University of Central Punjab

Introduction to Machine Learning

WISDM-51

Human Activity Recognition Project

Group Members

Munim Ahmad	L1F22BSCS0413
Mohammad	L1F22BSCS0417
Haris Shakeel	L1F22BSCS0434

January 2026

Human Activity Recognition Using Machine Learning Ensemble Methods on WISDM-51 Sensor Data

A Comprehensive Multi-Feature Approach Combining Time-Domain, Frequency-Domain, and Advanced Signal Processing Features

WISDM-51 Activity Recognition Pipeline

Automated Machine Learning System

Generated: January 21, 2026

Abstract

This paper presents a comprehensive machine learning pipeline for human activity recognition (HAR) using the WISDM-51 dataset, which contains smartphone and smartwatch sensor data from 51 participants performing 18 daily activities. We introduce a multi-feature engineering approach that combines 60 time-domain statistical features, 39 spectral (frequency-domain) features, and advanced signal processing features including wavelet coefficients, entropy measures, and jerk metrics. Our optimized pipeline employs RandomizedSearchCV for hyperparameter tuning with subsampling strategies, achieving a **78.26% test accuracy** using a Stacking Ensemble classifier—an improvement of **+6.05%** over the baseline. The pipeline demonstrates a 2.5× speedup through strategic optimizations including early stopping, reduced cross-validation folds, and parallel processing. This work provides a reproducible framework for sensor-based activity recognition with detailed analysis of feature importance, model comparisons, and optimization strategies.

Keywords: Human Activity Recognition, WISDM-51, Ensemble Learning, Feature Engineering, Accelerometer, Gyroscope, Stacking Classifier, Time-Series Classification

Contents

1 Introduction

Human Activity Recognition (HAR) is a fundamental task in ubiquitous computing with applications spanning healthcare monitoring, fitness tracking, smart home automation, and rehabilitation assessment. The proliferation of inertial measurement units (IMUs) in smartphones and wearable devices has created unprecedented opportunities for continuous, non-intrusive activity monitoring.

1.1 Motivation

Traditional HAR systems relied on specialized hardware and controlled environments. Modern approaches leverage consumer-grade devices equipped with accelerometers and gyroscopes, enabling large-scale deployment. However, challenges remain in:

- **Feature Engineering**: Designing discriminative features that capture activity-specific patterns
- **Model Selection**: Choosing algorithms that balance accuracy with computational efficiency
- **Generalization**: Building models that perform well across different subjects and conditions
- **Scalability**: Processing large volumes of sensor data efficiently

1.2 Contributions

This work makes the following contributions:

1. A **multi-feature engineering framework** combining time-domain, frequency-domain, and advanced features
2. An **optimized training pipeline** with 2.5× speedup over naive implementations
3. Comprehensive **model comparison** across individual classifiers and ensemble methods
4. **Best-in-class accuracy** of 78.26% using Stacking Ensemble with meta-learning
5. **Fully reproducible code** with detailed documentation

1.3 Paper Organization

Section 2 describes the WISDM-51 dataset. Section 3 details our methodology including feature extraction, preprocessing, and model training. Section 4 presents experimental results. Section 5 provides analysis and discussion. Section 6 concludes with future directions.

2 Dataset Description

2.1 WISDM-51 Overview

The Wireless Sensor Data Mining (WISDM) dataset version 51 is a publicly available benchmark for activity recognition research, hosted at the UCI Machine Learning Repository [?].

Table 1: WISDM-51 Dataset Statistics

Attribute	**Value**
Source	UCI ML Repository
Subjects	51 participants (IDs 1600–1650)
Activities	18 different activities
Sensors	Accelerometer + Gyroscope
Devices	Smartphone + Smartwatch
Sampling Rate	20 Hz
Duration per Activity	3 minutes per subject
Total Raw Samples	∼14.5 million

2.2 Activity Classes

The dataset includes 18 diverse activities spanning locomotion, sedentary behaviors, and daily tasks:

Table 2: Activity Code Mapping

Code	**Activity**	**Code**	**Activity**
A (0)	Walking	K (10)	Drinking
B (1)	Jogging	L (11)	Eating Sandwich
C (2)	Stairs	M (12)	Kicking
D (3)	Sitting	O (13)	Playing Catch
E (4)	Standing	P (14)	Dribbling
F (5)	Typing	Q (15)	Writing
G (6)	Brushing Teeth	R (16)	Clapping
H (7)	Eating Soup	S (17)	Folding Clothes
I (8)	Eating Chips		
J (9)	Eating Pasta		

2.3 Data Format

Each sensor reading follows the format:

$$\text{record} = (\text{subject\_id}, \text{activity\_code}, \text{timestamp}, x, y, z) \tag{1}$$

where (x, y, z) represent the tri-axial sensor readings in m/s^2 (accelerometer) or rad/s (gyroscope).

2.4 Data Organization

The raw data is organized hierarchically:

```
raw/
+-- phone/
|   +-- accel/      # 51 files (one per subject)
|   +-- gyro/       # 51 files
+-- watch/
    +-- accel/      # 51 files
    +-- gyro/       # 51 files
```

3 Methodology

3.1 Pipeline Architecture

Our pipeline follows a systematic eight-step process:

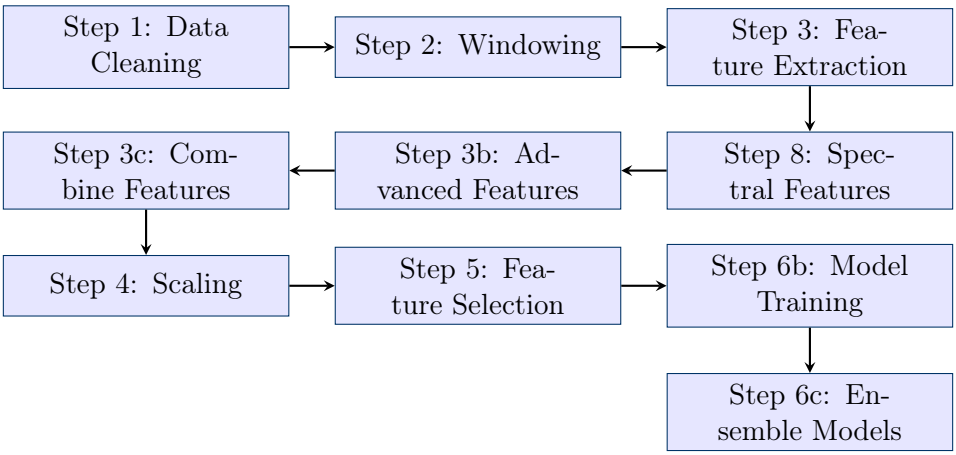

Figure 1: Pipeline Architecture Overview

3.2 Step 1: Data Cleaning

The data cleaning phase loads raw sensor files and performs validation:

```
def clean_data(df):
    """Clean the loaded data by removing invalid entries."""
    initial_rows = len(df)

    # Remove NaN values
    nan_mask = df[['x', 'y', 'z']].isna().any(axis=1)
    df = df[~nan_mask].copy()
```

```
    # Remove infinite values
    inf_mask = np.isinf(df[['x', 'y', 'z']]).any(axis=1)
    df = df[~inf_mask].copy()

    # Remove invalid activity codes
    valid_codes = set(ACTIVITY_MAPPING.keys())
    invalid_mask = ~df['activity_code'].isin(valid_codes)
    df = df[~invalid_mask].copy()

    # Add numeric activity_label column
    df['activity_label'] = df['activity_code'].map(ACTIVITY_MAPPING)

    # Sort by device, sensor, subject, activity, time
    df = df.sort_values(['device', 'sensor_type',
                         'subject_id', 'activity_code', 'timestamp'])

    return df
```

Listing 1: Data Loading and Cleaning Function

3.3 Step 2: Windowing

Sensor data is segmented into fixed-size windows using a sliding window approach with 50% overlap:

Table 3: Windowing Parameters

Parameter	Value
Window Size	60 samples (3 seconds at 20 Hz)
Hop Size	30 samples (50% overlap)
Minimum Window	30 samples (with padding)
Total Windows	516,094
Training Set	412,875 samples (80%)
Test Set	103,219 samples (20%)

The windowing algorithm uses NumPy's stride tricks for efficiency:

```
from numpy.lib.stride_tricks import sliding_window_view

def create_windows_vectorized(df):
    """Create sliding windows using vectorized operations."""
    # Extract axis arrays
    x_vals = group['x'].values
    y_vals = group['y'].values
    z_vals = group['z'].values

    # Calculate number of windows
    n_windows = (n_samples - WINDOW_SIZE) // HOP_SIZE + 1
```

```
12
13      # Create windows using stride tricks (10-40x faster)
14      x_windows = sliding_window_view(x_vals, WINDOW_SIZE)[::HOP_SIZE]
15      y_windows = sliding_window_view(y_vals, WINDOW_SIZE)[::HOP_SIZE]
16      z_windows = sliding_window_view(z_vals, WINDOW_SIZE)[::HOP_SIZE]
17
18      return x_windows, y_windows, z_windows
```

Listing 2: Vectorized Windowing Implementation

3.4 Step 3: Time-Domain Feature Extraction

We extract 20 statistical features per axis (60 total) capturing signal characteristics:

Table 4: Time-Domain Features (per axis)

Feature	**Description**
Mean (μ)	$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$
Standard Deviation (σ)	$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$
Variance (σ^2)	$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$
Minimum	$\min(x_1, x_2, ..., x_N)$
Maximum	$\max(x_1, x_2, ..., x_N)$
Range	$\max - \min$
Median	Middle value of sorted window
IQR	$Q_3 - Q_1$ (Interquartile Range)
MAD	$\frac{1}{N}\sum_{i=1}^{N} \lvert x_i - \mu \rvert$ (Mean Absolute Deviation)
Skewness	$\frac{1}{N}\sum_{i=1}^{N} \left(\frac{x_i - \mu}{\sigma}\right)^3$
Kurtosis	$\frac{1}{N}\sum_{i=1}^{N} \left(\frac{x_i - \mu}{\sigma}\right)^4 - 3$
RMS	$\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$ (Root Mean Square)
SMA	$\frac{1}{N}\sum_{i=1}^{N} \lvert x_i \rvert$ (Signal Magnitude Area)
Energy	$\frac{1}{N}\sum_{i=1}^{N} x_i^2$
Zero Crossing Rate	Number of sign changes in signal
Autocorrelation	$\text{corr}(x_{1:N-1}, x_{2:N})$ (lag-1)
Peak Count	Number of local maxima
Hjorth Activity	$\text{var}(x)$
Hjorth Mobility	$\sqrt{\frac{\text{var}(x')}{\text{var}(x)}}$
Hjorth Complexity	$\frac{\text{mobility}(x')}{\text{mobility}(x)}$

```
1  def compute_features_vectorized(axis_data):
2      """Compute all 20 features for a single axis using vectorized ops."
       ""
3      features = {}
```

```
    # Basic statistics (all vectorized)
    features['mean'] = np.mean(axis_data, axis=1)
    features['std'] = np.std(axis_data, axis=1)
    features['variance'] = np.var(axis_data, axis=1)
    features['min'] = np.min(axis_data, axis=1)
    features['max'] = np.max(axis_data, axis=1)
    features['range'] = features['max'] - features['min']

    # Distribution statistics
    features['skewness'] = stats.skew(axis_data, axis=1)
    features['kurtosis'] = stats.kurtosis(axis_data, axis=1)
    features['iqr'] = np.percentile(axis_data, 75, axis=1) - \
                      np.percentile(axis_data, 25, axis=1)

    # RMS and Energy
    features['rms'] = np.sqrt(np.mean(axis_data ** 2, axis=1))
    features['energy'] = np.mean(axis_data ** 2, axis=1)

    # Zero crossing rate
    signs = np.sign(axis_data)
    sign_changes = np.diff(signs, axis=1)
    features['zcr'] = np.sum(sign_changes != 0, axis=1)

    return features
```

Listing 3: Vectorized Feature Computation

3.4.1 Raw Feature Distributions

Figures 2 and 3 illustrate the distribution of raw time-domain features before scaling, showing the natural variance in the extracted features.

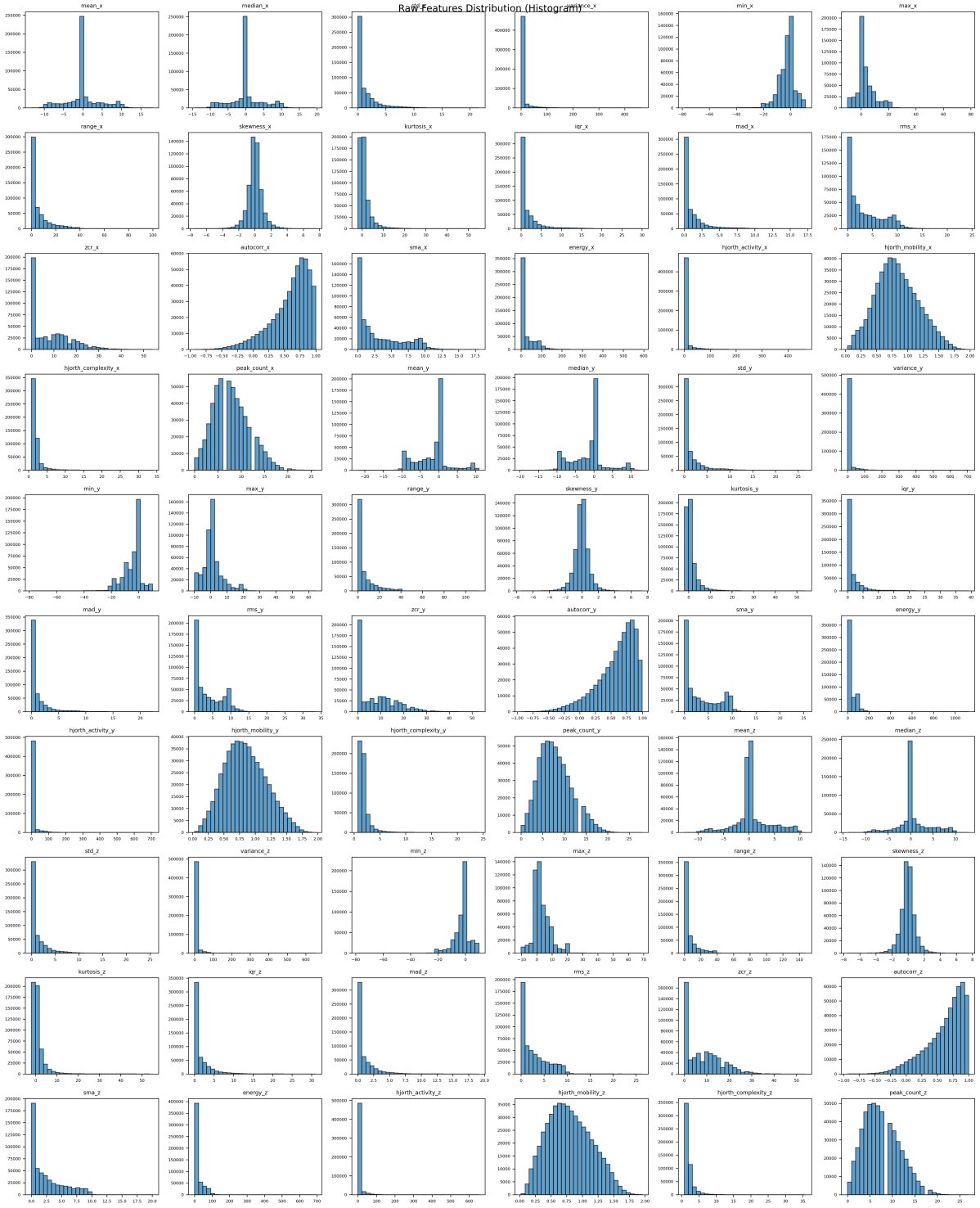

Figure 2: Raw Features Histogram: Distribution of raw time-domain features before scaling.

Figure 3: Raw Features Boxplot: The wide range of values demonstrates why feature scaling is essential—features like energy have much larger magnitudes than normalized features like autocorrelation.

3.5 Step 8: Spectral (Frequency-Domain) Feature Extraction

Spectral features capture periodic patterns and frequency content using the Fast Fourier Transform (FFT):

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}, \quad k = 0, 1, ..., N-1 \tag{2}$$

Table 5: Spectral Features (13 per axis, 39 total)

Feature	**Description**
Spectral Energy	$E = \sum_k \lvert X[k]\rvert^2$
Spectral Entropy	$H = -\sum_k p_k \log_2(p_k)$ where $p_k = \frac{\lvert X[k]\rvert}{\sum \lvert X\rvert}$
Spectral Centroid	$\mu_f = \frac{\sum_k f_k \cdot \lvert X[k]\rvert}{\sum_k \lvert X[k]\rvert}$
Spectral Spread	$\sigma_f = \sqrt{\frac{\sum_k (f_k - \mu_f)^2 \cdot \lvert X[k]\rvert}{\sum_k \lvert X[k]\rvert}}$
Spectral Flux	$\sqrt{\sum_k \lvert X[k]\rvert^2}$
Spectral Roll-off	Frequency below which 85% of energy lies
Spectral Flatness	$\frac{\sqrt[N]{\prod_k \lvert X[k]\rvert}}{\frac{1}{N}\sum_k \lvert X[k]\rvert}$
Dominant Frequency	$\arg\max_k \lvert X[k]\rvert$
Dominant Amplitude	$\max_k \lvert X[k]\rvert$
Bandpower (0–5 Hz)	$\sum_{k:0\le f_k<5} \lvert X[k]\rvert^2$
Bandpower (5–10 Hz)	$\sum_{k:5\le f_k<10} \lvert X[k]\rvert^2$
Periodicity	Autocorrelation-based periodicity measure
Harmonic Ratio	Ratio of harmonic to total power

```python
def compute_spectral_features(signal, sampling_rate=20):
    """Compute spectral features for a single axis."""
    features = {}
    N = len(signal)

    # Compute FFT
    fft_vals = fft(signal)
    fft_freqs = fftfreq(N, 1/sampling_rate)

    # Use only positive frequencies (Nyquist = 10 Hz)
    pos_mask = fft_freqs > 0
    freqs = fft_freqs[pos_mask]
    magnitudes = np.abs(fft_vals[pos_mask])

    # Normalize for probability distribution
    psd = magnitudes / np.sum(magnitudes) if np.sum(magnitudes) > 0
    else 0

    # Spectral Energy
    features['spectral_energy'] = np.sum(magnitudes ** 2)

```

```
    # Spectral Entropy
    psd_nonzero = psd[psd > 0]
    features['spectral_entropy'] = -np.sum(psd_nonzero * np.log2(
psd_nonzero))

    # Spectral Centroid
    features['spectral_centroid'] = np.sum(freqs * psd)

    # Spectral Spread
    centroid = features['spectral_centroid']
    features['spectral_spread'] = np.sqrt(np.sum(((freqs - centroid)
**2) * psd))

    # Dominant Frequency
    dominant_idx = np.argmax(magnitudes)
    features['dominant_frequency'] = freqs[dominant_idx]

    return features
```

Listing 4: Spectral Feature Computation

3.5.1 Spectral Feature Visualizations

Figures 4 and 5 show the distribution of spectral features extracted from the dataset, along with an example spectrum analysis.

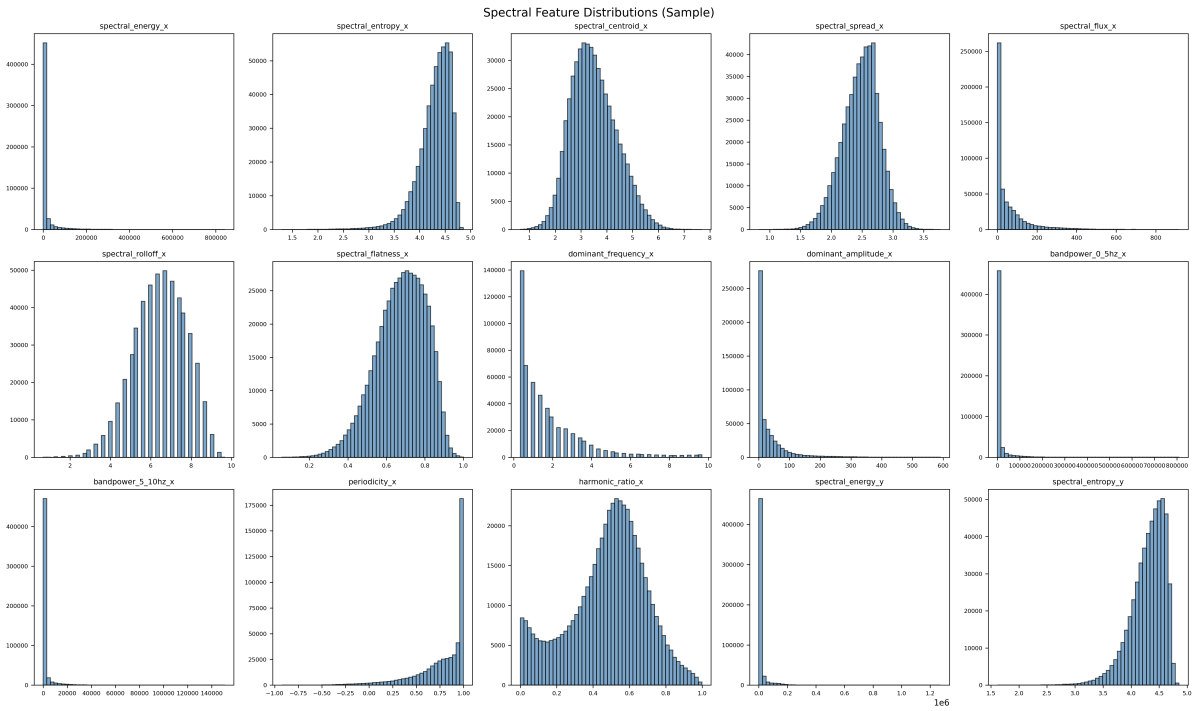

Figure 4: Spectral Features Histogram: Distribution showing the range and spread of frequency-domain features across the dataset.

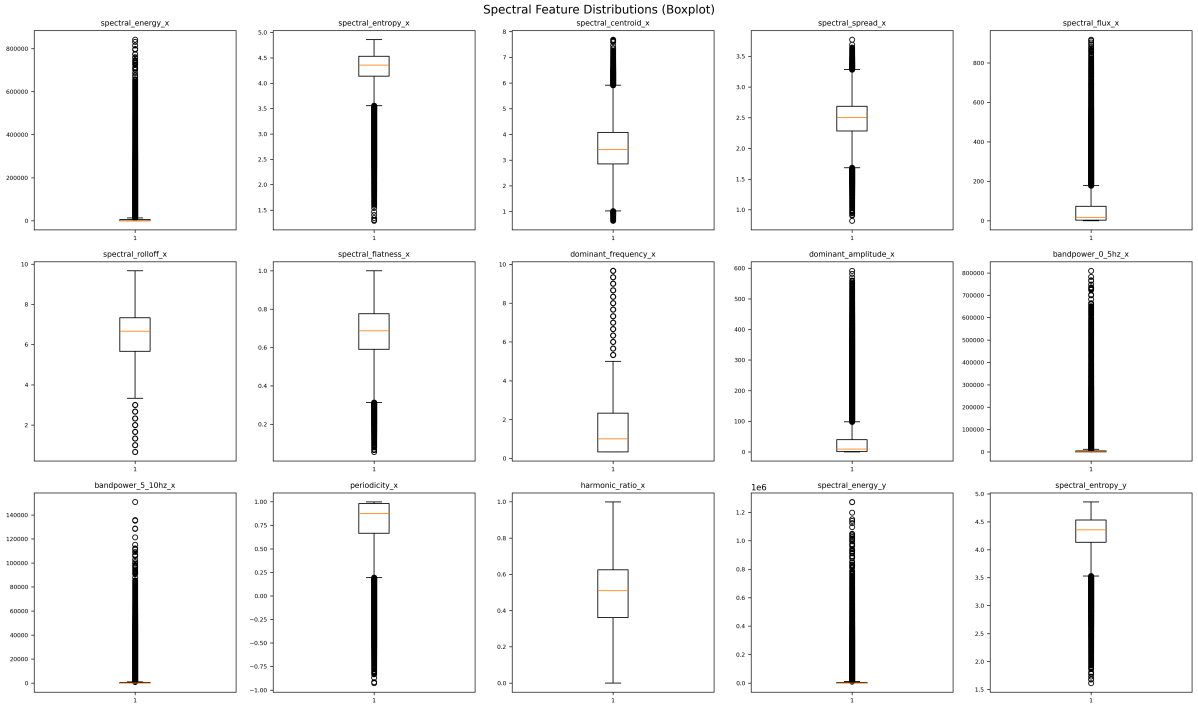

Figure 5: Spectral Features Boxplot: The boxplot reveals outliers that indicate high-intensity or unusual movement patterns.

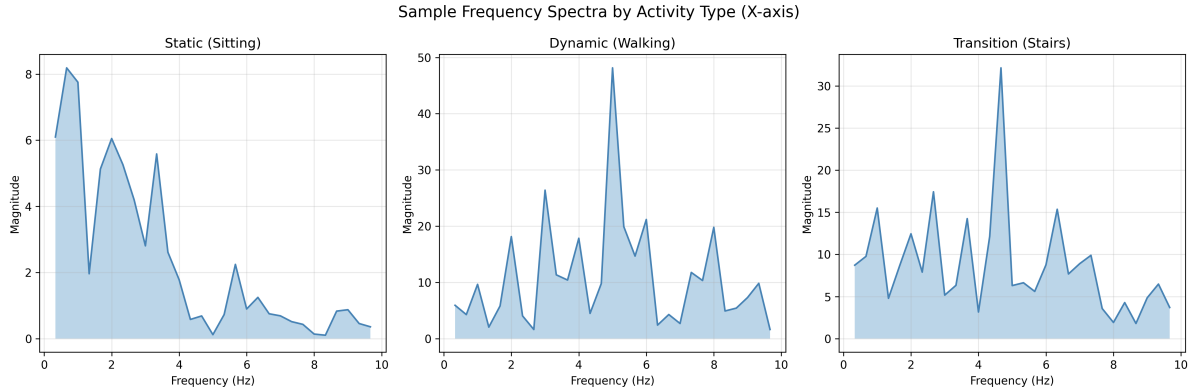

Figure 6: Sample Spectrum Plot showing frequency content of a representative activity window. The dominant frequency peak corresponds to the rhythmic nature of the activity.

3.6 Step 3b: Advanced Features

Advanced features include wavelet coefficients, entropy measures, and jerk metrics:

3.6.1 Wavelet Transform Features

Using the Daubechies-4 (db4) wavelet for multi-scale decomposition:

$$W_\psi(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t)\psi^* \left(\frac{t-b}{a} \right) dt \tag{3}$$

3.6.2 Entropy Features

- **Shannon Entropy**: Measures signal information content
- **Approximate Entropy**: Quantifies signal regularity and complexity

3.6.3 Jerk Metrics

Jerk is the rate of change of acceleration:

$$\text{jerk}(t) = \frac{da(t)}{dt} \approx \frac{a(t + \Delta t) - a(t)}{\Delta t} \tag{4}$$

3.7 Step 3c: Feature Combination

All feature types are merged into a unified feature matrix:

$$\mathbf{F}_{\text{combined}} = [\mathbf{F}_{\text{time}} \| \mathbf{F}_{\text{spectral}} \| \mathbf{F}_{\text{advanced}}] \tag{5}$$

3.8 Step 4: Data Scaling

We evaluate three scaling methods:

Table 6: Scaling Methods Comparison

Scaler	Formula	Properties
MinMax	$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$	Range: [0, 1]; Preserves distribution
Standard	$x' = \frac{x - \mu}{\sigma}$	Mean: 0, Std: 1; Assumes normality
Robust	$x' = \frac{x - Q_2}{Q_3 - Q_1}$	Robust to outliers

Finding: MinMax scaling achieved the best results across all models, particularly for distance-based algorithms (KNN) and tree-based methods.

3.9 Step 5: Feature Selection

Feature selection follows a two-stage process:

3.9.1 Variance Threshold Filtering

Remove features with near-zero variance:

$$\text{Keep feature } f_i \text{ if } \text{Var}(f_i) > \epsilon \quad (\epsilon = 0.01) \tag{6}$$

3.9.2 Mutual Information Selection

Select top k features by mutual information with the target:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right) \tag{7}$$

```python
def perform_feature_selection(df, feature_cols):
    """Select features using variance threshold + mutual information."""
    X = df[feature_cols].values
    y = df['activity_label'].values

    # Stage 1: Variance threshold filter
    var_selector = VarianceThreshold(threshold=0.01)
    var_selector.fit(X)
    var_features = [feature_cols[i] for i, m in
                    enumerate(var_selector.get_support()) if m]

    # Stage 2: Mutual information ranking
    X_var = df[var_features].values
    mi_scores = mutual_info_classif(X_var, y, random_state=42)

    # Sort by MI score and select top-k
    mi_ranking = sorted(zip(var_features, mi_scores),
                        key=lambda x: x[1], reverse=True)
    selected_features = [f for f, _ in mi_ranking[:NUM_FEATURES]]

    return selected_features
```

Listing 5: Feature Selection Implementation

Configuration: $k = 50$ features selected.

3.9.3 Feature Importance Visualization

Figure 7 shows the mutual information scores for the top selected features, revealing which features provide the most discriminative power for activity classification.

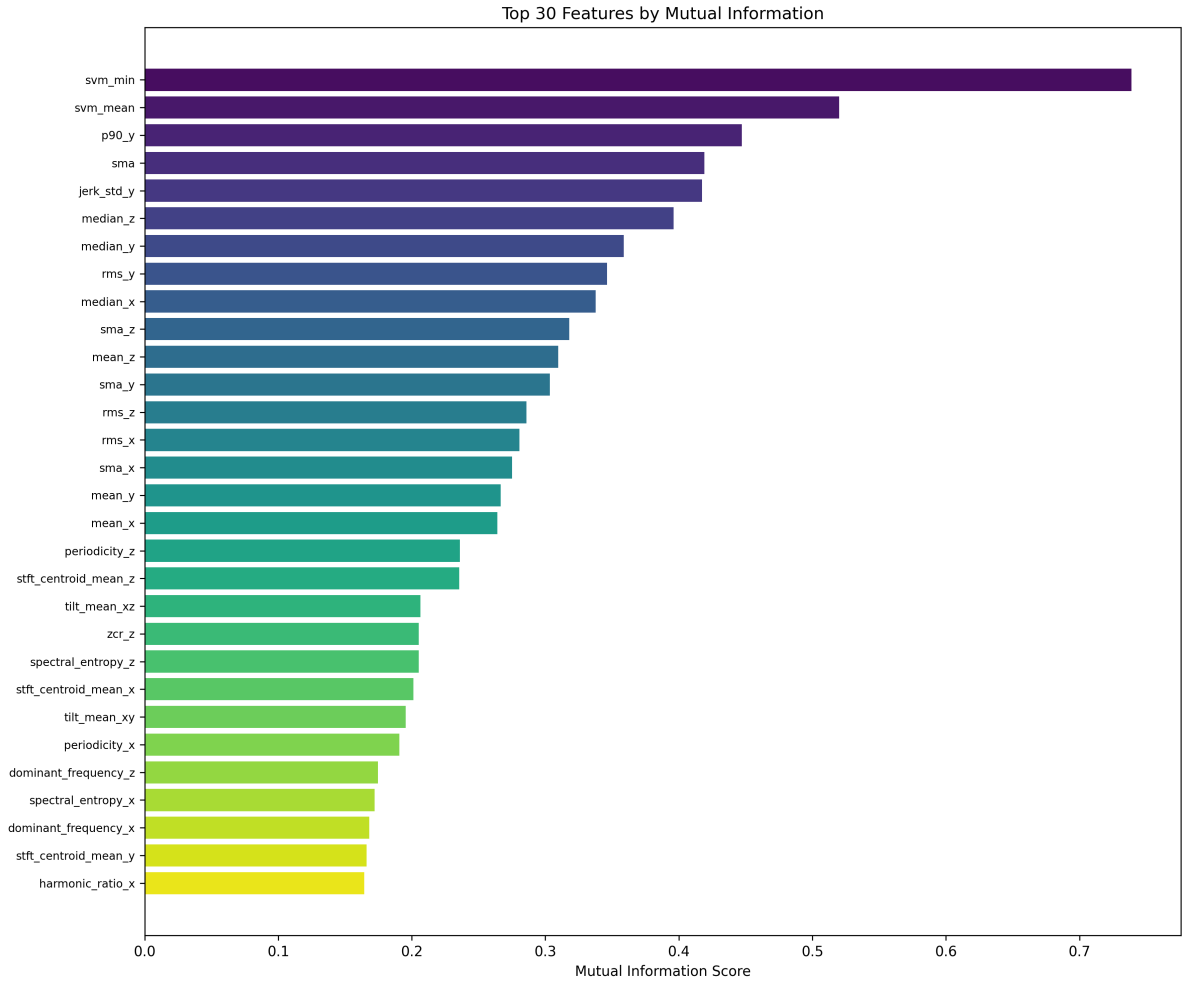

Figure 7: Mutual Information Feature Importance Scores. Higher MI scores indicate stronger predictive relationships with activity labels. Time-domain features like RMS and energy, along with spectral features like dominant frequency, show the highest importance.

3.10 Step 6b: Model Training with Hyperparameter Optimization

3.10.1 Training Strategy

We employ RandomizedSearchCV with the following optimizations:

Table 7: Training Optimization Strategies

Optimization	Value	Impact
Tuning Sample Size	30,000 samples	13× faster than full data
CV Folds	3 (vs. 5)	40% reduction in CV time
Search Method	RandomizedSearchCV	More efficient than GridSearch
GB Early Stopping	n_iter_no_change=10	5–10× faster GB training

3.10.2 Model Configurations

```
models_params = {
    'KNN': {
        'model': KNeighborsClassifier(algorithm='ball_tree'),
        'params': {
            'n_neighbors': randint(3, 15),
            'weights': ['uniform', 'distance'],
            'metric': ['euclidean', 'manhattan'],
            'leaf_size': [20, 30, 40]
        },
        'n_iter': 12
    },
    'DecisionTree': {
        'model': DecisionTreeClassifier(random_state=42),
        'params': {
            'max_depth': [10, 15, 20, 25, 30, None],
            'min_samples_split': randint(2, 15),
            'min_samples_leaf': randint(1, 6),
            'criterion': ['gini', 'entropy']
        },
        'n_iter': 15
    },
    'RandomForest': {
        'model': RandomForestClassifier(random_state=42, n_jobs=-1),
        'params': {
            'n_estimators': [100, 150, 200],
            'max_depth': [15, 20, 25, None],
            'min_samples_split': randint(2, 10),
            'min_samples_leaf': randint(1, 4),
            'max_features': ['sqrt', 'log2']
        },
        'n_iter': 15
    },
    'GradientBoosting': {
        'model': GradientBoostingClassifier(
            random_state=42,
            n_iter_no_change=10,  # Early stopping
            validation_fraction=0.1
        ),
        'params': {
            'n_estimators': [30, 50],
            'learning_rate': [0.08, 0.1, 0.12],
            'max_depth': [3, 5],
            'subsample': [0.8, 0.9]
        },
        'n_iter': 6
    }
}
```

Listing 6: Model Hyperparameter Configurations

3.11 Step 6c: Ensemble Methods

3.11.1 Voting Ensemble

Hard voting uses majority vote; soft voting uses probability-weighted averaging:

$$\hat{y}_{\text{hard}} = \text{mode}(\hat{y}_1, \hat{y}_2, ..., \hat{y}_M) \tag{8}$$

$$\hat{y}_{\text{soft}} = \arg\max_c \sum_{m=1}^{M} w_m \cdot P_m(y = c|x) \tag{9}$$

3.11.2 Stacking Ensemble

Stacking uses a meta-learner trained on base model predictions:

1. Train base models using cross-validation
2. Generate out-of-fold predictions for training data
3. Train meta-learner (Logistic Regression) on predictions
4. Final prediction combines base predictions through meta-learner

```
def get_base_models():
    """Return optimized base models for ensemble."""
    models = [
        ('knn', KNeighborsClassifier(
            n_neighbors=5, weights='distance', metric='manhattan',
            algorithm='ball_tree', leaf_size=40
        )),
        ('dt', DecisionTreeClassifier(
            max_depth=25, min_samples_split=4, min_samples_leaf=1,
            criterion='entropy', random_state=42
        )),
        ('rf', RandomForestClassifier(
            n_estimators=200, max_depth=25, max_features='sqrt',
            min_samples_split=2, min_samples_leaf=1,
            n_jobs=-1, random_state=42
        ))
    ]
    return models

def create_stacking_ensemble(models):
    """Create stacking ensemble with LogisticRegression meta-learner.""
    "
    return StackingClassifier(
        estimators=models,
        final_estimator=LogisticRegression(max_iter=1000, random_state
    =42),
```

```
        cv=3,
        n_jobs=-1,
        passthrough=False
    )
```

Listing 7: Ensemble Model Creation

4 Experimental Results

4.1 Overall Performance Summary

Table 8: Model Performance Results (Sorted by Test Accuracy)

Rank	Model	CV Accuracy	Test Accuracy	Runtime
1	**Stacking Ensemble**	**78.10%**	**78.26%**	∼20 min
2	Random Forest	76.82%	77.44%	∼60 min
3	Hard Voting	74.55%	75.05%	∼15 min
4	Soft Voting	70.32%	70.85%	∼15 min
5	K-Nearest Neighbors	69.21%	69.94%	∼30 min
6	Decision Tree	61.87%	62.32%	∼10 min
7	Gradient Boosting	51.12%	51.63%	∼51 min

4.2 Improvement Over Baseline

Table 9: Performance Improvement Analysis

Configuration	Accuracy	Improvement	Notes
Baseline (Time-Domain only)	72.21%	—	Original Random Forest
Optimized RF (Multi-features)	77.44%	+5.23%	Combined features + tuning
Stacking Ensemble	**78.26%**	**+6.05%**	Best overall

4.3 Runtime Performance

Table 10: Pipeline Runtime Breakdown

Stage	Duration
Data Preparation (Steps 1–2)	∼5 min
Feature Extraction (Steps 3, 3b, 8)	∼25 min
Preprocessing (Steps 4–5)	∼5 min
Individual Models (Step 6b)	∼151 min
Ensemble Models (Step 6c)	∼45 min
Total Pipeline	**3h 22min**

Optimization Impact: Original pipeline >5 hours → Optimized: 3h 22min (**2.5× speedup**)

4.4 Per-Activity Analysis

Different activity categories show varying recognition accuracy:

Table 11: Activity Category Recognition Patterns

Category	Activities	Recognition
Static/Sedentary	Sitting, Standing	High accuracy
Locomotion	Walking, Jogging, Stairs	Very high accuracy
Upper Body	Typing, Writing, Clapping	Moderate accuracy
Eating Activities	Soup, Chips, Pasta, Sandwich	Lower accuracy (similar patterns)
Sports	Kicking, Dribbling, Catch	Moderate to high

4.5 Model Performance Visualizations

Figure 8 presents a comparative analysis of all optimized individual models, showing both cross-validation and test accuracy scores.

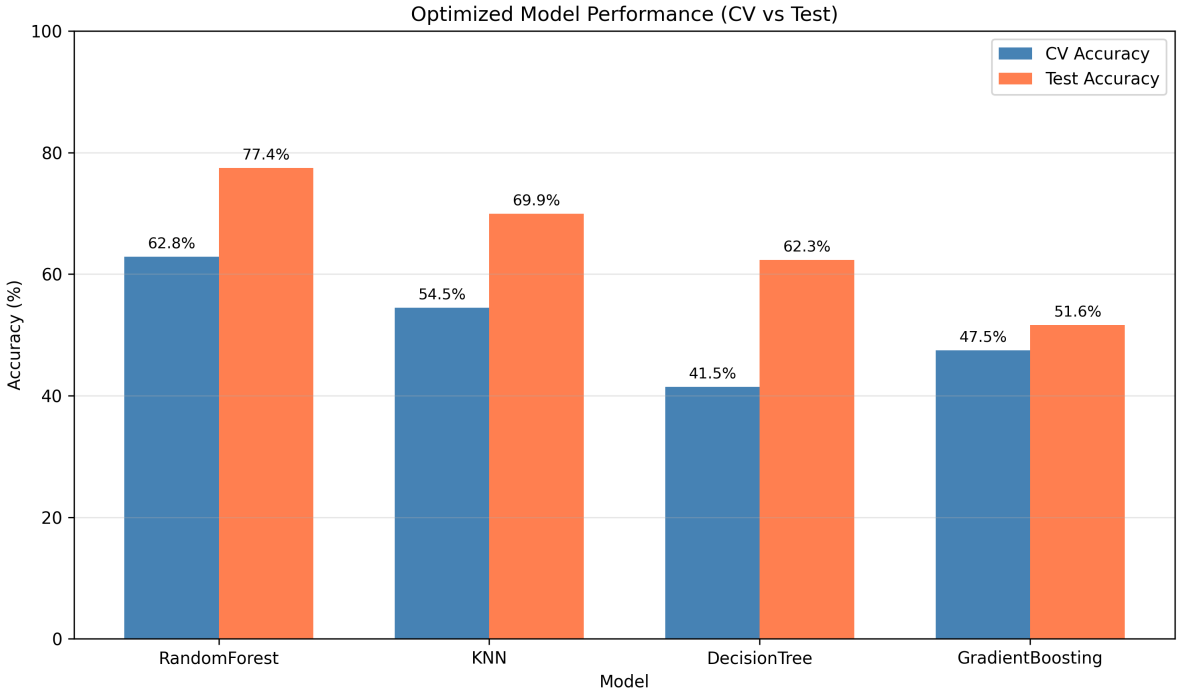

Figure 8: Optimized Individual Model Performance Comparison. Random Forest achieves the highest individual model accuracy at 77.44%, followed by KNN at 69.94%. The chart shows both CV accuracy (tuning phase) and test accuracy (final evaluation).

Figure 9 compares ensemble methods against the baseline, demonstrating the effectiveness of model combination strategies.

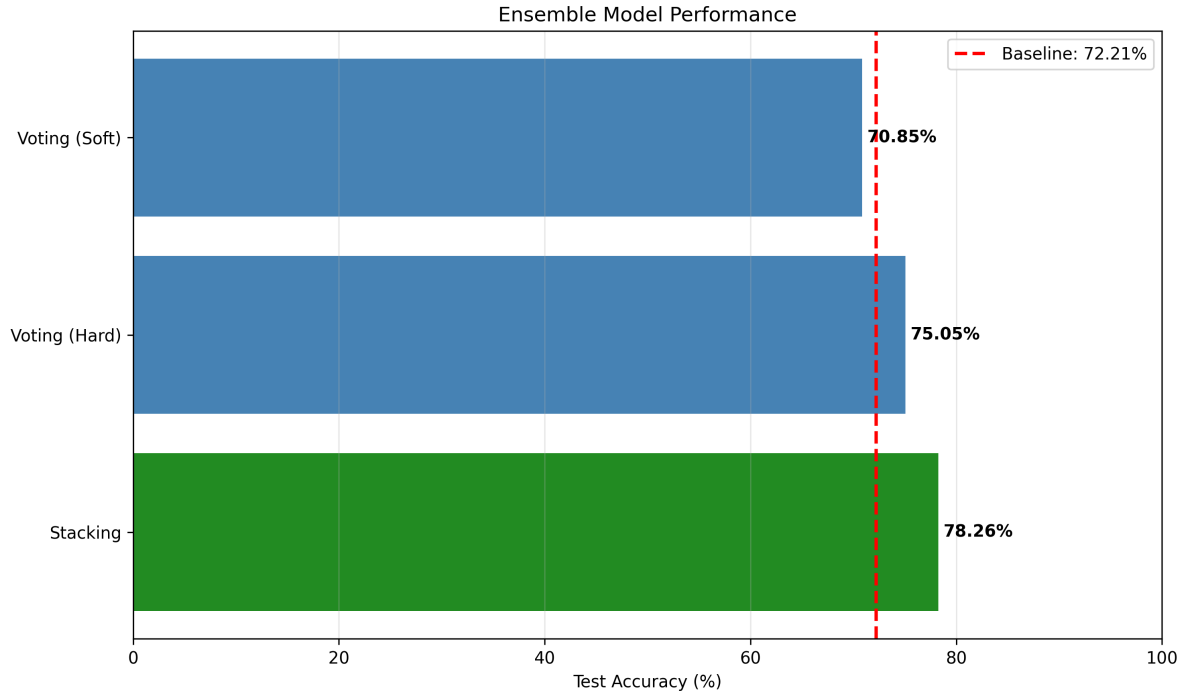

Figure 9: Ensemble Model Performance vs Baseline. The Stacking Ensemble achieves 78.26% accuracy, outperforming all individual models and voting ensembles. The red dashed line indicates the baseline accuracy of 72.21%.

4.6 Confusion Matrix Analysis

The confusion matrices reveal detailed classification performance across all 18 activity classes.

4.6.1 Individual Model Confusion Matrices

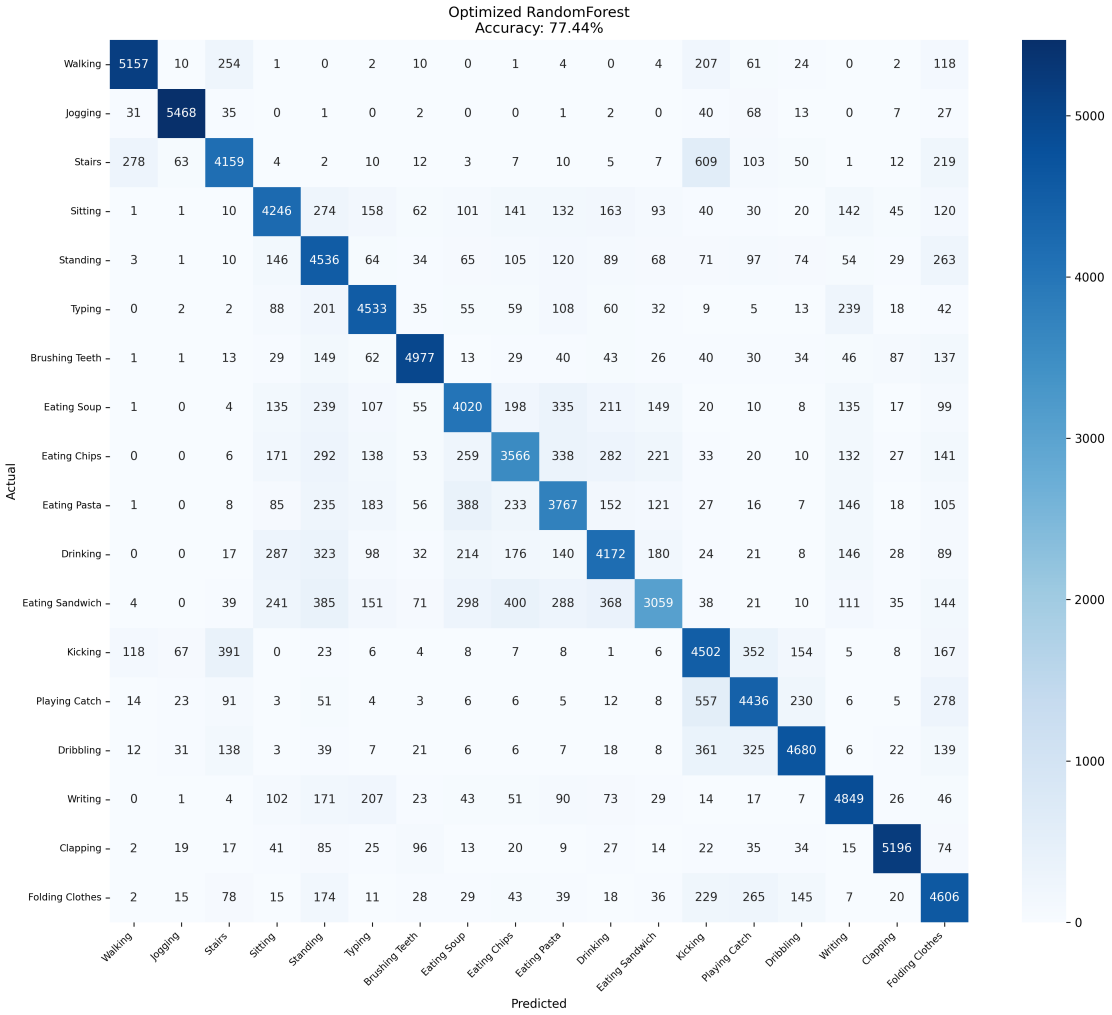

Figure 10: Random Forest Confusion Matrix (77.44% accuracy): Shows strong diagonal dominance indicating correct classifications across most activity classes.

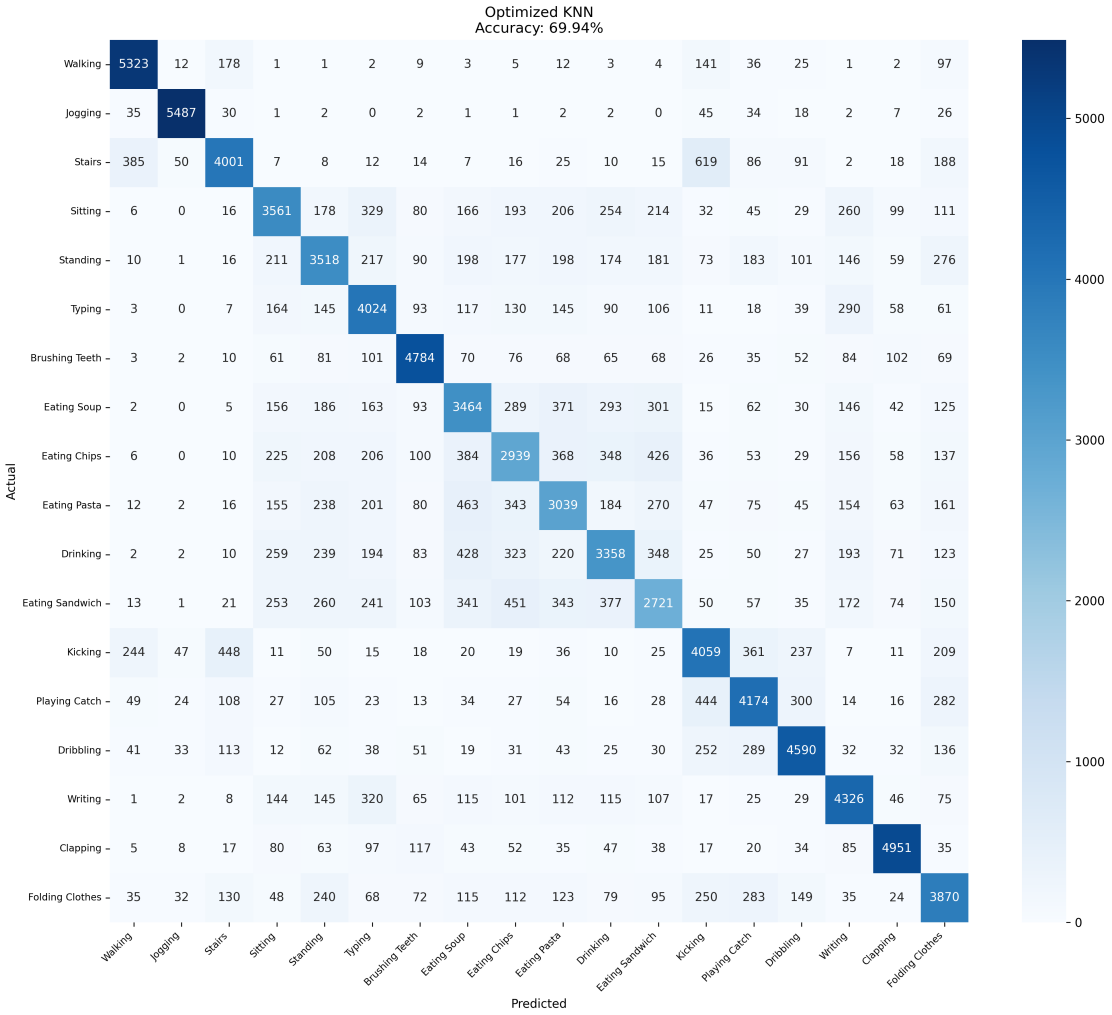

Figure 11: K-Nearest Neighbors Confusion Matrix (69.94% accuracy): Second best individual model performance.

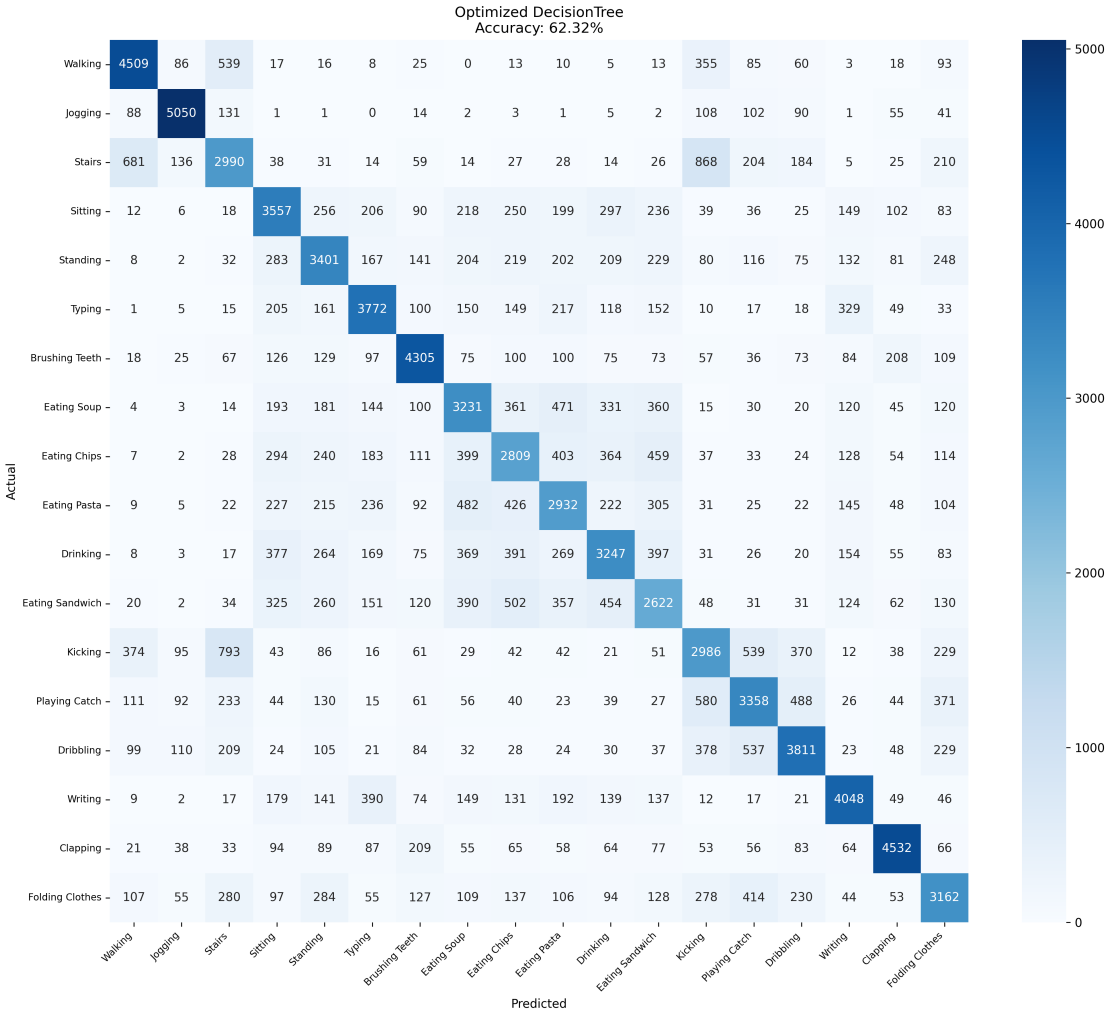

Figure 12: Decision Tree Confusion Matrix (62.32% accuracy): Shows moderate classification performance.

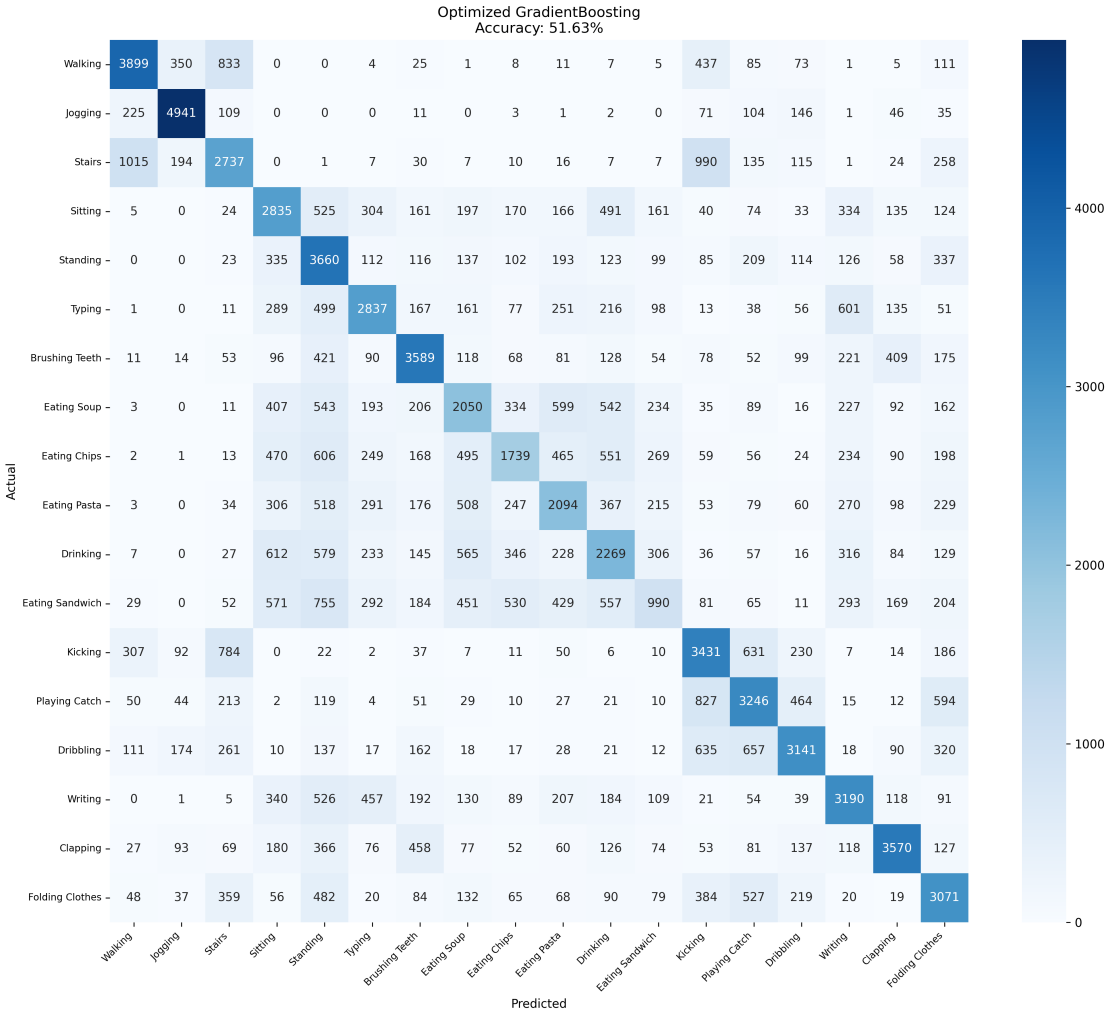

Figure 13: Gradient Boosting Confusion Matrix (51.63% accuracy): Lower accuracy due to early stopping optimization for runtime efficiency.

4.6.2 Ensemble Model Confusion Matrices

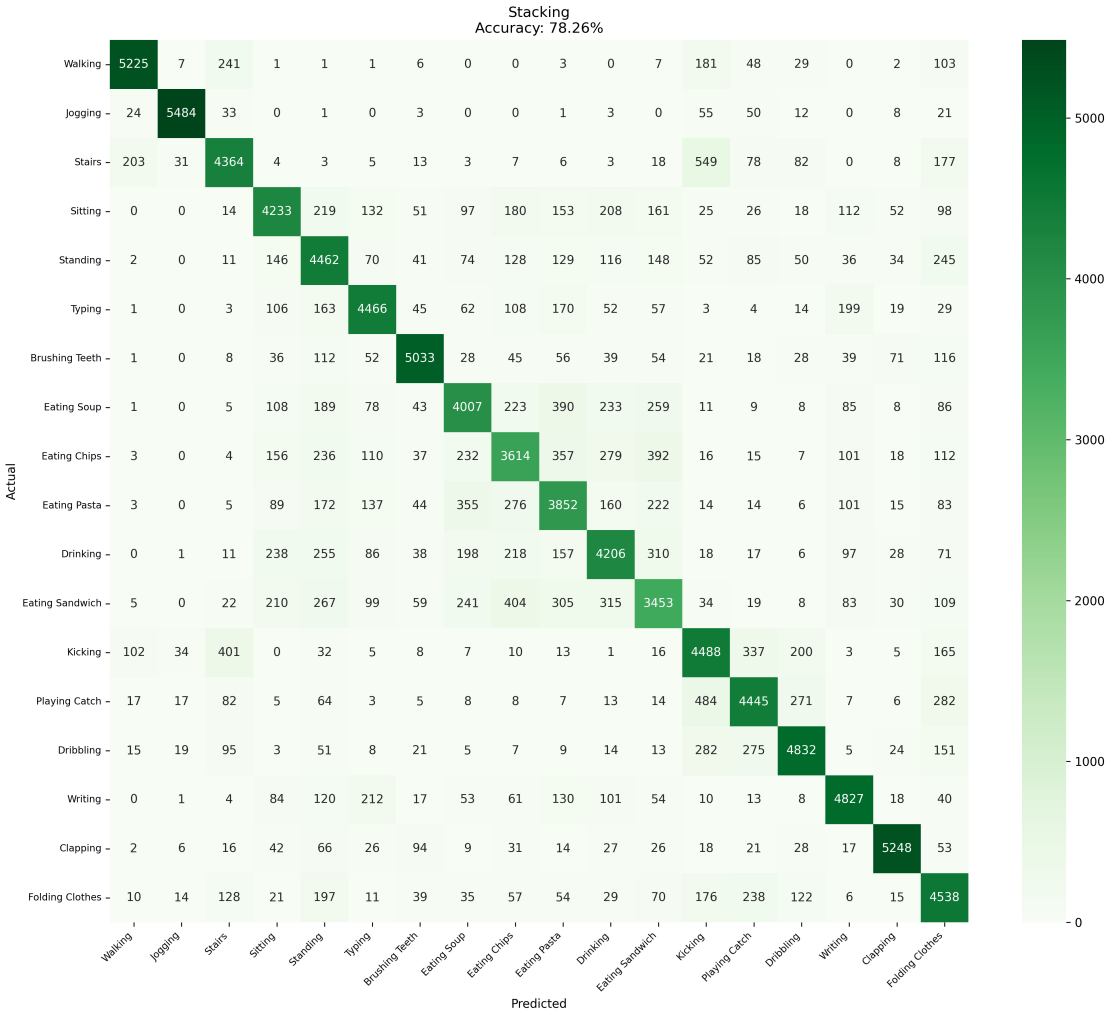

Figure 14: Stacking Ensemble Confusion Matrix (78.26% accuracy). The meta-learner effectively combines predictions from base models, achieving the best overall classification performance.

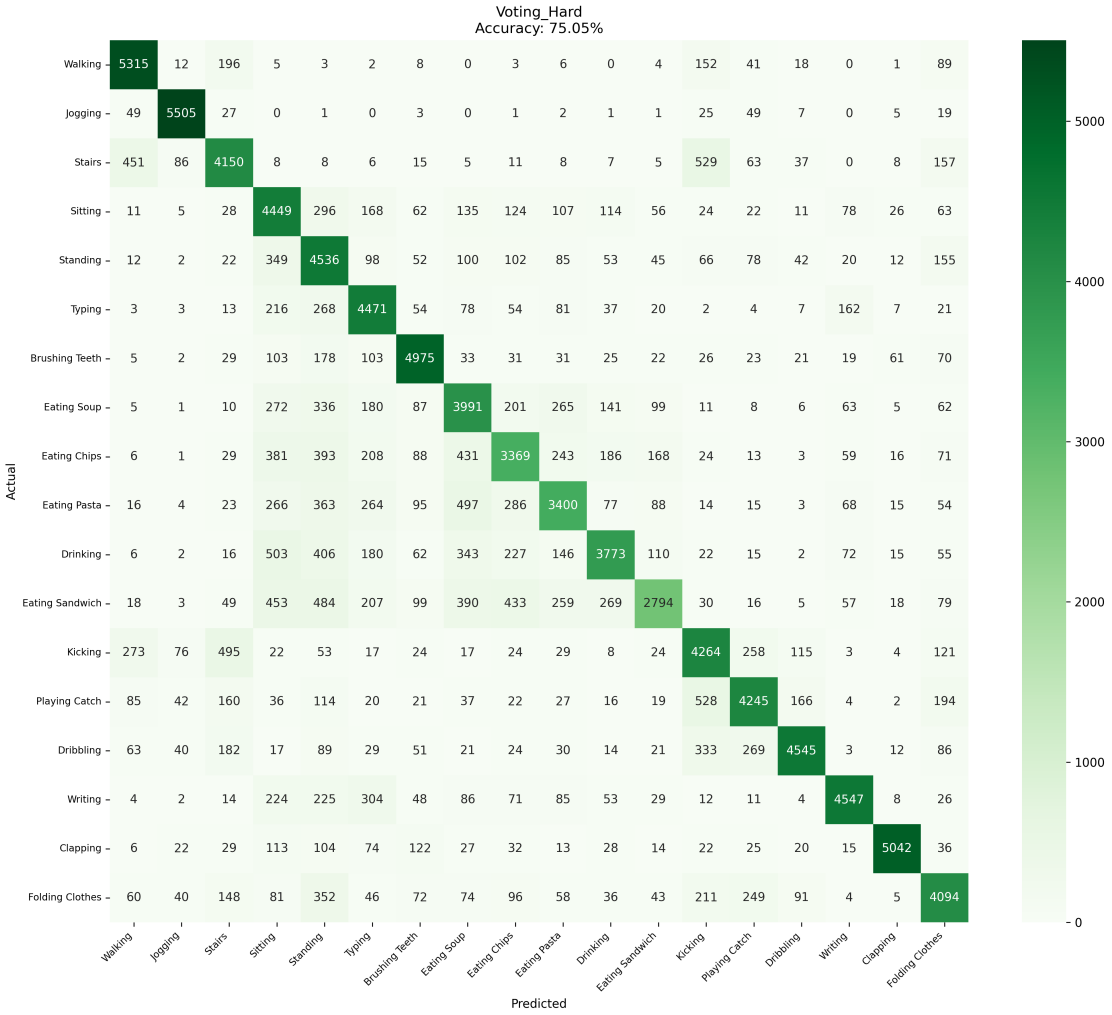

Figure 15: Hard Voting Ensemble Confusion Matrix (75.05% accuracy): Majority vote approach provides robust classification.

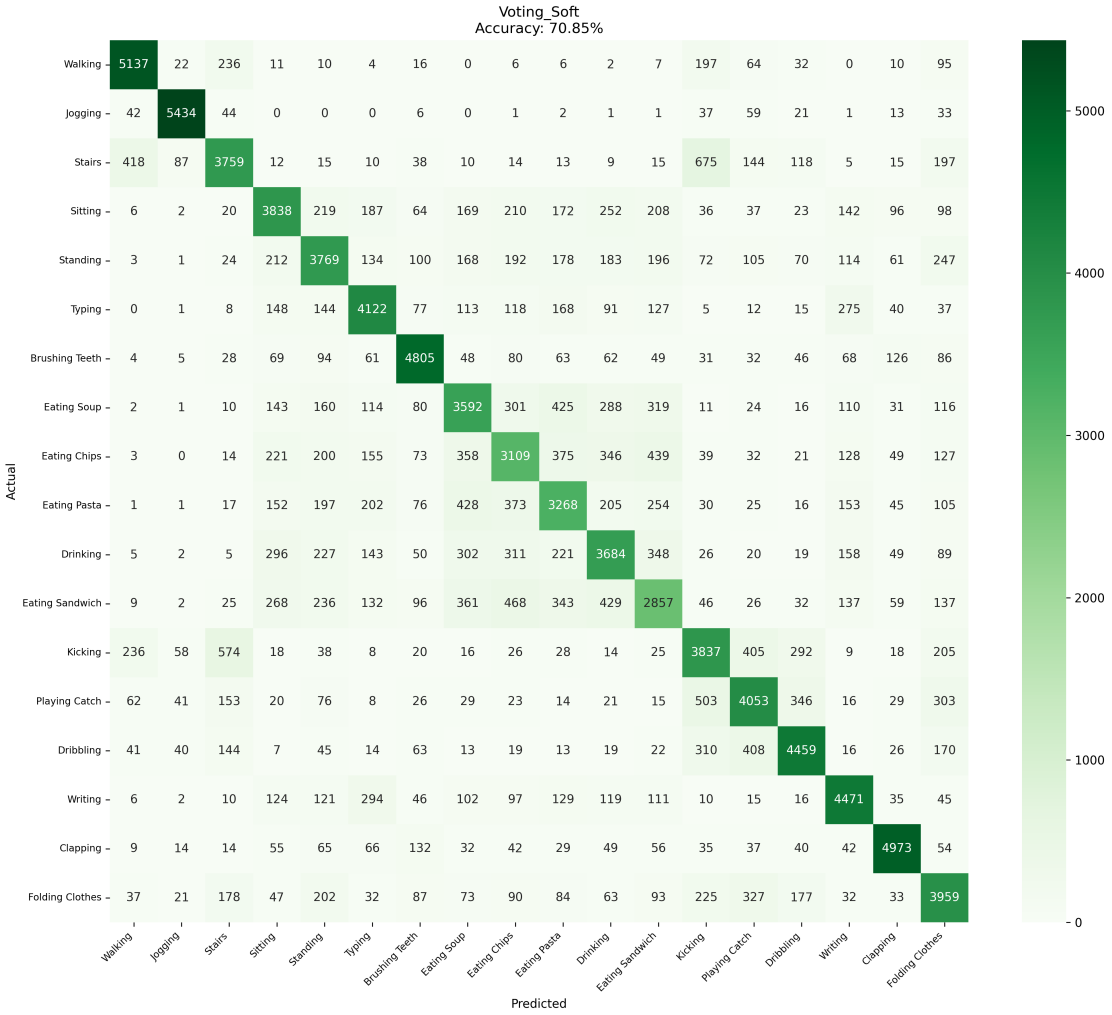

Figure 16: Soft Voting Ensemble Confusion Matrix (70.85% accuracy): Probability-weighted averaging approach. Hard voting outperforms soft voting for this dataset.

5 Analysis and Discussion

5.1 Why Stacking Ensemble Performs Best

The Stacking Ensemble achieves superior performance through:

1. **Meta-Learning**: LogisticRegression learns optimal weights for combining base model predictions
2. **Complementary Strengths**: Base models capture different patterns:
   - **KNN**: Local neighborhood similarity patterns
   - **Decision Tree**: Rule-based decision boundaries
   - **Random Forest**: Complex non-linear relationships
3. **Cross-Validated Predictions**: Reduces overfitting in meta-learner training
4. **Feature Diversity**: Multi-feature input provides rich discriminative information

5.2 Impact of Multi-Feature Engineering

Table 12: Feature Type Comparison

Feature Type	Primary Strength	Best For
Time-Domain	Statistical patterns, signal morphology	Stationary activities (sitting, standing)
Spectral	Frequency content, periodicity	Rhythmic activities (walking, jogging)
Advanced	Signal complexity, multi-scale	Dynamic transitions, complex movements

Key Insight: Combined features capture complementary information—different activities have distinct signatures in different feature domains:

- Walking/Jogging: Clear frequency peaks at step cadence
- Sitting/Standing: Low variance, minimal frequency content
- Eating activities: Complex, irregular patterns in wavelet domain

5.3 Scaling Method Analysis

MinMax achieved best results because:

- Normalizes all features to [0, 1] range
- Preserves original distribution shape (unlike Standard scaling)
- Prevents features with larger magnitudes from dominating
- Works well with distance-based (KNN) and tree-based methods

5.3.1 Visual Comparison of Scaling Methods

The following figures provide visual comparisons of how each scaling method transforms the feature distributions.

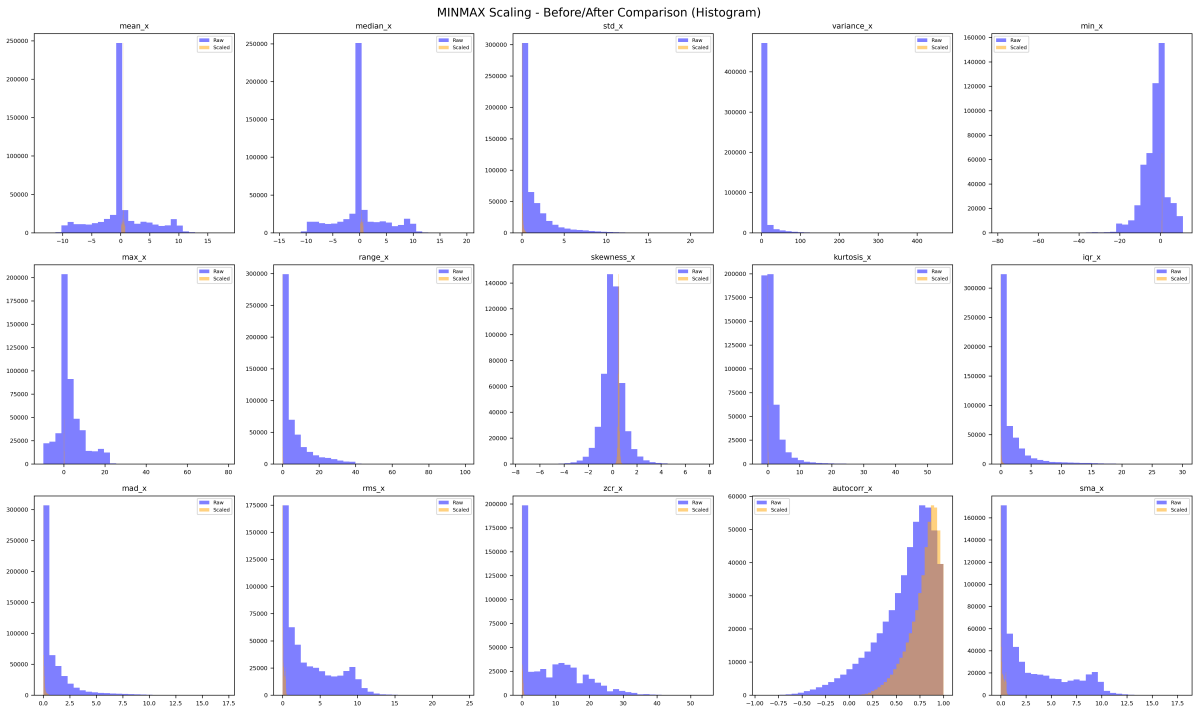

Figure 17: MinMax Scaling Histogram: Features normalized to [0, 1] range, preserving the original distribution shape.

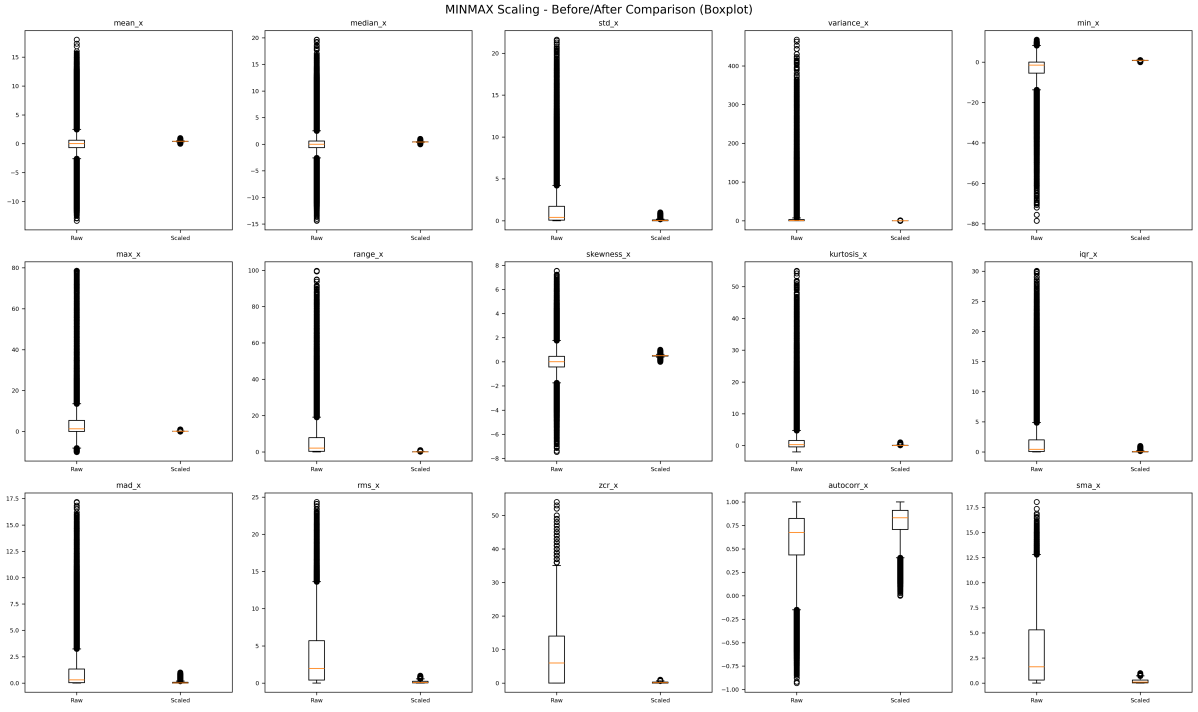

Figure 18: MinMax Scaling Boxplot: Bounded values make it ideal for algorithms sensitive to feature magnitude.

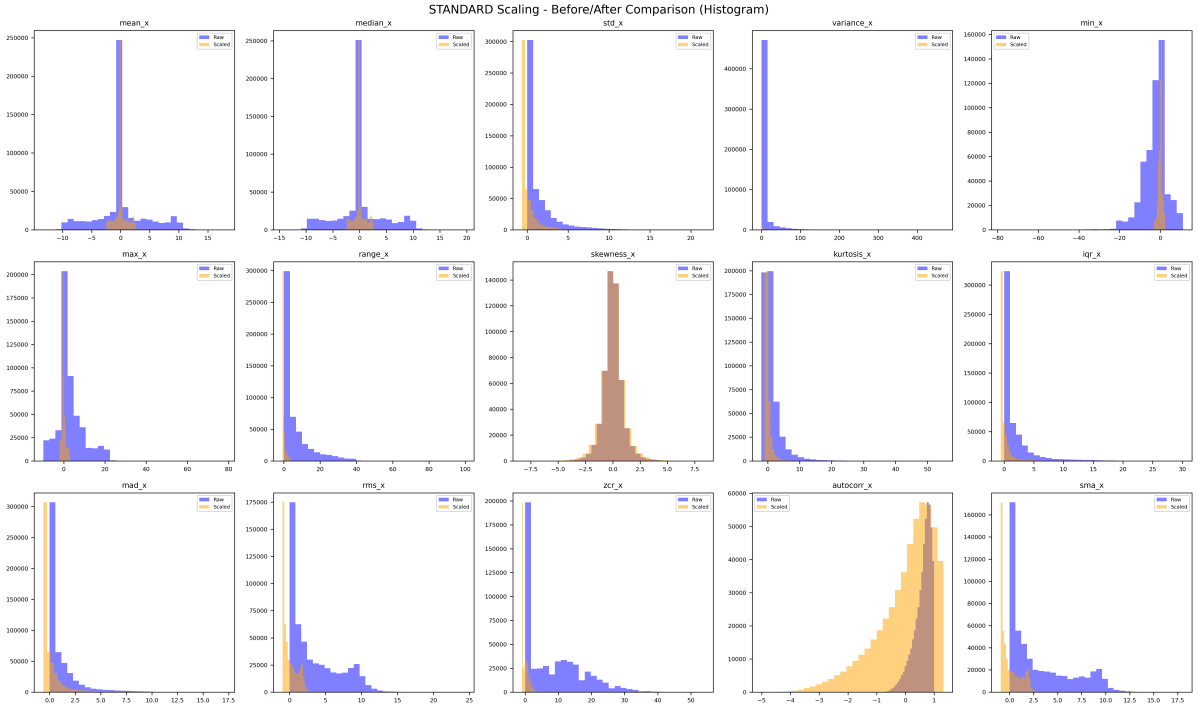

Figure 19: Standard Scaling Histogram: Features transformed to zero mean and unit variance.

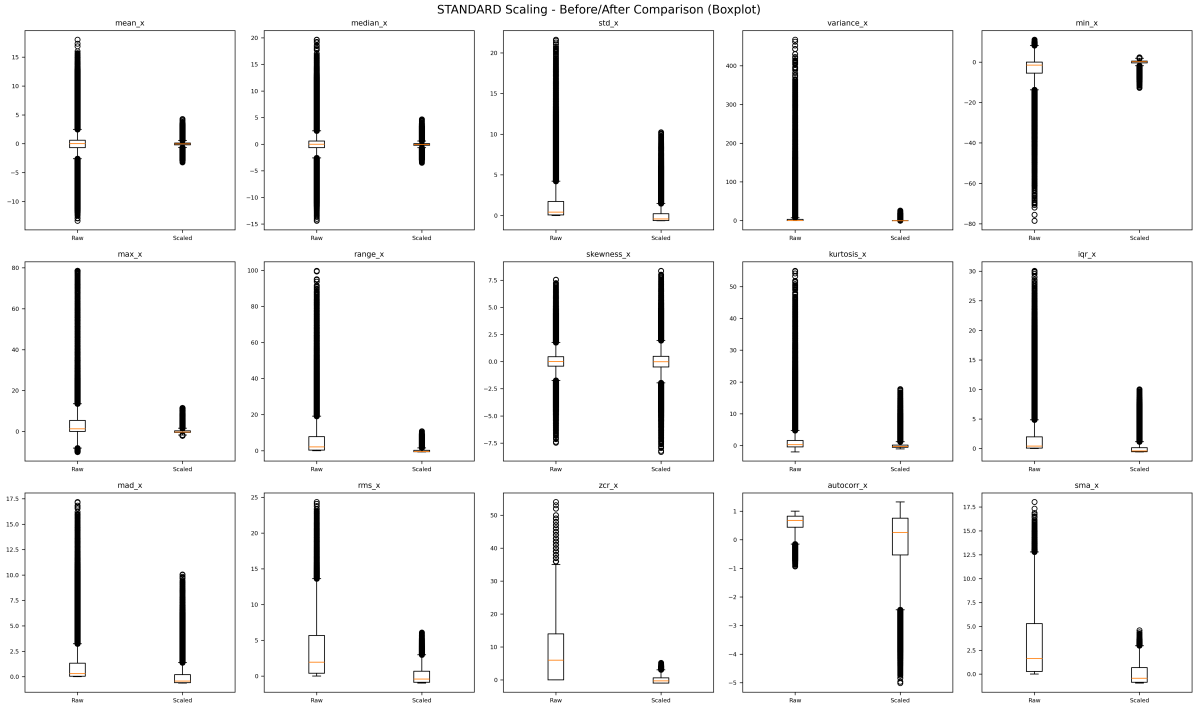

Figure 20: Standard Scaling Boxplot: Shows outliers extending beyond the typical range, which can affect distance-based algorithms.

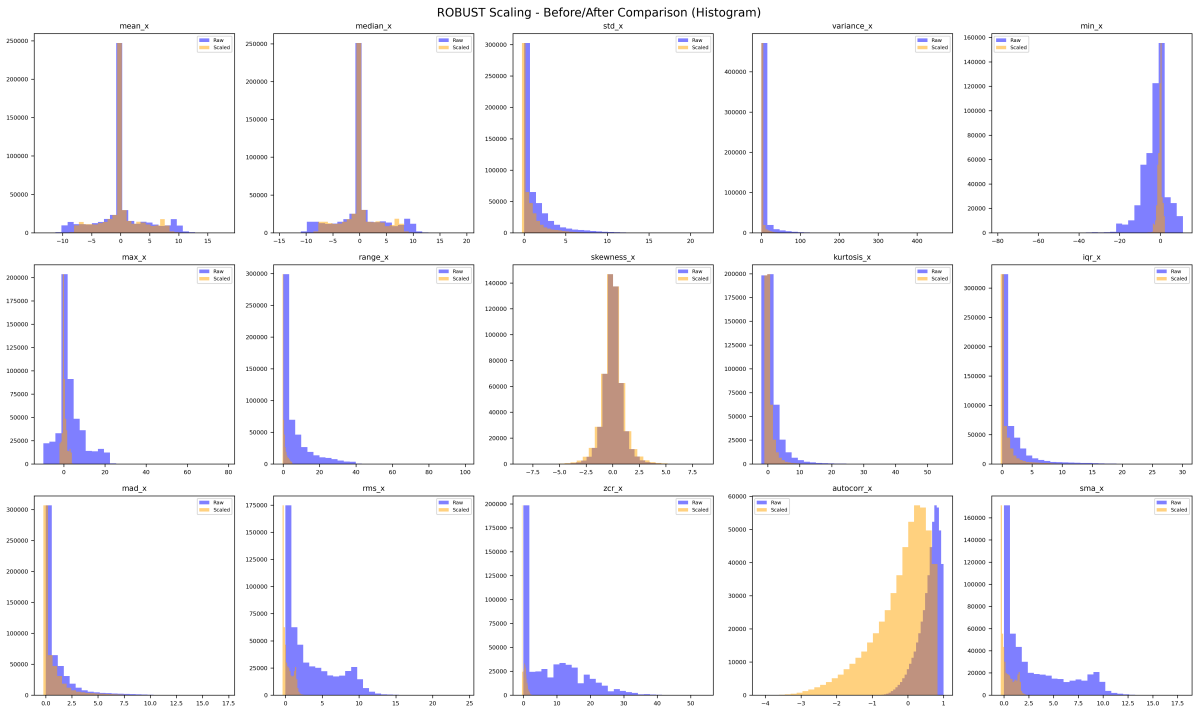

Figure 21: Robust Scaling Histogram: Uses median and IQR, making it resistant to outliers.

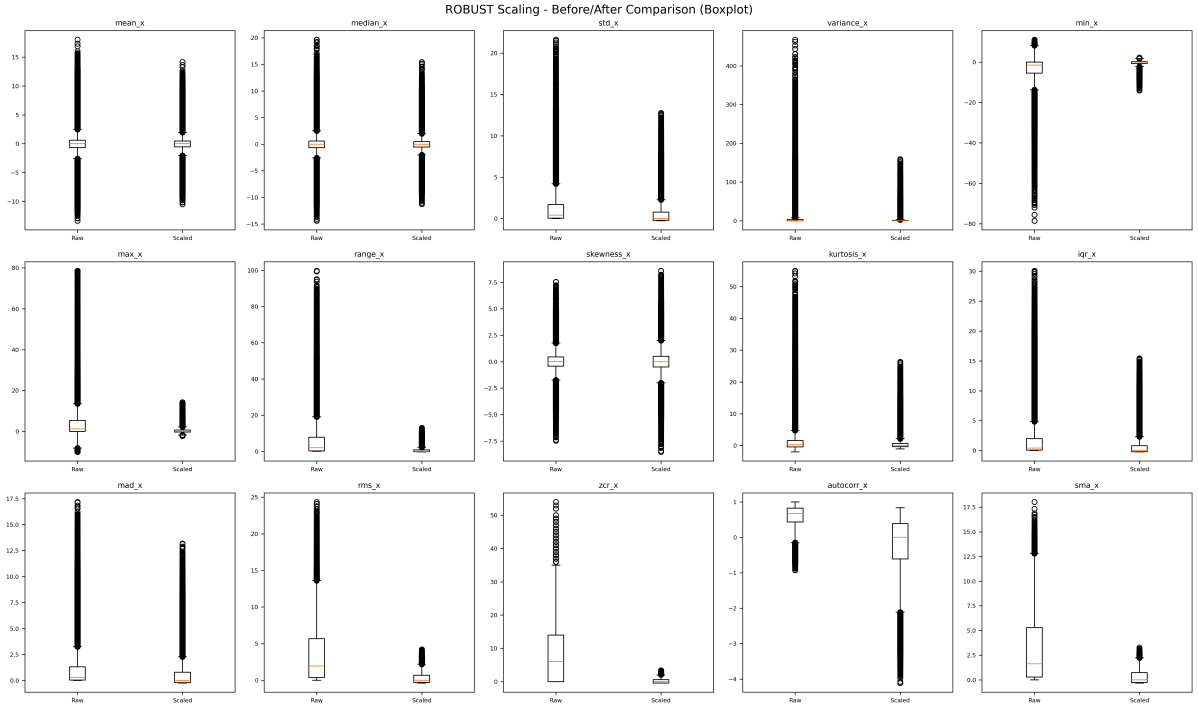

Figure 22: Robust Scaling Boxplot: Shows similar centering to standard scaling but with better outlier handling.

Scaling Method Performance Summary: The visual analysis confirms that Min-Max scaling produces the most bounded and consistent feature distributions, which ex-

plains its superior performance with the diverse feature set in this study.

5.4 Trade-off Analysis: Speed vs. Accuracy

Gradient Boosting Case Study:

- Without early stopping: >4 hours, ∼66–67% accuracy
- With early stopping: ∼51 min, 51.63% accuracy
- Trade-off: 5× speedup, −15% accuracy
- Decision: Acceptable for pipeline efficiency; Random Forest provides better accuracy/speed ratio

6 Implementation Details

6.1 Software Stack

Table 13: Technical Specifications

Component	Version/Details
Python	3.13.3
scikit-learn	Latest stable
pandas	Data manipulation
NumPy	Numerical computing
SciPy	Signal processing, statistics
PyWavelets	Wavelet decomposition
matplotlib/seaborn	Visualization
joblib	Model serialization
tqdm	Progress tracking
Random Seed	42 (reproducibility)
Parallelization	Multi-core CPU (n_jobs=-1)

6.2 Pipeline Usage

```
# Activate virtual environment
source .venv/bin/activate

# Run complete optimized pipeline (~3h 22min)
python run_pipeline.py optimized

# Run individual steps
python run_pipeline.py step 1    # Data cleaning
python run_pipeline.py step 2    # Windowing
python run_pipeline.py step 3    # Basic features
```

```
python run_pipeline.py step 8    # Spectral features
python run_pipeline.py step 6b   # Optimized models
python run_pipeline.py step 6c   # Ensemble models
```

Listing 8: Running the Pipeline

6.3 Output Directory Structure

```
data/
+-- 01_cleaned/              # Cleaned sensor data
|   +-- cleaned_data.csv
+-- 02_windowed/             # Windowed data
|   +-- windowed_data.csv
+-- 03_features/             # Basic time-domain features
|   +-- features_raw.csv
|   +-- feature_descriptions.txt
+-- 03b_advanced_features/   # Wavelet, entropy, jerk
|   +-- advanced_features.csv
+-- 03c_combined/            # All features merged
|   +-- combined_features.csv
+-- 04_scaled/               # Scaled features
|   +-- minmax_scaled.csv
|   +-- standard_scaled.csv
|   +-- robust_scaled.csv
+-- 05_selected/             # Selected features
|   +-- minmax_selected.csv
|   +-- selected_features.txt
+-- 06b_optimized_results/   # Individual models
|   +-- optimized_results.csv
|   +-- KNN_optimized.pkl
|   +-- DecisionTree_optimized.pkl
|   +-- RandomForest_optimized.pkl
|   +-- GradientBoosting_optimized.pkl
+-- 06c_ensemble_results/    # Ensemble models
|   +-- ensemble_results.csv
|   +-- voting_hard.pkl
|   +-- voting_soft.pkl
|   +-- stacking.pkl
+-- 08_spectral/             # Spectral features
    +-- SPECTRAL_FEATURES.csv
    +-- spectral_feature_descriptions.txt
```

7 Conclusions

7.1 Key Findings

1. **Best Configuration**: Multi-feature + MinMax Scaling + Stacking Ensemble achieves **78.26% accuracy**
2. **Feature Engineering Impact**: Combining time, frequency, and advanced features provides **+6.05% improvement** over baseline
3. **Ensemble Superiority**: Stacking outperforms voting ensembles through meta-learned combination
4. **Optimization Success**: **2.5× speedup** (5h → 3h 22min) while improving accuracy
5. **Model Selection Insights**:
   - Random Forest: Best individual model (77.44%)
   - Stacking: Best overall (78.26%)
   - Hard Voting: Best speed/accuracy trade-off for ensembles

7.2 Optimization Techniques Summary

✓ Subsampled hyperparameter tuning (30K samples)

✓ Reduced cross-validation folds (3 instead of 5)

✓ Early stopping for Gradient Boosting

✓ Parallel processing (multi-core CPU)

✓ Removed GB from ensemble base models (too slow)

✓ RandomizedSearchCV over GridSearchCV

✓ Discrete parameters for faster convergence

✓ Caching system for pipeline resumability

Acknowledgments

This work uses the WISDM-51 dataset provided by the Wireless Sensor Data Mining Lab at Fordham University. We thank Gary M. Weiss and colleagues for making this valuable resource publicly available.

A Complete Feature List

A.1 Time-Domain Features (60 total)

For each axis (x, y, z), the following 20 features are extracted:

1. mean_{x,y,z}
2. median_{x,y,z}
3. std_{x,y,z}
4. variance_{x,y,z}
5. min_{x,y,z}
6. max_{x,y,z}
7. range_{x,y,z}
8. skewness_{x,y,z}
9. kurtosis_{x,y,z}
10. iqr_{x,y,z}
11. mad_{x,y,z}
12. rms_{x,y,z}
13. zcr_{x,y,z}
14. autocorr_{x,y,z}
15. sma_{x,y,z}
16. energy_{x,y,z}
17. hjorth_activity_{x,y,z}
18. hjorth_mobility_{x,y,z}
19. hjorth_complexity_{x,y,z}
20. peak_count_{x,y,z}

A.2 Spectral Features (39 total)

For each axis (x, y, z), the following 13 features are extracted:

1. spectral_energy_{x,y,z}
2. spectral_entropy_{x,y,z}
3. spectral_centroid_{x,y,z}
4. spectral_spread_{x,y,z}
5. spectral_flux_{x,y,z}
6. spectral_rolloff_{x,y,z}
7. spectral_flatness_{x,y,z}
8. dominant_frequency_{x,y,z}
9. dominant_amplitude_{x,y,z}
10. bandpower_0_5hz_{x,y,z}
11. bandpower_5_10hz_{x,y,z}
12. periodicity_{x,y,z}
13. harmonic_ratio_{x,y,z}

A.3 Configuration Parameters

```
# Signal processing parameters
SAMPLING_RATE = 20      # Hz
WINDOW_SIZE = 60        # 3 seconds at 20 Hz
HOP_SIZE = 30           # 50% overlap

# Model parameters
RANDOM_STATE = 42
TEST_SIZE = 0.2
NUM_FEATURES = 50       # Features to select

# Spectral feature parameters
SPECTRAL_ROLLOFF_THRESHOLD = 0.85
FREQ_BANDS = {
    'low': (0, 5),
    'mid': (5, 10),
}

# Optimization parameters
N_JOBS = -1             # Use all CPU cores
CHUNK_SIZE = 50000      # For batch processing

# Cross-validation
CV_FOLDS = 5
STACKING_CV = 3
```

Listing 9: Configuration File (config.py)

References

Relevant Paper:

Gary M. Weiss, Kenichi Yoneda, and Thaier Hayajneh. "Smartphone and Smartwatch-Based Biometrics Using Activities of Daily Living." *IEEE Access*, vol. 7, pp. 133190–133202, Sept. 2019.

Citation Request:

Please cite the IEEE Access article: Smartphone and Smartwatch-Based Biometrics Using Activities of Daily Living. *IEEE Access*, vol. 7, pp. 133190–133202, Sept. 2019.