Machine Learning Models for Gait Classification Project

BMED 712 - IMU-Based Pathology Classification

📋 Table of Contents

---

1. CLASSICAL MACHINE LEARNING MODELS

Approach: Extract Features → Train Classifier

Pipeline:

```
Raw Time Series → Feature Extraction → Classical ML Model → Prediction
```

A. Random Forest (RF) ⭐ RECOMMENDED FOR BASELINE

Description:

- Ensemble of decision trees
- Votes across multiple trees for final prediction
- Handles high-dimensional data well

Pros:

- ✅ Excellent baseline (fast training)
- ✅ Feature importance built-in
- ✅ Handles class imbalance well (class_weight parameter)
- ✅ No need for feature scaling
- ✅ Interpretable (can visualize trees)
- ✅ Robust to overfitting

Cons:

- ❌ Requires manual feature engineering
- ❌ May miss temporal patterns
- ❌ Large models can be slow at inference

Typical Performance:

- Accuracy: 75-85% (with good features)
- Training time: Minutes

Python Implementation:

```python
from sklearn.ensemble import RandomForestClassifier

# Handle class imbalance
model = RandomForestClassifier(
    n_estimators=200,
    max_depth=20,
    min_samples_split=5,
    class_weight='balanced',  # Handles 3.7:1 imbalance
    random_state=42,
    n_jobs=-1
)

model.fit(X_train, y_train)
```

Feature Engineering Required:

- Time-domain: mean, std, min, max, range, RMS
- Frequency-domain: FFT, power spectral density, dominant frequency
- Statistical: skewness, kurtosis, entropy
- Sensor-specific: step length, stride time, cadence

B. Support Vector Machine (SVM)

Description:

- Finds optimal hyperplane separating classes

- Can use different kernels (linear, RBF, polynomial)

Pros:

- ✅ Effective in high-dimensional spaces
- ✅ Memory efficient
- ✅ Works well with clear margin of separation

Cons:

- ❌ Slow for large datasets (>10k samples)
- ❌ Requires feature scaling
- ❌ Sensitive to hyperparameters
- ❌ Not great for class imbalance

Typical Performance:

- Accuracy: 70-82%
- Training time: Minutes to hours

Python Implementation:

```python
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

# MUST scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

model = SVC(
    kernel='rbf',
    C=1.0,
    gamma='scale',
    class_weight='balanced',
    random_state=42
)

model.fit(X_train_scaled, y_train)
```

C. XGBoost (Extreme Gradient Boosting) ⭐ RECOMMENDED FOR BEST PERFORMANCE

Description:

- Advanced gradient boosting algorithm
- Sequential ensemble learning
- State-of-the-art for tabular data

Pros:

- ✅ Often best performance for structured data
- ✅ Built-in handling of missing values
- ✅ Feature importance
- ✅ Regularization prevents overfitting
- ✅ Handles imbalance with scale_pos_weight

Cons:

- ❌ Requires manual feature engineering
- ❌ Many hyperparameters to tune
- ❌ Can overfit if not careful

Typical Performance:

- Accuracy: 80-90% (with tuning)
- Training time: Minutes

Python Implementation:

```
python
```

```python
import xgboost as xgb

# Calculate scale_pos_weight for imbalance
n_neg = (y_train == 0).sum()
n_pos = (y_train == 1).sum()
scale_pos_weight = n_neg / n_pos  # e.g., 3.7 for your imbalance

model = xgb.XGBClassifier(
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    scale_pos_weight=scale_pos_weight,  # Handles imbalance
    random_state=42
)

model.fit(X_train, y_train)
```

D. k-Nearest Neighbors (kNN)

Description:

- Classifies based on k closest training examples
- Non-parametric, instance-based learning

Pros:

- ✅ Simple to understand
- ✅ No training phase
- ✅ Works well for small datasets

Cons:

- ❌ Slow prediction (searches all training data)
- ❌ Sensitive to feature scaling
- ❌ Poor with high dimensions ("curse of dimensionality")
- ❌ Sensitive to imbalanced data

Not Recommended for This Project - Better alternatives available

---

2. DEEP LEARNING MODELS

Approach: End-to-End Learning from Raw Time Series

Pipeline:

```
Raw Time Series → Deep Neural Network → Prediction
```

A. 1D Convolutional Neural Network (1D-CNN) ⭐ RECOMMENDED

Description:

- Applies convolutional filters along time axis
- Automatically learns relevant features
- Mimics image processing but for time series

Pros:

- ✅ No manual feature engineering
- ✅ Learns hierarchical features automatically
- ✅ Translation invariant (detects patterns anywhere in signal)
- ✅ Computationally efficient
- ✅ Works well for sensor data

Cons:

- ❌ Requires more data than classical ML
- ❌ Longer training time
- ❌ Needs GPU for large models
- ❌ Less interpretable

Typical Performance:

- Accuracy: 82-92%
- Training time: 10-60 minutes (GPU)

Architecture Example:

```
Input (37 channels × Time steps)
   ↓
Conv1D(64 filters, kernel=7) + ReLU + MaxPool
   ↓
Conv1D(128 filters, kernel=5) + ReLU + MaxPool
   ↓
Conv1D(256 filters, kernel=3) + ReLU + MaxPool
   ↓
GlobalAveragePooling
   ↓
Dense(128) + ReLU + Dropout(0.5)
   ↓
Dense(3) + Softmax
```

PyTorch Implementation:

```
python
```

```
import torch
import torch.nn as nn

class GaitCNN(nn.Module):
    def __init__(self, n_channels=37, n_classes=3):
        super().__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv1d(n_channels, 64, kernel_size=7, padding=3),
            nn.ReLU(),
            nn.MaxPool1d(2),

            nn.Conv1d(64, 128, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool1d(2),

            nn.Conv1d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool1d(1)
        )

        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, n_classes)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x

# Training with class weights
model = GaitCNN()
criterion = nn.CrossEntropyLoss(weight=class_weights)  # Handle imbalance
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

B. Long Short-Term Memory (LSTM)

Description:

- Recurrent neural network designed for sequences
- Has memory cells to capture long-term dependencies
- Processes time series sequentially

Pros:

- ✅ Excellent for temporal patterns
- ✅ Captures long-term dependencies
- ✅ Bidirectional option (look forward & backward)

Cons:

- ❌ Slower training than CNN
- ❌ Can be harder to train (vanishing gradients)
- ❌ Sequential processing (can't parallelize)
- ❌ More prone to overfitting

Typical Performance:

- Accuracy: 80-88%
- Training time: 20-90 minutes (GPU)

PyTorch Implementation:

```python
```

```python
class GaitLSTM(nn.Module):
    def __init__(self, n_channels=37, hidden_size=128, n_layers=2, n_classes=3):
        super().__init__()

        self.lstm = nn.LSTM(
            input_size=n_channels,
            hidden_size=hidden_size,
            num_layers=n_layers,
            batch_first=True,
            dropout=0.3,
            bidirectional=True
        )

        # *2 for bidirectional
        self.fc = nn.Linear(hidden_size * 2, n_classes)

    def forward(self, x):
        # x: (batch, time, channels)
        lstm_out, _ = self.lstm(x)
        # Take last time step
        last_output = lstm_out[:, -1, :]
        return self.fc(last_output)
```

C. CNN-LSTM Hybrid

Description:

- CNN extracts local features
- LSTM captures temporal dynamics
- Best of both worlds

Pros:

- ✅ Combines spatial and temporal learning
- ✅ Often best deep learning performance
- ✅ Hierarchical feature learning

Cons:

- ❌ Most complex architecture
- ❌ Longest training time

- ❌ More hyperparameters

Typical Performance:

- Accuracy: 85-93%
- Training time: 30-120 minutes (GPU)

D. Multi-Stream CNN (From Reference Paper) ⭐ PROJECT-SPECIFIC

Description:

- Separate CNN branch for each sensor
- Attention mechanism to weight sensor importance
- Late fusion of sensor streams

Pros:

- ✅ Sensor-specific feature learning
- ✅ Interpretable (shows which sensors matter)
- ✅ Handles missing sensors gracefully
- ✅ Used successfully in your reference paper

Cons:

- ❌ More complex implementation
- ❌ Requires careful architecture design

Architecture:

```
HE Sensor → CNN Branch → Feature Vector (128) ┐
LB Sensor → CNN Branch → Feature Vector (128) ├→ Attention → Fusion → Classifier
LF Sensor → CNN Branch → Feature Vector (128) │
RF Sensor → CNN Branch → Feature Vector (128) ┘
```

E. Transformer (Advanced)

Description:

- Attention-based architecture

- No recurrence or convolution
- Self-attention mechanism

Pros:

- ✅ State-of-the-art for sequences
- ✅ Parallelizable training
- ✅ Captures long-range dependencies

Cons:

- ❌ Requires large datasets (>10k samples)
- ❌ Computationally expensive
- ❌ Many hyperparameters
- ❌ Overkill for this project size

Not Recommended - Dataset too small, unnecessary complexity

---

3. MODEL COMPARISON & RECOMMENDATIONS

Performance vs Complexity Trade-off

Model	Accuracy	Training Time	Interpretability	Complexity	Recommended For
Random Forest	⭐⭐⭐	⚡⚡⚡	⭐⭐⭐	Low	**Baseline**
XGBoost	⭐⭐⭐⭐	⚡⚡	⭐⭐	Medium	**Best Classical**
SVM	⭐⭐	⚡	⭐	Medium	Not recommended
1D-CNN	⭐⭐⭐⭐	⚡⚡	⭐	Medium	**Best DL Start**
LSTM	⭐⭐⭐	⚡	⭐	Medium	Temporal focus
CNN-LSTM	⭐⭐⭐⭐⭐	⚡	⭐	High	Advanced
Multi-Stream CNN	⭐⭐⭐⭐⭐	⚡	⭐⭐⭐	High	**Project-Specific**

---

4. IMPLEMENTATION STRATEGY

Recommended Phased Approach

Phase 1: Baseline (Week 1)

Goal: Quick baseline to understand data

```
Model: Random Forest
Features: Simple statistical (mean, std, min, max per channel)
Expected Accuracy: 75-80%
Time: 1-2 days
```

Why?

- Fast to implement
- Establishes baseline
- Tests data pipeline
- Identifies obvious issues

Phase 2: Optimized Classical ML (Week 2)

Goal: Best classical ML performance

```
Model: XGBoost
Features: Comprehensive (time + frequency domain)
Expected Accuracy: 80-88%
Time: 3-4 days
```

Feature Engineering:

```python
```

```python
def extract_features(signal):
    features = {}

    # Time domain
    features['mean'] = np.mean(signal, axis=0)
    features['std'] = np.std(signal, axis=0)
    features['min'] = np.min(signal, axis=0)
    features['max'] = np.max(signal, axis=0)
    features['range'] = features['max'] - features['min']
    features['rms'] = np.sqrt(np.mean(signal**2, axis=0))
    features['skewness'] = skew(signal, axis=0)
    features['kurtosis'] = kurtosis(signal, axis=0)

    # Frequency domain (for each channel)
    for i in range(signal.shape[1]):
        fft = np.fft.fft(signal[:, i])
        features[f'fft_mean_{i}'] = np.mean(np.abs(fft))
        features[f'dominant_freq_{i}'] = np.argmax(np.abs(fft))

    return np.concatenate([v.flatten() for v in features.values()])
```

Phase 3: Deep Learning (Week 3)

Goal: End-to-end learning, best performance

Model: 1D-CNN or Multi-Stream CNN
Input: Raw time series (37 channels)
Expected Accuracy: 85-92%
Time: 5-7 days

Key Considerations:

1. **Data Augmentation:**

python

```
def augment_signal(signal):
    # Time warping
    # Magnitude warping
    # Adding Gaussian noise
    # Time shifting
    return augmented_signal
```

2. **Class Imbalance Handling:**

```
python

# Option 1: Class weights in loss
class_weights = torch.tensor([1.0, 3.7, 1.0])
criterion = nn.CrossEntropyLoss(weight=class_weights)

# Option 2: Weighted sampling
from torch.utils.data import WeightedRandomSampler
sampler = WeightedRandomSampler(weights, num_samples, replacement=True)
```

Sensor Ablation Study (All Models)

Test all 15 sensor combinations:

Configuration	Sensors Used	Hypothesis
All	HE+LB+LF+RF	Baseline
Single	HE only	Axial control
Single	LB only	Center of mass
Single	LF only	Left limb
Single	RF only	Right limb
Dual - Axial	HE+LB	Trunk dynamics
Dual - Feet	LF+RF	Bilateral comparison
Dual - Mixed	HE+RF	From paper (PD)
Triple	HE+LB+LF	Left-focused
Triple	HE+LB+RF	Right-focused
Triple	HE+LF+RF	Head + feet
Triple	LB+LF+RF	No head

📊 FINAL RECOMMENDATIONS FOR YOUR PROJECT

Option A: Fast & Robust (Recommended for tight deadline)

Week 1: Random Forest baseline (75-80%)
Week 2: XGBoost optimization (82-88%)
Week 3: Sensor ablation + analysis
Week 4: Report writing

Pros: Guaranteed results, interpretable, fast

Option B: High Performance (Recommended for learning)

Week 1: Random Forest baseline (75-80%)
Week 2: 1D-CNN (85-90%)

Week 3: Multi-Stream CNN + ablation (88-92%)
Week 4: Analysis + report

Pros: Best accuracy, learn deep learning, paper-like results

Option C: Comprehensive Comparison (Recommended for A+ grade)

Week 1-2: Both RF and XGBoost
Week 3-4: 1D-CNN and Multi-Stream CNN
Result: Compare 4 models across 15 sensor configs
= 60 experiments with full analysis

Pros: Most rigorous, publication-quality, demonstrates understanding

---

🎯 MY RECOMMENDATION

Start with: Random Forest (Day 1-2) **Then:** 1D-CNN (Day 3-7) **Finally:** Multi-Stream CNN with Attention (Day 8-14)

This gives you:

- ✅ Quick baseline to catch issues early
- ✅ Strong deep learning result
- ✅ Interpretable attention mechanism
- ✅ Matches reference paper approach
- ✅ Sensor ablation insights
- ✅ Publishable results

Time allocation:

- Data pipeline: 2 days
- RF baseline: 1 day
- Feature engineering (for RF): 1 day
- 1D-CNN: 3 days
- Multi-Stream CNN: 4 days
- Experiments & analysis: 3 days
- Report: 2 days

Total: ~16 days = realistic for 3-4 week project

💡 PRO TIPS

1. **Always start simple** - Don't jump to complex models
2. **Fix class imbalance** - Use class weights from Day 1
3. **Use stratified splits** - Never random splits!
4. **Track everything** - Log all experiments (use Weights & Biases or MLflow)
5. **Validate properly** - Leave-one-subject-out CV is gold standard
6. **Report right metrics** - Balanced accuracy, F1-score, per-class metrics
7. **Visualize errors** - Confusion matrix, t-SNE plots
8. **Document early** - Write as you go, not at the end

📚 Python Libraries You'll Need

```bash
# Classical ML
pip install scikit-learn xgboost

# Deep Learning
pip install torch torchvision  # PyTorch
# or
pip install tensorflow keras  # TensorFlow

# Data processing
pip install pandas numpy scipy

# Visualization
pip install matplotlib seaborn

# Feature extraction (optional)
pip install tsfresh

# Experiment tracking (optional but recommended)
pip install wandb
```

🚀 Ready to start coding!

Let me know which approach you want to pursue and I can help you implement it!

Would you like me to:

1. Create a complete Random Forest baseline implementation?
2. Build a 1D-CNN in PyTorch?
3. Implement the Multi-Stream CNN from the paper?
4. Create a complete experiment tracking system?

Just ask! 🎯