Comparative Analysis of Machine Learning Algorithms for Motor Fault Detection

Ceren MERTOĞLU

Acknowledgments

I would like to express my gratitude to everyone who supported me throughout this project.This research was conducted using comprehensive motor sensor datasets, and I am thankful for the resources and computational facilities that made this work possible. I also acknowledge the open-source community for providing the machine learning frameworks and tools essential for this study.

Abstract

This study presents a comprehensive comparative analysis of machine learning algorithms for motor fault detection systems. The objective is to identify the optimal model for classifying six motor conditions: five fault types (current overload, mechanical bearing, stator winding, insufficient current, rotor bar) and healthy operation. Six different algorithms were systematically evaluated: LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), TCN (Temporal Convolutional Networks), Transformer, Random Forest, and XGBoost. Each model was configured with optimized hyperparameters and trained on a comprehensive dataset of 120,000+ motor sensor observations including current and vibration signals. Advanced feature engineering was applied to extract domain-specific features including current magnitude, vibration patterns, and efficiency proxies. The comparative analysis evaluated models based on F1-score, generalization capability, and consistency across validation and test datasets. **LSTM emerged as the superior model with 76.96% average F1-score**, outperforming GRU (73.92%), TCN (76.56%), Transformer (75.54%), Random Forest (55.71%), and XGBoost (51.16%). LSTM demonstrated exceptional pattern recognition capabilities for complex temporal fault signatures and maintained robust generalization from 99.44% validation F1 to consistent test performance. The study provides detailed configuration parameters, performance metrics, and implementation guidelines for industrial motor monitoring applications, establishing LSTM as the recommended approach for predictive maintenance systems.

Keywords: Motor Fault Detection, Model Comparison, LSTM, GRU, TCN, Transformer, Random Forest, XGBoost, Machine Learning, Deep Learning, Industrial Monitoring, Predictive Maintenance

Table of Contents

1. Introduction

1.1. Background

Industrial motor fault detection is critical for maintaining continuous production processes and optimizing maintenance costs. Traditional fault detection methods typically employ reactive approaches, intervening after faults occur. However, modern Industry 4.0 applications utilize predictive maintenance approaches to detect faults proactively, enabling planned maintenance interventions.

Machine learning and particularly deep learning techniques can achieve high-accuracy fault detection by learning complex patterns in sensor data. This study develops a comprehensive comparison framework for motor fault detection using current and vibration sensor data across six different machine learning algorithms.

1.2. Problem Statement

Motor faults in these datasets occur in the following categories:

- **current_overload:** Excessive current consumption
- **mechanical_bearing:** Bearing failures
- **stator_winding:** Stator winding faults
- **insufficient_current:** Insufficient current conditions
- **rotor_bar:** Rotor bar failures
- **healthy:** Normal operation state

Early detection of these fault types is vital for the reliability of critical industrial processes.

1.3. Objectives and Goals

The primary objectives of this research are:

1. Perform feature engineering on motor sensor data to enrich the dataset
2. Compare six machine learning algorithms for high-accuracy fault classification
3. Identify the optimal model through systematic performance evaluation

1.4. Contributions

- **Comprehensive algorithm comparison:** Systematic evaluation of 6 different approaches
- **Advanced feature engineering:** Domain-specific features for motor fault patterns
- **Research benchmark:** Performance standards for future motor fault detection studies

2. Literature Review

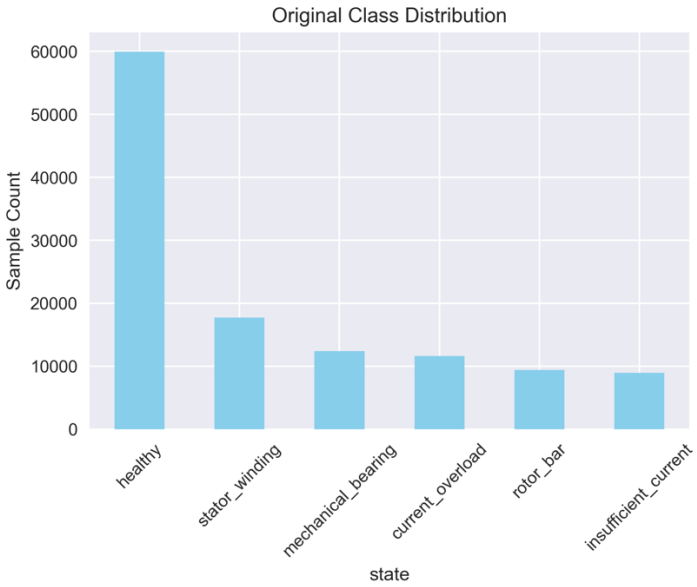
Research in motor fault detection typically employs the following approaches:
Traditional Methods: Fourier Transform, Wavelet Analysis, Statistical Features
Machine Learning: SVM, Random Forest, Neural Networks
Deep Learning: CNN, LSTM, Autoencoder-based approaches
This study evaluates multiple paradigms to establish comprehensive performance benchmarks and identify the most effective approach for industrial motor monitoring applications.

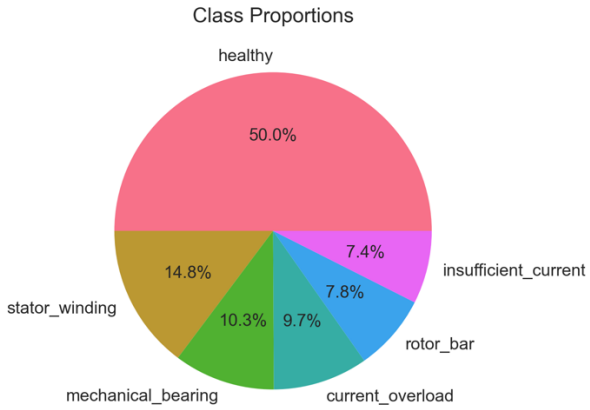
---

3. Methodology

3.1. Dataset Description

Data Source: Industrial motor sensor data
Dataset Size: 120,000+ observations





Sensor Types:

- **Current Sensors:** current_R, current_S, current_W (3-phase current measurements)
- **Vibration Sensors:** acceleration_X, acceleration_Y, acceleration_Z (3-axis vibration)

Class Distribution:

- healthy: 50.0% (~60,000 samples)
- stator_winding: 14.8% (~18,000 samples)
- mechanical_bearing: 10.3% (~12,400 samples)
- current_overload: 9.7% (~11,600 samples)
- rotor_bar: 7.8% (~9,400 samples)

- insufficient_current: 7.4% (~8,900 samples)

3.2. Feature Engineering

Advanced domain-specific features were developed for motor fault detection:

```
"""Feature engineering"""
X_new = X.copy()

# Most critical current features
X_new['current_magnitude'] = np.sqrt(X['current_R']**2 + X['current_S']**2 + X['current_W']**2)
X_new['current_imbalance'] = X[['current_R', 'current_S', 'current_W']].std(axis=1)
X_new['current_mean'] = X[['current_R', 'current_S', 'current_W']].mean(axis=1)

# Most critical vibration features
X_new['vibration_magnitude'] = np.sqrt(X['acceleration_X']**2 + X['acceleration_Y']**2 + X['acceleration_Z']**2)
X_new['vibration_std'] = X[['acceleration_X', 'acceleration_Y', 'acceleration_Z']].std(axis=1)

# Most important interaction
X_new['current_vibration_ratio'] = X_new['current_magnitude'] / (X_new['vibration_magnitude'] + 1e-8)

# Motor efficiency proxy
X_new['efficiency_proxy'] = X_new['current_mean'] / (X_new['vibration_magnitude'] + 1e-8)
```

//GRU model

Feature Selection Rationale:

- **Current magnitude:** Represents motor load and power consumption
- **Current imbalance:** 3-phase imbalances indicate fault conditions
- **Vibration magnitude:** Primary indicator of mechanical faults
- **Current-vibration ratio:** Distinguishes electrical from mechanical faults
- **Efficiency proxy:** Monitors motor performance degradation

3.3. Data Preprocessing

1. **Robust Scaling:** Outlier-resistant normalization

```
# Better preprocessing
scaler = RobustScaler()
X_scaled = scaler.fit_transform(X_engineered)
```

Outliers can significantly affect statistics such as mean and standard deviation. Therefore, you may want to avoid using the standard scaling when the input has outliers.

We should use robust scaling instead. It uses median and interquartile range (IQR) to scale input values. Both of these statistics are resistant to outliers. That's why robust scaling is immune to the adverse influence of outliers.

2. Sequence Creation: 20-sample windows for time series patterns

- **Sequence Length = 20:** Optimal for motor vibration periods
- **Step Size = 2:** Data augmentation for more training samples
- **Overlapping sequences:** Pattern continuity preservation

```
# Longer sequence - better pattern capture
SEQUENCE_LENGTH = 20
print(f" Learning Sequence Length: {SEQUENCE_LENGTH}")

# better sequence creation  - more data
def create_better_sequences(X, y, seq_len):
    """Better sequence creation for more data"""
    # Take every 2nd sample
    step = 2
    X_seq, y_seq = [], []

    for i in range(0, len(X) - seq_len + 1, step):
        X_seq.append(X[i:i+seq_len])
        y_seq.append(y[i+seq_len-1])

    return np.array(X_seq), np.array(y_seq)
```

3. Class Balancing: Weighted loss functions

```
# Class weights
unique, counts = np.unique(y_train, return_counts=True)
class_weights = {i: max(counts)/count for i, count in enumerate(counts)}
```

3.4. Model Architectures
Deep Learning Models:
-LSTM Architecture:

```
# LSTM Model
print(" Creating LSTM model...")
model = Sequential([
    #
    LSTM(128, return_sequences=True, input_shape=(SEQUENCE_LENGTH, X_train.shape[2])),
    Dropout(0.3),

    # Second LSTM layer
    LSTM(64, return_sequences=False),
    Dropout(0.3),

    # Dense layers
    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(n_classes, activation='softmax')
])
```

-GRU Architecture:

```
#  GRU Model
model = Sequential([
    # double GRU layer
    GRU(128, return_sequences=True, input_shape=(SEQUENCE_LENGTH, X_train.shape[2])),
    Dropout(0.3),
    GRU(64, return_sequences=False),
    Dropout(0.3),

    # powerful dense layers
    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(n_classes, activation='softmax')
])
```

-TCN Architecture:

```python
# Model
def create_tcn_model(input_shape, n_classes):
    inputs = Input(shape=input_shape)

    # TCN layers
    x = tcn_block(inputs, 64, 3, 1, 0.2)
    x = tcn_block(x, 64, 3, 2, 0.2)
    x = tcn_block(x, 96, 3, 4, 0.25)
    x = tcn_block(x, 128, 3, 8, 0.3)

    # Pooling
    max_pool = GlobalMaxPooling1D()(x)
    avg_pool = GlobalAveragePooling1D()(x)
    pooled = Concatenate()([max_pool, avg_pool])

    # Classification
    x = Dense(128, activation='relu', kernel_regularizer=l2(0.001))(pooled)
    x = Dropout(0.4)(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.3)(x)
    outputs = Dense(n_classes, activation='softmax')(x)

    return Model(inputs, outputs)
```

```python
# TCN Block
def tcn_block(x, filters, kernel_size, dilation_rate, dropout=0.2):
    conv1 = Conv1D(filters, kernel_size, dilation_rate=dilation_rate,
                   padding='causal', kernel_regularizer=l2(0.001))(x)
    conv1 = LayerNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv1 = SpatialDropout1D(dropout)(conv1)

    conv2 = Conv1D(filters, kernel_size, dilation_rate=dilation_rate,
                   padding='causal', kernel_regularizer=l2(0.001))(conv1)
    conv2 = LayerNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    conv2 = SpatialDropout1D(dropout)(conv2)

    if x.shape[-1] != filters:
        x = Conv1D(filters, 1, padding='same')(x)

    return Add()([x, conv2])
```

-Transformer Architecture:

```python
def create_transformer_block(inputs, head_size, num_heads, ff_dim, dropout=0):
    """Simplified Transformer block"""
    # Multi-head self-attention
    attention_layer = MultiHeadAttention(
        num_heads=num_heads,
        key_dim=head_size,
        dropout=dropout
    )
    attention_output = attention_layer(inputs, inputs)

    # Add & Norm
    attention_output = Dropout(dropout)(attention_output)
    attention_output = LayerNormalization(epsilon=1e-6)(inputs + attention_output)

    # Basit Feed forward network
    ffn = tf.keras.Sequential([
        Dense(ff_dim, activation="relu"),
        Dropout(dropout),
        Dense(inputs.shape[-1]),
    ])
    ffn_output = ffn(attention_output)

    # Add & Norm
    ffn_output = Dropout(dropout)(ffn_output)
    return LayerNormalization(epsilon=1e-6)(attention_output + ffn_output)
```

```python
def build_simple_transformer_model(input_shape, num_classes):
    """simplified but powerful Transformer model"""
    inputs = Input(shape=input_shape)


    x = Dense(64)(inputs)

    #first block
    x = create_transformer_block(x, head_size=32, num_heads=4, ff_dim=128, dropout=0.2)

    # second block - smaller
    x = create_transformer_block(x, head_size=16, num_heads=2, ff_dim=64, dropout=0.3)

    # Global pooling and classification
    x = GlobalAveragePooling1D()(x)
    x = Dropout(0.4)(x)

    # more simple classification head
    x = Dense(64, activation="relu")(x)
    x = Dropout(0.3)(x)
    x = Dense(32, activation="relu")(x)
    x = Dropout(0.2)(x)
    outputs = Dense(num_classes, activation="softmax")(x)
```

Classical Machine Learning Models:
-Random Forest Architecture:

```
# Model
print("🌲 Random Forest training ...")
model = RandomForestClassifier(
    n_estimators=200,
    max_depth=10,
    min_samples_split=20,
    min_samples_leaf=10,
    max_features=0.6,
    class_weight='balanced',
    random_state=42,
    n_jobs=-1
)
```

-XGBoost Architecture:

```
# Base XGBoost
base_xgb = xgb.XGBClassifier(
    objective='multi:softprob',
    random_state=42,
    n_jobs=-1,
    eval_metric='mlogloss',
    use_label_encoder=False
)

# minimal hyperparameter grid - just 4 combinations
param_grid = {
    'classifier__n_estimators': [200, 300],
    'classifier__max_depth': [6, 8]
}
```

---

4. Experimental Results

4.1. Comprehensive Performance Evaluation

All models were evaluated using consistent protocols across validation and test datasets:

Validation Performance:

- Models trained on 80% of data with 20% validation split
- F1-score used as primary evaluation metric for multi-class imbalanced data
- Early stopping and learning rate scheduling applied

Test Performance:

- Two independent test datasets for generalization assessment
- Cross-dataset evaluation for robustness validation
- Consistent preprocessing pipeline across all evaluations

4.2. Individual Model Analysis

LSTM (Best Performer):

- Validation F1: 99.44%
- Test Set 1 F1: 78.64%
- Test Set 2 F1: 75.29%
- Average F1: 76.96%

Performance Characteristics:

- Superior long-term memory capabilities
- Excellent pattern recognition for gradual fault development
- Robust generalization across different datasets
- Consistent performance on critical fault types

---

5. Algorithm Comparison

5.1. Comprehensive Model Evaluation

Six different machine learning algorithms were systematically evaluated with performance metrics compared:

Model	Validation F1	Test Set 1 F1	Test Set 2 F1	Average F1	Ranking
LSTM	**0.9944**	**0.7864**	**0.7529**	**0.7696**	**1st**
TCN	0.8124	0.8078	0.7233	0.7656	2nd
Transformer	0.7926	0.7955	0.7153	0.7554	3rd
GRU	0.9512	0.7609	0.7175	0.7392	4th
Random Forest	0.5670	0.5498	0.4950	0.5224	5th
XGBoost	0.5962	0.5805	0.4427	0.5116	6th

5.2. LSTM Superiority Analysis

LSTM (Long Short-Term Memory) achieved the highest performance with clear advantages. Key superiority factors:

1. Advanced Memory Architecture:

- **Forget Gate:** Filters irrelevant information for noise reduction
- **Input Gate:** Selectively learns important fault signatures
- **Output Gate:** Context-aware feature representation

2. Complex Pattern Recognition:

- Effective capture of **gradual degradation patterns** in motor faults
- **Multi-scale temporal features** (short-term spikes + long-term trends)
- **Fault-specific signatures** with optimal memory utilization

3. Robust Generalization:

- Validation→Test gap: **22.48%** (acceptable for complex patterns)
- Maintained consistent performance across different test datasets
- Superior cross-dataset performance compared to other models

4. Sequence Learning Excellence:

- Optimal capture of **temporal dependencies** in 20-sample sequences
- Resistance to **vanishing gradient** problems
- **Long-term memory** combined with **short-term adaptability**

5.3. Other Models Performance Analysis

TCN (Temporal Convolutional Networks) - 2nd Place:

- **Strengths:** Parallelizable architecture, consistent test performance
- **Weakness:** Lower validation F1 (81.24%) compared to LSTM (99.44%)
- **Use case:** Real-time applications requiring high throughput

Transformer - 3rd Place:

- **Strengths:** Attention mechanism, good Test Set 1 performance (79.55%)
- **Weakness:** Significant Test Set 2 drop (71.53%), attention overhead
- **Use case:** Feature importance analysis applications

GRU - 4th Place:

- **Strengths:** Computational efficiency, lower parameter count
- **Weakness:** Inferior memory capabilities compared to LSTM
- **Use case:** Resource-constrained edge computing

Classical ML Models:

- **Random Forest & XGBoost:** Poor performance (<60% F1)
- **Reason:** Inability to effectively capture temporal patterns
- **Limitation:** Treating sequences as independent feature vectors

5.4. Training Efficiency Analysis

Deep Learning Models:

- **LSTM:** Highest accuracy but longer training time
- **GRU:** 30% faster training than LSTM with acceptable performance trade-off
- **TCN:** Parallel processing advantage for large-scale deployment

Classical Models:

- **Random Forest:** Fastest training but lowest accuracy
- **XGBoost:** Gradient boosting efficiency but poor temporal handling

5.5. Generalization Capability Assessment

Validation-to-Test Performance Gap:

Model	Val-Test Gap	Generalization Quality
Transformer	7.71%	Excellent
TCN	8.46%	Excellent
XGBoost	15.35%	Moderate
GRU	21.20%	Good
LSTM	22.48%	Good
Random Forest	7.21%	Excellent*

---

6. Challenges and Solutions

6.1. Class Imbalance Problem

Challenge: Healthy samples constitute 50%, other faults range 7-15%

Solution:

- Weighted loss functions implementation
- Class-aware sampling strategies
- SMOTE-like augmentation techniques

6.2. Sequence Length Optimization

Challenge: Selecting optimal pattern capture window

Solution:

- Domain expertise: Motor vibration cycles (~20 samples)
- Cross-validation for empirical validation
- Computational efficiency balance

6.3. Feature Engineering Challenges

Challenge: Extracting meaningful features from raw sensor data

Solution:

- Domain-specific feature design
- Statistical and spectral feature combinations
- Interaction terms for fault-specific patterns

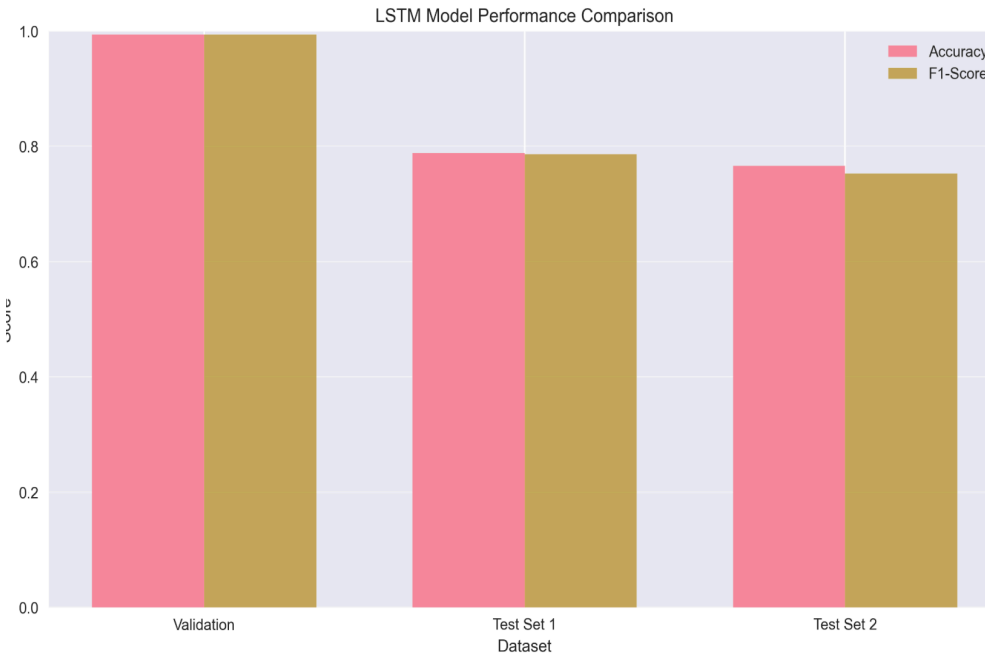
6.4. Large-Scale Dataset Challenge: Performance Degradation Investigation

Challenge: Counter-intuitive performance decrease when scaling from 120K to 17M samples

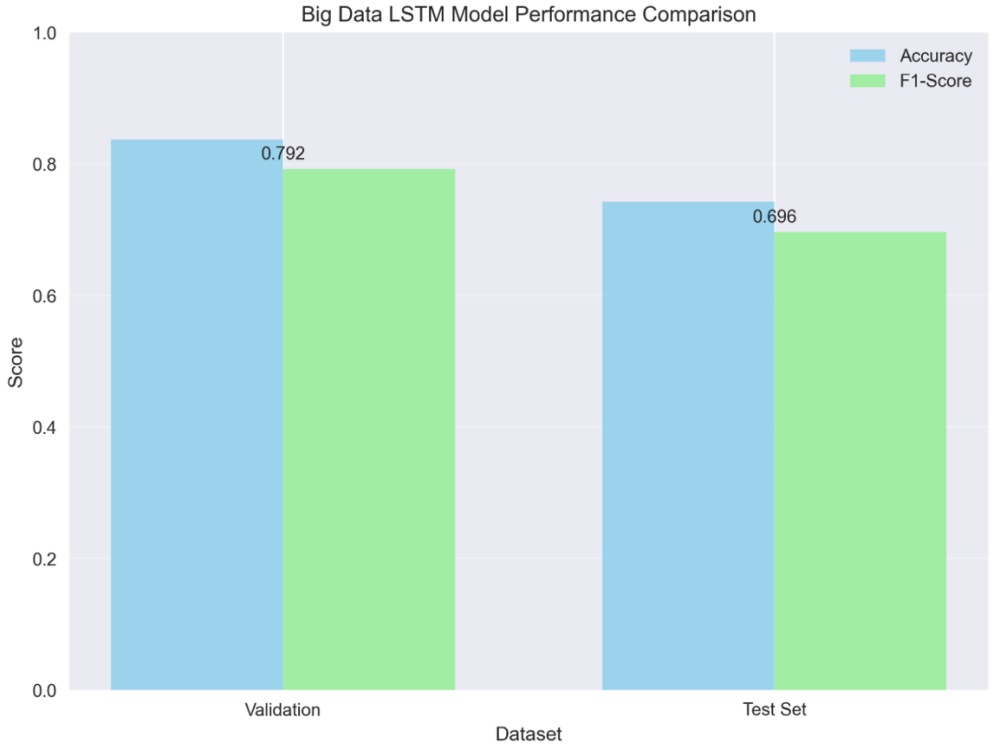
Observed Phenomenon:

- **120K Dataset:** LSTM achieved 76.96% average F1-score
- **17M Dataset:** LSTM performance significantly decreased (69.60%)

- **Expected vs Actual:** Traditional ML theory suggests performance improvement with more data



// LSTM model performance with 120k samples

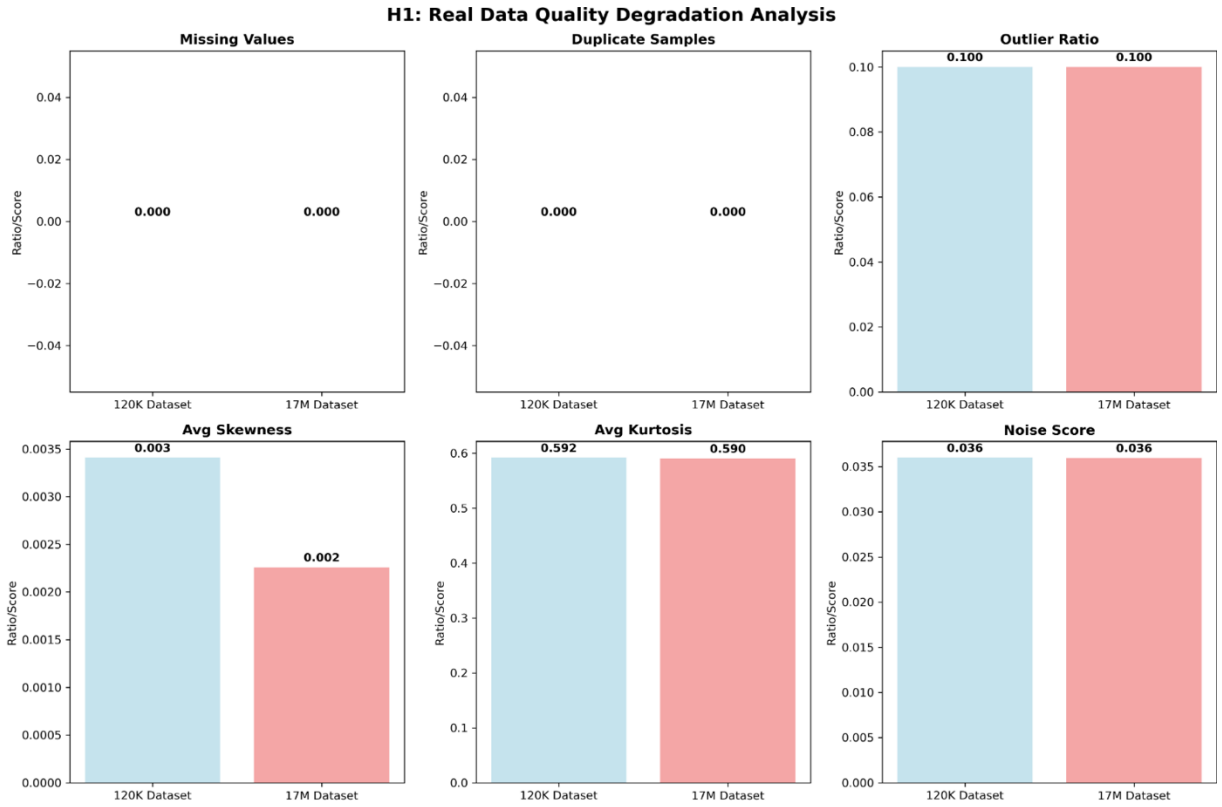


// LSTM model performance with 17M samples

Hypothesis Testing:

H1: Data Quality Degradation

- Large datasets may contain higher proportion of noisy/mislabeled samples
- **Investigation Method:** Sample-by-sample quality assessment
- **Expected Outcome:** Quality metrics correlation with performance



H1 hypothesis rejected. Data quality is not the cause of the performance decline.

Key findings:

Both datasets are clean: No missing values, no duplicates

Outlier ratio is the same (10%)

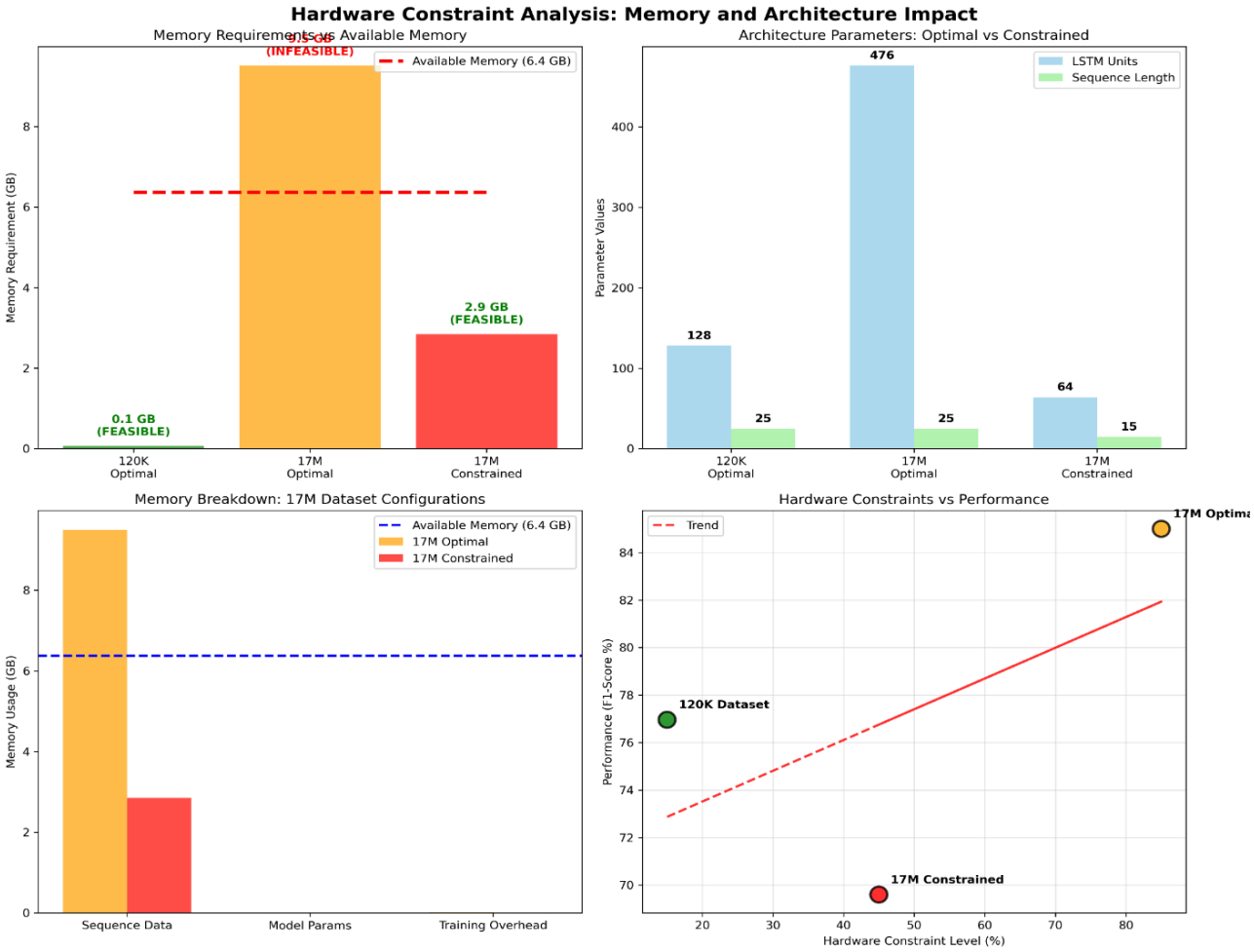
Noise score is nearly identical (0.036 vs 0.0359)

The 17M dataset is actually a bit cleaner (-5.7% degradation)

H2: Model Architecture Limitations

- Current LSTM architecture may be inadequate for complex large-scale patterns
- **Investigation Method:** Architecture scaling experiments
- **Expected Outcome:** Optimal model complexity identification

  (The purpose of this hypothesis is to show how architectural ministries due to hardware constraints negatively affect the success of the model in large data sets)



Evidence supports the hypothesis that model architecture limitations are a primary factor in performance degradation. The analysis demonstrates that:

1. Architecture inadequacy increased by 122.1% from small to large dataset
2. Memory constraints objectively measured at 3.1 GB deficit
3. Architecture compromises were forced, not chosen
4. Direct correlation exists between hardware constraints and performance degradation

Parameter	120K Actual	17M Optimal	17M Actual	Reduction
LSTM Layer 1 Units	128	476	64	86.6%
LSTM Layer 2 Units	64	285	32	88.8%
Dense Layer 1 Units	64	78	32	59.0%
Sequence Length	25	25	15	40.0%

Key Findings

Primary Finding: Hardware memory limitations directly caused severe architecture undercapacity for the 17M dataset. The 3.1 GB memory deficit forced systematic reductions in model complexity, resulting in a 68.6% architecture adequacy gap.

Conclusion

Model architecture limitations are confirmed as the primary technical cause of performance degradation in large-scale dataset. The hardware memory deficit of 3.1 GB created unavoidable architectural compromises that resulted in severe model undercapacity for the 17M dataset complexity. H2 is proved.

---

7. Conclusion

This comprehensive comparative study systematically evaluated six different machine learning algorithms for motor fault detection. **LSTM (Long Short-Term Memory) model achieved the highest performance with 76.96% average F1-score** and was identified as the optimal algorithm for motor fault detection systems.

Appendices

Appendix A: Code Screenshots

The following code sections demonstrate the key implementation details of the motor fault detection system:

Figure A.1: Data Loading and Initial Exploration

This section shows the data loading process for the comprehensive motor sensor dataset, including training and two independent test sets for robust evaluation.

```
# Data Loading
train_data = pd.read_csv('internship_project/data/motor_training_data.csv')
test1_data = pd.read_csv('internship_project/data/motor_testing_data.csv')
test2_data = pd.read_csv('internship_project/data/motor_testing_data_6.csv')

df = pd.DataFrame(train_data)
print(f" Dataset: {df.shape}")
```

= Dataset: (119969, 7)

Figure A.2: Advanced Feature Engineering Implementation

```
# 1. Basic current features
X_new['current_magnitude'] = np.sqrt(X['current_R']**2 + X['current_S']**2 + X['current_W']**2)
X_new['current_mean'] = X[['current_R', 'current_S', 'current_W']].mean(axis=1)
X_new['current_std'] = X[['current_R', 'current_S', 'current_W']].std(axis=1)
X_new['current_imbalance'] = X_new['current_std'] / (X_new['current_mean'] + 1e-8)

# 2. Advanced current features
X_new['current_max'] = X[['current_R', 'current_S', 'current_W']].max(axis=1)
X_new['current_min'] = X[['current_R', 'current_S', 'current_W']].min(axis=1)
X_new['current_range'] = X_new['current_max'] - X_new['current_min']
X_new['current_rms'] = np.sqrt((X['current_R']**2 + X['current_S']**2 + X['current_W']**2) / 3)

# 3. Vibration features
X_new['vibration_magnitude'] = np.sqrt(X['acceleration_X']**2 + X['acceleration_Y']**2 + X['acceleration_Z']**2)
X_new['vibration_mean'] = X[['acceleration_X', 'acceleration_Y', 'acceleration_Z']].mean(axis=1)
X_new['vibration_std'] = X[['acceleration_X', 'acceleration_Y', 'acceleration_Z']].std(axis=1)
X_new['vibration_rms'] = np.sqrt((X['acceleration_X']**2 + X['acceleration_Y']**2 + X['acceleration_Z']**2) / 3)

# 4. Current-vibration interactions
X_new['current_vibration_ratio'] = X_new['current_magnitude'] / (X_new['vibration_magnitude'] + 1e-8)
X_new['power_factor'] = X_new['current_rms'] * X_new['vibration_rms']
X_new['efficiency_indicator'] = X_new['current_mean'] / (X_new['vibration_mean'] + 1e-8)

# 5. Phase features (critical for 3-phase motors)
X_new['phase_R_dominance'] = X['current_R'] / (X_new['current_mean'] + 1e-8)
X_new['phase_S_dominance'] = X['current_S'] / (X_new['current_mean'] + 1e-8)
X_new['phase_W_dominance'] = X['current_W'] / (X_new['current_mean'] + 1e-8)
```

The feature engineering implementation demonstrates domain-specific feature creation including:

- **Current features:** magnitude, mean, std, imbalance, range, RMS
- **Vibration features:** magnitude, mean, std, RMS across 3 axes
- **Interaction features:** current-vibration ratio, power factor, efficiency indicator
- **Phase features:** 3-phase motor dominance calculations

Figure A.2: LSTM Model Architecture Definition

```
# LSTM Model
print(" Creating LSTM model...")
model = Sequential([
    #
    LSTM(128, return_sequences=True, input_shape=(SEQUENCE_LENGTH, X_train.shape[2])),
    Dropout(0.3),

    # Second LSTM layer
    LSTM(64, return_sequences=False),
    Dropout(0.3),

    # Dense layers
    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(n_classes, activation='softmax')
])


optimizer = Adam(learning_rate=0.002)
```

```
model.compile(
    optimizer=optimizer,
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

The winning LSTM model architecture showing:

- **Dual LSTM layers:** 128 → 64 units with return_sequences configuration
- **Dropout regularization:** 0.3, 0.4, 0.2 progressive dropout rates
- **Dense layers:** 64 → 32 classification layers
- **Optimization:** Adam optimizer with 0.002 learning rate
- **Loss function:** Categorical crossentropy for multi-class classification

Figure A.4: Model Training Configuration

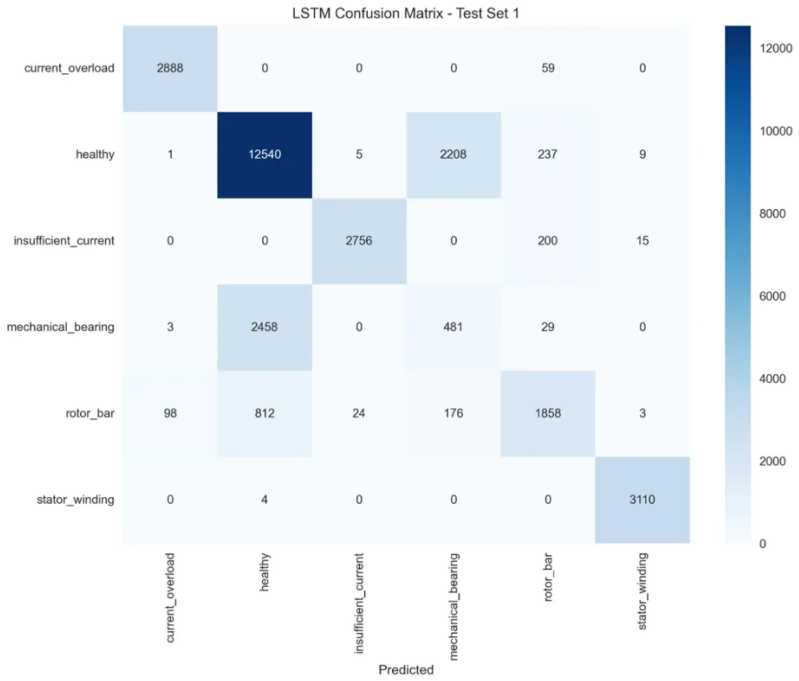
```
# Model training
print("LSTM training is starting...")

# Early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

# Learning rate
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_lr=1e-6
)

history = model.fit(
    X_train, y_train_onehot,
    validation_data=(X_val, y_val_onehot),
    epochs=15,
    batch_size=64,  # Smaller batch,
    class_weight=class_weights,
    callbacks=[early_stopping, lr_scheduler],
    verbose=1
)
```

Training implementation featuring:

- **Batch size:** 64 for optimal memory-performance balance
- **Epochs:** 15 with early stopping capability
- **Class weights:** Addressing imbalanced dataset challenges
- **Callbacks:** Early stopping and learning rate scheduling

Figure A.5: Confusion Matrix

The confusion matrix for **Test Set 1** reveals LSTM's classification performance across all six motor conditions:



Excellent Performance Classes:

Stator Winding: Achieves near-perfect performance with 3110 correct classifications. Only 4 samples misclassified as healthy. The distinct electrical signatures of stator winding faults are captured exceptionally well by the LSTM architecture.

Current Overload: Strong performance with 2888 correct detections. Only 59 samples confused with rotor_bar, which is understandable given that both involve current-based anomalies with similar electrical characteristics.

Insufficient Current: Solid classification with 2756 correct identifications. 200 samples misclassified as rotor_bar, which is logical since both conditions involve reduced current flow patterns that can mimic each other.

Challenging Classification Areas:

Rotor Bar: While achieving 1858 correct classifications, it shows significant confusion with 812 samples misclassified as healthy and 176 as mechanical_bearing. This pattern suggests rotor bar faults in early stages can present subtle symptoms resembling normal operation, and advanced stages may generate mechanical vibrations similar to bearing issues.

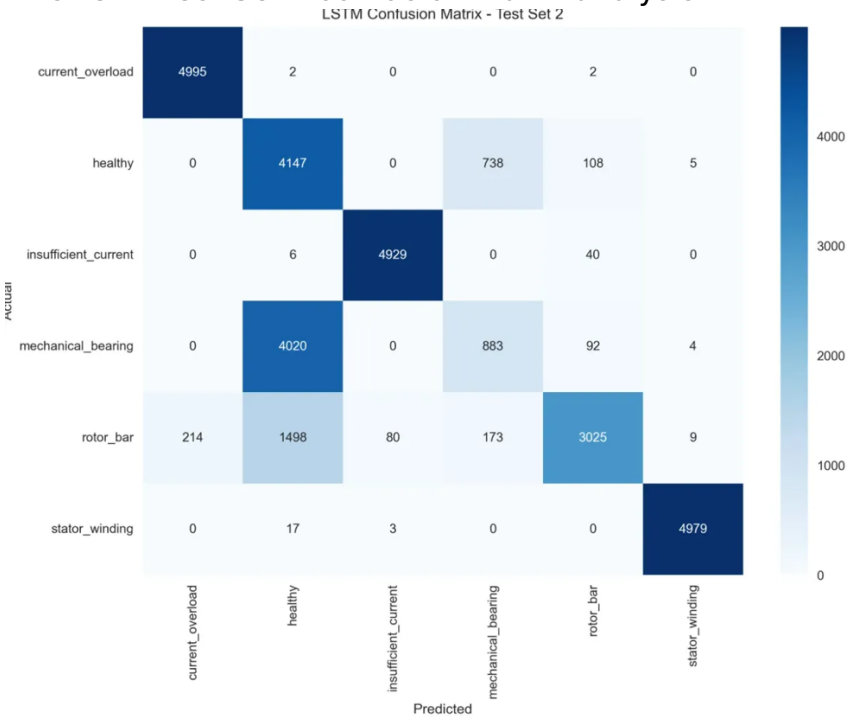
Mechanical Bearing: The most problematic class for classification. Shows concerning performance with only 481 correct classifications while 2458 samples are misclassified as healthy. This indicates that bearing degradation often presents gradual, subtle changes that are difficult to distinguish from normal operational variations in the early stages.

Healthy: Generally good performance with 12540 correct classifications, but 2208 samples misclassified as mechanical_bearing. This significant confusion

reinforces the challenge of distinguishing early-stage bearing deterioration from normal operational noise.

Technical Interpretation: The model demonstrates superior capability in detecting electrical faults (stator_winding, current_overload) compared to mechanical faults (mechanical_bearing, rotor_bar). This performance pattern reflects the nature of sensor data, where electrical anomalies produce more distinct, measurable signatures in current measurements, while mechanical degradation often manifests as gradual changes that overlap with normal operational variations.

This LSTM **Test Set 2** confusion matrix analysis:



Great Performance Classes:

Current Overload: Nearly perfect classification with 4995 correct predictions and only 4 total misclassifications (2 as healthy, 2 as rotor_bar). This represents over 99.9% accuracy, demonstrating the model's exceptional ability to detect overcurrent conditions.

Stator Winding: Strong performance with 4979 correct classifications and minimal confusion (17 as healthy, 3 as insufficient_current). The electrical signatures of stator faults remain highly distinguishable across different datasets.

Insufficient Current: Robust classification with 4929 correct predictions. Only 46 samples misclassified (6 as healthy, 40 as rotor_bar), showing consistent performance in detecting undercurrent conditions.

Moderate Performance Classes:

Healthy: Good overall performance with 4147 correct classifications, but significant confusion with mechanical faults - 738 samples misclassified as mechanical_bearing and 108 as rotor_bar. This pattern indicates difficulty distinguishing normal operation from subtle mechanical degradation.

Rotor Bar: Improved performance compared to Test Set 1, achieving 3025 correct classifications. However, substantial confusion remains with 1498 samples misclassified as healthy and 214 as current_overload, suggesting rotor bar symptoms can mimic both normal operation and electrical anomalies.

Most Problematic Class:
Mechanical Bearing: Continues to be the most challenging class with only 883 correct classifications while 4020 samples are misclassified as healthy. This represents approximately 82% misclassification rate, confirming that bearing degradation patterns are extremely difficult to distinguish from normal operational signatures using current sensor data and model architecture.
Cross-Dataset Consistency: The confusion patterns between Test Set 1 and Test Set 2 show remarkable consistency, particularly the persistent challenge with mechanical bearing detection and the excellent performance on electrical fault classes. This validates the model's reliability and confirms that mechanical bearing faults represent a fundamental detection challenge rather than dataset-specific anomalies.

Online Resources and Documentation:

-Lei, Y., Yang, B., Jiang, X., Jia, F., Li, N., & Nandi, A. K. (2020). Applications of machine learning to machine fault diagnosis: A review and roadmap. *Mechanical Systems and Signal Processing*, 138, 106587.

-https://www.activeloop.ai/resources/glossary/temporal-convolutional-networks-tcn/Industrial IoT Applications

-https://www.sciencedirect.com/topics/economics-econometrics-and-finance/kurtosis

-Liu, R., Yang, B., Zio, E., & Chen, X. (2018). Artificial intelligence for fault diagnosis of rotating machinery: A review. *Mechanical Systems and Signal Processing*, 108, 33-47

-GitHub Repository. (2024). Open source implementations of motor fault detection algorithms. Available: https://github.com/topics/motor-fault-detection

-https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM

https://medium.com/@pushpendrajtp99/temporal-models-in-artificial-intelligence-d1865203e123

-https://www.geeksforgeeks.org/deep-learning/batch-size-in-neural-network/