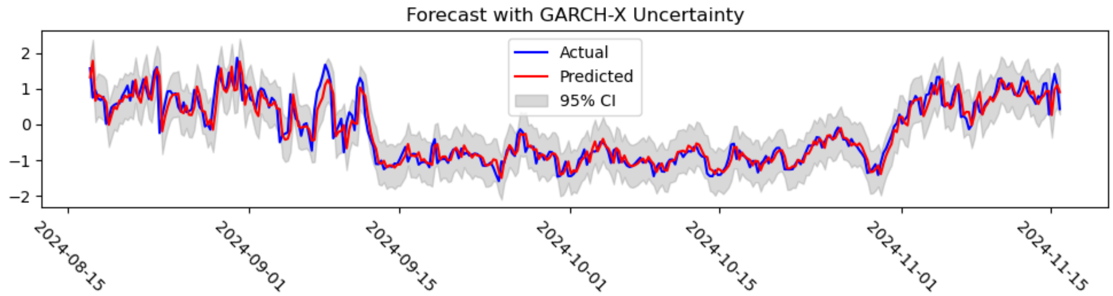
```
/var/folders/w1/xfz4_86j0zg0zz10g620bhz00000gn/T/ipykernel_74815/3796071886.py:3
2: OptimizeWarning: Unknown solver options: gtol, xtol
  result = minimize(garch_exog_loglik, init_params,
/opt/anaconda3/lib/python3.12/site-packages/scipy/optimize/_slsqp_py.py:437:
RuntimeWarning: Values in x were outside bounds during a minimize step, clipping
to bounds
  fx = wrapped_fun(x)
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 2371.966948487659
            Iterations: 36
            Function evaluations: 221
            Gradient evaluations: 35
=== Convergence Analysis ===
Convergence achieved: True
Termination message: Optimization terminated successfully
Number of iterations: 36
Function evaluations: 221

=== Tolerance Checks ===
Final function value: 2371.966948487659
Function tolerance: 1e-09
Relative function change (total): 1.78e-01

Final gradient norm: 0.000813738502561852
Gradient tolerance: 1e-06
Gradient criterion met: False

Parameter change norm (total): 1.07e+00
Parameter tolerance: 1e-06

Final parameters: [0.17850484 0.79573642 0.0070291  0.04038951]
GARCH(1,1) with Exogenous Variable Results:
omega (constant): 0.178505
alpha (ARCH): 0.795736
beta (GARCH): 0.007029
gamma (exogenous): 0.040390
Log-likelihood: -2371.97
```

5.3.1 Forecast Volatility on Test Set

[81]:
```
x_test = dry_splits["dry_X_test"][garch_x_col]
volatility_forecast = forecast_volatility(result, y.values, x_test)
```

[83]:
```
y_pred_test = dry_y_pred
y_test = dry_splits['dry_y_test']

# Create confidence bands
ci_bands = create_confidence_bands(y_test, y_pred_test, volatility_forecast,␣
↪confidence_level)

# Plot everything together
rescaled_vol = rescale(volatility_forecast, dry_y_res.mean(), dry_y_res.std())
cond_vol = plot_forecast_with_uncertainty(y_test, y_pred_test, rescaled_vol)
```



[85]:
```
# Rescaled variance range on testing dataset

rescale(volatility_forecast.min(), dry_y_res.mean(), dry_y_res.std()),␣
↪rescale(volatility_forecast.max(), dry_y_res.mean(), dry_y_res.std())
```

[85]: (0.28393599397947666, 0.3289850736354223)

[ ]: