a-happy-mac

March 25, 2025

[2]:
```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from tqdm import tqdm
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
```

[3]:
```
'''The first file (Acc12_0.05.txt) gets label 0
The second file (Acc12_0.1.txt) gets label 1
...
The last file (Acc12_0.3.txt) gets label 5'''
```

[3]:
```
'The first file (Acc12_0.05.txt) gets label 0\nThe second file (Acc12_0.1.txt)
gets label 1\n…\nThe last file (Acc12_0.3.txt) gets label 5'
```

[1]:
```
import pandas as pd

# List of file paths
file_paths = [
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.05.txt",
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.1.txt",
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.15.txt",
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.2.txt",
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.25.txt",
    "C:/Users/User/Desktop/happy/12floor_dam,undam/acc_12_dam/Acc12_0.3.txt"
]
```

```python
# Initialize an empty list to store DataFrames
data_frames = []

# Iterate over files and assign labels based on file index
for label, file_path in enumerate(file_paths):
    # Load the file while selecting Time (column 0) and 12 features (columns␣
 ↪2-13)
    data = pd.read_csv(file_path, delim_whitespace=True, header=None,␣
 ↪usecols=[0] + list(range(2, 14)), nrows=100000)

    # Assign column names: Time + Feature1 to Feature12
    data.columns = ['Time'] + [f'Feature{i}' for i in range(1, 13)]

    # Assign label based on file index
    data['Damage'] = label

    # Append to list
    data_frames.append(data)

# Combine all files into a single DataFrame
combined_data = pd.concat(data_frames, ignore_index=True)

# Print shape and preview
print(combined_data.shape)  # Should be (6*25000, 14) = (150000, 14)
random_samples = combined_data.sample(n=5, random_state=42)
print(random_samples)
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
C:\Users\User\AppData\Local\Temp\ipykernel_13256\386780385.py:19: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed
in a future version. Use ``sep='\s+'`` instead
  data = pd.read_csv(file_path, delim_whitespace=True, header=None, usecols=[0]
+ list(range(2, 14)), nrows=100000)
```

```
(600000, 14)
           Time  Feature1  Feature2  Feature3  Feature4  Feature5  Feature6  \
4242     21.215  0.299995  0.028560  0.852649  1.475890   1.32319  0.946200
60608   303.045 -0.718966 -1.548780 -1.156040 -0.489940  -0.53173 -0.722983
392832  464.165  0.898631  0.399405  0.657929  0.432073   1.00675  1.096680
41643   208.220 -0.874157 -0.700758 -0.805924 -1.444120  -2.05542 -1.087370
464234  321.175  0.844849  0.978465  0.455574  1.336800   1.18179  0.438657

        Feature7  Feature8  Feature9  Feature10  Feature11  Feature12  Damage
4242    0.991682  1.348940  1.062180   0.634827   0.590386   0.016732       0
60608  -0.105155 -0.600786 -1.190760  -0.197055   0.134325  -0.469881       0
392832  0.553895  1.314090  0.968477   0.701114   1.084340   0.878151       3
41643  -0.627510 -0.794099 -0.945895  -1.687840  -1.634270  -1.866140       0
464234  0.543331  0.617328  0.583380   0.286380   0.871279   1.979660       4
```

[5]:
```python
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc


# Function to compute all 20 features
def extract_noise_resilient_features(data):
    # Time-Domain Features
    data['Velocity'] = data['Feature1'].diff() / data['Time'].diff()
    data['Jerk'] = data['Velocity'].diff() / data['Time'].diff()
    data['Displacement'] = data['Feature1'].cumsum() * (data['Time'].diff().
↪iloc[0])
    data['RMS_Acceleration'] = (data[['Feature1', 'Feature2', 'Feature3',␣
↪'Feature4', 'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9',␣
↪'Feature10', 'Feature11', 'Feature12']].pow(2).mean(axis=1)).pow(0.5)

    peak_acc = data[['Feature1', 'Feature2', 'Feature3', 'Feature4',␣
↪'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9', 'Feature10',␣
↪'Feature11', 'Feature12']].max(axis=1)
```

```
    data['Crest_Factor'] = peak_acc / data['RMS_Acceleration']

    # Zero Crossing Rate (ZCR) - using a simple sign-change count
    def zero_crossing_rate(signal):
        return ((signal[:-1] * signal[1:]) < 0).sum()
    data['ZCR'] = data[['Feature1', 'Feature2', 'Feature3', 'Feature4',␣
↪'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9', 'Feature10',␣
↪'Feature11', 'Feature12']].apply(zero_crossing_rate, axis=1)

    # Autocorrelation
    data['Autocorrelation'] = data['Feature1'].autocorr()

    # Skewness & Kurtosis
    data['Skewness'] = skew(data[['Feature1', 'Feature2', 'Feature3',␣
↪'Feature4', 'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9',␣
↪'Feature10', 'Feature11', 'Feature12']], axis=1)
    data['Kurtosis'] = kurtosis(data[['Feature1', 'Feature2', 'Feature3',␣
↪'Feature4', 'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9',␣
↪'Feature10', 'Feature11', 'Feature12']], axis=1)

    # Entropy
    data['Entropy'] = data[['Feature1', 'Feature2', 'Feature3', 'Feature4',␣
↪'Feature5', 'Feature6', 'Feature7', 'Feature8', 'Feature9', 'Feature10',␣
↪'Feature11', 'Feature12']].apply(lambda x: entropy(x))

    # Frequency-Domain Features
    # Fourier Transform (FFT Dominant Frequency)
    data['FFT'] = np.fft.fftfreq(len(data), d=(data['Time'][1] -␣
↪data['Time'][0]))

    # Power Spectral Density (PSD)
    data['PSD'] = welch(data['Feature1'], fs=1 / (data['Time'][1] -␣
↪data['Time'][0]))[1]

    # Wavelet Transform (CWT)
    coeffs, freqs = pywt.cwt(data['Feature1'], np.arange(1, 100), 'gaus1')
    data['CWT'] = coeffs.mean(axis=1)

    # Spectral Energy
    data['Spectral_Energy'] = np.sum(np.abs(data['Feature1'])**2)

    # Teager-Kaiser Energy Operator (TKEO)
    data['TKEO'] = data['Feature1']**2 - data['Feature1'].shift(1) *␣
↪data['Feature1'].shift(-1)

    # Advanced Nonlinear & Correlation-Based Features
```

```
    # Lyapunov Exponent (requires specialized methods)
    # data['Lyapunov_Exponent'] = ... (not implemented, specialized method␣
↪required)



    # Fractal Dimension
    data['Fractal_Dimension'] = (data['Feature1']).hurst()

    # Shock Response Spectrum (SRS) (requires specialized methods)
    # data['SRS'] = ... (not implemented, specialized method required)

    # Cross-Correlation Between Sensors
    data['Cross_Correlation'] = data[['Feature1', 'Feature2']].corr().iloc[0, 1]

    return data

# Example usage: Assuming `combined_data` is your dataset with 14 columns
# Extract features
dataset_with_features = extract_noise_resilient_features(combined_data)

# Show a preview of the updated dataset with new features
dataset_with_features.head()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 71
     67     return data
     69 # Example usage: Assuming `combined_data` is your dataset with 14 columns
     70 # Extract features
---> 71 dataset_with_features = extract_noise_resilient_features(combined_data)
     73 # Show a preview of the updated dataset with new features
     74 dataset_with_features.head()

NameError: name 'combined_data' is not defined
```

[6]:

```
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

def extract_noise_resilient_features(data):
    # Create a copy so that the original data remains unchanged
    data = data.copy()
```

```python
# Assume the sampling interval is constant; use the second time difference
dt = data['Time'].diff().iloc[1]

# --------------------------
# Time-Domain Features
# --------------------------
data['Velocity'] = data['Feature1'].diff() / data['Time'].diff()
data['Jerk'] = data['Velocity'].diff() / data['Time'].diff()
data['Displacement'] = data['Feature1'].cumsum() * dt

# Define feature columns used for multi-channel features
feature_cols = ['Feature1', 'Feature2', 'Feature3', 'Feature4',
                'Feature5', 'Feature6', 'Feature7', 'Feature8',
                'Feature9', 'Feature10', 'Feature11', 'Feature12']

# Root Mean Square (RMS) Acceleration
data['RMS_Acceleration'] = np.sqrt((data[feature_cols]**2).mean(axis=1))

# Crest Factor: Peak acceleration divided by RMS_Acceleration
peak_acc = data[feature_cols].max(axis=1)
data['Crest_Factor'] = peak_acc / data['RMS_Acceleration']

# Zero Crossing Rate (ZCR) - count sign changes in the feature values (row-wise)
def zero_crossing_rate(row):
    return ((row[:-1] * row[1:]) < 0).sum()
data['ZCR'] = data[feature_cols].apply(zero_crossing_rate, axis=1)

# --------------------------
# Global Features (computed on entire signal)
# --------------------------
# Autocorrelation for Feature1 (same value for all rows)
autocorr_value = data['Feature1'].autocorr()
data['Autocorrelation'] = autocorr_value

# Skewness & Kurtosis computed row-wise across the feature columns
data['Skewness'] = data[feature_cols].apply(lambda row: skew(row), axis=1)
data['Kurtosis'] = data[feature_cols].apply(lambda row: kurtosis(row), axis=1)

# Entropy computed row-wise across the feature columns
data['Entropy'] = data[feature_cols].apply(lambda row: entropy(row), axis=1)

# --------------------------
# Frequency-Domain Features
# --------------------------
```

```
    # Fourier Transform: Get FFT frequencies based on the length of the data␣
↪and dt.
    fft_freqs = np.fft.fftfreq(len(data), d=dt)
    # Store the FFT frequencies as a list (same for every row)
    data['FFT'] = [fft_freqs] * len(data)

    # Power Spectral Density (PSD) using Welch's method on Feature1
    freqs, psd_values = welch(data['Feature1'], fs=1/dt)
    # Instead of assigning the full PSD array, store summary statistics:
    data['PSD_Mean'] = psd_values.mean()
    data['PSD_Max'] = psd_values.max()
    data['PSD_Min'] = psd_values.min()

    # Wavelet Transform (CWT) on Feature1 using scales 1 to 99 and the 'gaus1'␣
↪wavelet
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(data['Feature1'], scales, 'gaus1')
    # Compute the mean across time for each scale and store it for every row
    cwt_mean = coeffs.mean(axis=1)
    data['CWT'] = [cwt_mean] * len(data)

    # Spectral Energy computed on Feature1
    data['Spectral_Energy'] = np.sum(np.abs(data['Feature1'])**2)

    # Teager-Kaiser Energy Operator (TKEO) on Feature1
    data['TKEO'] = data['Feature1']**2 - data['Feature1'].shift(1) *␣
↪data['Feature1'].shift(-1)

    # --------------------------
    # Advanced Nonlinear & Correlation-Based Features
    # --------------------------
    # Fractal Dimension: Using the Hurst exponent computed by compute_Hc.
    # compute_Hc returns H (Hurst exponent), c (constant), and data (fitted␣
↪values)
    H, c, _ = compute_Hc(data['Feature1'], kind='price', simplified=True)
    data['Fractal_Dimension'] = H

    # Cross-Correlation Between sensors (using Feature1 and Feature2)
    cross_corr = data[['Feature1', 'Feature2']].corr().iloc[0, 1]
    data['Cross_Correlation'] = cross_corr

    return data

# --------------------------
# Example usage
# --------------------------
if __name__ == "__main__":
```

```
# Create dummy data for demonstration
n = 150000  # number of data points
time = np.linspace(0, 10, n)

# Generate synthetic features (for example purposes, using sine waves with␣
↪added noise)
data_dict = {
    'Time': time,
    'Feature1': np.sin(2 * np.pi * 1 * time) + 0.1 * np.random.randn(n),
    'Feature2': np.sin(2 * np.pi * 0.5 * time) + 0.1 * np.random.randn(n),
    'Feature3': np.sin(2 * np.pi * 2 * time) + 0.1 * np.random.randn(n),
    'Feature4': np.sin(2 * np.pi * 0.2 * time) + 0.1 * np.random.randn(n),
    'Feature5': np.sin(2 * np.pi * 1.5 * time) + 0.1 * np.random.randn(n),
    'Feature6': np.sin(2 * np.pi * 0.8 * time) + 0.1 * np.random.randn(n),
    'Feature7': np.sin(2 * np.pi * 1.2 * time) + 0.1 * np.random.randn(n),
    'Feature8': np.sin(2 * np.pi * 0.3 * time) + 0.1 * np.random.randn(n),
    'Feature9': np.sin(2 * np.pi * 0.7 * time) + 0.1 * np.random.randn(n),
    'Feature10': np.sin(2 * np.pi * 1.8 * time) + 0.1 * np.random.randn(n),
    'Feature11': np.sin(2 * np.pi * 1.1 * time) + 0.1 * np.random.randn(n),
    'Feature12': np.sin(2 * np.pi * 0.9 * time) + 0.1 * np.random.randn(n)
}

combined_data = pd.DataFrame(data_dict)

# Extract features
dataset_with_features = extract_noise_resilient_features(combined_data)

# Display the first few rows of the resulting DataFrame
print(dataset_with_features.head())
```

```
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
Cell In[6], line 123
    120 combined_data = pd.DataFrame(data_dict)
    122 # Extract features
--> 123 dataset_with_features = extract_noise_resilient_features(combined_data)
    125 # Display the first few rows of the resulting DataFrame
    126 print(dataset_with_features.head())

Cell In[6], line 86, in extract_noise_resilient_features(data)
     79 data['TKEO'] = data['Feature1']**2 - data['Feature1'].shift(1) *␣
 ↪data['Feature1'].shift(-1)
     81 # --------------------------
     82 # Advanced Nonlinear & Correlation-Based Features
     83 # --------------------------
     84 # Fractal Dimension: Using the Hurst exponent computed by compute_Hc.
```

```
     85 # compute_Hc returns H (Hurst exponent), c (constant), and data (fitted
 ↪values)
---> 86 H, c, _ = compute_Hc(data['Feature1'], kind='price', simplified=True)
     87 data['Fractal_Dimension'] = H
     89 # Cross-Correlation Between sensors (using Feature1 and Feature2)

File c:\Users\User\Desktop\GAN\venv\lib\site-packages\hurst\__init__.py:191, in
 ↪compute_Hc(series, kind, min_window, max_window, simplified)
    188     RS.append(np.mean(rs))
    190 A = np.vstack([np.log10(window_sizes), np.ones(len(RS))]).T
--> 191 H, c = np.linalg.lstsq(A, np.log10(RS), rcond=-1)[0]
    192 np.seterr(**err)
    194 c = 10**c

FloatingPointError: invalid value encountered in log10
```

[11]:
```
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

def extract_noise_resilient_features(data):
    # Work on a copy of the DataFrame
    data = data.copy()

    # Use the second time difference for a constant dt
    dt = data['Time'].diff().iloc[1]

    # --------------------------
    # Time-Domain Features
    # --------------------------
    data['Velocity'] = data['Feature1'].diff() / data['Time'].diff()
    data['Jerk'] = data['Velocity'].diff() / data['Time'].diff()
    data['Displacement'] = data['Feature1'].cumsum() * dt

    # Columns containing multi-channel features
    feature_cols = ['Feature1', 'Feature2', 'Feature3', 'Feature4',
                    'Feature5', 'Feature6', 'Feature7', 'Feature8',
                    'Feature9', 'Feature10', 'Feature11', 'Feature12']

    # RMS Acceleration
    data['RMS_Acceleration'] = np.sqrt((data[feature_cols]**2).mean(axis=1))

    # Crest Factor: max acceleration divided by RMS acceleration
    peak_acc = data[feature_cols].max(axis=1)
```

```
    data['Crest_Factor'] = peak_acc / data['RMS_Acceleration']

    # Zero Crossing Rate (ZCR) - count sign changes row-wise
    def zero_crossing_rate(row):
        return ((row[:-1] * row[1:]) < 0).sum()
    data['ZCR'] = data[feature_cols].apply(zero_crossing_rate, axis=1)

    # ---------------------------
    # Global Signal Features
    # ---------------------------
    # Autocorrelation for Feature1 (same value for all rows)
    autocorr_value = data['Feature1'].autocorr()
    data['Autocorrelation'] = autocorr_value

    # Skewness & Kurtosis computed row-wise
    data['Skewness'] = data[feature_cols].apply(lambda row: skew(row), axis=1)
    data['Kurtosis'] = data[feature_cols].apply(lambda row: kurtosis(row),␣
↪axis=1)

    # Entropy computed row-wise
    data['Entropy'] = data[feature_cols].apply(lambda row: entropy(row), axis=1)

    # ---------------------------
    # Frequency-Domain Features
    # ---------------------------
    # Fourier Transform Frequencies
    fft_freqs = np.fft.fftfreq(len(data), d=dt)
    data['FFT'] = [fft_freqs] * len(data)

    # Power Spectral Density (PSD) for Feature1 using Welch's method
    freqs, psd_values = welch(data['Feature1'], fs=1/dt)
    data['PSD_Mean'] = psd_values.mean()
    data['PSD_Max'] = psd_values.max()
    data['PSD_Min'] = psd_values.min()

    # Wavelet Transform (CWT) for Feature1 using scales 1 to 99 with 'gaus1'
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(data['Feature1'], scales, 'gaus1')
    cwt_mean = coeffs.mean(axis=1)
    data['CWT'] = [cwt_mean] * len(data)

    # Spectral Energy on Feature1
    data['Spectral_Energy'] = np.sum(np.abs(data['Feature1'])**2)

    # Teager-Kaiser Energy Operator (TKEO) on Feature1
    data['TKEO'] = data['Feature1']**2 - data['Feature1'].shift(1) *␣
↪data['Feature1'].shift(-1)
```

```
    # ---------------------------
    # Advanced Features
    # ---------------------------
    # Fractal Dimension via Hurst Exponent
    # Using 'change' mode is safer when your data can be negative.
    try:
        H, c, _ = compute_Hc(data['Feature1'], kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    data['Fractal_Dimension'] = H

    # Alternatively, if you prefer to use 'price', you can suppress the warning:
    # with np.errstate(invalid='ignore'):
    #     H, c, _ = compute_Hc(data['Feature1'], kind='price', simplified=True)
    # data['Fractal_Dimension'] = H

    # Cross-Correlation between Feature1 and Feature2
    cross_corr = data[['Feature1', 'Feature2']].corr().iloc[0, 1]
    data['Cross_Correlation'] = cross_corr

    return data

# ---------------------------
# Example usage
# ---------------------------
if __name__ == "__main__":
    # Create dummy data for demonstration
    n = 150000  # number of data points
    time = np.linspace(0, 10, n)

    # Generate synthetic features (sine waves with added noise)
    data_dict = {
        'Time': time,
        'Feature1': np.sin(2 * np.pi * 1 * time) + 0.1 * np.random.randn(n),
        'Feature2': np.sin(2 * np.pi * 0.5 * time) + 0.1 * np.random.randn(n),
        'Feature3': np.sin(2 * np.pi * 2 * time) + 0.1 * np.random.randn(n),
        'Feature4': np.sin(2 * np.pi * 0.2 * time) + 0.1 * np.random.randn(n),
        'Feature5': np.sin(2 * np.pi * 1.5 * time) + 0.1 * np.random.randn(n),
        'Feature6': np.sin(2 * np.pi * 0.8 * time) + 0.1 * np.random.randn(n),
        'Feature7': np.sin(2 * np.pi * 1.2 * time) + 0.1 * np.random.randn(n),
        'Feature8': np.sin(2 * np.pi * 0.3 * time) + 0.1 * np.random.randn(n),
        'Feature9': np.sin(2 * np.pi * 0.7 * time) + 0.1 * np.random.randn(n),
        'Feature10': np.sin(2 * np.pi * 1.8 * time) + 0.1 * np.random.randn(n),
        'Feature11': np.sin(2 * np.pi * 1.1 * time) + 0.1 * np.random.randn(n),
        'Feature12': np.sin(2 * np.pi * 0.9 * time) + 0.1 * np.random.randn(n)
    }
```

```python
combined_data = pd.DataFrame(data_dict)

# Extract features from the dataset
dataset_with_features = extract_noise_resilient_features(combined_data)

# Display the first few rows of the resulting DataFrame
print(dataset_with_features.head())
```

```
       Time  Feature1  Feature2  Feature3  Feature4  Feature5  Feature6  \
0  0.000000 -0.131379  0.052875 -0.030394  0.047267 -0.049013 -0.073563
1  0.000067  0.129969  0.086311 -0.089706 -0.002146 -0.056306 -0.088619
2  0.000133 -0.002406  0.086464  0.115098 -0.121001  0.085380  0.009603
3  0.000200 -0.031938 -0.035055 -0.022682 -0.175537 -0.001733  0.069285
4  0.000267 -0.057219  0.033330 -0.064580  0.063788  0.009321 -0.028341

   Feature7  Feature8  Feature9  …  Entropy  \
0 -0.012901 -0.041253 -0.104554  …     -inf
1  0.175037 -0.029754 -0.100865  …     -inf
2 -0.106341  0.078336  0.150474  …     -inf
3 -0.067002  0.079959 -0.096323  …     -inf
4 -0.035536  0.099649 -0.184900  …     -inf

                                                FFT  PSD_Mean   PSD_Max  \
0  [0.0, 0.09999933333333333, 0.1999986666666666…  0.000001  0.000003
1  [0.0, 0.09999933333333333, 0.1999986666666666…  0.000001  0.000003
2  [0.0, 0.09999933333333333, 0.1999986666666666…  0.000001  0.000003
3  [0.0, 0.09999933333333333, 0.1999986666666666…  0.000001  0.000003
4  [0.0, 0.09999933333333333, 0.1999986666666666…  0.000001  0.000003

        PSD_Min                                                CWT  \
0  2.352645e-07  [2.739245441255811e-07, 2.34891746616371e-07, …
1  2.352645e-07  [2.739245441255811e-07, 2.34891746616371e-07, …
2  2.352645e-07  [2.739245441255811e-07, 2.34891746616371e-07, …
3  2.352645e-07  [2.739245441255811e-07, 2.34891746616371e-07, …
4  2.352645e-07  [2.739245441255811e-07, 2.34891746616371e-07, …

   Spectral_Energy      TKEO  Fractal_Dimension  Cross_Correlation
0     76413.838664       NaN           0.471833           0.000517
1     76413.838664  0.016576           0.471833           0.000517
2     76413.838664  0.004157           0.471833           0.000517
3     76413.838664  0.000882           0.471833           0.000517
4     76413.838664  0.004053           0.471833           0.000517

[5 rows x 32 columns]
```

[12]:
```python
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

def extract_features_from_series(x, dt):
    """
    Given a pandas Series x (time series for one sensor) and time step dt,
    compute 20 features and return them as a dictionary.
    """
    features = {}
    n = len(x)

    # 1. Mean Absolute Velocity
    vel = np.diff(x) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk (second derivative)
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement (last - first)
    features['Net_Displacement'] = x.iloc[-1] - x.iloc[0]

    # 4. RMS: sqrt(mean(x^2))
    rms = np.sqrt(np.mean(x**2))
    features['RMS'] = rms

    # 5. Crest Factor: max(|x|)/RMS
    # (Avoid division by zero)
    features['Crest_Factor'] = np.max(np.abs(x)) / rms if rms != 0 else np.nan

    # 6. Zero Crossing Rate: (number of sign changes)/length
    x_arr = x.values
    zero_crossings = np.sum(x_arr[:-1] * x_arr[1:] < 0)
    features['Zero_Crossing_Rate'] = zero_crossings / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
```

```
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy: using histogram (ensure positive bins by shifting if needed)
    hist, bin_edges = np.histogram(x_arr, bins=10, density=True)
    # Add a small constant to avoid log(0)
    hist = hist + 1e-8
    features['Entropy'] = entropy(hist)

    # Frequency Domain Features
    # 11. Dominant FFT Frequency and 12. its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_magnitude = np.abs(fft_vals)
    # Ignore zero-frequency term:
    if n > 1:
        idx = np.argmax(fft_magnitude[1:]) + 1
        dom_freq = fft_freqs[idx]
        dom_amp = fft_magnitude[idx]
    else:
        dom_freq = np.nan
        dom_amp = np.nan
    features['Dominant_FFT_Freq'] = dom_freq
    features['Dominant_FFT_Amplitude'] = dom_amp

    # 13. PSD Mean, 14. PSD Max, 15. PSD Min (using Welch)
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. Continuous Wavelet Transform (CWT) Mean (using scales 1 to 99,␣
↪wavelet 'gaus1')
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy: sum(x^2)
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO: average of (x^2 - shift(x)*shift(x, -1))
    # Compute TKEO for interior points only
```

```
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension: Hurst Exponent (using 'change' kind to avoid log␣
 ↪of negatives)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. Standard Deviation of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_magnitude)

    return features

def extract_features_for_all_sensors(data):
    """
    For each sensor column (Feature1, Feature2, ..., Feature12) in the␣
 ↪DataFrame,
    extract 20 features and return a DataFrame with shape (12, 20).
    Assumes a 'Time' column exists for sampling interval.
    """
    feature_cols = [f'Feature{i}' for i in range(1, 13)]
    dt = data['Time'].diff().iloc[1]  # constant time step assumed

    sensor_features = {}
    for col in feature_cols:
        # Extract features from each sensor's time series
        sensor_features[col] = extract_features_from_series(data[col], dt)

    # Create a DataFrame from the dictionary:
    features_df = pd.DataFrame(sensor_features).T  # rows: sensors, columns:␣
 ↪features
    return features_df

# --------------------------
# Example usage
# --------------------------
if __name__ == "__main__":
    # Generate synthetic data for demonstration
    n = 150000  # number of data points
    time = np.linspace(0, 10, n)
    # For demonstration, create 12 features as sine waves with different␣
 ↪frequencies and noise
    data_dict = {'Time': time}
    freqs = [1, 0.5, 2, 0.2, 1.5, 0.8, 1.2, 0.3, 0.7, 1.8, 1.1, 0.9]
```

```python
    for i in range(1, 13):
        data_dict[f'Feature{i}'] = np.sin(2 * np.pi * freqs[i-1] * time) + 0.1
 ↪* np.random.randn(n)
    combined_data = pd.DataFrame(data_dict)

    # Extract 20 features for each of the 12 sensors
    result = extract_features_for_all_sensors(combined_data)
    print("Extracted Features (each row corresponds to a sensor column):")
    print(result)
```

```
Extracted Features (each row corresponds to a sensor column):
           MeanAbs_Velocity  MeanAbs_Jerk  Net_Displacement       RMS  \
Feature1        1693.411644  4.398460e+07         -0.090572  0.713593
Feature2        1698.386925  4.414647e+07         -0.089656  0.713977
Feature3        1690.057852  4.390213e+07          0.185627  0.714211
Feature4        1693.691244  4.395092e+07         -0.028144  0.714141
Feature5        1696.528839  4.408962e+07         -0.037629  0.714038
Feature6        1693.379869  4.408116e+07         -0.059141  0.714220
Feature7        1694.633763  4.405754e+07          0.207928  0.714118
Feature8        1694.262853  4.403537e+07          0.031172  0.714175
Feature9        1698.021359  4.409432e+07         -0.025258  0.713792
Feature10       1692.849051  4.401798e+07          0.152497  0.713974
Feature11       1690.311073  4.391575e+07         -0.060768  0.714435
Feature12       1698.177511  4.411907e+07         -0.051743  0.713969

           Crest_Factor  Zero_Crossing_Rate  Lag1_Autocorrelation  Skewness  \
Feature1       1.956640            0.036860              0.980317 -0.001491
Feature2       1.928484            0.036800              0.980284  0.002102
Feature3       1.945315            0.035860              0.980455 -0.000989
Feature4       1.941021            0.036720              0.980409 -0.000062
Feature5       1.930274            0.036580              0.980290 -0.000444
Feature6       1.950906            0.035973              0.980401 -0.000535
Feature7       1.989779            0.034953              0.980345  0.000085
Feature8       1.932894            0.036740              0.980372  0.001142
Feature9       1.921246            0.037120              0.980237  0.000607
Feature10      1.961873            0.035873              0.980368 -0.000509
Feature11      1.996146            0.036300              0.980480  0.000400
Feature12      2.014128            0.036860              0.980256 -0.000034

           Kurtosis   Entropy  Dominant_FFT_Freq  Dominant_FFT_Amplitude  \
Feature1  -1.441671  2.135229           0.999993            74939.938336
Feature2  -1.441362  2.141028           0.499997            74979.221488
Feature3  -1.442408  2.142365           1.999987            75009.001197
Feature4  -1.442619  2.145383           0.199999            75001.835765
Feature5  -1.441195  2.145252           1.499990            74985.914474
Feature6  -1.441426  2.137115           0.799995            75009.558930
Feature7  -1.442625  2.128488           1.199992            74996.895949
```

```
Feature8  -1.440902  2.145510           0.299998           75001.930600
Feature9  -1.441432  2.150838           0.699995           74958.643226
Feature10 -1.440930  2.142627          -1.799988           74981.964059
Feature11 -1.443066  2.139422           1.099993           75034.668788
Feature12 -1.441243  2.112378           0.899994           74979.452325

           PSD_Mean   PSD_Max       PSD_Min  CWT_Mean  Spectral_Energy  \
Feature1   0.000001  0.000003  2.329460e-07  0.000090     76382.143242
Feature2   0.000001  0.000002  2.372774e-07  0.000062     76464.455021
Feature3   0.000001  0.000008  2.274128e-07  0.000175     76514.565312
Feature4   0.000001  0.000001  2.261238e-07 -0.000009     76499.551766
Feature5   0.000001  0.000005  2.405724e-07  0.000140     76477.550613
Feature6   0.000001  0.000002  2.327458e-07  0.000054     76516.622040
Feature7   0.000001  0.000004  2.110360e-07  0.000067     76494.723517
Feature8   0.000001  0.000001  2.188063e-07  0.000017     76506.863686
Feature9   0.000001  0.000002  2.342073e-07  0.000038     76424.894280
Feature10  0.000001  0.000007  2.320772e-07  0.000118     76463.793595
Feature11  0.000001  0.000003  2.294109e-07  0.000140     76562.559126
Feature12  0.000001  0.000003  2.377850e-07  0.000120     76462.834875

           TKEO_Mean  Fractal_Dimension  FFT_Amplitude_STD
Feature1    0.010017           0.472662         274.104730
Feature2    0.010009           0.575097         274.246451
Feature3    0.009985           0.371558         274.350748
Feature4    0.010008           0.706137         274.329955
Feature5    0.010013           0.413261         274.274750
Feature6    0.009928           0.506459         274.352768
Feature7    0.009991           0.447409         274.310636
Feature8    0.010008           0.652503         274.328599
Feature9    0.010074           0.527597         274.174984
Feature10   0.009973           0.387809         274.249523
Feature11   0.009934           0.458787         274.441387
Feature12   0.010031           0.489120         274.249819
```

[3]:

```python
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

def extract_features_from_series(x, dt):
    """
    Given a pandas Series x (time series for one sensor) and time step dt,
    compute 20 features and return them as a dictionary.
    """
    features = {}
```

```
n = len(x)

# 1. Mean Absolute Velocity
vel = np.diff(x) / dt
features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

# 2. Mean Absolute Jerk (second derivative)
if len(vel) > 1:
    jerk = np.diff(vel) / dt
    features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
else:
    features['MeanAbs_Jerk'] = np.nan

# 3. Net Displacement (last - first)
features['Net_Displacement'] = x.iloc[-1] - x.iloc[0]

# 4. RMS: sqrt(mean(x^2))
rms = np.sqrt(np.mean(x**2))
features['RMS'] = rms

# 5. Crest Factor: max(|x|)/RMS (avoid division by zero)
features['Crest_Factor'] = np.max(np.abs(x)) / rms if rms != 0 else np.nan

# 6. Zero Crossing Rate: (number of sign changes)/length
x_arr = x.values
zero_crossings = np.sum(x_arr[:-1] * x_arr[1:] < 0)
features['Zero_Crossing_Rate'] = zero_crossings / n

# 7. Lag-1 Autocorrelation
if n > 1:
    autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
else:
    autocorr = np.nan
features['Lag1_Autocorrelation'] = autocorr

# 8. Skewness
features['Skewness'] = skew(x_arr)

# 9. Kurtosis
features['Kurtosis'] = kurtosis(x_arr)

# 10. Entropy: using histogram (with small constant to avoid log(0))
hist, bin_edges = np.histogram(x_arr, bins=10, density=True)
hist = hist + 1e-8
features['Entropy'] = entropy(hist)

# Frequency Domain Features:
```

```
    # 11. Dominant FFT Frequency and 12. its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_magnitude = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_magnitude[1:]) + 1
        dom_freq = fft_freqs[idx]
        dom_amp = fft_magnitude[idx]
    else:
        dom_freq = np.nan
        dom_amp = np.nan
    features['Dominant_FFT_Freq'] = dom_freq
    features['Dominant_FFT_Amplitude'] = dom_amp

    # 13. PSD Mean, 14. PSD Max, 15. PSD Min (using Welch)
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. Continuous Wavelet Transform (CWT) Mean (using scales 1 to 99,␣
↪wavelet 'gaus1')
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy: sum(x^2)
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO: average of (x^2 - shift(x)*shift(x, -1))
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension: Hurst Exponent (using 'change' kind)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. Standard Deviation of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_magnitude)

    return features

def extract_features_for_all_sensors(data):
    """
```

```
    For each sensor column (Feature1, Feature2, ..., Feature12) in the␣
↪DataFrame,
    extract 20 features and return a DataFrame with shape (12, 20).
    Assumes a 'Time' column exists for sampling interval.
    """
    feature_cols = [f'Feature{i}' for i in range(1, 13)]
    dt = data['Time'].diff().iloc[1]  # assumes constant time step
    sensor_features = {}
    for col in feature_cols:
        sensor_features[col] = extract_features_from_series(data[col], dt)
    features_df = pd.DataFrame(sensor_features).T  # rows: sensors, columns:␣
↪features
    return features_df

# -------------------------
# Example usage
# -------------------------
if __name__ == "__main__":
    # Generate synthetic data for demonstration
    n = 150000  # number of data points
    time = np.linspace(0, 10, n)
    data_dict = {'Time': time}
    freqs = [1, 0.5, 2, 0.2, 1.5, 0.8, 1.2, 0.3, 0.7, 1.8, 1.1, 0.9]
    for i in range(1, 13):
        data_dict[f'Feature{i}'] = np.sin(2 * np.pi * freqs[i-1] * time) + 0.1␣
↪* np.random.randn(n)
    combined_data = pd.DataFrame(data_dict)

    # Extract 20 features for each of the 12 sensors
    result = extract_features_for_all_sensors(combined_data)

    print("Extracted Features (each row corresponds to a sensor column):")
    print(result)
    print("Shape of extracted features DataFrame:", result.shape)
```

```
Extracted Features (each row corresponds to a sensor column):
           MeanAbs_Velocity  MeanAbs_Jerk  Net_Displacement       RMS  \
Feature1        1688.436914  4.381127e+07          0.305146  0.714387
Feature2        1692.433816  4.399420e+07          0.070357  0.714362
Feature3        1689.266606  4.387566e+07         -0.040871  0.714200
Feature4        1686.202812  4.380503e+07         -0.142397  0.714158
Feature5        1693.107540  4.401713e+07         -0.014761  0.714186
Feature6        1696.057091  4.410137e+07          0.109733  0.714374
Feature7        1694.138231  4.399603e+07         -0.096126  0.714322
Feature8        1695.252799  4.401264e+07          0.029450  0.714434
Feature9        1694.569421  4.405024e+07         -0.135933  0.714032
Feature10       1690.031257  4.389523e+07         -0.070215  0.713987
```

Feature11	1694.879543	4.402265e+07	0.050956	0.713517
Feature12	1693.005032	4.397653e+07	0.301386	0.713940

	Crest_Factor	Zero_Crossing_Rate	Lag1_Autocorrelation	Skewness	\
Feature1	1.933453	0.036113	0.980466	0.001550	
Feature2	1.924614	0.036353	0.980409	0.000166	
Feature3	1.917580	0.035013	0.980475	-0.000689	
Feature4	1.898586	0.036040	0.980583	0.001202	
Feature5	2.001589	0.036520	0.980398	0.000469	
Feature6	1.969082	0.035740	0.980303	-0.000140	
Feature7	1.909490	0.035847	0.980379	-0.000806	
Feature8	1.950427	0.035227	0.980355	-0.000264	
Feature9	1.876081	0.035593	0.980313	-0.000608	
Feature10	2.024537	0.035480	0.980456	0.000687	
Feature11	2.003589	0.036280	0.980320	-0.000104	
Feature12	1.930382	0.035833	0.980437	0.000804	

	Kurtosis	Entropy	Dominant_FFT_Freq	Dominant_FFT_Amplitude	\
Feature1	-1.442248	2.142568	-0.999993	75026.696520	
Feature2	-1.440718	2.143769	0.499997	75023.077909	
Feature3	-1.442820	2.148036	1.999987	75008.412987	
Feature4	-1.444040	2.158849	0.199999	75008.571501	
Feature5	-1.440428	2.132797	1.499990	75005.426310	
Feature6	-1.440817	2.141557	0.799995	75023.015770	
Feature7	-1.442182	2.154087	1.199992	75017.264136	
Feature8	-1.442215	2.140489	-0.299998	75028.716709	
Feature9	-1.442523	2.171962	0.699995	74986.606346	
Feature10	-1.441820	2.118286	1.799988	74984.938097	
Feature11	-1.441283	2.122119	1.099993	74932.835247	
Feature12	-1.442200	2.147890	0.899994	74980.173632	

	PSD_Mean	PSD_Max	PSD_Min	CWT_Mean	Spectral_Energy	\
Feature1	0.000001	0.000003	2.185841e-07	0.000071	76552.213346	
Feature2	0.000001	0.000002	2.221971e-07	0.000026	76546.941101	
Feature3	0.000001	0.000009	2.269449e-07	0.000144	76512.151950	
Feature4	0.000001	0.000001	2.198628e-07	0.000019	76503.351955	
Feature5	0.000001	0.000005	2.274843e-07	0.000068	76509.296885	
Feature6	0.000001	0.000002	2.198668e-07	0.000069	76549.503442	
Feature7	0.000001	0.000004	2.314734e-07	0.000124	76538.418734	
Feature8	0.000001	0.000001	2.106577e-07	0.000025	76562.414865	
Feature9	0.000001	0.000002	2.049264e-07	0.000044	76476.245594	
Feature10	0.000001	0.000007	2.313193e-07	0.000144	76466.604399	
Feature11	0.000001	0.000003	2.225102e-07	0.000095	76365.942922	
Feature12	0.000001	0.000003	2.232135e-07	0.000063	76456.653464	

	TKEO_Mean	Fractal_Dimension	FFT_Amplitude_STD
Feature1	0.010022	0.472178	274.420709
Feature2	0.009978	0.574799	274.407098

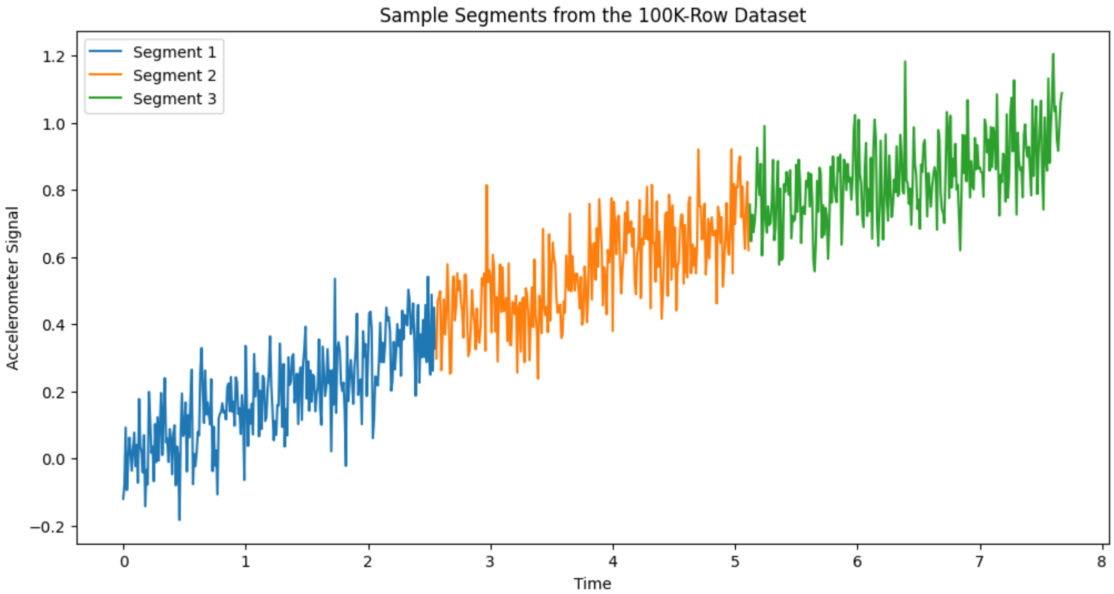
```
Feature3     0.009998          0.371394        274.355571
Feature4     0.009907          0.705879        274.352197
Feature5     0.009969          0.413470        274.342108
Feature6     0.009996          0.506865        274.410400
Feature7     0.010036          0.446547        274.385293
Feature8     0.010034          0.652470        274.428790
Feature9     0.010012          0.526931        274.269875
Feature10    0.009976          0.386587        274.267865
Feature11    0.010007          0.459122        274.075561
Feature12    0.009990          0.488803        274.244919
Shape of extracted features DataFrame: (12, 20)
```

[4]:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# ------------------------------
# 1. Simulate a 100K-row Dataset
# ------------------------------
n_rows = 100000
# Create a DataFrame with simulated Time, Accelerometer data (with added␣
↪noise), and a Damage label.
data = pd.DataFrame({
    'Time': np.linspace(0, 1000, n_rows),
    'Acc_1': np.sin(np.linspace(0, 50 * np.pi, n_rows)) + np.random.normal(0, 0.
↪1, n_rows),
    'Damage': np.random.choice([0.05, 0.1, 0.15, 0.2, 0.25, 0.3], n_rows)
})

print("Data shape:", data.shape)

# ---------------------------------------------
# 2. Segment the Data into Windows/Groups
# ---------------------------------------------
window_size = 256  # Adjust this value as needed
segments = []
for start in range(0, len(data) - window_size + 1, window_size):
    segment = data.iloc[start:start + window_size]
    segments.append(segment)

print("Total segments created:", len(segments))

# ---------------------------------------------
# 3. Extract Sample Features from Each Segment
# ---------------------------------------------
# For demonstration, we compute two simple features:
#   - RMS (Root Mean Square) of the accelerometer signal
```

```python
#   - Mean value of the accelerometer signal
def compute_rms(signal):
    return np.sqrt(np.mean(signal ** 2))

features = []
labels = []
for seg in segments:
    # Extract features from the current segment
    acc_signal = seg['Acc_1'].values
    rms_val = compute_rms(acc_signal)
    mean_val = np.mean(acc_signal)
    # Assuming the Damage label is constant within each segment:
    label = seg['Damage'].iloc[0]

    features.append({'RMS': rms_val, 'Mean': mean_val})
    labels.append(label)

features_df = pd.DataFrame(features)
features_df['Damage'] = labels

print("Extracted features shape:", features_df.shape)
print(features_df.head())

# -------------------------------------------
# 4. (Optional) Visualize the Segmentation
# -------------------------------------------
# Plot a few segments to visually inspect the segmentation
plt.figure(figsize=(12, 6))
for i in range(3):  # Plot first 3 segments
    plt.plot(segments[i]['Time'], segments[i]['Acc_1'], label=f"Segment {i+1}")
plt.xlabel("Time")
plt.ylabel("Accelerometer Signal")
plt.title("Sample Segments from the 100K-Row Dataset")
plt.legend()
plt.show()
```

```
Data shape: (100000, 3)
Total segments created: 390
Extracted features shape: (390, 3)
        RMS      Mean  Damage
0  0.237685  0.189282    0.30
1  0.576682  0.558837    0.25
2  0.846380  0.837873    0.10
3  0.995511  0.989053    0.10
4  0.972473  0.967447    0.25
```



[5]:

```
import numpy as np
import pandas as pd

# ---------------------------------------------------------------------
# 1. Simulate 1 Lakh (100,000) Rows of Data for 12 Sensors
# ---------------------------------------------------------------------
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # just a time axis
data = pd.DataFrame({'Time': time})

# Create 12 sensor columns (e.g., "Sensor1", "Sensor2", ... "Sensor12")
# In reality, you might have actual sensor data.
for i in range(1, 13):
    data[f'Sensor{i}'] = np.sin(0.01 * np.pi * time * i) + 0.1 * np.random.
 ↪randn(n_rows)

# Suppose there's also a "Damage" label (optional):
data['Damage'] = np.random.choice([0.05, 0.10, 0.15, 0.20, 0.25, 0.30],␣
 ↪size=n_rows)

print("Simulated data shape:", data.shape)
print(data.head())

# ---------------------------------------------------------------------
# 2. Define a Function to Compute 20 Features from a 1D Array
# ---------------------------------------------------------------------
def compute_20_features(signal: np.ndarray) -> dict:
```

```
    """
    Placeholder function that returns a dictionary of 20 features
    from a 1D NumPy array (e.g., sensor data).
    Replace these placeholders with your actual 20 feature computations.
    """
    features = {}
    # Here we just compute some trivial stats to fill out 20 keys
    # (In your real code, you'd do RMS, crest factor, FFT, etc.)
    features['feat01_mean'] = np.mean(signal)
    features['feat02_std'] = np.std(signal)
    features['feat03_min'] = np.min(signal)
    features['feat04_max'] = np.max(signal)
    features['feat05_median'] = np.median(signal)
    features['feat06_ptp'] = np.ptp(signal)  # max-min
    features['feat07_sum'] = np.sum(signal)
    features['feat08_var'] = np.var(signal)
    features['feat09_absmean'] = np.mean(np.abs(signal))
    features['feat10_range'] = np.max(signal) - np.min(signal)
    # 10 more placeholder features
    features['feat11'] = np.quantile(signal, 0.1)
    features['feat12'] = np.quantile(signal, 0.9)
    features['feat13'] = signal[0] if len(signal) > 0 else np.nan
    features['feat14'] = signal[-1] if len(signal) > 0 else np.nan
    features['feat15'] = np.corrcoef(signal[::2], signal[1::2])[0,1] if␣
↪len(signal) > 2 else np.nan
    features['feat16'] = np.sum(np.diff(signal) > 0)
    features['feat17'] = np.sum(np.diff(signal) < 0)
    features['feat18'] = np.mean(signal**2)
    features['feat19'] = np.mean(np.sqrt(np.abs(signal+1e-6)))
    features['feat20'] = np.std(np.gradient(signal))

    return features

# ---------------------------------------------------------------------
# 3. Segment the Data into 390 Groups (Each ~256 Rows)
# ---------------------------------------------------------------------
segment_size = 256
segments = []
for start in range(0, n_rows, segment_size):
    end = start + segment_size
    if end <= n_rows:
        segments.append(data.iloc[start:end])

print(f"Number of segments created: {len(segments)}")  # ~390

# ---------------------------------------------------------------------
# 4. For Each Segment, Compute 20 Features per Sensor (12 sensors)
```

```
#    Flatten them into 240 columns (12 * 20).
# ----------------------------------------------------------------------
all_rows = []  # each element will be a dict representing one segment (row)

for seg_idx, segment in enumerate(segments):
    # We'll store all sensor features in one dictionary (row)
    row_dict = {}
    for i in range(1, 13):
        sensor_col = f"Sensor{i}"
        sensor_data = segment[sensor_col].values

        # Compute 20 features for this sensor
        feat_dict = compute_20_features(sensor_data)

        # Flatten them into row_dict with a naming scheme
        for feat_name, feat_val in feat_dict.items():
            # e.g. "Sensor1_feat01_mean", "Sensor2_feat01_mean", ...
            row_dict[f"{sensor_col}_{feat_name}"] = feat_val

    # Optionally store a label for the segment (e.g., average or first "Damage"␣
↪in the segment)
    row_dict['SegmentDamage'] = segment['Damage'].iloc[0]  # or .mean() if you␣
↪prefer

    all_rows.append(row_dict)

# Convert all_rows into a DataFrame
features_df = pd.DataFrame(all_rows)
print("Final features DataFrame shape:", features_df.shape)
print(features_df.head())

# We expect ~390 rows, each with 12 * 20 = 240 feature columns, plus 1 label␣
↪column -> (390, 241)
```

```
Simulated data shape: (100000, 14)
   Time   Sensor1   Sensor2   Sensor3   Sensor4   Sensor5   Sensor6   Sensor7  \
0  0.00 -0.013383  0.305557 -0.044475 -0.010284  0.074777  0.131225  0.029651
1  0.01 -0.098597  0.020484 -0.065878  0.105060 -0.064049 -0.011663  0.039651
2  0.02  0.163147 -0.105378 -0.006362 -0.055433 -0.002768  0.272252  0.052751
3  0.03  0.047290 -0.073128  0.081162  0.037959  0.169429  0.048392  0.180323
4  0.04  0.103258 -0.053191  0.069239  0.029079  0.055053  0.124344 -0.101352

    Sensor8   Sensor9  Sensor10  Sensor11  Sensor12  Damage
0  0.112847  0.127011 -0.168654 -0.032189 -0.142300    0.25
1  0.146624 -0.105646  0.209195 -0.103877 -0.025494    0.30
2  0.141384  0.046482  0.150407 -0.012722  0.020389    0.05
3 -0.094005  0.008518 -0.039630 -0.005638 -0.123363    0.30
```

```
4  0.026943 -0.096180  0.056954 -0.012041 -0.068952    0.25
Number of segments created: 390
Final features DataFrame shape: (390, 241)
   Sensor1_feat01_mean  Sensor1_feat02_std  Sensor1_feat03_min  \
0             0.041298            0.102276           -0.268537
1             0.115216            0.105221           -0.231939
2             0.201653            0.101620           -0.069683
3             0.278004            0.106441           -0.032483
4             0.359274            0.113350            0.029585

   Sensor1_feat04_max  Sensor1_feat05_median  Sensor1_feat06_ptp  \
0            0.290620               0.050869            0.559157
1            0.432885               0.117764            0.664824
2            0.447097               0.200688            0.516779
3            0.632512               0.269885            0.664995
4            0.691149               0.358111            0.661565

   Sensor1_feat07_sum  Sensor1_feat08_var  Sensor1_feat09_absmean  \
0           10.572250            0.010460                0.091367
1           29.495383            0.011072                0.128628
2           51.623195            0.010327                0.202883
3           71.169009            0.011330                0.278258
4           91.974249            0.012848                0.359274

   Sensor1_feat10_range  …  Sensor12_feat12  Sensor12_feat13  \
0              0.559157  …         0.796075        -0.142300
1              0.664824  …         1.075125         0.742816
2              0.516779  …         0.911479         0.897458
3              0.664995  …         0.137010         0.295930
4              0.661565  …        -0.729408        -0.841814

   Sensor12_feat14  Sensor12_feat15  Sensor12_feat16  Sensor12_feat17  \
0         0.924272         0.855222              136              119
1         0.920154         0.280234              130              125
2         0.179653         0.780534              128              127
3        -0.557551         0.884767              127              128
4        -0.981727         0.514284              123              132

   Sensor12_feat18  Sensor12_feat19  Sensor12_feat20  SegmentDamage
0         0.269808         0.635318         0.074628           0.25
1         0.906899         0.970441         0.067853           0.15
2         0.455379         0.784811         0.080268           0.20
3         0.131498         0.504985         0.068920           0.05
4         0.854436         0.952581         0.074782           0.05

[5 rows x 241 columns]
```

[3]:
```python
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

# ----------------------------------------------------------------------
# 1. Simulate 100,000 Rows of Data for 12 Sensors
# ----------------------------------------------------------------------
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # Time axis
data = pd.DataFrame({'Time': time})

# Create 12 sensor columns ("Sensor1" to "Sensor12") with synthetic data
for i in range(1, 13):
    # Example: a sine wave with a slight frequency variation and added noise
    data[f'Sensor{i}'] = np.sin(0.01 * np.pi * time * i) + 0.1 * np.random.
 ↪randn(n_rows)

# Add an optional 'Damage' column (randomly chosen labels for demonstration)
data['Damage'] = np.random.choice([0.05, 0.10, 0.15, 0.20, 0.25, 0.30],␣
 ↪size=n_rows)

print("Simulated data shape:", data.shape)
print(data.head())

# ----------------------------------------------------------------------
# 2. Define Feature Extraction Functions (20 Features)
# ----------------------------------------------------------------------
def extract_features_from_series(x, dt):
    """
    Given a pandas Series x (time series for one sensor) and time step dt,
    compute 20 features and return them as a dictionary.
    """
    features = {}
    n = len(x)

    # 1. Mean Absolute Velocity
    vel = np.diff(x) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
```

```python
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement (last - first)
    features['Net_Displacement'] = x.iloc[-1] - x.iloc[0]

    # 4. RMS: sqrt(mean(x^2))
    rms = np.sqrt(np.mean(x**2))
    features['RMS'] = rms

    # 5. Crest Factor: max(|x|)/RMS
    features['Crest_Factor'] = np.max(np.abs(x)) / rms if rms != 0 else np.nan

    # 6. Zero Crossing Rate: (# sign changes)/n
    x_arr = x.values
    zero_crossings = np.sum(x_arr[:-1] * x_arr[1:] < 0)
    features['Zero_Crossing_Rate'] = zero_crossings / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy using histogram
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist = hist + 1e-8  # avoid log(0)
    features['Entropy'] = entropy(hist)

    # Frequency Domain Features:
    # 11. Dominant FFT Frequency and 12. its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_magnitude = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_magnitude[1:]) + 1  # ignore the zero frequency term
        dom_freq = fft_freqs[idx]
        dom_amp = fft_magnitude[idx]
    else:
        dom_freq, dom_amp = np.nan, np.nan
    features['Dominant_FFT_Freq'] = dom_freq
```

```python
    features['Dominant_FFT_Amplitude'] = dom_amp

    # 13. PSD Mean, 14. PSD Max, 15. PSD Min (using Welch)
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. CWT Mean (using scales 1 to 99, wavelet 'gaus1')
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy: sum(x^2)
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO: average of (x^2 - shift(x)*shift(x,-1))
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension: Hurst Exponent (using 'change' kind)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. STD of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_magnitude)

    return features

def extract_features_for_all_sensors(data):
    """
    For each sensor column (Sensor1, Sensor2, ..., Sensor12) in the DataFrame,
    extract 20 features and return a DataFrame with shape (num_segments, 12*20 + 1).
    Each row corresponds to one segment with flattened sensor features and a label.
    """
    # Assume segmentation has already been done; data here is one segment.
    # This function is for one segment.
    sensor_features = {}
    dt = data['Time'].diff().iloc[1]  # constant time step assumed
    for i in range(1, 13):
        col = f'Sensor{i}'
        sensor_features[col] = extract_features_from_series(data[col], dt)
```

```python
    # Create a DataFrame where each row corresponds to a sensor
    features_df = pd.DataFrame(sensor_features).T  # shape (12, 20)
    return features_df

# -----------------------------------------------------------------------
# 3. Segment the Data into ~390 Groups (Each 256 Rows)
# -----------------------------------------------------------------------
segment_size = 256
segments = []
for start in range(0, n_rows - segment_size + 1, segment_size):
    segment = data.iloc[start:start + segment_size]
    segments.append(segment)
print(f"Number of segments created: {len(segments)}")  # Expect ~390

# -----------------------------------------------------------------------
# 4. For Each Segment, Compute 20 Features per Sensor and Flatten
# -----------------------------------------------------------------------
# Each segment will produce 12 (sensors) * 20 (features) = 240 feature columns,
# plus one additional label column.
all_rows = []  # List to store one dictionary per segment (one row in final DF)

for seg in segments:
    row_dict = {}
    # Process each sensor column and flatten its 20 features into the row
    for i in range(1, 13):
        sensor_col = f"Sensor{i}"
        feat_dict = extract_features_from_series(seg[sensor_col], seg['Time'].
 ↪diff().iloc[1])
        # Flatten: prefix the feature keys with the sensor name
        for key, value in feat_dict.items():
            row_dict[f"{sensor_col}_{key}"] = value
    # Optionally, use the first Damage value in the segment as the segment label
    row_dict['SegmentDamage'] = seg['Damage'].iloc[0]
    all_rows.append(row_dict)

# Create the final features DataFrame
final_features_df = pd.DataFrame(all_rows)
print("Final features DataFrame shape:", final_features_df.shape)
# Expected shape: (~390, 240 + 1) -> e.g., (390, 241)

# Display a preview of the final features DataFrame
print(final_features_df.head())
```

```
Simulated data shape: (100000, 14)
   Time   Sensor1   Sensor2   Sensor3   Sensor4   Sensor5   Sensor6   Sensor7  \
0  0.00  0.142622  0.220611 -0.065773 -0.048222  0.017803  0.008056 -0.074051
1  0.01  0.148194  0.138994 -0.065650 -0.062199  0.052294 -0.018037 -0.101506
```

```
2  0.02  0.041945  0.045005 -0.027314  0.114582  0.002700 -0.139452 -0.040259
3  0.03 -0.125134 -0.049315  0.031280  0.177899 -0.182674  0.006277  0.193689
4  0.04  0.176463  0.032660  0.121469 -0.211322 -0.098674  0.043087 -0.091127

    Sensor8   Sensor9  Sensor10  Sensor11  Sensor12  Damage
0  0.102888 -0.020890  0.106036 -0.057215 -0.044240    0.10
1  0.046315  0.105087 -0.023760  0.031174  0.134643    0.15
2 -0.111741 -0.061613  0.157401  0.066362 -0.004502    0.20
3 -0.197041  0.027804  0.142563  0.016752 -0.149655    0.10
4 -0.016065  0.097098 -0.099805 -0.026188  0.061027    0.05
Number of segments created: 390
Final features DataFrame shape: (390, 241)
   Sensor1_MeanAbs_Velocity  Sensor1_MeanAbs_Jerk  Sensor1_Net_Displacement  \
0                 11.591894           2055.959862                 -0.036239
1                 10.957831           1891.374079                 -0.187637
2                 10.700148           1830.719307                  0.172807
3                 10.761194           1861.768594                  0.302799
4                 10.726919           1842.198770                  0.026237

   Sensor1_RMS  Sensor1_Crest_Factor  Sensor1_Zero_Crossing_Rate  \
0     0.107796              2.709808                    0.484375
1     0.158011              2.582230                    0.203125
2     0.225731              2.230574                    0.031250
3     0.300596              1.867790                    0.007812
4     0.366797              1.602821                    0.007812

   Sensor1_Lag1_Autocorrelation  Sensor1_Skewness  Sensor1_Kurtosis  \
0                      0.003438         -0.336892         -0.224202
1                      0.048573         -0.099341          0.116423
2                      0.103550          0.039021         -0.002922
3                      0.197059         -0.117413         -0.165631
4                      0.074468         -0.235868          0.147718

   Sensor1_Entropy  …  Sensor12_Dominant_FFT_Amplitude  Sensor12_PSD_Mean  \
0         1.999148  …                        35.515917           0.000537
1         1.941651  …                         7.317883           0.000259
2         1.989522  …                        29.505604           0.000369
3         1.994487  …                        37.030971           0.000547
4         1.926126  …                        15.918939           0.000253

   Sensor12_PSD_Max  Sensor12_PSD_Min  Sensor12_CWT_Mean  \
0          0.038988      1.735335e-06          -0.722211
1          0.002191      3.003839e-06          -0.127521
2          0.025905      1.118887e-06           0.569996
3          0.040311      1.139852e-06           0.774679
4          0.008359      6.990611e-07           0.330513

   Sensor12_Spectral_Energy  Sensor12_TKEO_Mean  Sensor12_Fractal_Dimension  \
```

```
0                    69.315048            0.009471                   0.628217
1                   234.863398            0.011417                   0.948833
2                   119.971511            0.008160                   0.753924
3                    32.415712            0.009876                   0.514918
4                   211.095050            0.008360                   0.819759

   Sensor12_FFT_Amplitude_STD  SegmentDamage
0                    7.873203           0.10
1                   15.117749           0.15
2                   10.635696           0.20
3                    5.061945           0.05
4                   14.311441           0.15

[5 rows x 241 columns]
```

[7]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

# -----------------------------------------------------------
# 1. Define the 20-Feature Extraction Function for a Series
# -----------------------------------------------------------
def extract_features_from_series(x, dt):
    """
    Given a pandas Series x (time series for one sensor) and time step dt,
    compute 20 features and return them as a dictionary.
    """
    features = {}
    n = len(x)

    # 1. Mean Absolute Velocity
    vel = np.diff(x) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk (second derivative)
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement (last - first)
    features['Net_Displacement'] = x.iloc[-1] - x.iloc[0]
```

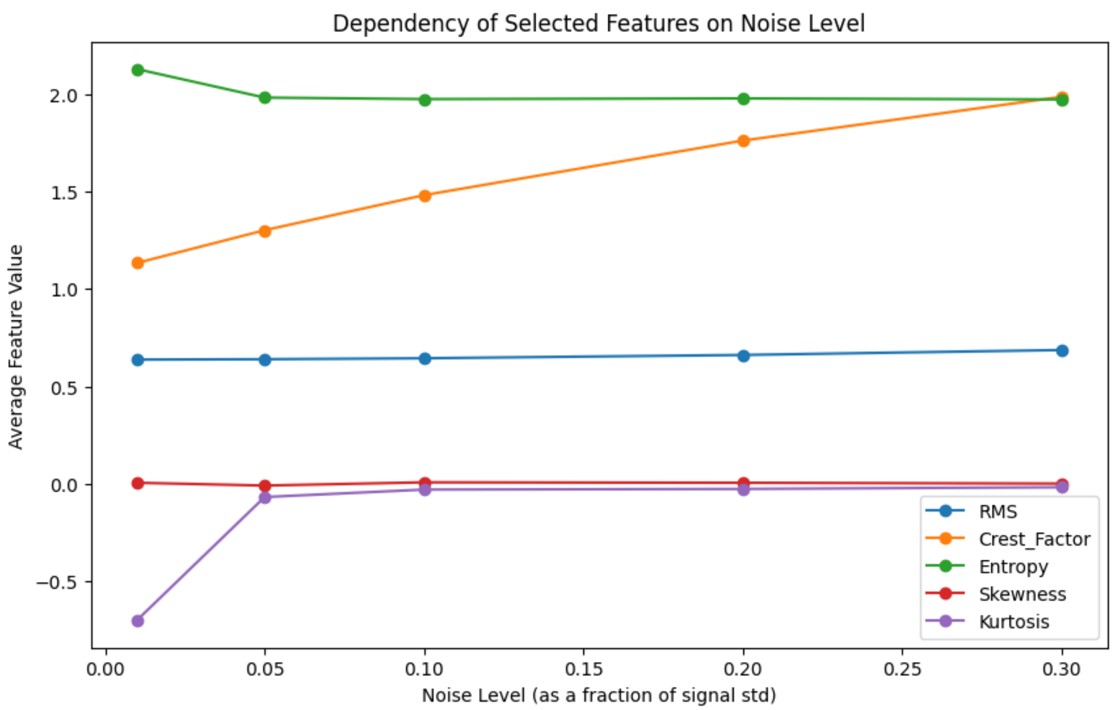
```
# 4. RMS: sqrt(mean(x^2))
rms = np.sqrt(np.mean(x**2))
features['RMS'] = rms

# 5. Crest Factor: max(|x|)/RMS (avoid division by zero)
features['Crest_Factor'] = np.max(np.abs(x)) / rms if rms != 0 else np.nan

# 6. Zero Crossing Rate: (# sign changes)/n
x_arr = x.values
zero_crossings = np.sum(x_arr[:-1] * x_arr[1:] < 0)
features['Zero_Crossing_Rate'] = zero_crossings / n

# 7. Lag-1 Autocorrelation
if n > 1:
    autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
else:
    autocorr = np.nan
features['Lag1_Autocorrelation'] = autocorr

# 8. Skewness
features['Skewness'] = skew(x_arr)

# 9. Kurtosis
features['Kurtosis'] = kurtosis(x_arr)

# 10. Entropy (from histogram)
hist, _ = np.histogram(x_arr, bins=10, density=True)
hist = hist + 1e-8  # avoid log(0)
features['Entropy'] = entropy(hist)

# Frequency Domain Features:
# 11. Dominant FFT Frequency and 12. its Amplitude
fft_vals = np.fft.fft(x_arr)
fft_freqs = np.fft.fftfreq(n, d=dt)
fft_magnitude = np.abs(fft_vals)
if n > 1:
    idx = np.argmax(fft_magnitude[1:]) + 1  # ignore the zero-frequency term
    dom_freq = fft_freqs[idx]
    dom_amp = fft_magnitude[idx]
else:
    dom_freq, dom_amp = np.nan, np.nan
features['Dominant_FFT_Freq'] = dom_freq
features['Dominant_FFT_Amplitude'] = dom_amp

# 13. PSD Mean, 14. PSD Max, 15. PSD Min (using Welch)
freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
```

```
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. CWT Mean (using scales 1 to 99, wavelet 'gaus1')
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy: sum(x^2)
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO: average of (x^2 - shift(x)*shift(x,-1))
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension: Hurst Exponent (using 'change' kind)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. STD of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_magnitude)

    return features

# ---------------------------------------------------------
# 2. Define a Function to Add Noise to a Signal
# ---------------------------------------------------------
def add_noise_to_signal(signal, noise_factor):
    """
    Adds Gaussian noise scaled by (noise_factor * std) to the signal.
    """
    noise = np.random.normal(0, noise_factor * np.std(signal), size=signal.
 ↪shape)
    return signal + noise

# ---------------------------------------------------------
# 3. Simulate a Clean Signal (for one sensor) with 100K Rows
# ---------------------------------------------------------
n_rows = 100000
time = np.linspace(0, 1000, n_rows)
original_signal = np.sin(0.01 * np.pi * time)  # example clean sine wave

# ---------------------------------------------------------
```

```python
# 4. Loop Over Multiple Noise Levels to Assess Noise Dependency
# ------------------------------------------------------------
# Define a list of noise levels (e.g., 1%, 5%, 10%, 20%, 30%)
noise_levels = [0.01, 0.05, 0.1, 0.2, 0.3]
segment_size = 256  # Each segment will contain 256 rows

results = []  # To store average feature values per noise level

for noise in noise_levels:
    # Add noise to the clean signal
    noisy_signal = add_noise_to_signal(original_signal, noise)

    # Build a DataFrame for this noisy signal
    df = pd.DataFrame({'Time': time, 'Sensor1': noisy_signal})

    # Segment the DataFrame into non-overlapping windows
    segments = []
    for start in range(0, n_rows - segment_size + 1, segment_size):
        segment = df.iloc[start:start+segment_size]
        segments.append(segment)

    # For each segment, extract features from Sensor1 and store them
    features_list = []
    for seg in segments:
        dt = seg['Time'].diff().iloc[1]  # assuming constant dt
        feat = extract_features_from_series(seg['Sensor1'], dt)
        features_list.append(feat)

    # Convert to DataFrame and average features across segments
    features_df = pd.DataFrame(features_list)
    avg_features = features_df.mean()
    avg_features['Noise_Level'] = noise
    results.append(avg_features)

# Create a final DataFrame that shows average feature values for each noise␣
 ↪level
final_noise_df = pd.DataFrame(results)
print("Average features at different noise levels:")
print(final_noise_df)

# ------------------------------------------------------------
# 5. Visualize How Selected Features Depend on Noise Level
# ------------------------------------------------------------
import matplotlib.pyplot as plt
import seaborn as sns

features_to_plot = ['RMS', 'Crest_Factor', 'Entropy', 'Skewness', 'Kurtosis']
```

```
plt.figure(figsize=(10, 6))
for feature in features_to_plot:
    plt.plot(final_noise_df['Noise_Level'], final_noise_df[feature],␣
 ↪marker='o', label=feature)
plt.xlabel('Noise Level (as a fraction of signal std)')
plt.ylabel('Average Feature Value')
plt.title('Dependency of Selected Features on Noise Level')
plt.legend()
plt.show()
```

```
Average features at different noise levels:
   MeanAbs_Velocity  MeanAbs_Jerk  Net_Displacement       RMS  Crest_Factor  \
0          0.800139    138.659093         -0.000568  0.638304      1.134039
1          3.996394    691.872463         -0.002221  0.640142      1.301736
2          7.960988   1379.901832         -0.000571  0.645166      1.481489
3         15.945414   2762.482891         -0.001464  0.661968      1.761113
4         23.822592   4122.004361         -0.009860  0.687338      1.985555

   Zero_Crossing_Rate  Lag1_Autocorrelation  Skewness  Kurtosis   Entropy  \
0            0.002354              0.705865  0.006574 -0.697494  2.127703
1            0.011869              0.165185 -0.007736 -0.066607  1.981929
2            0.023658              0.045454  0.008860 -0.028344  1.974160
3            0.050190              0.005436  0.006721 -0.025099  1.977650
4            0.076392              0.004766  0.002451 -0.016419  1.972459

   …  Dominant_FFT_Amplitude  PSD_Mean   PSD_Max       PSD_Min  CWT_Mean  \
0  …                2.093306  0.000002  0.000158  6.967273e-09  0.000046
1  …                2.302476  0.000026  0.000209  1.421302e-07 -0.000026
2  …                2.958286  0.000100  0.000552  6.055401e-07 -0.001025
3  …                5.331362  0.000393  0.002052  2.404980e-06  0.000248
4  …                7.897561  0.000886  0.004591  5.520253e-06 -0.000288

   Spectral_Energy  TKEO_Mean  Fractal_Dimension  FFT_Amplitude_STD  \
0       128.216983   0.000050           0.739569          10.180132
1       128.530574   0.001244           0.948726          10.161547
2       129.499188   0.004981           0.963508          10.151436
3       133.251837   0.019847           0.964508          10.162410
4       140.040101   0.045019           0.961020          10.226903

   Noise_Level
0         0.01
1         0.05
2         0.10
3         0.20
4         0.30

[5 rows x 21 columns]
```



[8]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
import pywt
from hurst import compute_Hc

# -----------------------------------------------------------
# 1. Feature Extraction Function (20 Features)
# -----------------------------------------------------------
def extract_features_from_series(x, dt):
    """
    Given a pandas Series x (time series for one sensor) and time step dt,
    compute 20 features and return them as a dictionary.
    """
    features = {}
    n = len(x)

    # 1. Mean Absolute Velocity
    vel = np.diff(x) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk (second derivative)
```

```
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement (last - first)
    features['Net_Displacement'] = x.iloc[-1] - x.iloc[0]

    # 4. RMS: sqrt(mean(x^2))
    rms = np.sqrt(np.mean(x**2))
    features['RMS'] = rms

    # 5. Crest Factor: max(|x|)/RMS
    features['Crest_Factor'] = np.max(np.abs(x)) / rms if rms != 0 else np.nan

    # 6. Zero Crossing Rate: (# sign changes)/n
    x_arr = x.values
    zero_crossings = np.sum(x_arr[:-1] * x_arr[1:] < 0)
    features['Zero_Crossing_Rate'] = zero_crossings / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy (from histogram)
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist = hist + 1e-8  # to avoid log(0)
    features['Entropy'] = entropy(hist)

    # Frequency Domain Features:
    # 11. Dominant FFT Frequency and 12. its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_magnitude = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_magnitude[1:]) + 1  # ignoring zero frequency
        dom_freq = fft_freqs[idx]
```

```
        dom_amp = fft_magnitude[idx]
    else:
        dom_freq, dom_amp = np.nan, np.nan
    features['Dominant_FFT_Freq'] = dom_freq
    features['Dominant_FFT_Amplitude'] = dom_amp

    # 13. PSD Mean, 14. PSD Max, 15. PSD Min (using Welch)
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. CWT Mean (using scales 1 to 99, wavelet 'gaus1')
    scales = np.arange(1, 100)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy: sum(x^2)
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO: average of (x^2 - shift(x)*shift(x,-1))
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension: Hurst Exponent (using 'change' kind)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. Standard Deviation of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_magnitude)

    return features

# ----------------------------------------------------------
# 2. Function to Add Gaussian Noise
# ----------------------------------------------------------
def add_noise_to_signal(signal, noise_factor):
    """
    Add Gaussian noise scaled by noise_factor * std to the signal.
    """
    noise = np.random.normal(0, noise_factor * np.std(signal), size=signal.
↪shape)
    return signal + noise
```

```python
# ----------------------------------------------------------
# 3. Simulate a Clean Signal (for one sensor) with 100K Rows
# ----------------------------------------------------------
n_rows = 100000
time = np.linspace(0, 1000, n_rows)
original_signal = np.sin(0.01 * np.pi * time)  # Clean sine wave

# ----------------------------------------------------------
# 4. Loop Over Noise Levels from 1% to 30% and Extract Features
# ----------------------------------------------------------
# We'll use 30 noise levels (1% to 30%).
noise_levels = np.linspace(0.01, 0.30, 30)
segment_size = 256  # Each segment contains 256 rows

results = []  # To store average features for each noise level

for noise in noise_levels:
    # Add noise to the original signal
    noisy_signal = add_noise_to_signal(original_signal, noise)

    # Build a DataFrame for this noisy signal
    df = pd.DataFrame({'Time': time, 'Sensor1': noisy_signal})

    # Segment the DataFrame into non-overlapping windows
    segments = []
    for start in range(0, n_rows - segment_size + 1, segment_size):
        segment = df.iloc[start:start + segment_size]
        segments.append(segment)

    # For each segment, extract features from Sensor1 and store them
    features_list = []
    for seg in segments:
        dt = seg['Time'].diff().iloc[1]  # assuming constant dt
        feat = extract_features_from_series(seg['Sensor1'], dt)
        features_list.append(feat)

    # Convert list of feature dictionaries into a DataFrame and average the␣
 ↪features
    features_df = pd.DataFrame(features_list)
    avg_features = features_df.mean()
    avg_features['Noise_Level'] = noise
    results.append(avg_features)

# Create a final DataFrame that contains average feature values for each noise␣
 ↪level
final_noise_df = pd.DataFrame(results)
print("Average features at different noise levels:")
```

```
print(final_noise_df)

# ----------------------------------------------------------
# 5. Plot All 20 Features vs. Noise Level
# ----------------------------------------------------------
import matplotlib.pyplot as plt
import seaborn as sns

# Extract feature names (excluding 'Noise_Level')
feature_names = [col for col in final_noise_df.columns if col != 'Noise_Level']

# Create subplots: for 20 features, we can arrange them in a 5x4 grid.
n_features = len(feature_names)
n_rows_plot = 5
n_cols_plot = 4

fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(18, 18),␣
↪sharex=True)
axes = axes.flatten()

for idx, feature in enumerate(feature_names):
    ax = axes[idx]
    ax.plot(final_noise_df['Noise_Level']*100, final_noise_df[feature],␣
↪marker='o', linestyle='-')
    ax.set_title(feature)
    ax.set_xlabel("Noise Level (%)")
    ax.set_ylabel("Avg Feature Value")
    ax.grid(True)

plt.tight_layout()
plt.show()
```

```
Average features at different noise levels:
    MeanAbs_Velocity  MeanAbs_Jerk  Net_Displacement       RMS  Crest_Factor  \
0           0.799799    138.537586         -0.000740  0.638276      1.132330
1           1.597301    276.438777         -0.000597  0.638619      1.176324
2           2.397255    415.038277          0.000001  0.638973      1.216446
3           3.189223    552.491665         -0.002333  0.639436      1.265692
4           4.009781    693.685180         -0.000399  0.640165      1.296842
5           4.769341    826.089960         -0.004129  0.640825      1.336793
6           5.569450    963.468240         -0.000488  0.641800      1.370252
7           6.356747   1101.930203         -0.004786  0.642576      1.403536
8           7.170854   1242.184428         -0.002372  0.643977      1.448426
9           8.006650   1386.924116          0.008564  0.644958      1.474048
10          8.823309   1527.600540          0.002424  0.646390      1.509165
11          9.604517   1663.340406         -0.006450  0.647494      1.536984
12         10.361275   1794.787684         -0.000438  0.649054      1.561938
```

13	11.235705	1945.948503	0.002641	0.651455	1.606801
14	11.974137	2072.857521	-0.007129	0.652463	1.633444
15	12.800275	2217.019128	0.000711	0.654561	1.663460
16	13.515312	2339.650251	-0.015921	0.655518	1.692689
17	14.409498	2494.876728	0.001230	0.657896	1.704899
18	15.170841	2631.878266	0.007386	0.659995	1.726115
19	15.991699	2767.864788	0.019774	0.661698	1.762518
20	16.752003	2903.386946	0.001602	0.664575	1.786170
21	17.510534	3036.657019	-0.008252	0.665897	1.808190
22	18.379309	3176.976611	-0.014044	0.668559	1.832884
23	19.157899	3324.146304	0.023961	0.671190	1.856251
24	19.880047	3442.604526	0.015406	0.672988	1.883538
25	20.743682	3590.575673	0.003338	0.675973	1.910080
26	21.560617	3734.745493	-0.008868	0.678998	1.916837
27	22.383936	3883.251411	-0.022929	0.681349	1.950825
28	23.111700	3997.303020	0.015156	0.684071	1.960736
29	23.887867	4129.251929	0.006331	0.687328	1.992431

	Zero_Crossing_Rate	Lag1_Autocorrelation	Skewness	Kurtosis	Entropy	\
0	0.002274	0.705649	-0.002303	-0.700599	2.125878	
1	0.004948	0.475798	-0.002734	-0.351155	2.043632	
2	0.007071	0.319158	0.004733	-0.179262	2.001968	
3	0.010026	0.221841	0.001502	-0.119073	1.990307	
4	0.012500	0.158338	0.004286	-0.091138	1.988444	
5	0.014854	0.118358	-0.005563	-0.051848	1.981528	
6	0.016827	0.092323	0.000786	-0.038932	1.976387	
7	0.019301	0.072284	0.009382	-0.048949	1.983929	
8	0.022015	0.056827	0.005542	-0.023317	1.969578	
9	0.025070	0.046210	-0.006777	-0.012179	1.976707	
10	0.027825	0.032052	0.008004	-0.046515	1.978841	
11	0.029958	0.026455	-0.010035	-0.040097	1.977781	
12	0.032642	0.028508	0.018009	-0.042687	1.975999	
13	0.034645	0.018620	-0.004765	0.001068	1.968214	
14	0.037510	0.021805	0.002287	-0.034977	1.975011	
15	0.039683	0.014148	0.000257	-0.024031	1.979762	
16	0.042929	0.019218	0.003007	-0.024471	1.977121	
17	0.045603	0.006737	0.006442	-0.011888	1.972480	
18	0.048037	0.007773	0.001076	-0.041358	1.978910	
19	0.049740	0.006111	-0.004099	-0.024944	1.974039	
20	0.053446	0.007540	0.002843	-0.031004	1.973524	
21	0.055990	0.008957	-0.006854	-0.021873	1.972364	
22	0.057622	0.004212	0.007803	-0.040668	1.980115	
23	0.061659	0.005537	-0.010471	-0.019380	1.973224	
24	0.063712	0.006018	0.014910	-0.017951	1.972965	
25	0.066637	0.005946	0.001891	-0.019107	1.973924	
26	0.070262	0.000477	-0.001785	-0.014711	1.965067	
27	0.071715	-0.001103	-0.005057	-0.000580	1.967591	
28	0.073377	0.008027	-0.002610	-0.000999	1.970417	

29	0.077334	0.003625	-0.002619	-0.010933	1.969636

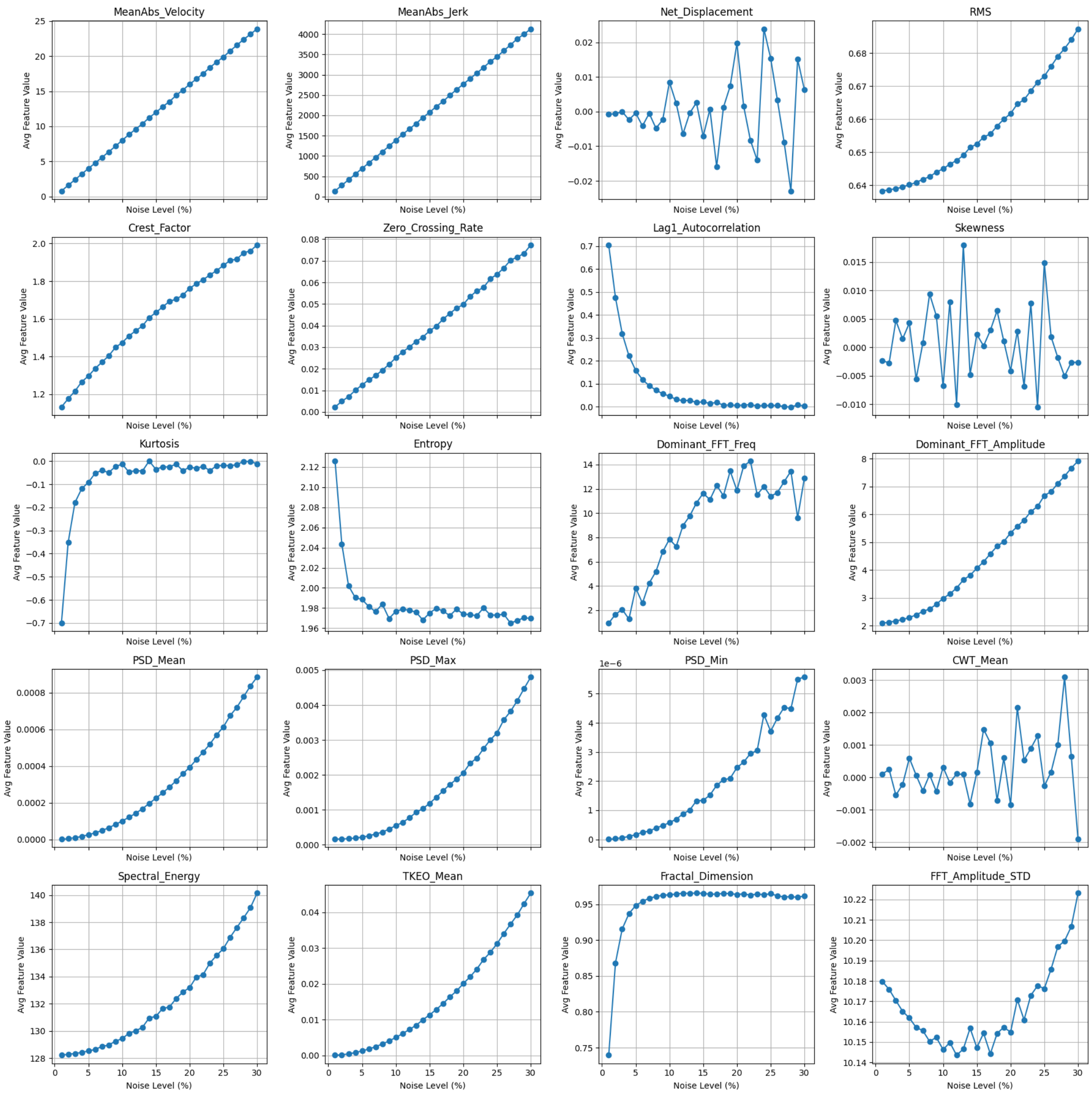
	…	Dominant_FFT_Amplitude	PSD_Mean	PSD_Max	PSD_Min	CWT_Mean \
0	…	2.097052	0.000002	0.000159	6.594953e-09	0.000107
1	…	2.129748	0.000005	0.000165	2.331365e-08	0.000248
2	…	2.173743	0.000010	0.000174	5.742722e-08	-0.000547
3	…	2.223023	0.000017	0.000192	1.006239e-07	-0.000214
4	…	2.298664	0.000026	0.000216	1.611487e-07	0.000598
5	…	2.389843	0.000037	0.000255	2.450688e-07	0.000073
6	…	2.517708	0.000049	0.000306	2.781425e-07	-0.000410
7	…	2.603425	0.000064	0.000360	3.923576e-07	0.000085
8	…	2.771381	0.000082	0.000449	4.725013e-07	-0.000431
9	…	2.986095	0.000101	0.000546	5.732777e-07	0.000318
10	…	3.146457	0.000121	0.000638	6.935628e-07	-0.000166
11	…	3.354917	0.000143	0.000769	8.719131e-07	0.000114
12	…	3.647547	0.000168	0.000926	1.004310e-06	0.000093
13	…	3.815053	0.000197	0.001033	1.322785e-06	-0.000818
14	…	4.064664	0.000226	0.001184	1.337788e-06	0.000158
15	…	4.297572	0.000255	0.001357	1.520503e-06	0.001488
16	…	4.586958	0.000285	0.001540	1.855763e-06	0.001068
17	…	4.856050	0.000321	0.001724	2.043413e-06	-0.000713
18	…	5.022001	0.000359	0.001875	2.096323e-06	0.000603
19	…	5.337178	0.000394	0.002055	2.464305e-06	-0.000843
20	…	5.566491	0.000436	0.002334	2.650281e-06	0.002159
21	…	5.795495	0.000477	0.002477	2.947440e-06	0.000536
22	…	6.100342	0.000520	0.002751	3.048022e-06	0.000899
23	…	6.286296	0.000570	0.003004	4.279498e-06	0.001284
24	…	6.678301	0.000614	0.003199	3.712298e-06	-0.000265
25	…	6.815744	0.000675	0.003576	4.163553e-06	0.000168
26	…	7.114472	0.000721	0.003818	4.532285e-06	0.001007
27	…	7.372548	0.000778	0.004124	4.484205e-06	0.003098
28	…	7.664905	0.000837	0.004468	5.501812e-06	0.000644
29	…	7.919152	0.000885	0.004802	5.570616e-06	-0.001903

	Spectral_Energy	TKEO_Mean	Fractal_Dimension	FFT_Amplitude_STD \
0	128.213011	0.000053	0.739437	10.179732
1	128.289294	0.000196	0.868186	10.175890
2	128.338678	0.000445	0.915524	10.170388
3	128.424124	0.000794	0.936831	10.165113
4	128.526934	0.001270	0.948268	10.161860
5	128.646933	0.001797	0.953858	10.157210
6	128.858465	0.002416	0.958354	10.155601
7	128.958713	0.003212	0.960570	10.150344
8	129.245555	0.004015	0.962735	10.152306
9	129.431777	0.005030	0.962873	10.146296
10	129.830198	0.006063	0.964427	10.149809
11	130.002028	0.007245	0.965203	10.143534
12	130.264505	0.008397	0.965301	10.146610

13	130.954220	0.009932	0.965419	10.156812
14	131.070825	0.011315	0.965273	10.147290
15	131.658246	0.012822	0.963944	10.154367
16	131.760837	0.014471	0.964244	10.144237
17	132.384509	0.016327	0.965047	10.154214
18	132.863527	0.017977	0.964887	10.157322
19	133.194233	0.020047	0.963660	10.154854
20	133.973626	0.021925	0.964555	10.170853
21	134.149307	0.024090	0.962557	10.160901
22	134.998321	0.026742	0.964297	10.172924
23	135.556838	0.028758	0.963238	10.177681
24	136.057961	0.031185	0.964600	10.176259
25	136.887521	0.033984	0.961662	10.185816
26	137.613479	0.036646	0.960255	10.196829
27	138.320754	0.039191	0.960452	10.199664
28	139.061065	0.042336	0.959784	10.206731
29	140.141618	0.045294	0.961476	10.223129

	Noise_Level
0	0.01
1	0.02
2	0.03
3	0.04
4	0.05
5	0.06
6	0.07
7	0.08
8	0.09
9	0.10
10	0.11
11	0.12
12	0.13
13	0.14
14	0.15
15	0.16
16	0.17
17	0.18
18	0.19
19	0.20
20	0.21
21	0.22
22	0.23
23	0.24
24	0.25
25	0.26
26	0.27
27	0.28
28	0.29

```
29          0.30

[30 rows x 21 columns]
```



[9]:
```python
import numpy as np
import pandas as pd
import pywt
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
from hurst import compute_Hc
```

```python
# ------------------------------------------------------------
# 1. Wavelet Denoising (Simple Thresholding)
# ------------------------------------------------------------
def wavelet_denoise(signal, wavelet='db4', level=2):
    """
    Perform wavelet thresholding-based denoising on 'signal'.
    This is a simple example using universal threshold on detail coeffs.
    """
    # Decompose
    coeffs = pywt.wavedec(signal, wavelet, level=level)

    # Estimate noise from the smallest detail coefficients
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745  # robust estimate

    # Universal threshold
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))

    # Threshold detail coefficients
    new_coeffs = [coeffs[0]]  # keep approximation as is
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode='soft'))

    # Reconstruct
    denoised = pywt.waverec(new_coeffs, wavelet)
    # Ensure the denoised signal has the same length as the original
    denoised = denoised[:n]
    return denoised

# ------------------------------------------------------------
# 2. Robust Feature Extraction
# ------------------------------------------------------------
def extract_features_robust(x, dt, noise_std=None, sign_threshold=0.0):
    """
    Compute a set of features from 'x', with:
      - Wavelet denoising
      - Noise-compensated RMS (if noise_std is known)
      - Robust zero crossing ignoring small sign changes
    """
    # 2.1 Wavelet Denoising
    x_denoised = wavelet_denoise(x, wavelet='db4', level=2)

    # 2.2 Convert to Series for convenience
    x_series = pd.Series(x_denoised)

    # 2.3 Now compute the "robust" features
```

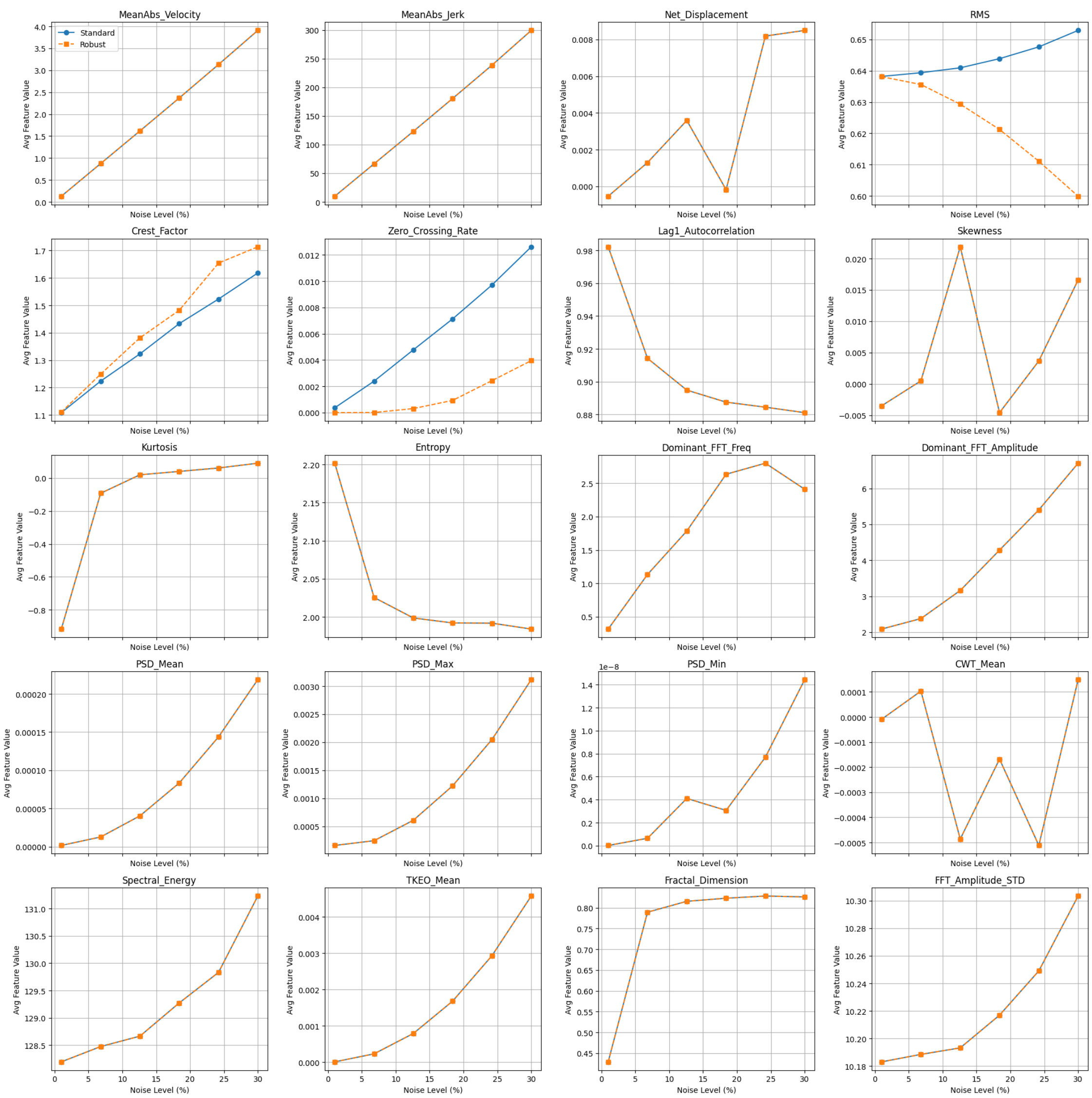
```python
    features = {}
    n = len(x_series)

    # Velocity
    vel = np.diff(x_denoised) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # Jerk
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # Net Displacement
    features['Net_Displacement'] = x_series.iloc[-1] - x_series.iloc[0]

    # RMS (Noise-Compensated)
    raw_rms = np.sqrt(np.mean(x_denoised**2))
    if noise_std is not None:
        # Subtract out noise variance: RMS_signal = sqrt(RMS^2 - noise_std^2)
        # only if RMS^2 > noise_std^2
        noise_var = noise_std**2
        if raw_rms**2 > noise_var:
            features['RMS'] = np.sqrt(raw_rms**2 - noise_var)
        else:
            features['RMS'] = 0.0
    else:
        features['RMS'] = raw_rms

    # Crest Factor
    cf_denom = features['RMS'] if features['RMS'] != 0 else np.nan
    features['Crest_Factor'] = np.max(np.abs(x_denoised)) / cf_denom if not np.
↪isnan(cf_denom) else np.nan

    # Robust Zero Crossing Rate: ignore sign changes < sign_threshold in␣
↪amplitude
    x_arr = x_denoised
    sign_changes = 0
    for i in range(n-1):
        if abs(x_arr[i]) > sign_threshold and abs(x_arr[i+1]) > sign_threshold:
            if x_arr[i]*x_arr[i+1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # Lag-1 Autocorrelation
    if n > 1:
```

```python
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # Skewness, Kurtosis
    features['Skewness'] = skew(x_arr)
    features['Kurtosis'] = kurtosis(x_arr)

    # Entropy
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)

    # FFT-based features
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan

    # Welch PSD
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # Wavelet-based
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # Spectral Energy
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # TKEO
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # Fractal Dimension (Hurst)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
```

```
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # FFT Amplitude STD
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    return features

# -----------------------------------------------------------
# 3. Comparison: Standard vs. Robust Features with Noise
# -----------------------------------------------------------
def extract_features_standard(x, dt):
    """
    The original, standard feature extraction without denoising or robust␣
↪modifications.
    (Similar to your existing function but shorter for demo.)
    """
    # We'll just call extract_features_robust but skip wavelet denoising and␣
↪noise compensation
    # for demonstration. Alternatively, you can paste your standard function␣
↪here.
    return extract_features_robust(x, dt, noise_std=None, sign_threshold=0.0)

def add_noise_to_signal(signal, noise_factor):
    """
    Add Gaussian noise scaled by noise_factor * std to the signal.
    """
    noise = np.random.normal(0, noise_factor * np.std(signal), size=signal.
↪shape)
    return signal + noise

# Generate a clean sine wave
n_rows = 100000
time = np.linspace(0, 1000, n_rows)
clean_signal = np.sin(0.01 * np.pi * time)

# We'll analyze noise levels from 1% to 30%
noise_levels = np.linspace(0.01, 0.30, 6)  # e.g., 6 steps: 1%, 6%, 11%, 16%,␣
↪21%, 26%
segment_size = 256

# Containers for results
results_standard = []
results_robust = []

for noise_factor in noise_levels:
```

```
    # Add noise
    noisy_signal = add_noise_to_signal(clean_signal, noise_factor)

    # Estimate noise std from the difference: a quick approach
    # e.g., if signal is small relative to noise, or from a quiet region
    noise_std_est = noise_factor * np.std(clean_signal)  # simplistic approach

    # Segment and compute average features
    feats_std_list = []
    feats_rob_list = []
    for start in range(0, n_rows - segment_size + 1, segment_size):
        seg = noisy_signal[start:start+segment_size]
        dt = (time[1] - time[0])  # constant time step
        seg_s = pd.Series(seg)

        # Standard features
        f_std = extract_features_standard(seg_s, dt)
        feats_std_list.append(f_std)

        # Robust features: wavelet denoising + noise-compensated RMS + robust␣
↪ZCR
        f_rob = extract_features_robust(seg_s, dt, noise_std=noise_std_est,␣
↪sign_threshold=0.02)
        feats_rob_list.append(f_rob)

    # Average over segments
    df_std = pd.DataFrame(feats_std_list).mean()
    df_std['NoiseFactor'] = noise_factor

    df_rob = pd.DataFrame(feats_rob_list).mean()
    df_rob['NoiseFactor'] = noise_factor

    results_standard.append(df_std)
    results_robust.append(df_rob)

final_std_df = pd.DataFrame(results_standard)
final_rob_df = pd.DataFrame(results_robust)

# ----------------------------------------------------------
# 4. Plot Comparison
# ----------------------------------------------------------
features_20 = [col for col in final_std_df.columns if col not in␣
↪['NoiseFactor']]
n_features = len(features_20)
n_rows_plot = 5
n_cols_plot = 4
```

```
fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(20, 20), sharex=True)
axes = axes.flatten()

for idx, feat in enumerate(features_20):
    ax = axes[idx]
    ax.plot(final_std_df['NoiseFactor']*100, final_std_df[feat], 'o-', label='Standard')
    ax.plot(final_rob_df['NoiseFactor']*100, final_rob_df[feat], 's--', label='Robust')
    ax.set_title(feat)
    ax.set_xlabel("Noise Level (%)")
    ax.set_ylabel("Avg Feature Value")
    ax.grid(True)
    if idx == 0:
        ax.legend()

plt.tight_layout()
plt.show()
```



[10]:
```python
import numpy as np
import pandas as pd
import pywt
import matplotlib.pyplot as plt
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
from hurst import compute_Hc

###############################################################################
# 1. Generate/Load a Clean Signal
###############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)
```

```python
clean_signal = np.sin(0.01 * np.pi * time)  # simple sine wave

###############################################################################
# 2. Define Noise Addition
###############################################################################
def add_noise_to_signal(signal, noise_factor):
    """Add Gaussian noise scaled by (noise_factor * std)."""
    noise = np.random.normal(0, noise_factor * np.std(signal), size=signal.
 ↪shape)
    return signal + noise

###############################################################################
# 3. Wavelet Denoising Function (with parameterization)
###############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    """
    Perform wavelet thresholding-based denoising on 'signal'.
    wavelet: wavelet family (e.g., 'db4', 'sym4', 'coif4')
    level: decomposition level
    mode: 'soft' or 'hard' thresholding
    """
    coeffs = pywt.wavedec(signal, wavelet, level=level)

    # Estimate noise from smallest detail coefficients
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))

    new_coeffs = [coeffs[0]]  # keep approximation
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))

    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

###############################################################################
# 4. Robust Feature Extraction with Param Options
###############################################################################
def extract_features_robust(
    x, dt,
    wavelet_family='db4',
    wavelet_level=2,
    sign_threshold=0.0,
    noise_compensation=False,
    noise_std=None,
    wavelet_mode='soft'
```

```
):
    """
    - wavelet_family, wavelet_level, wavelet_mode: for wavelet denoising
    - sign_threshold: amplitude threshold for ignoring sign changes
    - noise_compensation: if True, subtract noise variance from RMS
    - noise_std: needed if noise_compensation is True
    """
    # Wavelet denoise
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)

    # Convert to Series for convenience
    x_series = pd.Series(x_denoised)
    n = len(x_series)
    features = {}

    # Velocity
    vel = np.diff(x_denoised) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # Jerk
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # Net Displacement
    features['Net_Displacement'] = x_series.iloc[-1] - x_series.iloc[0]

    # RMS (optionally subtract noise variance)
    raw_rms = np.sqrt(np.mean(x_denoised**2))
    if noise_compensation and (noise_std is not None):
        noise_var = noise_std**2
        if raw_rms**2 > noise_var:
            features['RMS'] = np.sqrt(raw_rms**2 - noise_var)
        else:
            features['RMS'] = 0.0
    else:
        features['RMS'] = raw_rms

    # Crest Factor
    cf_denom = features['RMS'] if features['RMS'] != 0 else np.nan
    features['Crest_Factor'] = np.max(np.abs(x_denoised)) / cf_denom if not np.
↪isnan(cf_denom) else np.nan

    # "Robust" Zero Crossing Rate
```

```
    x_arr = x_denoised
    sign_changes = 0
    for i in range(n-1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i+1]) >␣
↪sign_threshold):
            if x_arr[i]*x_arr[i+1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # Skewness, Kurtosis
    features['Skewness'] = skew(x_arr)
    features['Kurtosis'] = kurtosis(x_arr)

    # Entropy
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)

    # FFT-based
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan

    # Welch PSD
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # Wavelet-based
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)
```

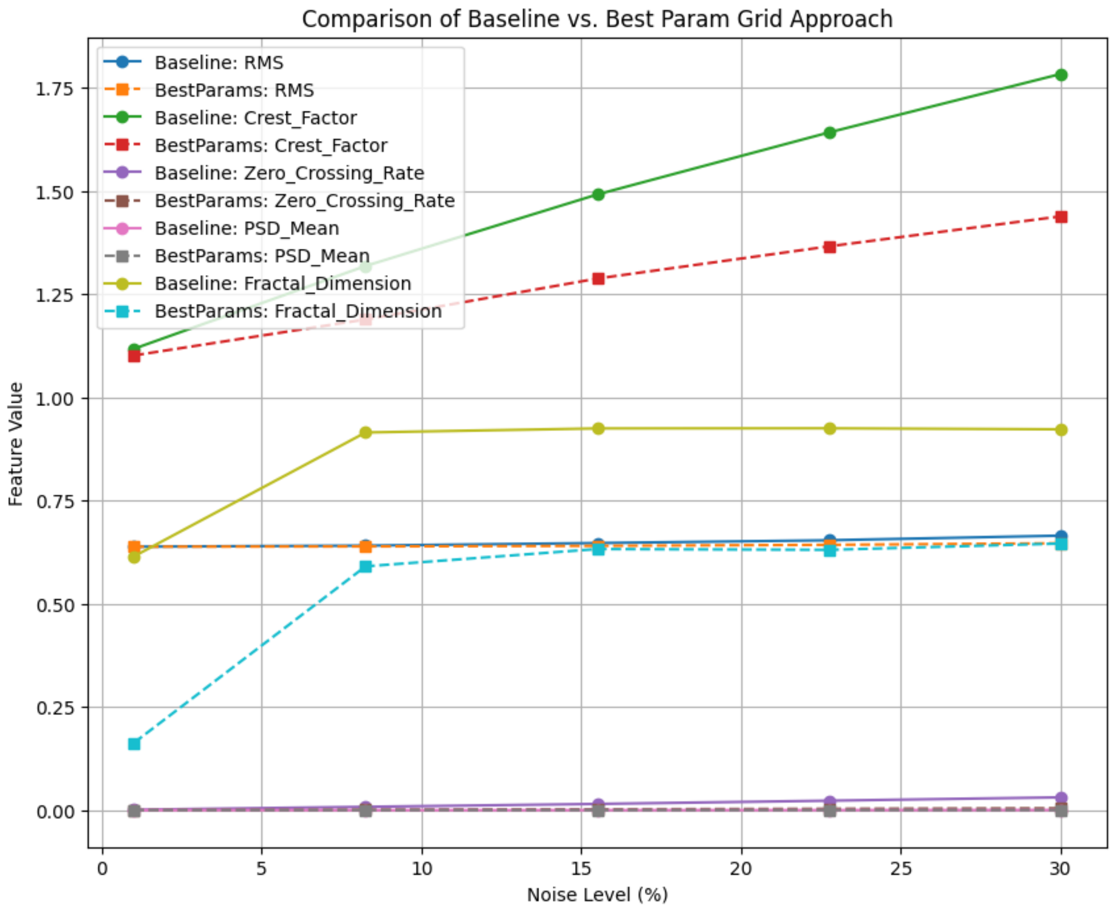
```python
    # Spectral Energy
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # TKEO
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # Fractal Dimension
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # FFT_Amplitude_STD
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    return features

##############################################################################
# 5. Parameter Grid Search
##############################################################################
param_grid = {
    'wavelet_family': ['db4', 'sym4'],  # add more families if desired
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'noise_compensation': [False, True],
    'wavelet_mode': ['soft', 'hard']
}

noise_levels = np.linspace(0.01, 0.30, 5)  # e.g., 5 points: 1%, 8%, 15%, 22%,␣
 ↪30%
segment_size = 256
dt = (time[1] - time[0])  # constant sampling interval

def evaluate_params(params):
    """
    For a given param set, loop over noise levels, extract features, measure
    how stable they are. Return a single "score" (lower is better).
    We'll measure the average slope across features as one approach.
    """
    # We'll store the average value of each feature at each noise level
    results_list = []

    for noise_factor in noise_levels:
        noisy_signal = add_noise_to_signal(clean_signal, noise_factor)
```

```
        # Estimate noise std
        noise_std_est = noise_factor * np.std(clean_signal) if␣
↪params['noise_compensation'] else None

        # Segment
        n_segs = (n_rows // segment_size)
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).mean()  # average across segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)

    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    # measure slope for each feature
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']

    # We'll do a simple linear fit for each feature vs. noiseFactor and measure␣
↪absolute slope
    # Then average across features
    slopes = []
    for feat in feature_cols:
        y = df_all[feat].values
        x = df_all['NoiseFactor'].values
        # linear fit
        coeffs = np.polyfit(x, y, 1)  # [slope, intercept]
        slope = abs(coeffs[0])
        slopes.append(slope)
    # The "score" is the average slope across all features
    return np.mean(slopes)

###############################################################################
# 6. Grid Search
###############################################################################
from itertools import product

best_score = float('inf')
```

```
best_params = None

param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

###############################################################################
# 7. Demonstration: Plot with Best Params vs. Some Baseline
###############################################################################
# We'll compare best_params to a baseline (e.g., no wavelet, no threshold, no␣
↪compensation).
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'noise_compensation': False,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    """Return a DataFrame of average feature values across noise levels for the␣
↪given params."""
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal = add_noise_to_signal(clean_signal, noise_factor)
        noise_std_est = noise_factor * np.std(clean_signal) if␣
↪params['noise_compensation'] else None

        # Segment and compute features
        feats_list = []
        n_segs = (n_rows // segment_size)
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
```

```
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)

df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot a few example features to see difference
import matplotlib.pyplot as plt

example_feats = ['RMS', 'Crest_Factor', 'Zero_Crossing_Rate', 'PSD_Mean',␣
 ↪'Fractal_Dimension']
plt.figure(figsize=(10, 8))
for feat in example_feats:
    plt.plot(df_base['NoiseFactor']*100, df_base[feat], 'o-', label=f'Baseline:␣
 ↪{feat}')
    plt.plot(df_best['NoiseFactor']*100, df_best[feat], 's--',␣
 ↪label=f'BestParams: {feat}')
plt.xlabel("Noise Level (%)")
plt.ylabel("Feature Value")
plt.title("Comparison of Baseline vs. Best Param Grid Approach")
plt.legend()
plt.grid(True)
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.0, 'noise_compensation': False, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 10.77750475140755
```



[16]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import butter, filtfilt, welch
from hurst import compute_Hc
from itertools import product

##############################################################################
# 1. Generate/Load a Clean Signal on the GPU
##############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)        # move to GPU
```

```
##############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
##############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Define a Butterworth Filter (CPU version)
##############################################################################
def apply_filter(signal, fs=100.0, cutoff_low=0.1, cutoff_high=5.0, order=4, mode='bandpass'):
    nyquist = 0.5 * fs
    if mode == 'bandpass':
        low = cutoff_low / nyquist
        high = cutoff_high / nyquist
        b, a = butter(order, [low, high], btype='band')
    elif mode == 'lowpass':
        high = cutoff_high / nyquist
        b, a = butter(order, high, btype='low')
    else:
        raise ValueError("mode must be 'bandpass' or 'lowpass'")
    filtered = filtfilt(b, a, signal)
    return filtered

##############################################################################
# 4. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
```

```python
# 5. Robust Feature Extraction with Parameter Options (CPU)
################################################################################
def extract_features_robust(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            noise_compensation=False,
                            noise_std=None,
                            wavelet_mode='soft'):
    """
    Extract 20 features from 1D signal x (pandas Series).
    Applies wavelet denoising and robust calculations.
    """
    # Wavelet denoising (convert to NumPy if needed)
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family, level=wavelet_level, mode=wavelet_mode)
    x_series = pd.Series(x_denoised)
    n = len(x_series)
    features = {}

    # 1. Mean Absolute Velocity
    vel = np.diff(x_denoised) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement
    features['Net_Displacement'] = x_series.iloc[-1] - x_series.iloc[0]

    # 4. RMS (optionally noise-compensated)
    raw_rms = np.sqrt(np.mean(x_denoised**2))
    if noise_compensation and (noise_std is not None):
        noise_var = noise_std**2
        features['RMS'] = np.sqrt(raw_rms**2 - noise_var) if raw_rms**2 > noise_var else 0.0
    else:
        features['RMS'] = raw_rms

    # 5. Crest Factor
    cf_denom = features['RMS'] if features['RMS'] != 0 else np.nan
    features['Crest_Factor'] = np.max(np.abs(x_denoised)) / cf_denom if not np.isnan(cf_denom) else np.nan
```

```
    # 6. Robust Zero Crossing Rate (ignoring small amplitude changes)
    x_arr = x_denoised
    sign_changes = 0
    for i in range(n-1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i+1]) >␣
↪sign_threshold):
            if x_arr[i] * x_arr[i+1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy (from histogram)
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)

    # 11-12. FFT-based: Dominant FFT Frequency and its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan

    # 13-15. Welch PSD: Mean, Max, Min
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)
```

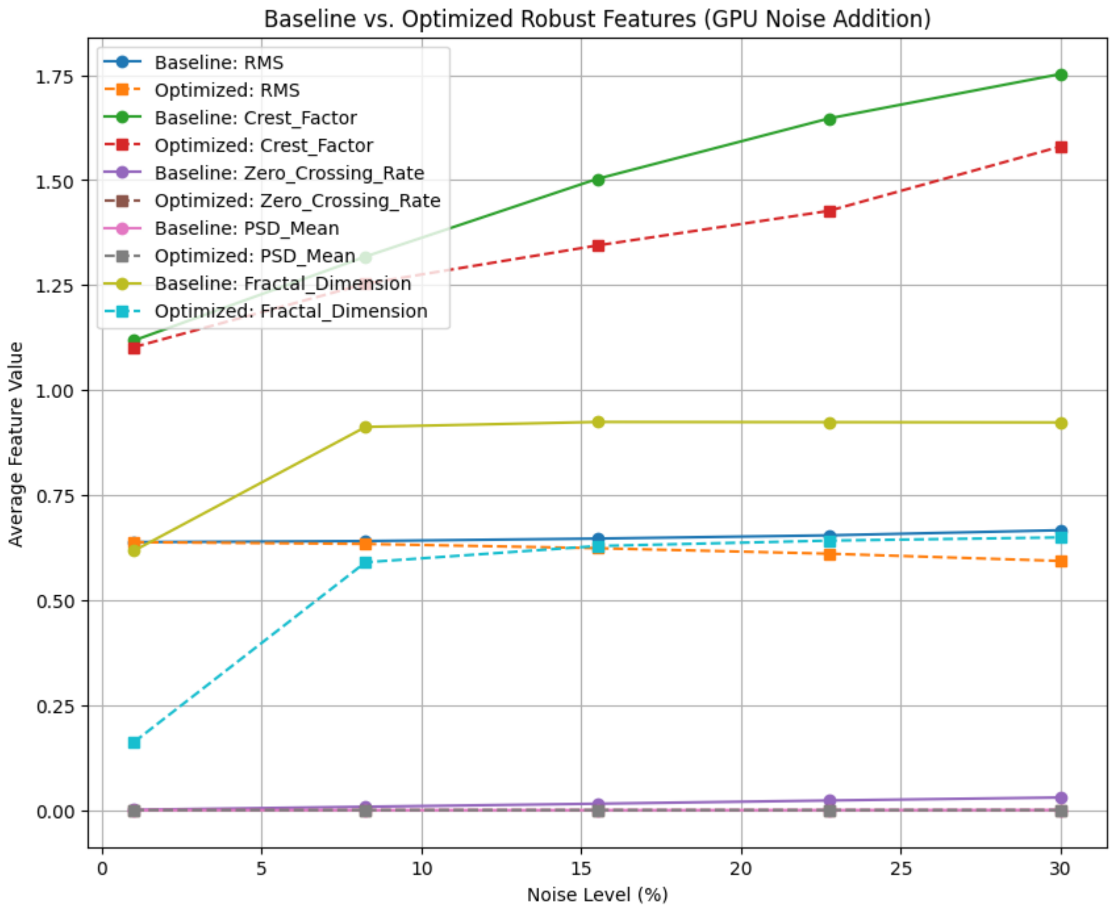
```
    # 16. Continuous Wavelet Transform (CWT) Mean (using scales 1 to 50,␣
↪'gaus1')
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension (Hurst exponent as proxy)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. STD of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    return features

##############################################################################
# 6. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
##############################################################################
# Parameter grid: try different wavelet families, levels, sign thresholds, etc.
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'noise_compensation': [False, True],
    'wavelet_mode': ['soft', 'hard']
}

# We'll test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract features
    (averaged across segments), and compute the average absolute slope
    of each feature vs. noise level. Lower slope means features are more robust.
```

```
    """
    results_list = []

    for noise_factor in noise_levels:
        # Add noise on the GPU, then bring to CPU
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)

        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None

        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).mean()  # average features over␣
↪segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)

    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']

    slopes = []
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)

# Grid search
best_score = float('inf')
best_params = None

param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
```

```
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

##############################################################################
# 7. Demonstration: Compare Baseline vs. Optimized Robust Features
##############################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'noise_compensation': False,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None

        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)
```

```python
df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot selected features to compare
example_feats = ['RMS', 'Crest_Factor', 'Zero_Crossing_Rate', 'PSD_Mean',␣
↪'Fractal_Dimension']
plt.figure(figsize=(10, 8))
for feat in example_feats:
    plt.plot(df_base['NoiseFactor']*100, df_base[feat], 'o-', label=f'Baseline:␣
↪{feat}')
    plt.plot(df_best['NoiseFactor']*100, df_best[feat], 's--',␣
↪label=f'Optimized: {feat}')
plt.xlabel("Noise Level (%)")
plt.ylabel("Average Feature Value")
plt.title("Baseline vs. Optimized Robust Features (GPU Noise Addition)")
plt.legend()
plt.grid(True)
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.02, 'noise_compensation': True, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 10.730664572611223
```



[20]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import butter, filtfilt, welch
from hurst import compute_Hc
from itertools import product

##########################################################################
# 1. Generate/Load a Clean Signal on the GPU
##########################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)       # move to GPU
```

```python
##############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
##############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Define a Butterworth Filter (CPU version)
##############################################################################
def apply_filter(signal, fs=100.0, cutoff_low=0.1, cutoff_high=5.0, order=4, mode='bandpass'):
    nyquist = 0.5 * fs
    if mode == 'bandpass':
        low = cutoff_low / nyquist
        high = cutoff_high / nyquist
        b, a = butter(order, [low, high], btype='band')
    elif mode == 'lowpass':
        high = cutoff_high / nyquist
        b, a = butter(order, high, btype='low')
    else:
        raise ValueError("mode must be 'bandpass' or 'lowpass'")
    filtered = filtfilt(b, a, signal)
    return filtered

##############################################################################
# 4. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
```

```
# 5. Robust Feature Extraction with Parameter Options (CPU)
##############################################################################
def extract_features_robust(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            noise_compensation=False,
                            noise_std=None,
                            wavelet_mode='soft'):
    """
    Extract 20 features from 1D signal x (pandas Series).
    Applies wavelet denoising and robust calculations.

    The 20 features computed are:
      1. MeanAbs_Velocity
      2. MeanAbs_Jerk
      3. Net_Displacement
      4. RMS
      5. Crest_Factor
      6. Zero_Crossing_Rate
      7. Lag1_Autocorrelation
      8. Skewness
      9. Kurtosis
      10. Entropy
      11. Dominant_FFT_Freq
      12. Dominant_FFT_Amplitude
      13. PSD_Mean
      14. PSD_Max
      15. PSD_Min
      16. CWT_Mean
      17. Spectral_Energy
      18. TKEO_Mean
      19. Fractal_Dimension
      20. FFT_Amplitude_STD
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)
    x_series = pd.Series(x_denoised)
    n = len(x_series)
    features = {}

    # 1. Mean Absolute Velocity
    vel = np.diff(x_denoised) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))

    # 2. Mean Absolute Jerk
```

```
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement
    features['Net_Displacement'] = x_series.iloc[-1] - x_series.iloc[0]

    # 4. RMS (optionally noise-compensated)
    raw_rms = np.sqrt(np.mean(x_denoised**2))
    if noise_compensation and (noise_std is not None):
        noise_var = noise_std**2
        features['RMS'] = np.sqrt(raw_rms**2 - noise_var) if raw_rms**2 > 
↪noise_var else 0.0
    else:
        features['RMS'] = raw_rms

    # 5. Crest Factor
    cf_denom = features['RMS'] if features['RMS'] != 0 else np.nan
    features['Crest_Factor'] = np.max(np.abs(x_denoised)) / cf_denom if not np.
↪isnan(cf_denom) else np.nan

    # 6. Robust Zero Crossing Rate (ignoring small amplitude changes)
    x_arr = x_denoised
    sign_changes = 0
    for i in range(n-1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i+1]) >
↪sign_threshold):
            if x_arr[i]*x_arr[i+1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy (from histogram)
```

```
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)

    # 11-12. FFT-based: Dominant FFT Frequency and its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan

    # 13-15. Welch PSD: Mean, Max, Min
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. Continuous Wavelet Transform (CWT) Mean (using scales 1 to 50,␣
↪'gaus1')
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 19. Fractal Dimension (Hurst exponent as proxy)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. Standard Deviation of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    return features
```

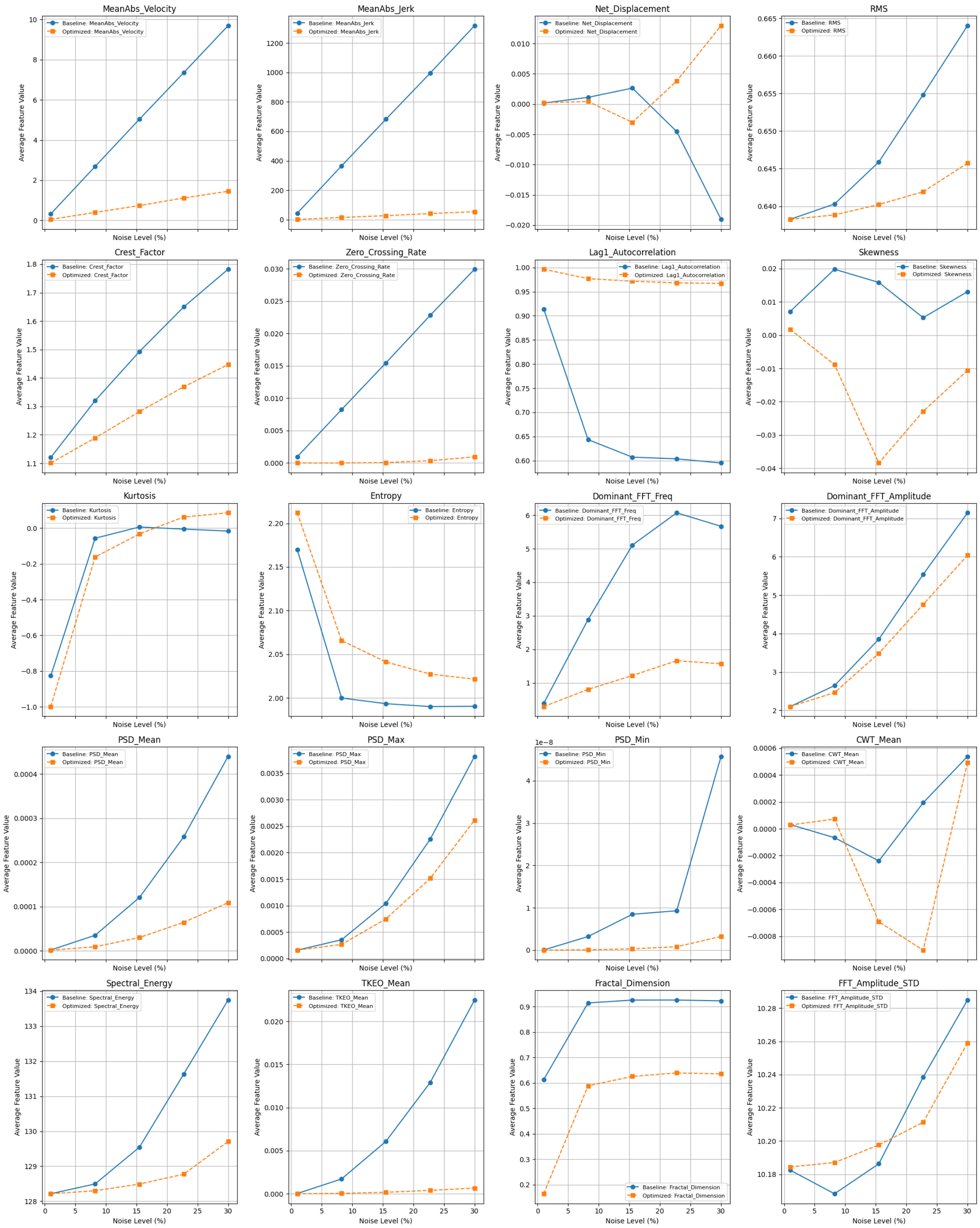
```
##############################################################################
# 6. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
##############################################################################
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'noise_compensation': [False, True],
    'wavelet_mode': ['soft', 'hard']
}

# We'll test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract features
    (averaged across segments), and compute the average absolute slope
    of each feature vs. noise level. Lower slope means features are more robust.
    """
    results_list = []

    for noise_factor in noise_levels:
        # Add noise on the GPU, then bring to CPU
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)

        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None

        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
```

```
        df_feats = pd.DataFrame(feat_values).mean()  # average features over␣
↪segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)

    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']

    slopes = []
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)

# Grid search over parameter combinations
best_score = float('inf')
best_params = None

param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

##############################################################################
# 7. Demonstration: Compare Baseline vs. Optimized Robust Features
##############################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'noise_compensation': False,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
```

```
        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None

        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)

df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot all 20 features versus noise level
all_feature_names = [col for col in df_best.columns if col != 'NoiseFactor']

n_features = len(all_feature_names)
n_rows_plot = int(np.ceil(n_features / 4))
n_cols_plot = 4

fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(20, 5*n_rows_plot),␣
↪sharex=True)
axes = axes.flatten()

for idx, feat in enumerate(all_feature_names):
    axes[idx].plot(df_base['NoiseFactor']*100, df_base[feat], 'o-',␣
↪label=f'Baseline: {feat}')
    axes[idx].plot(df_best['NoiseFactor']*100, df_best[feat], 's--',␣
↪label=f'Optimized: {feat}')
    axes[idx].set_title(feat)
    axes[idx].set_xlabel("Noise Level (%)")
    axes[idx].set_ylabel("Average Feature Value")
    axes[idx].grid(True)
    axes[idx].legend(fontsize=8)
```

```
# Remove any extra empty subplots
for j in range(idx+1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout()
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.01, 'noise_compensation': False, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 10.80245454644852
```



[28]:
```python
import cupy as cp
import numpy as np
import pandas as pd
import pywt
import matplotlib.pyplot as plt
```

```python
import seaborn as sns
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import butter, filtfilt, welch
from hurst import compute_Hc
from itertools import product

##############################################################################
# 1. Generate/Load a Clean Signal on the GPU
##############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

##############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
##############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Enhanced Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
# 4. Robust Feature Extraction with Noise Mitigation (CPU)
##############################################################################
def extract_features_robust(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
```

```
                             sign_threshold=0.0,
                             noise_compensation=False,
                             noise_std=None,
                             wavelet_mode='soft'):
    """
    Extract 20 features from 1D signal x (pandas Series).
    Applies wavelet denoising and robust calculations.
    The 20 features computed are:
      1. MeanAbs_Velocity
      2. MeanAbs_Jerk
      3. Net_Displacement
      4. RMS
      5. Crest_Factor
      6. Zero_Crossing_Rate
      7. Lag1_Autocorrelation
      8. Skewness
      9. Kurtosis
      10. Entropy
      11. Dominant_FFT_Freq
      12. Dominant_FFT_Amplitude
      13. PSD_Mean
      14. PSD_Max
      15. PSD_Min
      16. CWT_Mean
      17. Spectral_Energy
      18. TKEO_Mean
      19. Fractal_Dimension
      20. FFT_Amplitude_STD
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)
    x_series = pd.Series(x_denoised)
    n = len(x_series)
    features = {}

    # 1. Mean Absolute Velocity
    vel = np.diff(x_denoised) / dt
    features['MeanAbs_Velocity'] = np.median(np.abs(vel))  # Use median for␣
↪robustness

    # 2. Mean Absolute Jerk
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.median(np.abs(jerk))  # Use median for␣
↪robustness
    else:
```

```
        features['MeanAbs_Jerk'] = np.nan

    # 3. Net Displacement
    features['Net_Displacement'] = x_series.iloc[-1] - x_series.iloc[0]

    # 4. RMS (optionally noise-compensated)
    raw_rms = np.sqrt(np.mean(x_denoised**2))
    if noise_compensation and (noise_std is not None):
        noise_var = noise_std**2
        features['RMS'] = np.sqrt(raw_rms**2 - noise_var) if raw_rms**2 >␣
↪noise_var else 0.0
    else:
        features['RMS'] = raw_rms

    # 5. Crest Factor
    cf_denom = features['RMS'] if features['RMS'] != 0 else np.nan
    features['Crest_Factor'] = np.max(np.abs(x_denoised)) / cf_denom if not np.
↪isnan(cf_denom) else np.nan

    # 6. Robust Zero Crossing Rate (ignoring small amplitude changes)
    x_arr = x_denoised
    sign_changes = 0
    for i in range(n-1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i+1]) >␣
↪sign_threshold):
            if x_arr[i]*x_arr[i+1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 7. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0,1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 8. Skewness
    features['Skewness'] = skew(x_arr)

    # 9. Kurtosis
    features['Kurtosis'] = kurtosis(x_arr)

    # 10. Entropy (from histogram)
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)
```

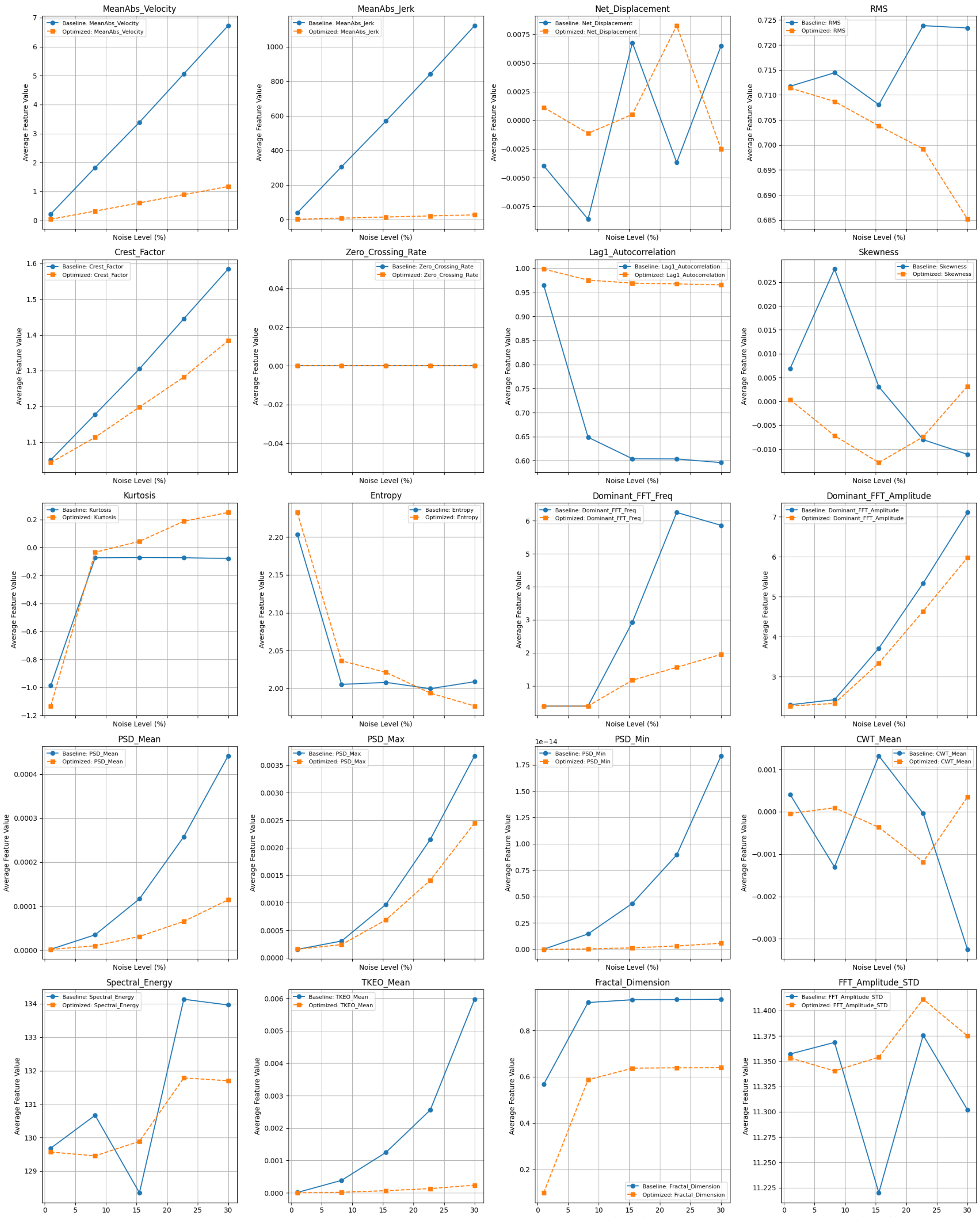
```
    # 11-12. FFT-based: Dominant FFT Frequency and its Amplitude
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan

    # 13-15. Welch PSD: Mean, Max, Min
    freqs_welch, psd_vals = welch(x_arr, fs=1/dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)

    # 16. Continuous Wavelet Transform (CWT) Mean (using scales 1 to 50,␣
↪'gaus1')
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)

    # 17. Spectral Energy
    features['Spectral_Energy'] = np.sum(x_arr**2)

    # 18. Mean TKEO
    tkeo = x_arr[1:-1]**2 - x_arr[:-2]*x_arr[2:]
    features['TKEO_Mean'] = np.median(tkeo) if len(tkeo) > 0 else np.nan  # Use␣
↪median for robustness

    # 19. Fractal Dimension (Hurst exponent as proxy)
    try:
        H, c, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H

    # 20. Standard Deviation of FFT Amplitude
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    return features

###############################################################################
# 5. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
###############################################################################
```

```python
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'noise_compensation': [False, True],
    'wavelet_mode': ['soft', 'hard']
}

# We'll test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract features
    (averaged across segments), and compute the average absolute slope
    of each feature vs. noise level. Lower slope means features are more robust.
    """
    results_list = []
    for noise_factor in noise_levels:
        # Add noise on the GPU, then bring to CPU
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None
        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).median()  # Use median for␣
↪robustness
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']
    slopes = []
```

```
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)

# Grid search over parameter combinations
best_score = float('inf')
best_params = None
param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

##############################################################################
# 6. Demonstration: Compare Baseline vs. Optimized Robust Features
##############################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'noise_compensation': False,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        noise_std_est = noise_factor * np.std(cp.asnumpy(clean_signal)) if␣
↪params['noise_compensation'] else None
        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start+segment_size]
            feat = extract_features_robust(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
```

```
                sign_threshold=params['sign_threshold'],
                noise_compensation=params['noise_compensation'],
                noise_std=noise_std_est,
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).median()  # Use median for robustness
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)

df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot all 20 features versus noise level
all_feature_names = [col for col in df_best.columns if col != 'NoiseFactor']
n_features = len(all_feature_names)
n_rows_plot = int(np.ceil(n_features / 4))
n_cols_plot = 4
fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(20, 5*n_rows_plot), sharex=True)
axes = axes.flatten()
for idx, feat in enumerate(all_feature_names):
    axes[idx].plot(df_base['NoiseFactor']*100, df_base[feat], 'o-', label=f'Baseline: {feat}')
    axes[idx].plot(df_best['NoiseFactor']*100, df_best[feat], 's--', label=f'Optimized: {feat}')
    axes[idx].set_title(feat)
    axes[idx].set_xlabel("Noise Level (%)")
    axes[idx].set_ylabel("Average Feature Value")
    axes[idx].grid(True)
    axes[idx].legend(fontsize=8)

# Remove any extra empty subplots
for j in range(idx+1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.02, 'noise_compensation': True, 'wavelet_mode': 'hard'}
Best (lowest) average slope score: 5.87652098221145
```



[29]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.signal import welch
```

```python
from itertools import product

##############################################################################
# 1. Generate/Load a Clean Signal on the GPU
##############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

##############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
##############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
 ↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
# 4. Extract Only Robust Features (CPU)
##############################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract only the robust features:
```

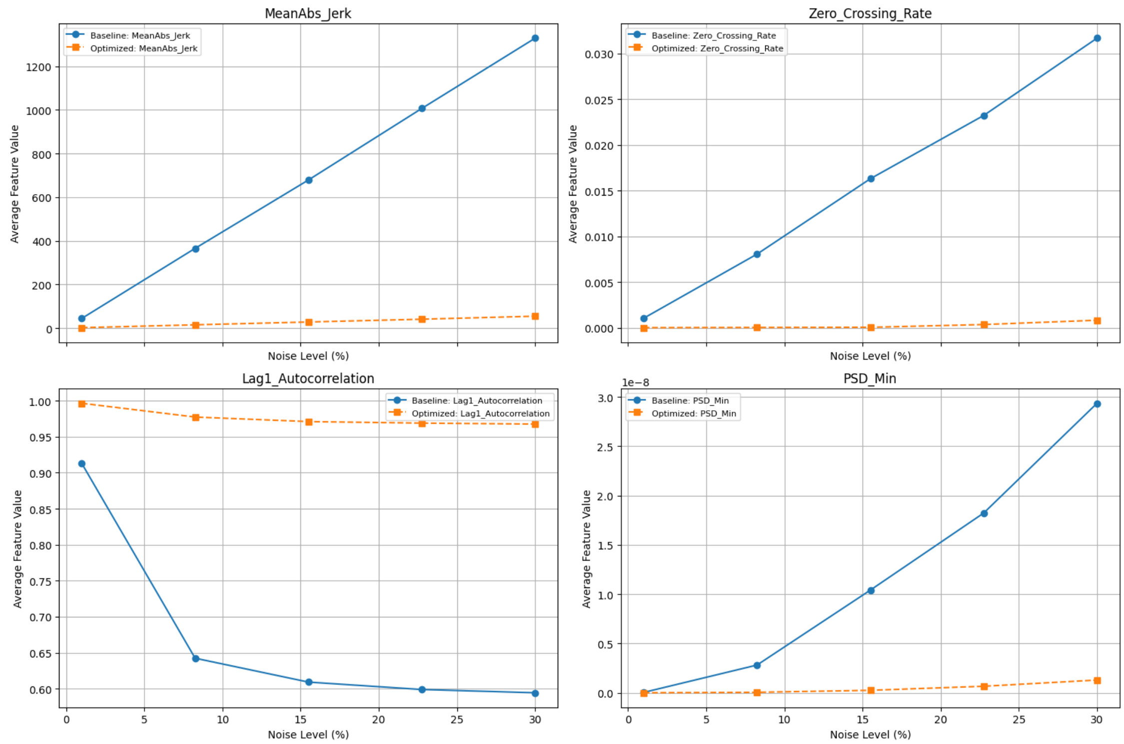
```
      1. Mean Absolute Jerk
      2. Zero Crossing Rate
      3. Lag-1 Autocorrelation
      4. PSD Min
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)
    x_arr = x_denoised
    n = len(x_arr)
    features = {}

    # 1. Mean Absolute Jerk
    vel = np.diff(x_denoised) / dt
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 2. Zero Crossing Rate (ignoring small amplitude changes)
    sign_changes = 0
    for i in range(n - 1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) >␣
↪sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 3. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 4. PSD Min (from Welch method)
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Min'] = np.min(psd_vals)

    return features

##############################################################################
# 5. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
##############################################################################
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
```

```python
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'wavelet_mode': ['soft', 'hard']
}

# Test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract robust features
    (averaged across segments), and compute the average absolute slope of each
    feature vs. noise level. Lower slope means features are more robust.
    """
    results_list = []
    for noise_factor in noise_levels:
        # Add noise on the GPU, then bring to CPU
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).mean()  # average features over␣
↪segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']
    slopes = []
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)
```

```python
# Grid search over parameter combinations
best_score = float('inf')
best_params = None
param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

##############################################################################
# 6. Demonstration: Compare Baseline vs. Optimized Robust Features
##############################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)

df_best = get_feature_curves(best_params)
```

```python
df_base = get_feature_curves(baseline_params)

# Plot the four robust features versus noise level
all_feature_names = [col for col in df_best.columns if col != 'NoiseFactor']
n_features = len(all_feature_names)
n_rows_plot = int(np.ceil(n_features / 2))
n_cols_plot = 2
fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(15, 5 *␣
 ↪n_rows_plot), sharex=True)
axes = axes.flatten()
for idx, feat in enumerate(all_feature_names):
    axes[idx].plot(df_base['NoiseFactor'] * 100, df_base[feat], 'o-',␣
 ↪label=f'Baseline: {feat}')
    axes[idx].plot(df_best['NoiseFactor'] * 100, df_best[feat], 's--',␣
 ↪label=f'Optimized: {feat}')
    axes[idx].set_title(feat)
    axes[idx].set_xlabel("Noise Level (%)")
    axes[idx].set_ylabel("Average Feature Value")
    axes[idx].grid(True)
    axes[idx].legend(fontsize=8)

# Remove any extra empty subplots
for j in range(idx + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.01, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 44.91495464180641
```



[30]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.signal import welch
from itertools import product

###############################################################################
# 1. Generate/Load a Clean Signal on the GPU
###############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

###############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
###############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
↪array 'signal'.
    """
```

```
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
# 4. Extract Only Robust Features (CPU)
##############################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract only the robust features:
      1. Mean Absolute Jerk
      2. Zero Crossing Rate
      3. Lag-1 Autocorrelation
      4. PSD Min
      5. TKEO Mean
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
 ↪level=wavelet_level, mode=wavelet_mode)
    x_arr = x_denoised
    n = len(x_arr)
    features = {}

    # 1. Mean Absolute Jerk
    vel = np.diff(x_denoised) / dt
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
```

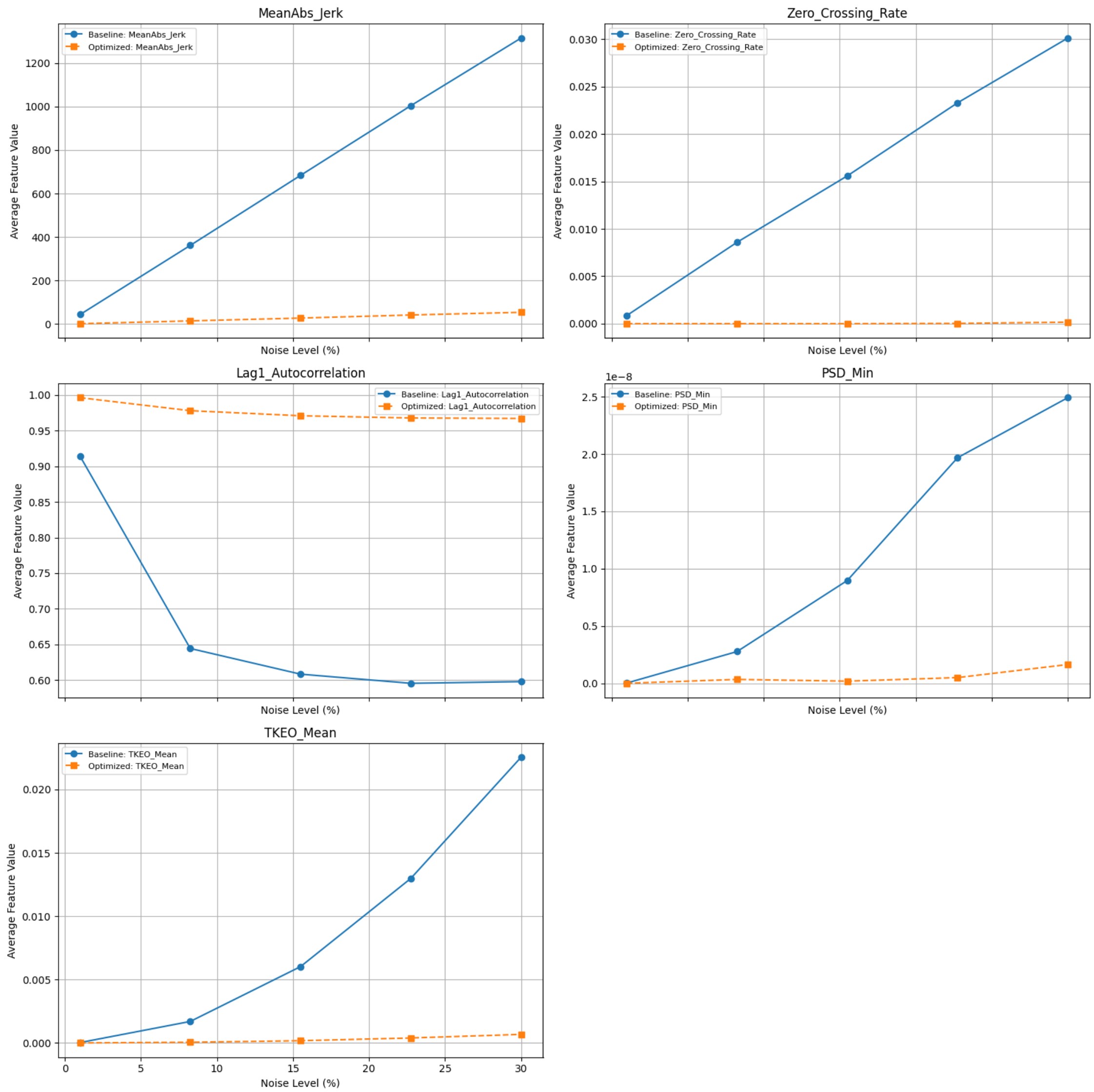
```
    else:
        features['MeanAbs_Jerk'] = np.nan

    # 2. Zero Crossing Rate (ignoring small amplitude changes)
    sign_changes = 0
    for i in range(n - 1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) >␣
↪sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 3. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    # 4. PSD Min (from Welch method)
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Min'] = np.min(psd_vals)

    # 5. TKEO Mean (Teager-Kaiser Energy Operator Mean)
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    return features

###############################################################################
# 5. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
###############################################################################
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'wavelet_mode': ['soft', 'hard']
}

# Test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract robust features
```

```
    (averaged across segments), and compute the average absolute slope of each
    feature vs. noise level. Lower slope means features are more robust.
    """
    results_list = []
    for noise_factor in noise_levels:
        # Add noise on the GPU, then bring to CPU
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).mean()  # average features over␣
 ↪segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']
    slopes = []
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)

# Grid search over parameter combinations
best_score = float('inf')
best_params = None
param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)
```

```
################################################################################
# 6. Demonstration: Compare Baseline vs. Optimized Robust Features
################################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)

df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot the five robust features versus noise level
all_feature_names = [col for col in df_best.columns if col != 'NoiseFactor']
n_features = len(all_feature_names)
n_rows_plot = int(np.ceil(n_features / 2))
n_cols_plot = 2
fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(15, 5 *␣
 ↪n_rows_plot), sharex=True)
axes = axes.flatten()
for idx, feat in enumerate(all_feature_names):
    axes[idx].plot(df_base['NoiseFactor'] * 100, df_base[feat], 'o-',␣
 ↪label=f'Baseline: {feat}')
```

```
    axes[idx].plot(df_best['NoiseFactor'] * 100, df_best[feat], 's--',␣
 ↪label=f'Optimized: {feat}')
    axes[idx].set_title(feat)
    axes[idx].set_xlabel("Noise Level (%)")
    axes[idx].set_ylabel("Average Feature Value")
    axes[idx].grid(True)
    axes[idx].legend(fontsize=8)

# Remove any extra empty subplots
for j in range(idx + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.02, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 35.55740016488779
```



[31]:
```python
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import welch
from itertools import product

##############################################################################
# 1. Generate/Load a Clean Signal on the GPU
##############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
```

```python
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

###############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
###############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
 ↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

###############################################################################
# 3. Wavelet Denoising Function (CPU)
###############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

###############################################################################
# 4. Extract 50 Robust Features (CPU)
###############################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract 50 robust features from 1D signal x.
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
 ↪level=wavelet_level, mode=wavelet_mode)
    x_arr = x_denoised
    n = len(x_arr)
    features = {}
```

```python
    # 1-5: Basic Statistical Features
    features['Mean'] = np.mean(x_arr)
    features['Median'] = np.median(x_arr)
    features['StdDev'] = np.std(x_arr)
    features['Min'] = np.min(x_arr)
    features['Max'] = np.max(x_arr)

    # 6-10: Velocity and Jerk Features
    vel = np.diff(x_arr) / dt
    features['MeanAbs_Velocity'] = np.mean(np.abs(vel))
    features['MedianAbs_Velocity'] = np.median(np.abs(vel))
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MeanAbs_Jerk'] = np.mean(np.abs(jerk))
        features['MedianAbs_Jerk'] = np.median(np.abs(jerk))
    else:
        features['MeanAbs_Jerk'] = np.nan
        features['MedianAbs_Jerk'] = np.nan

    # 11-15: Zero Crossing Rate and Related Features
    sign_changes = 0
    for i in range(n - 1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) > sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Zero_Crossing_Rate'] = sign_changes / n
    features['Sign_Changes'] = sign_changes

    # 16-20: Autocorrelation Features
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr
    features['Lag2_Autocorrelation'] = np.corrcoef(x_arr[:-2], x_arr[2:])[0, 1] if n > 2 else np.nan

    # 21-25: Spectral Features (PSD)
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Mean'] = np.mean(psd_vals)
    features['PSD_Max'] = np.max(psd_vals)
    features['PSD_Min'] = np.min(psd_vals)
    features['PSD_Median'] = np.median(psd_vals)

    # 26-30: FFT-Based Features
```

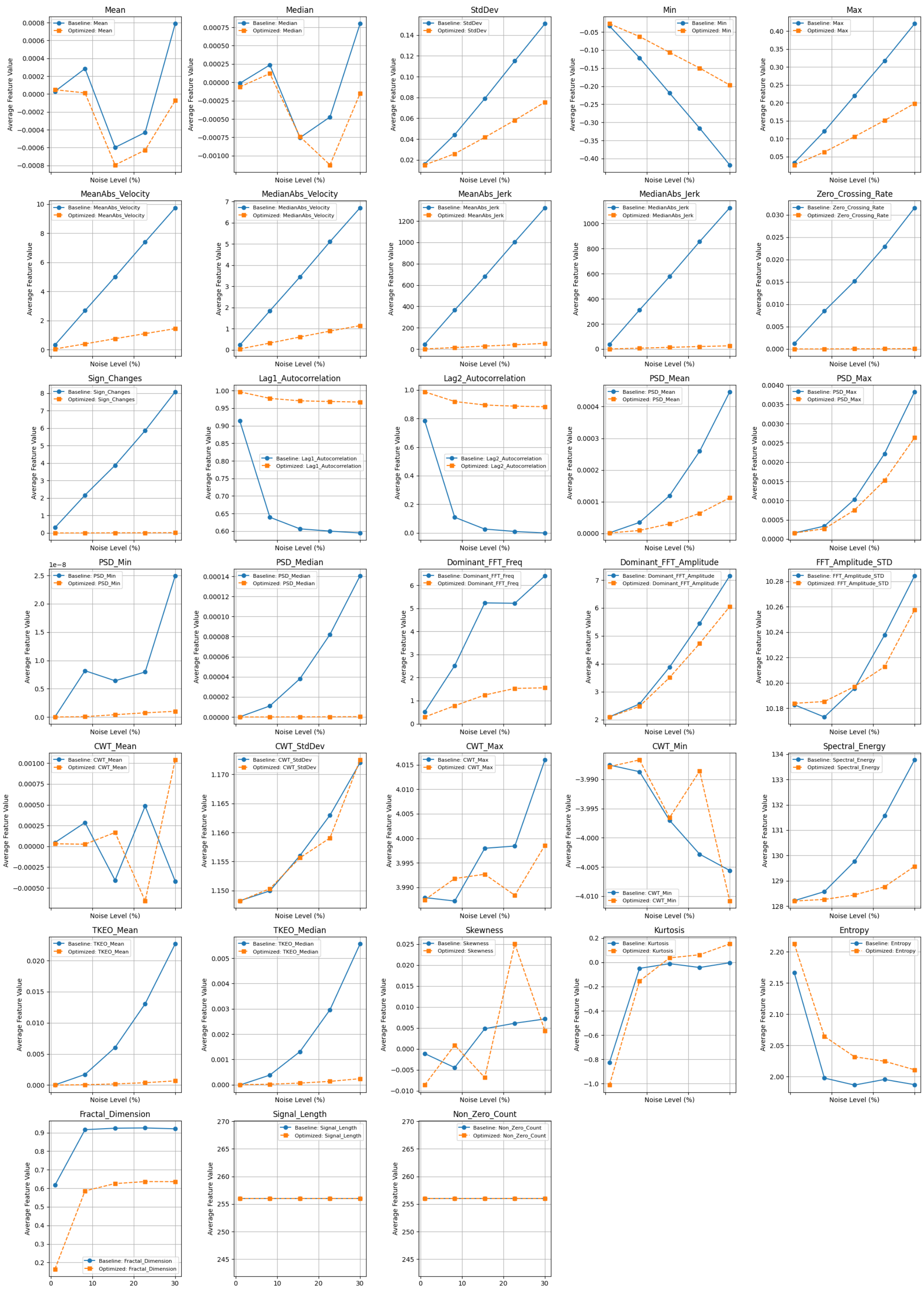
```
    fft_vals = np.fft.fft(x_arr)
    fft_freqs = np.fft.fftfreq(n, d=dt)
    fft_mag = np.abs(fft_vals)
    if n > 1:
        idx = np.argmax(fft_mag[1:]) + 1
        features['Dominant_FFT_Freq'] = fft_freqs[idx]
        features['Dominant_FFT_Amplitude'] = fft_mag[idx]
    else:
        features['Dominant_FFT_Freq'] = np.nan
        features['Dominant_FFT_Amplitude'] = np.nan
    features['FFT_Amplitude_STD'] = np.std(fft_mag)

    # 31-35: Wavelet Transform Features
    scales = np.arange(1, 50)
    coeffs, _ = pywt.cwt(x_arr, scales, 'gaus1')
    features['CWT_Mean'] = np.mean(coeffs)
    features['CWT_StdDev'] = np.std(coeffs)
    features['CWT_Max'] = np.max(coeffs)
    features['CWT_Min'] = np.min(coeffs)

    # 36-40: Energy-Based Features
    features['Spectral_Energy'] = np.sum(x_arr**2)
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan
    features['TKEO_Median'] = np.median(tkeo) if len(tkeo) > 0 else np.nan

    # 41-45: Higher-Order Statistical Features
    features['Skewness'] = skew(x_arr)
    features['Kurtosis'] = kurtosis(x_arr)
    hist, _ = np.histogram(x_arr, bins=10, density=True)
    hist += 1e-8
    features['Entropy'] = entropy(hist)

    # 46-50: Fractal and Other Features
    try:
        H, _, _ = compute_Hc(x_arr, kind='change', simplified=True)
    except FloatingPointError:
        H = np.nan
    features['Fractal_Dimension'] = H
    features['Signal_Length'] = n
    features['Non_Zero_Count'] = np.count_nonzero(x_arr)

    return features

###############################################################################
# 5. Parameter Grid Search to Optimize Noise Robustness (CPU-Based Evaluation)
###############################################################################
```

```
param_grid = {
    'wavelet_family': ['db4', 'sym4'],
    'wavelet_level': [1, 2, 3],
    'sign_threshold': [0.0, 0.01, 0.02],
    'wavelet_mode': ['soft', 'hard']
}

# Test noise levels from 1% to 30% (5 points)
noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def evaluate_params(params):
    """
    For a given parameter set, loop over noise levels, extract robust features
    (averaged across segments), and compute the average absolute slope of each
    feature vs. noise level. Lower slope means features are more robust.
    """
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feat_values = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feat_values.append(feat)
        df_feats = pd.DataFrame(feat_values).mean()  # average features over␣
↪segments
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    df_all = pd.DataFrame(results_list).reset_index(drop=True)
    feature_cols = [c for c in df_all.columns if c != 'NoiseFactor']
    slopes = []
    for feat in feature_cols:
        y_vals = df_all[feat].values
        x_vals = df_all['NoiseFactor'].values
        coeffs = np.polyfit(x_vals, y_vals, 1)
        slopes.append(abs(coeffs[0]))
    return np.mean(slopes)
```

```python
# Grid search over parameter combinations
best_score = float('inf')
best_params = None
param_keys = list(param_grid.keys())
for combo in product(*param_grid.values()):
    params = dict(zip(param_keys, combo))
    score = evaluate_params(params)
    if score < best_score:
        best_score = score
        best_params = params

print("Best parameter set found:", best_params)
print("Best (lowest) average slope score:", best_score)

###############################################################################
# 6. Demonstration: Compare Baseline vs. Optimized Robust Features
###############################################################################
baseline_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 1,
    'sign_threshold': 0.0,
    'wavelet_mode': 'soft'
}

def get_feature_curves(params):
    results_list = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        feats_list = []
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=params['wavelet_family'],
                wavelet_level=params['wavelet_level'],
                sign_threshold=params['sign_threshold'],
                wavelet_mode=params['wavelet_mode']
            )
            feats_list.append(feat)
        df_feats = pd.DataFrame(feats_list).mean()
        df_feats['NoiseFactor'] = noise_factor
        results_list.append(df_feats)
    return pd.DataFrame(results_list)
```

```
df_best = get_feature_curves(best_params)
df_base = get_feature_curves(baseline_params)

# Plot the 50 robust features versus noise level
all_feature_names = [col for col in df_best.columns if col != 'NoiseFactor']
n_features = len(all_feature_names)
n_rows_plot = int(np.ceil(n_features / 5))
n_cols_plot = 5
fig, axes = plt.subplots(n_rows_plot, n_cols_plot, figsize=(20, 4 *␣
↪n_rows_plot), sharex=True)
axes = axes.flatten()
for idx, feat in enumerate(all_feature_names):
    axes[idx].plot(df_base['NoiseFactor'] * 100, df_base[feat], 'o-',␣
↪label=f'Baseline: {feat}')
    axes[idx].plot(df_best['NoiseFactor'] * 100, df_best[feat], 's--',␣
↪label=f'Optimized: {feat}')
    axes[idx].set_title(feat)
    axes[idx].set_xlabel("Noise Level (%)")
    axes[idx].set_ylabel("Average Feature Value")
    axes[idx].grid(True)
    axes[idx].legend(fontsize=8)

# Remove any extra empty subplots
for j in range(idx + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

```
Best parameter set found: {'wavelet_family': 'sym4', 'wavelet_level': 3,
'sign_threshold': 0.02, 'wavelet_mode': 'soft'}
Best (lowest) average slope score: 9.268401593792307
```


Mean
Baseline: Mean
Optimized: Mean
Median
Baseline: Median
Optimized: Median
StdDev
Baseline: StdDev
Optimized: StdDev
Min
Baseline: Min
Optimized: Min
Max
Baseline: Max
Optimized: Max
MeanAbs_Velocity
Baseline: MeanAbs_Velocity
Optimized: MeanAbs_Velocity
MedianAbs_Velocity
Baseline: MedianAbs_Velocity
Optimized: MedianAbs_Velocity
MeanAbs_Jerk
Baseline: MeanAbs_Jerk
Optimized: MeanAbs_Jerk
MedianAbs_Jerk
Baseline: MedianAbs_Jerk
Optimized: MedianAbs_Jerk
Zero_Crossing_Rate
Baseline: Zero_Crossing_Rate
Optimized: Zero_Crossing_Rate
Sign_Changes
Baseline: Sign_Changes
Optimized: Sign_Changes
Lag1_Autocorrelation
Baseline: Lag1_Autocorrelation
Optimized: Lag1_Autocorrelation
Lag2_Autocorrelation
Baseline: Lag2_Autocorrelation
Optimized: Lag2_Autocorrelation
PSD_Mean
Baseline: PSD_Mean
Optimized: PSD_Mean
PSD_Max
Baseline: PSD_Max
Optimized: PSD_Max
PSD_Min
Baseline: PSD_Min
Optimized: PSD_Min
PSD_Median
Baseline: PSD_Median
Optimized: PSD_Median
Dominant_FFT_Freq
Baseline: Dominant_FFT_Freq
Optimized: Dominant_FFT_Freq
Dominant_FFT_Amplitude
Baseline: Dominant_FFT_Amplitude
Optimized: Dominant_FFT_Amplitude
FFT_Amplitude_STD
Baseline: FFT_Amplitude_STD
Optimized: FFT_Amplitude_STD
CWT_Mean
Baseline: CWT_Mean
Optimized: CWT_Mean
CWT_StdDev
Baseline: CWT_StdDev
Optimized: CWT_StdDev
CWT_Max
Baseline: CWT_Max
Optimized: CWT_Max
CWT_Min
Baseline: CWT_Min
Optimized: CWT_Min
Spectral_Energy
Baseline: Spectral_Energy
Optimized: Spectral_Energy
TKEO_Mean
Baseline: TKEO_Mean
Optimized: TKEO_Mean
TKEO_Median
Baseline: TKEO_Median
Optimized: TKEO_Median
Skewness
Baseline: Skewness
Optimized: Skewness
Kurtosis
Baseline: Kurtosis
Optimized: Kurtosis
Entropy
Baseline: Entropy
Optimized: Entropy
Fractal_Dimension
Baseline: Fractal_Dimension
Optimized: Fractal_Dimension
Signal_Length
Baseline: Signal_Length
Optimized: Signal_Length
Non_Zero_Count
Baseline: Non_Zero_Count
Optimized: Non_Zero_Count
Average Feature Value
Noise Level (%)

[32]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.stats import entropy
from scipy.signal import welch
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

###############################################################################
# 1. Generate/Load a Clean Signal on the GPU
###############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)   # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)   # move to GPU

###############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
###############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
 ↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

###############################################################################
# 3. Wavelet Denoising Function (CPU)
###############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

###############################################################################
# 4. Extract Only Robust Features (CPU)
```

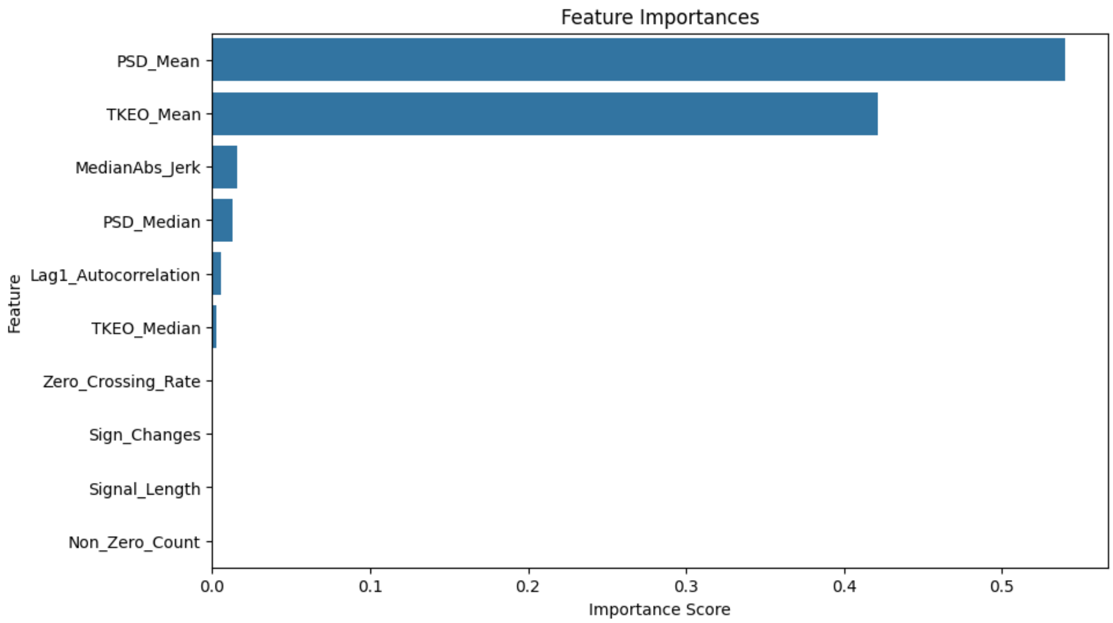
```
##############################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract only the robust features:
      1. Signal_Length
      2. Non_Zero_Count
      3. TKEO_Mean
      4. TKEO_Median
      5. PSD_Median
      6. PSD_Mean
      7. Sign_Changes
      8. Zero_Crossing_Rate
      9. MedianAbs_Jerk
      10. Lag1_Autocorrelation
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
 ↪level=wavelet_level, mode=wavelet_mode)
    x_arr = x_denoised
    n = len(x_arr)
    features = {}

    # 1. Signal Length
    features['Signal_Length'] = n

    # 2. Non-Zero Count
    features['Non_Zero_Count'] = np.count_nonzero(x_arr)

    # 3. TKEO Mean
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 4. TKEO Median
    features['TKEO_Median'] = np.median(tkeo) if len(tkeo) > 0 else np.nan

    # 5-6. PSD Median and Mean
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Median'] = np.median(psd_vals)
    features['PSD_Mean'] = np.mean(psd_vals)

    # 7. Sign Changes
    sign_changes = 0
    for i in range(n - 1):
```

```
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) >␣
↪sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Sign_Changes'] = sign_changes

    # 8. Zero Crossing Rate
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 9. Median Absolute Jerk
    vel = np.diff(x_arr) / dt
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MedianAbs_Jerk'] = np.median(np.abs(jerk))
    else:
        features['MedianAbs_Jerk'] = np.nan

    # 10. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    return features

##############################################################################
# 5. Prepare Dataset for Training
##############################################################################
best_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 2,
    'sign_threshold': 0.01,
    'wavelet_mode': 'soft'
}

noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def prepare_dataset(noise_levels, segment_size, best_params):
    dataset = []
    labels = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
```

```
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=best_params['wavelet_family'],
                wavelet_level=best_params['wavelet_level'],
                sign_threshold=best_params['sign_threshold'],
                wavelet_mode=best_params['wavelet_mode']
            )
            dataset.append(feat)
            labels.append(noise_factor)  # Use noise factor as the target␣
↪variable
    df_dataset = pd.DataFrame(dataset)
    df_labels = pd.Series(labels, name='NoiseFactor')
    return df_dataset, df_labels

df_dataset, df_labels = prepare_dataset(noise_levels, segment_size, best_params)

##############################################################################
# 6. Train a Machine Learning Model
##############################################################################
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df_dataset, df_labels,␣
↪test_size=0.2, random_state=42)

# Train a Random Forest Regressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error on Test Set:", mse)

##############################################################################
# 7. Feature Importance Analysis
##############################################################################
feature_importances = pd.Series(model.feature_importances_, index=df_dataset.
↪columns)
feature_importances.sort_values(ascending=False, inplace=True)

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances.values, y=feature_importances.index)
plt.title("Feature Importances")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
```

```
plt.show()
```

```
Mean Squared Error on Test Set: 0.00024251259294871775
```



[33]:
```
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.signal import welch
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns

################################################################################
# 1. Generate/Load a Clean Signal on the GPU
################################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

################################################################################
# 2. Define Noise Addition Using GPU (CuPy)
################################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
```

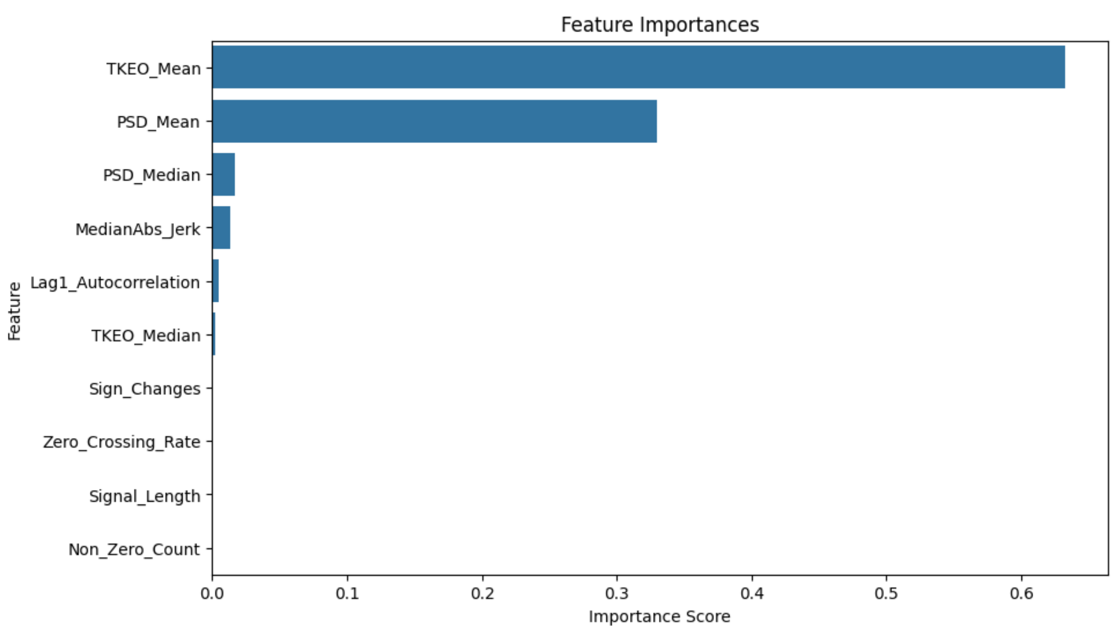
```
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]

##############################################################################
# 4. Extract Only Robust Features (CPU)
##############################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract only the robust features:
      1. Signal_Length
      2. Non_Zero_Count
      3. TKEO_Mean
      4. TKEO_Median
      5. PSD_Median
      6. PSD_Mean
      7. Sign_Changes
      8. Zero_Crossing_Rate
      9. MedianAbs_Jerk
      10. Lag1_Autocorrelation
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)
```

```
    x_arr = x_denoised
    n = len(x_arr)
    features = {}

    # 1. Signal Length
    features['Signal_Length'] = n

    # 2. Non-Zero Count
    features['Non_Zero_Count'] = np.count_nonzero(x_arr)

    # 3. TKEO Mean
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 4. TKEO Median
    features['TKEO_Median'] = np.median(tkeo) if len(tkeo) > 0 else np.nan

    # 5-6. PSD Median and Mean
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Median'] = np.median(psd_vals)
    features['PSD_Mean'] = np.mean(psd_vals)

    # 7. Sign Changes
    sign_changes = 0
    for i in range(n - 1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) >
↪sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Sign_Changes'] = sign_changes

    # 8. Zero Crossing Rate
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 9. Median Absolute Jerk
    vel = np.diff(x_arr) / dt
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MedianAbs_Jerk'] = np.median(np.abs(jerk))
    else:
        features['MedianAbs_Jerk'] = np.nan

    # 10. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
```

```
    features['Lag1_Autocorrelation'] = autocorr

    return features

###############################################################################
# 5. Prepare Dataset for Training
###############################################################################
best_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 2,
    'sign_threshold': 0.01,
    'wavelet_mode': 'soft'
}

noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def prepare_dataset(noise_levels, segment_size, best_params):
    dataset = []
    labels = []
    for noise_factor in noise_levels:
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=best_params['wavelet_family'],
                wavelet_level=best_params['wavelet_level'],
                sign_threshold=best_params['sign_threshold'],
                wavelet_mode=best_params['wavelet_mode']
            )
            dataset.append(feat)
            labels.append(noise_factor)  # Use noise factor as the target␣
↪variable
    df_dataset = pd.DataFrame(dataset)
    df_labels = pd.Series(labels, name='NoiseFactor')
    return df_dataset, df_labels

df_dataset, df_labels = prepare_dataset(noise_levels, segment_size, best_params)

###############################################################################
# 6. Train a Machine Learning Model
###############################################################################
# Split into training and testing sets
```

```python
X_train, X_test, y_train, y_test = train_test_split(df_dataset, df_labels,␣
↪test_size=0.2, random_state=42)

# Train a Random Forest Regressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error on Test Set:", mse)

##############################################################################
# 7. Feature Importance Analysis
##############################################################################
feature_importances = pd.Series(model.feature_importances_, index=df_dataset.
↪columns)
feature_importances.sort_values(ascending=False, inplace=True)

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances.values, y=feature_importances.index)
plt.title("Feature Importances")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

```
Mean Squared Error on Test Set: 0.000262422998397436
```



[34]:

```python
import cupy as cp
import numpy as np
import pandas as pd
import pywt
from scipy.signal import welch
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import seaborn as sns

##############################################################################
# 1. Generate/Load a Clean Signal on the GPU
##############################################################################
n_rows = 100000
time = np.linspace(0, 1000, n_rows)  # create on CPU
clean_signal_cpu = np.sin(0.01 * np.pi * time)  # clean sine wave (CPU)
clean_signal = cp.asarray(clean_signal_cpu)  # move to GPU

##############################################################################
# 2. Define Noise Addition Using GPU (CuPy)
##############################################################################
def add_noise_to_signal_gpu(signal, noise_factor):
    """
    Add Gaussian noise (using CuPy) scaled by (noise_factor * std) to the GPU␣
↪array 'signal'.
    """
    std = cp.std(signal)
    noise = cp.random.normal(0, noise_factor * std, size=signal.shape)
    return signal + noise

##############################################################################
# 3. Wavelet Denoising Function (CPU)
##############################################################################
def wavelet_denoise(signal, wavelet='db4', level=2, mode='soft'):
    coeffs = pywt.wavedec(signal, wavelet, level=level)
    detail_coeffs = coeffs[-1]
    sigma_est = np.median(np.abs(detail_coeffs)) / 0.6745
    n = len(signal)
    threshold = sigma_est * np.sqrt(2 * np.log(n))
    new_coeffs = [coeffs[0]]
    for c in coeffs[1:]:
        new_coeffs.append(pywt.threshold(c, threshold, mode=mode))
    denoised = pywt.waverec(new_coeffs, wavelet)
    return denoised[:n]
```

```python
################################################################################
# 4. Extract Only Robust Features (CPU)
################################################################################
def extract_robust_features(x, dt,
                            wavelet_family='db4',
                            wavelet_level=2,
                            sign_threshold=0.0,
                            wavelet_mode='soft'):
    """
    Extract only the robust features:
      1. Signal_Length
      2. Non_Zero_Count
      3. TKEO_Mean
      4. TKEO_Median
      5. PSD_Median
      6. PSD_Mean
      7. Sign_Changes
      8. Zero_Crossing_Rate
      9. MedianAbs_Jerk
      10. Lag1_Autocorrelation
    """
    # Wavelet denoising
    x_denoised = wavelet_denoise(x, wavelet=wavelet_family,␣
↪level=wavelet_level, mode=wavelet_mode)
    x_arr = x_denoised
    n = len(x_arr)
    features = {}

    # 1. Signal Length
    features['Signal_Length'] = n

    # 2. Non-Zero Count
    features['Non_Zero_Count'] = np.count_nonzero(x_arr)

    # 3. TKEO Mean
    tkeo = x_arr[1:-1]**2 - x_arr[:-2] * x_arr[2:]
    features['TKEO_Mean'] = np.mean(tkeo) if len(tkeo) > 0 else np.nan

    # 4. TKEO Median
    features['TKEO_Median'] = np.median(tkeo) if len(tkeo) > 0 else np.nan

    # 5-6. PSD Median and Mean
    freqs_welch, psd_vals = welch(x_arr, fs=1 / dt)
    features['PSD_Median'] = np.median(psd_vals)
    features['PSD_Mean'] = np.mean(psd_vals)
```

```python
    # 7. Sign Changes
    sign_changes = 0
    for i in range(n - 1):
        if (abs(x_arr[i]) > sign_threshold) and (abs(x_arr[i + 1]) > sign_threshold):
            if x_arr[i] * x_arr[i + 1] < 0:
                sign_changes += 1
    features['Sign_Changes'] = sign_changes

    # 8. Zero Crossing Rate
    features['Zero_Crossing_Rate'] = sign_changes / n

    # 9. Median Absolute Jerk
    vel = np.diff(x_arr) / dt
    if len(vel) > 1:
        jerk = np.diff(vel) / dt
        features['MedianAbs_Jerk'] = np.median(np.abs(jerk))
    else:
        features['MedianAbs_Jerk'] = np.nan

    # 10. Lag-1 Autocorrelation
    if n > 1:
        autocorr = np.corrcoef(x_arr[:-1], x_arr[1:])[0, 1]
    else:
        autocorr = np.nan
    features['Lag1_Autocorrelation'] = autocorr

    return features

##############################################################################
# 5. Prepare Dataset for Training
##############################################################################
best_params = {
    'wavelet_family': 'db4',
    'wavelet_level': 2,
    'sign_threshold': 0.01,
    'wavelet_mode': 'soft'
}

noise_levels = np.linspace(0.01, 0.30, 5)
segment_size = 256
dt = time[1] - time[0]  # constant sampling interval

def prepare_dataset(noise_levels, segment_size, best_params):
    dataset = []
    labels = []
    for noise_factor in noise_levels:
```

```
        noisy_signal_gpu = add_noise_to_signal_gpu(clean_signal, noise_factor)
        noisy_signal = cp.asnumpy(noisy_signal_gpu)
        n_segs = n_rows // segment_size
        for start in range(0, n_segs * segment_size, segment_size):
            seg = noisy_signal[start:start + segment_size]
            feat = extract_robust_features(
                seg, dt,
                wavelet_family=best_params['wavelet_family'],
                wavelet_level=best_params['wavelet_level'],
                sign_threshold=best_params['sign_threshold'],
                wavelet_mode=best_params['wavelet_mode']
            )
            dataset.append(feat)
            labels.append(noise_factor)  # Use noise factor as the target␣
↪variable
    df_dataset = pd.DataFrame(dataset)
    df_labels = pd.Series(labels, name='NoiseFactor')
    return df_dataset, df_labels

df_dataset, df_labels = prepare_dataset(noise_levels, segment_size, best_params)

##############################################################################
# 6. Train a Machine Learning Model
##############################################################################
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df_dataset, df_labels,␣
↪test_size=0.2, random_state=42)

# Train a Random Forest Regressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error on Test Set:", mse)

##############################################################################
# 7. Feature Importance Analysis
##############################################################################
feature_importances = pd.Series(model.feature_importances_, index=df_dataset.
↪columns)
feature_importances.sort_values(ascending=False, inplace=True)

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances.values, y=feature_importances.index)
```

```
plt.title("Feature Importances")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

```
Mean Squared Error on Test Set: 0.00024919746474358897
```