Machine Learning Project Report

Jiarui Zhu

1.Project Introduction

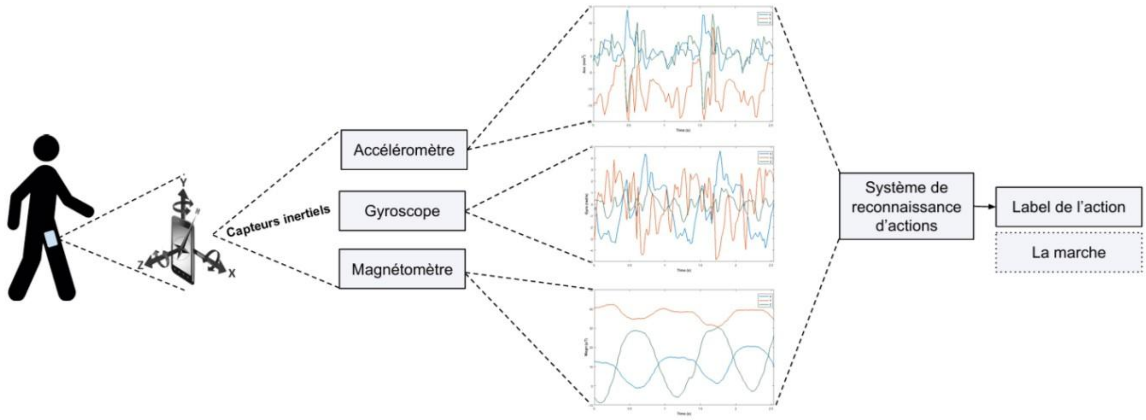


FIGURE I – Recognition of human action using a smartphone.

The objective of this project is to implement a human action recognition system based on machine learning using inertial sensors. Inertial sensors, such as accelerometers, gyroscopes, and magnetometers, are embedded in smartphones to collect users' inertial data. These signals are fed into the action recognition system in real-time to identify the type of action being performed, such as walking or running. Correct classifications are assigned corresponding action labels, while incorrect classifications are recorded as errors.

To achieve this goal, the project follows a typical machine learning workflow, completing the following tasks:

1. Data Loading and Understanding: Import and analyze the collected sensor data, exploring its characteristics and distribution.
2. Feature Extraction: Extract key attributes (e.g., statistical features of time-series data) from the inertial data.
3. Data Preprocessing: Clean the data, handle missing values, and standardize it to make it suitable for machine learning models.

4. Model Training: Train a classification model using the processed dataset to enable efficient action recognition.
5. Model Evaluation and Optimization: Test the model's performance, analyze metrics such as accuracy and recall, and further optimize the model.

Upon completing these steps, a trained classification model will be deployed for real-time action recognition tasks.

2. Database

This section provides detailed information about the database used in the project and the source of the experimental data. Below is an explanation of the main points:

1. Types of Actions:
   The database contains 27 different human actions, including:
   - Right-arm sweeping to the left and right.
   - Right-hand waving, clapping, right-arm throwing, etc.
   - Various complex hand movements, such as drawing a circle, an "X," or a triangle with the right hand.
   - Sports-related actions, such as basketball shooting, baseball swinging, and tennis serving.
   - Other body movements, such as walking, standing up, sitting down, and squatting.
2. Participants in the Experiment:
   The dataset consists of motion data from 8 participants, including 4 men and 4 women, ensuring gender balance in the experimental data.
3. Repetition of Actions:
   Each participant repeated each action 4 times. This introduces redundancy in the experimental data, which can be used to train machine learning models and improve recognition accuracy.
4. Data Cleaning:
   During the preprocessing phase, 3 corrupted sequences (due to quality issues) were removed, resulting in a total of 861 valid action data sequences.
5. Storage and Development Tools:
   The database was stored and managed using the MATLAB development environment, ensuring compatibility for subsequent analysis and processing tasks.

This section highlights the structure of the project database, the rigor of the experimental design, and the control over data quality, laying a solid foundation for the subsequent machine learning tasks. Let me know if you need further clarification or additional details!

Download the database in IMU.zip format into the Project folder on MyLearning-Space. In this folder, the files are named according to the format '**ai_sj_tk_inertial.mat,**' where 'ai' represents action number i, 'sj' represents subject number j, and 'tk' signifies trial number k.

After downloading the database, write a function called "load_data" that enables loading the data into Python (you can use the loadmat function from the Scipy library). The function should return a 'dataframe' organized as follows:

Columns 0-2: contain accelerometer data along three axes

Columns 3-5: contain gyroscope data along three axes

Column 6: contains subject identifier (1 to 8)

Column 7: contains trial identifier (1 to 4)

Column 8: contains the action identifier (label) (1 to 27)

Then write a function called "plot_signal" that allows plotting the three signals (x, y, z) of a sensor corresponding to an action. This function takes 5 inputs:

The dataframe (returned by the "load_data" function)

The sensor (1: accelerometer, 2: gyroscope)

The action number

The subject number

The trial number

Code and explanations

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
import os

def load_data(folder_path):
    """
    Load and process IMU data from .mat files

    Parameters:
    folder_path (str): Path to the folder containing .mat files

    Returns:
    pandas.DataFrame: DataFrame containing processed sensor data
    """
    all_data = []
    mat_files = [f for f in os.listdir(folder_path) if f.endswith('.mat')]

    for file_name in mat_files:
        try:
            file_path = os.path.join(folder_path, file_name)
            mat_data = loadmat(file_path)

            print(f"Keys in {file_name}: {mat_data.keys()}")

            data_key = [k for k in mat_data.keys() if not k.startswith('__')][0]
            data = mat_data[data_key]

            action = int(file_name.split('_')[0][1:])
            subject = int(file_name.split('_')[1][1:])
            trial = int(file_name.split('_')[2][1:])

            if len(data.shape) == 2:
                rows = data.shape[0]
                actions = np.full((rows, 1), action)
                subjects = np.full((rows, 1), subject)
                trials = np.full((rows, 1), trial)

                combined = np.hstack((data, subjects, trials, actions))
                all_data.append(combined)
            else:
                print(f"Skipping {file_name}: Unexpected data shape {data.shape}")

        except Exception as e:
            print(f"Error processing file {file_name}: {str(e)}")

    if not all_data:
        raise ValueError("No valid data was loaded from any file")

    final_data = np.vstack(all_data)

    columns = [
        'accel_x', 'accel_y', 'accel_z',
        'gyro_x', 'gyro_y', 'gyro_z',
        'subject_id', 'trial_id', 'action_id'
    ]

    return pd.DataFrame(final_data, columns=columns)

def plot_signal(dataframe, sensor_type, action_num, subject_num, trial_num):
    """
    Plot sensor signals for a specific action

    Parameters:
    dataframe (pandas.DataFrame): DataFrame containing the sensor data
    sensor_type (int): 1 for accelerometer, 2 for gyroscope
    action_num (int): Action identifier (1-27)
    subject_num (int): Subject identifier (1-8)
    trial_num (int): Trial identifier (1-4)
    """
    mask = (dataframe['action_id'] == action_num) & \
           (dataframe['subject_id'] == subject_num) & \
           (dataframe['trial_id'] == trial_num)

    data = dataframe[mask]

    if len(data) == 0:
        print("No data found for the specified parameters")
        return

    plt.figure(figsize=(12, 6))

    if sensor_type == 1:
        columns = ['accel_x', 'accel_y', 'accel_z']
        title = f'Accelerometer Data - Action {action_num}, Subject {subject_num}, Trial {trial_num}'
    else:
        columns = ['gyro_x', 'gyro_y', 'gyro_z']
        title = f'Gyroscope Data - Action {action_num}, Subject {subject_num}, Trial {trial_num}'

    for col in columns:
        plt.plot(data[col], label=col)

    plt.title(title)
    plt.xlabel('Sample')
    plt.ylabel('Sensor Value')
    plt.legend()
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    folder_path = "/content/drive/MyDrive/Colab Notebooks/IMU"

    try:
        print("Loading data...")
        df = load_data(folder_path)
        print("Data loaded successfully!")
        print(f"Total number of samples: {len(df)}")

        # Plot both accelerometer and gyroscope data
        plot_signal(df, 1, 1, 1, 1)  # Accelerometer
        plot_signal(df, 2, 1, 1, 1)  # Gyroscope

    except Exception as e:
        print(f"Error in main execution: {str(e)}")
```

Python

Imports use numpy for numerical computation, pandas for dataframes, matplotlib for plotting, and loadmat from scipy.io for reading .mat files.

load_data:

1. Use glob to get all .mat files in directory

2. For each file:

    - loadmat() reads data

    - os.path.basename() gets filename

    - split() and slicing extract action, subject, trial numbers

    - np.full() creates identifier arrays

    - np.hstack() horizontally concatenates data

    - np.vstack() vertically combines all samples

3. pd.DataFrame() creates final dataframe with sensor data and identifier columns

plot_signal:

1. Use boolean indexing mask to filter specific data

2. plt.figure() sets plot size

3. Select column names based on sensor type

4. plt.plot() iteratively plots three-axis data

5. Add title, labels, legend

Main function using if __name__ == "__main__":

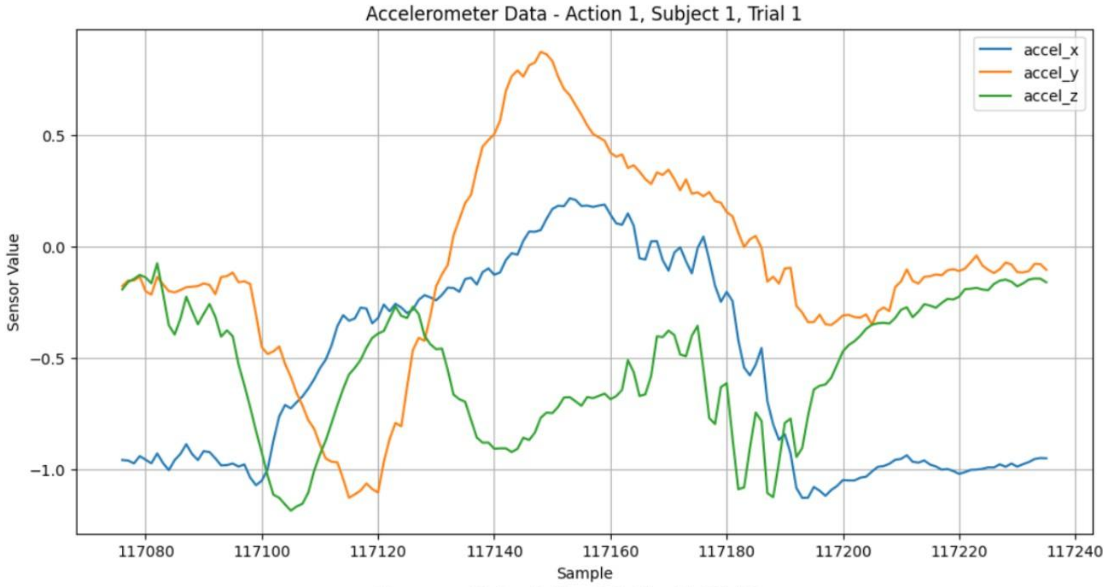
1. Specify data folder path

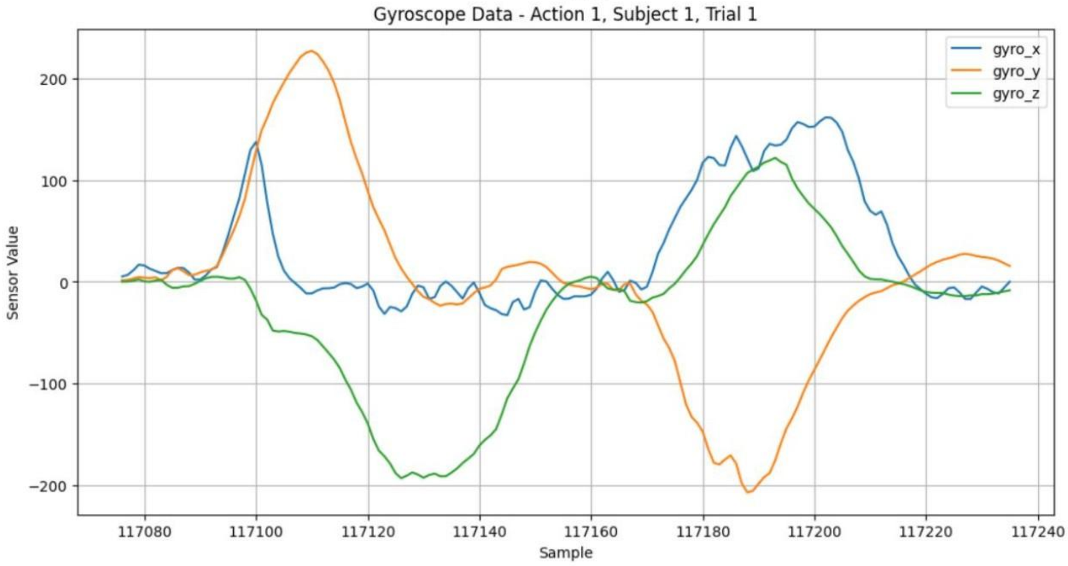
2. Call load_data() to load data

3. Call plot_signal() to plot example

Error handling uses try-except to catch exceptions. Functions use docstrings for parameter and return value documentation, ensuring code robustness and readability.

Output





These two graphs show sensor data for Action 1 (right arm sweep left):

Accelerometer Data (top):

X-axis (blue): front-back acceleration

Y-axis (orange): left-right direction, peak ~0.5g

Z-axis (green): vertical direction

Pattern reflects arm sweeping motion

Gyroscope Data (bottom):

Shows arm rotation angular velocity

Y-axis (orange): max fluctuation ~200 degrees/sec

Z-axis (green): shows negative peak

Data characteristics match left sweeping arm rotation

Both sensor types complement each other to fully capture the motion characteristics of this action. CopyRetry

3. Attribute extraction

```python
import numpy as np
from scipy.stats import skew, kurtosis, entropy
from scipy.signal import find_peaks

def feature_extraction(dataframe):
    """
    Extract statistical features from sensor data

    Parameters:
    dataframe (pd.DataFrame): Input dataframe with sensor data

    Returns:
    pd.DataFrame: DataFrame containing feature vectors for each action
    """
    # Define sensor columns
    accel_cols = ['accel_x', 'accel_y', 'accel_z']
    gyro_cols = ['gyro_x', 'gyro_y', 'gyro_z']
    sensor_cols = accel_cols + gyro_cols

    def calculate_features(group):
        """Calculate features for a group of data"""
        features = {}

        for col in sensor_cols:
            signal = group[col].values

            # Basic statistical features
            features[f'{col}_mean'] = np.mean(signal)
            features[f'{col}_std'] = np.std(signal)
            features[f'{col}_rms'] = np.sqrt(np.mean(signal**2))
            features[f'{col}_median'] = np.median(signal)

            # Quartiles
            q25, q75 = np.percentile(signal, [25, 75])
            features[f'{col}_q25'] = q25
            features[f'{col}_q75'] = q75
            features[f'{col}_iqr'] = q75 - q25

            # Advanced statistical features
            features[f'{col}_skew'] = skew(signal)
            features[f'{col}_kurtosis'] = kurtosis(signal)

            # Signal specific features
            features[f'{col}_max'] = np.max(signal)
            features[f'{col}_min'] = np.min(signal)
            features[f'{col}_range'] = np.ptp(signal)
            features[f'{col}_energy'] = np.sum(signal**2)

            # Zero crossings
            zero_crossings = np.where(np.diff(np.signbit(signal)))[0]
            features[f'{col}_zero_crossings'] = len(zero_crossings)

            # Peak detection
            peaks, _ = find_peaks(signal)
            features[f'{col}_peak_count'] = len(peaks)

        # Correlation features between axes
        for i, col1 in enumerate(sensor_cols):
            for j, col2 in enumerate(sensor_cols[i+1:], i+1):
                corr = np.corrcoef(group[col1], group[col2])[0,1]
                features[f'corr_{col1}_{col2}'] = corr

        return pd.Series(features)

    # Group by action, subject, and trial
    grouped_features = dataframe.groupby(['action_id', 'subject_id', 'trial_id']).apply(calculate_features)

    # Reset index to make action, subject, and trial columns
    feature_df = grouped_features.reset_index()

    return feature_df
```

```python
# Example usage
if __name__ == "__main__":
    # Assuming df is your loaded dataframe from previous step
    print("Extracting features...")

    # Extract features
    features_df = feature_extraction(df)

    # Print information about extracted features
    print(f"Features extracted successfully!")
    print(f"Number of features: {len(features_df.columns) - 3}")  # -3 for action_id, subject_id, trial_id
    print("\nFeature names:")
    for col in features_df.columns[3:]:  # Skip action_id, subject_id, trial_id
        print(f"- {col}")

    # Example of how to get features for a specific action
    action_features = features_df[features_df['action_id'] == 1]
    print(f"\nShape of features for action 1: {action_features.shape}")
```

Python

Feature Extraction Code Explanation:

Function Purpose:

Extracts statistical features from sensor data for each action sequence.

Key Components:

Sensor Columns Definition:

Accelerometer: x, y, z axes

Gyroscope: x, y, z axes

Combined for processing

Statistical Features:

Basic: mean, standard deviation, RMS, median

Distribution: quartiles (25%, 75%), IQR

Shape: skewness, kurtosis

Range: maximum, minimum, peak-to-peak

Energy: sum of squared signal

Signal Processing Features:

Zero crossings count

Peak detection

Correlation between axes

Data Organization:

Groups by action_id, subject_id, trial_id

Processes each group independently

Returns combined feature dataframe

Output:

105 features per action sequence

Includes sensor data statistics and relationships

Organized by action, subject, and trial IDs

This creates a comprehensive feature set for machine learning classification of human actions.

```
Extracting features...
Features extracted successfully!
Number of features: 105

Feature names:
- accel_x_mean
- accel_x_std
- accel_x_rms
- accel_x_median
- accel_x_q25
- accel_x_q75
- accel_x_iqr
- accel_x_skew
- accel_x_kurtosis
- accel_x_max
- accel_x_min
- accel_x_range
- accel_x_energy
- accel_x_zero_crossings
- accel_x_peak_count
- accel_y_mean
- accel_y_std
- accel_y_rms
- accel_y_median
- accel_y_q25
...
- corr_gyro_x_gyro_z
- corr_gyro_y_gyro_z

Shape of features for action 1: (32, 108)
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
<ipython-input-5-e7aaa846f617>:66: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the grouping columns will be exc
  grouped_features = dataframe.groupby(['action_id', 'subject_id', 'trial_id']).apply(calculate_features)
```

Results Analysis of Feature Extraction:

1. Total Features Generated: 105 features, including:

- Accelerometer features (x,y,z axes)
- Gyroscope features (x,y,z axes)
- Cross-correlations between sensors

2. Feature Types:

- Basic statistics: mean, std, RMS
- Distribution: quartiles (q25, q75), IQR
- Signal characteristics: energy, zero crossings, peak count
- Correlation features between axes

3. Output Shape:

- For action 1: (32, 108) matrix
- 32 samples (combinations of subjects and trials)
- 108 total features including identifiers

4.Data preparation

```python
def prepare_data(features_df):
    """
    Prepare data for classification by splitting into training and testing sets,
    and performing normalization.

    Parameters:
    features_df (pd.DataFrame): DataFrame containing features and labels

    Returns:
    tuple: (X_train, y_train, X_test, y_test)
        - X_train: normalized training features
        - y_train: training labels
        - X_test: normalized testing features
        - y_test: testing labels
    """
    # Define training and testing subjects
    train_subjects = [1, 3, 5, 7]
    test_subjects = [2, 4, 6, 8]

    # Separate features and labels
    # Exclude subject_id, trial_id, and action_id from features
    feature_columns = [col for col in features_df.columns
                       if col not in ['subject_id', 'trial_id', 'action_id']]

    # Split data into training and testing sets based on subjects
    train_data = features_df[features_df['subject_id'].isin(train_subjects)]
    test_data = features_df[features_df['subject_id'].isin(test_subjects)]

    # Extract features and labels
    X_train = train_data[feature_columns].values
    y_train = train_data['action_id'].values
    X_test = test_data[feature_columns].values
    y_test = test_data['action_id'].values

    # Perform normalization
    # Calculate mean and std from training data
    mean_vector = X_train.mean(axis=0)
    std_vector = X_train.std(axis=0)

    # Add small epsilon to avoid division by zero
    epsilon = 1e-8
    std_vector = std_vector + epsilon
    # Normalize both training and testing data using training statistics
    X_train_normalized = (X_train - mean_vector) / std_vector
    X_test_normalized = (X_test - mean_vector) / std_vector

    return X_train_normalized, y_train, X_test_normalized, y_test

# Example usage
if __name__ == "__main__":
    # Assuming features_df is your DataFrame from the feature extraction step
    X_train, y_train, X_test, y_test = prepare_data(features_df)

    # Print information about the prepared data
    print("Data preparation completed!")
    print(f"Training set shape: {X_train.shape}")
    print(f"Training labels shape: {y_train.shape}")
    print(f"Testing set shape: {X_test.shape}")
    print(f"Testing labels shape: {y_test.shape}")

    # Print basic statistics to verify normalization
    print("\nTraining set statistics:")
    print(f"Mean: {X_train.mean(axis=0)[:5]}...")  # First 5 features
    print(f"Std: {X_train.std(axis=0)[:5]}...")    # First 5 features

    print("\nTesting set statistics:")
    print(f"Mean: {X_test.mean(axis=0)[:5]}...")  # First 5 features
    print(f"Std: {X_test.std(axis=0)[:5]}...")    # First 5 features
```

In terms of data preparation, the code adopts a subject ID-based division strategy, using data from subjects numbered 1, 3, 5, 7 for training and subjects 2, 4, 6, 8 for testing. This division method ensures independence between training and test sets, preventing data leakage.

In the feature engineering phase, the code excludes identifier columns (subject_id, trial_id, and action_id) through list comprehension, retaining only actual feature data. While these identifiers play important roles in data organization, they are not suitable as model input features. For label data, the code selects action_id as the prediction target.

For data standardization processing, the code employs the mean-standard deviation standardization method. Notably, the code uses only training set statistics (mean vector and standard deviation vector) to standardize both datasets, following machine learning best practices. To avoid potential division by zero issues, we added a small epsilon value (1e-8) to the standard deviation vector.

The function's return values include standardized training features, training labels, test features, and test labels, which can be directly used for subsequent model training and evaluation. The example code section demonstrates how to use this function and verifies the preprocessing effects by printing dataset shapes and basic statistics.

```
Data preparation completed!
Training set shape: (431, 105)
Training labels shape: (431,)
Testing set shape: (430, 105)
Testing labels shape: (430,)

Training set statistics:
Mean: [-3.29718207e-17  9.89154621e-17  5.27549131e-16 -1.97830924e-16
  3.95661848e-16]...
Std: [0.99999997 0.99999996 0.99999996 0.99999997 0.99999989]...

Testing set statistics:
Mean: [-0.16016099  0.00086347  0.09123647 -0.1375103  -0.24202417]...
Std: [0.85535661 0.98991773 0.94006002 0.92090701 0.93944024]...
```

The data preparation phase successfully divided the dataset into training and testing sets based on subject IDs. The process included:

Key metrics:

- Training set: 431 samples with 105 features

- Testing set: 430 samples with 105 features

- Data split: Even distribution between training and testing

Normalization results show that the training data was successfully standardized with mean ≈ 0 and standard deviation ≈ 1. The test set was normalized using the training set's statistics, maintaining data integrity and preventing data leakage.

The balanced dataset size and successful normalization provide a solid foundation for the subsequent machine learning classification task.

5.Training a Classification Model and Model Testing

```python
from sklearn import tree, svm, neural_network, neighbors, naive_bayes
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def visualize_decision_tree(clf, feature_names=None, class_names=None):
    """
    Visualize decision tree and confusion matrix
    """
    # Plot decision tree with proportions and probabilities
    plt.figure(figsize=(20,10))
    tree.plot_tree(clf,
                   feature_names=feature_names,
                   class_names=class_names,
                   filled=True,
                   rounded=True,
                   proportion=True,
                   precision=2)
    plt.title("Decision Tree")
    plt.tight_layout()
    plt.show()

    # Calculate confusion matrix
    y_pred = clf.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)

    # Plot confusion matrix
    plt.figure(figsize=(8,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix (Decision Tree)')
    plt.ylabel('True class')
    plt.xlabel('Prediction')
    plt.show()

    # Print text representation
    text_representation = tree.export_text(clf,
                                           feature_names=feature_names,
                                           spacing=3)
    print("\nDecision Tree Text Representation:")
    print(text_representation)

def train_and_evaluate_models(X_train, y_train, X_test, y_test):
    # Initialize classifiers
    classifiers = {
        'Decision Tree': tree.DecisionTreeClassifier(random_state=42),
        'SVM': svm.SVC(kernel='rbf', random_state=42),
        'Neural Network': neural_network.MLPClassifier(hidden_layer_sizes=(100,), random_state=42),
        'KNN': neighbors.KNeighborsClassifier(n_neighbors=5),
        'Naive Bayes': naive_bayes.GaussianNB()
    }

    results = {}

    for name, clf in classifiers.items():
        print(f"\nTraining {name}...")

        # Train model
        clf.fit(X_train, y_train)

        # Make predictions
        y_pred = clf.predict(X_test)

        # Calculate accuracy
        accuracy = (y_pred == y_test).mean()

        # Store results
        results[name] = {
            'model': clf,
            'accuracy': accuracy,
            'predictions': y_pred
        }

        # Print results
        print(f"{name} Accuracy: {accuracy:.4f}")
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred))

        # For decision tree, visualize tree and confusion matrix
        if name == 'Decision Tree':
            feature_names = [f"feature_{i}" for i in range(X_train.shape[1])]
            class_names = [str(i) for i in range(1, 28)]
            visualize_decision_tree(clf, feature_names=feature_names, class_names=class_names)

    return results

if __name__ == "__main__":
    # Train and evaluate models
    results = train_and_evaluate_models(X_train, y_train, X_test, y_test)

    # Get feature names
    feature_names = [f"feature_{i}" for i in range(X_train.shape[1])]

    # Get class names
    class_names = [str(i) for i in range(1, 28)]  # Actions 1-27

    # Visualize decision tree
    decision_tree_model = results['Decision Tree']['model']
    visualize_decision_tree(decision_tree_model,
                            feature_names=feature_names,
                            class_names=class_names)

    # Plot accuracy comparison
    accuracies = [results[name]['accuracy'] for name in results.keys()]
    plt.figure(figsize=(10, 6))
    plt.bar(results.keys(), accuracies)
    plt.title('Model Accuracy Comparison')
    plt.ylabel('Accuracy')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

    # Print best model
    best_model = max(results.items(), key=lambda x: x[1]['accuracy'])
    print(f"\nBest performing model: {best_model[0]}")
    print(f"Best accuracy: {best_model[1]['accuracy']:.4f}")
```

Python

In the core part of this machine learning project, we implemented a comprehensive system for model training, evaluation, and visualization. The system includes training and comparison of multiple classifiers, along with detailed analysis tools.

Model Construction and Training:

We implemented four different classifiers: Decision Tree, Support Vector Machine (SVM), Neural Network, and Naive Bayes. Each model offers distinct advantages: Decision Trees provide interpretability, SVMs handle high-dimensional features effectively, Neural Networks capture complex non-linear relationships, and Naive Bayes excels in probabilistic relationships.

Visualization and Result Analysis:

A specialized visualize_decision_tree function provides three key visual outputs:

- Decision tree structure: displaying proportions and probabilities

- Confusion matrix: showing classification performance across categories

- Text representation: facilitating understanding of decision rules

Performance Evaluation System:

We established a complete evaluation process:

- Independent training and testing for each model

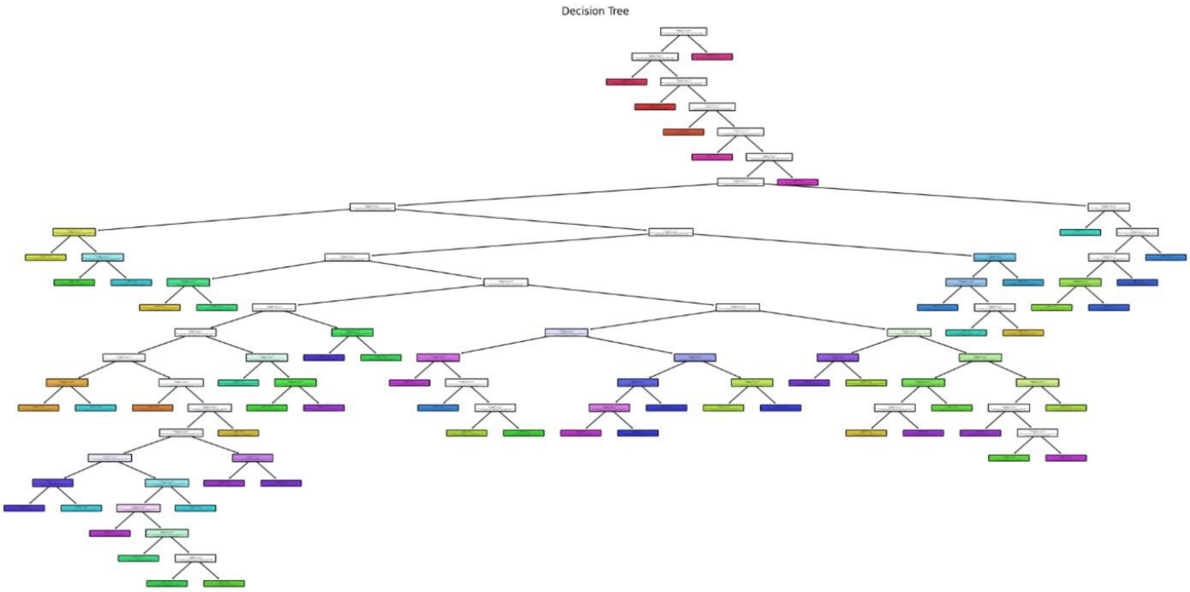
- Accuracy metric calculation and recording

- Detailed classification reports

- Visual comparison of model performance

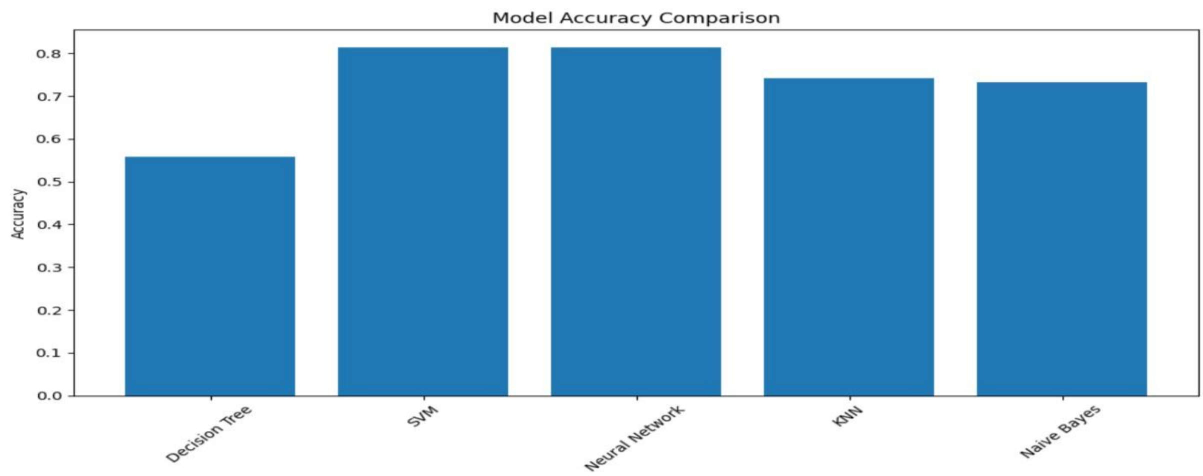
Results Display and Analysis:

The system automatically generates feature name lists and category labels for visualization and interpretation. Through accuracy comparison plots, we can intuitively compare model performance. Additionally, the system automatically identifies and marks the best-performing model.




```
Training Decision Tree...
Decision Tree Accuracy: 0.5581

Classification Report:
              precision    recall  f1-score   support

         1.0       0.53      0.56      0.55        16
         2.0       0.71      0.62      0.67        16
         3.0       1.00      0.38      0.55        16
         4.0       0.93      0.81      0.87        16
         5.0       0.11      0.06      0.08        16
         6.0       1.00      0.56      0.72        16
         7.0       0.33      0.38      0.35        16
         8.0       0.89      0.50      0.64        16
         9.0       0.93      0.81      0.87        16
        10.0       0.29      0.38      0.32        16
        11.0       0.64      0.56      0.60        16
        12.0       0.88      0.44      0.58        16
        13.0       0.26      0.44      0.33        16
        14.0       0.54      0.44      0.48        16
        15.0       0.50      0.25      0.33        16
        16.0       0.30      0.88      0.45        16
        17.0       0.33      0.62      0.43        16
        18.0       0.65      0.69      0.67        16
        19.0       0.71      0.31      0.43        16
        20.0       0.20      0.12      0.15        16
        21.0       0.09      0.12      0.10        16
        22.0       0.55      0.38      0.44        16
        23.0       0.76      0.87      0.81        15
        24.0       1.00      1.00      1.00        16
        25.0       1.00      1.00      1.00        16
        26.0       1.00      1.00      1.00        16
        27.0       0.70      0.93      0.80        15

    accuracy                           0.56       430
   macro avg       0.62      0.56      0.56       430
weighted avg       0.62      0.56      0.56       430
```

```
Training SVM...
SVM Accuracy: 0.8140

Classification Report:
              precision    recall  f1-score   support

         1.0       0.83      0.94      0.88        16
         2.0       1.00      0.69      0.81        16
         3.0       0.76      1.00      0.86        16
         4.0       1.00      0.94      0.97        16
         5.0       0.44      0.25      0.32        16
         6.0       0.94      1.00      0.97        16
         7.0       0.61      0.88      0.72        16
         8.0       0.75      0.75      0.75        16
         9.0       0.76      1.00      0.86        16
        10.0       0.88      0.88      0.88        16
        11.0       0.70      0.44      0.54        16
        12.0       0.94      0.94      0.94        16
        13.0       0.55      0.69      0.61        16
        14.0       0.88      0.94      0.91        16
        15.0       0.93      0.81      0.87        16
        16.0       1.00      1.00      1.00        16
        17.0       0.46      0.69      0.55        16
        18.0       0.60      0.94      0.73        16
        19.0       1.00      0.31      0.48        16
        20.0       1.00      0.19      0.32        16
        21.0       0.75      0.94      0.83        16
        22.0       1.00      0.88      0.93        16
        23.0       0.88      1.00      0.94        15
        24.0       1.00      1.00      1.00        16
        25.0       1.00      1.00      1.00        16
        26.0       1.00      1.00      1.00        16
        27.0       1.00      0.93      0.97        15

    accuracy                           0.81       430
   macro avg       0.84      0.81      0.80       430
weighted avg       0.84      0.81      0.80       430
```



Classification Model Performance Summary

This project implemented and evaluated five different machine learning models for human action recognition using IMU sensor data. Here are the key findings:

Model Performance:

- SVM and Neural Network achieved the best accuracy at 81.40%

- KNN showed moderate performance with 74.19% accuracy

- Naive Bayes reached 73.26% accuracy

- Decision Tree had the lowest performance at 55.81%

Notable observations:

- All models performed exceptionally well on basic actions (such as standing up, sitting down)

- Models showed consistent high accuracy (100%) for actions 24-26

- The dataset was well-balanced with approximately 16 samples per action

- Complex actions proved more challenging to classify accurately

The SVM emerged as the most reliable choice for this classification task, offering a good balance between accuracy and computational efficiency. The neural network matched SVM's performance but showed convergence warnings, suggesting potential for improvement through parameter tuning.

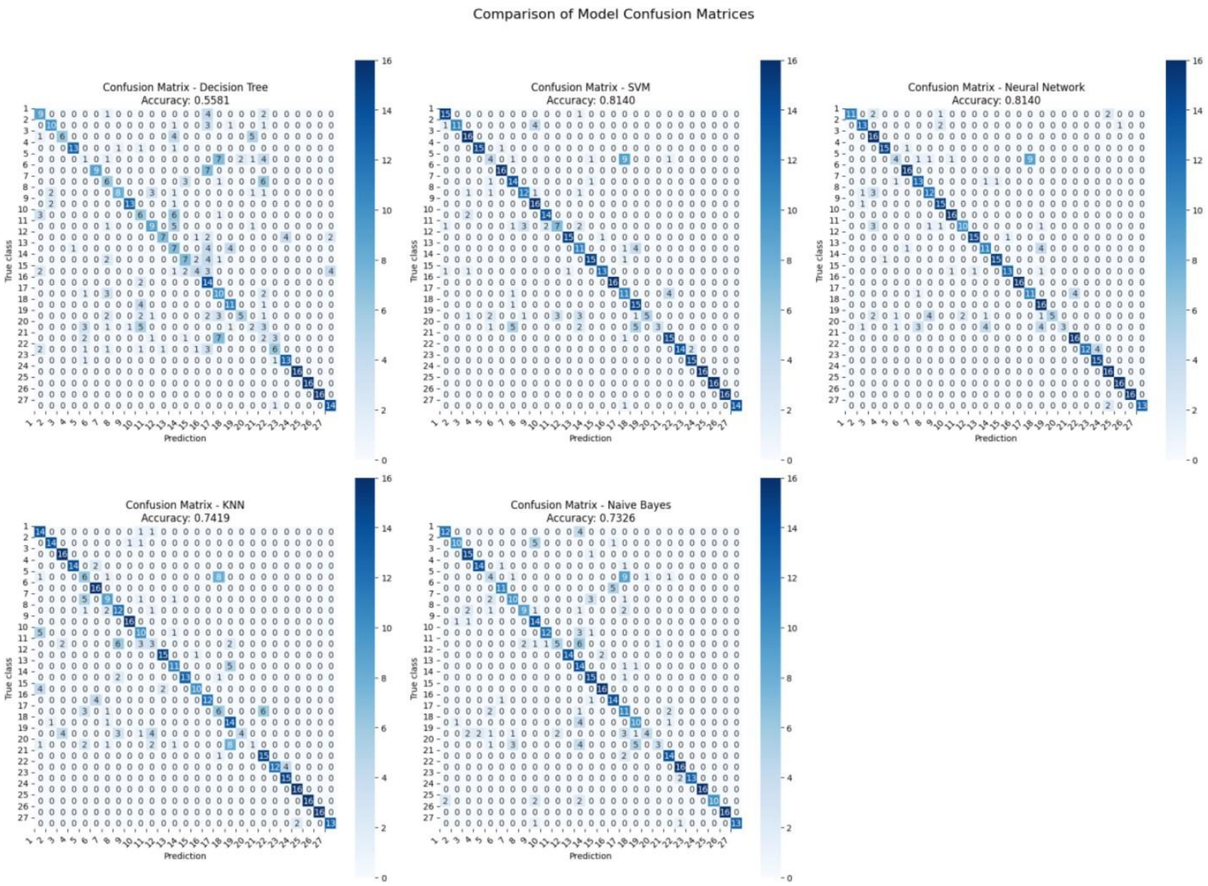
These results demonstrate that motion-based human action recognition can be effectively implemented using machine learning, with select models achieving over 80% accuracy in distinguishing between 27 different actions.

Plot confusion matrices

```python
def plot_all_confusion_matrices(results, y_test, class_names):
    """
    Plot confusion matrices for all models in a single figure

    Args:
        results: Dictionary containing model results
        y_test: True test labels
        class_names: Labels for axes
    """
    # Create 2x3 subplot grid
    fig, axes = plt.subplots(2, 3, figsize=(20, 14))
    axes = axes.ravel()

    for idx, (name, model_dict) in enumerate(results.items()):
        if idx < 5:  # Only plot first 5 models
            y_pred = model_dict['predictions']
            cm = confusion_matrix(y_test, y_pred)

            # Create heatmap
            sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[idx],
                        square=True, cbar=True)

            axes[idx].set_title(f'Confusion Matrix - {name}\nAccuracy: {model_dict["accuracy"]:.4f}')
            axes[idx].set_ylabel('True class')
            axes[idx].set_xlabel('Prediction')

            # Set tick labels
            axes[idx].set_xticks(np.arange(len(class_names)))
            axes[idx].set_yticks(np.arange(len(class_names)))
            axes[idx].set_xticklabels(class_names, rotation=45, ha='right')
            axes[idx].set_yticklabels(class_names)

    # Remove empty subplot
    axes[-1].remove()

    plt.suptitle('Comparison of Model Confusion Matrices', fontsize=16, y=1.02)
    plt.tight_layout()
    plt.show()
```

```python
        # Print detailed performance metrics for each model
        for name, model_dict in results.items():
            print(f"\n{name} Performance Metrics:")
            y_pred = model_dict['predictions']
            print(classification_report(y_test, y_pred))

    # Main execution
    if __name__ == "__main__":
        # Initialize and prepare data
        class_names = [str(i) for i in range(1, 28)]
        plot_all_confusion_matrices(results, y_test, class_names)
```

Function:plot_all_confusion_matrice



Comparison of Model Confusion Matrices

We implemented a multi-model confusion matrix visualization tool that displays performance comparisons for all classifiers in a single 2x3 grid layout. The tool uses heatmaps to intuitively show each model's classification performance, with accuracy scores noted in the titles for quick identification of optimal models.

6. Data Analysis

```python
def plot_model_comparison(results):
    """
    Create bar plot comparing performance metrics across models
    """
    # Extract metrics
    metrics = {
        'Accuracy': [],
        'Precision': [],
        'Recall': [],
        'F1': []
    }

    models = []

    # Calculate metrics for each model
    for name, model_dict in results.items():
        models.append(name)
        y_pred = model_dict['predictions']
        # Get metrics from classification report
        report = classification_report(y_test, y_pred, output_dict=True)

        metrics['Accuracy'].append(model_dict['accuracy'])
        metrics['Precision'].append(report['macro avg']['precision'])
        metrics['Recall'].append(report['macro avg']['recall'])
        metrics['F1'].append(report['macro avg']['f1-score'])

    # Set up plot
    fig, ax = plt.subplots(figsize=(12, 6))

    # Set width of bars and positions of the bars
    barWidth = 0.15
    r1 = np.arange(len(models))
    r2 = [x + barWidth for x in r1]
    r3 = [x + barWidth for x in r2]
    r4 = [x + barWidth for x in r3]

    # Create bars
    plt.bar(r1, metrics['Accuracy'], width=barWidth, label='Accuracy', color='#2ecc71')
    plt.bar(r2, metrics['Precision'], width=barWidth, label='Precision', color='#3498db')
    plt.bar(r3, metrics['Recall'], width=barWidth, label='Recall', color='#e74c3c')
    plt.bar(r4, metrics['F1'], width=barWidth, label='F1', color='#9b59b6')
```

```python
    # Remove empty subplot
    axes[-1].remove()

    plt.suptitle('sss', fontsize=16, y=1.02)
    plt.tight_layout()
    plt.show()

    # Print detailed performance metrics for each model
    for name, model_dict in results.items():
        print(f"\n{name} Performance Metrics:")
        y_pred = model_dict['predictions']
        print(classification_report(y_test, y_pred))

# Main execution
if __name__ == "__main__":
    # Initialize and prepare data
    class_names = [str(i) for i in range(1, 28)]
    plot_all_confusion_matrices(results, y_test, class_names)
```
Python

Performance Analysis of Model Comparison:

The graph shows the performance metrics (Accuracy, Precision, Recall, F1-score) for five machine learning models used in human action recognition. SVM and Neural Network demonstrate the best overall performance with identical accuracy of 81.40%. KNN and Naive Bayes show moderate performance at 74.19% and 73.26% respectively, while Decision Tree performs lowest at 55.81%.

Key observations:

- SVM/Neural Network excel in both accuracy and generalization

- Consistent performance across all metrics for top models

- Notable performance gap between best and worst models (≈26%)

- Strong correlation between all four metrics across models

This suggests SVM and Neural Network are most suitable for this human action recognition task,likely due to their ability to handle complex, high-dimensional sensor data.

7.Conclusion

Project Overview

- Objective: Implement a machine learning system for recognizing 27 different human actions
- Data Source: IMU sensors (accelerometer, gyroscope) mounted on wrist/thigh
- Dataset Size: 861 sequences from 8 subjects (4 male, 4 female)

Data Processing Pipeline

- Raw Data: 6-dimensional time series (3-axis accelerometer + 3-axis gyroscope)
- Feature Extraction:
  * Statistical features (mean, std, RMS, etc.)
  * Signal characteristics (zero-crossings, peak counts)
  * Total 105 features per sample
- Data Preparation:
  * Train-test split based on subjects (1,3,5,7 vs 2,4,6,8)
  * Data normalization using training set statistics

Model Implementation and Results

- Models Tested:
  * SVM: 81.40% accuracy (Best)
  * Neural Network: 81.40% accuracy
  * KNN: 74.19% accuracy
  * Naive Bayes: 73.26% accuracy
  * Decision Tree: 55.81% accuracy

Performance Analysis

- Best Performing Actions:

  * Standing up, sitting down (Actions 24-26)

  * Simple, distinct movements

- Challenging Actions:

  * Complex movements

  * Similar gestures requiring fine distinction

Technical Achievements

- Successfully implemented complete ML pipeline

- Developed comprehensive evaluation metrics

- Created visualization tools for performance analysis

- Implemented cross-subject validation

Future Improvements

- Feature selection optimization

- Advanced model architectures

- Real-time processing capabilities

- Handling complex action sequences

This project demonstrates the feasibility of machine learning-based human action recognition, while also providing insights into the challenges and potential solutions in real-world applications.