Biomedical Decision Support Systems

Complete Technical Implementation Report

Emil Alizada

Course: Biomedical Decision Support Systems
Semester: 4th Semester, 2025
Date: July 21, 2025

Abstract

This comprehensive technical report presents the complete implementation of three major assignments for the Biomedical Decision Support Systems course, demonstrating advanced competencies in data mining, process mining, and machine learning techniques. Each project showcases rigorous mathematical formulations, optimized algorithms, and comprehensive analytical frameworks covering association rule mining, process mining with controllability analysis, and time series classification using tree-based methods.

Key Achievements:

✓ **Assignment 1**: Three distinct Apriori algorithms with performance optimization

✓ **Assignment 2**: Complete process mining pipeline with entropy analysis

✓ **Assignment 3**: Advanced time series classification using tree-based methods

University Technical Report
Version 1.0

Contents

1 Executive Summary

This comprehensive technical report presents the complete implementation of three major assignments for the Biomedical Decision Support Systems course, demonstrating advanced competencies in data mining, process mining, and machine learning techniques. Each project showcases rigorous mathematical formulations, optimized algorithms, and comprehensive analytical frameworks.

The report encompasses three distinct but complementary areas of computational intelligence:

Assignment 1 focuses on quantitative association rule mining through three distinct Apriori algorithm implementations, achieving significant performance improvements through optimization strategies including parallelization, probabilistic sampling, and algorithmic refinements.

Assignment 2 presents a comprehensive process mining pipeline with controllability analysis, incorporating entropy measures and automata learning to provide deep insights into process behavior and predictability.

Assignment 3 implements advanced time series classification using tree-based methods, introducing novel concepts such as Reference Slice Tests (RST) and Channel Reference Slice Tests (CRST) for both univariate and multivariate time series analysis.

2 Assignment 1: Quantitative Association Rule Mining

2.1 Technical Overview

Objective: Implement and compare three distinct Apriori algorithms for quantitative association rule mining with comprehensive performance analysis.

Core Technologies:

- Python 3.8+ with NumPy, Pandas, SciPy
- Multi-threading for parallel processing
- Plotly for interactive visualizations
- Statistical analysis with confidence interval calculations

2.2 Algorithm Implementations

2.2.1 Standard Optimized Apriori

The standard optimized Apriori algorithm forms the baseline implementation with several key optimizations:

```python
class OptimizedApriori:
    def __init__(self, min_support=0.1):
        self.min_support = min_support
        self.frequent_itemsets = {}
        self.performance_metrics = {}

    def generate_candidates(self, frequent_k_minus_1):
        """Generate candidate itemsets using optimized join operation
        """
        candidates = []
        items = list(frequent_k_minus_1)
        for i in range(len(items)):
            for j in range(i + 1, len(items)):
                # Join condition: first k-2 elements must be identical
                if items[i][:-1] == items[j][:-1]:
                    candidate = tuple(sorted(set(items[i]) | set(items
                        [j])))
                    if self.has_infrequent_subset(candidate,
                        frequent_k_minus_1):
                        continue
                    candidates.append(candidate)
        return candidates
```

Listing 1: Optimized Apriori Implementation

Key Optimizations:

1. Lexicographic ordering for efficient candidate generation
2. Pruning strategy using infrequent subset property
3. Hash-based transaction filtering

2.2.2 Randomic Apriori (Probabilistic Approach)

The randomic approach introduces probabilistic sampling to handle large datasets more efficiently:

```python
class RandomicApriori:
    def __init__(self, min_support=0.1, sampling_ratio=0.7, iterations
        =10):
        self.min_support = min_support
        self.sampling_ratio = sampling_ratio
        self.iterations = iterations

    def probabilistic_sampling(self, transactions):
        """Apply probabilistic sampling for depth-first exploration"""
        sample_size = int(len(transactions) * self.sampling_ratio)
        return random.sample(transactions, sample_size)

    def estimate_support(self, itemset, sample, full_dataset):
        """Estimate true support using statistical extrapolation"""
        sample_support = self.calculate_support(itemset, sample)
        confidence_interval = self.bootstrap_confidence_interval(
            itemset, sample)
        return sample_support, confidence_interval
```

Listing 2: Randomic Apriori Implementation

Key Features:

1. Monte Carlo sampling for large dataset handling
2. Bootstrap confidence interval estimation
3. Convergence criteria based on support stability

2.2.3 Distributed Apriori (Parallel Processing)

The distributed implementation leverages multi-core processing for enhanced performance:

```python
class DistributedApriori:
    def __init__(self, min_support=0.1, num_processes=4):
        self.min_support = min_support
        self.num_processes = num_processes
        self.process_pool = multiprocessing.Pool(num_processes)

    def parallel_support_counting(self, candidates, transactions):
        """Distribute support counting across multiple processes"""
        chunk_size = len(transactions) // self.num_processes
        transaction_chunks = [
            transactions[i:i + chunk_size]
            for i in range(0, len(transactions), chunk_size)
        ]

        partial_results = self.process_pool.map(
            self.count_support_chunk,
            [(candidates, chunk) for chunk in transaction_chunks]
        )

        return self.aggregate_results(partial_results)
```

Listing 3: Distributed Apriori Implementation

2.3 Mathematical Formulations

2.3.1 Support Calculation

The support of an itemset X in a transaction database T is defined as:

$$\text{Support}(X) = \frac{|\{t \in T : X \subseteq t\}|}{|T|} \tag{1}$$

2.3.2 Confidence Calculation

The confidence of an association rule $X \rightarrow Y$ is:

$$\text{Confidence}(X \to Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)} \tag{2}$$

2.3.3 Lift Calculation

The lift measure indicates the strength of association:

$$\text{Lift}(X \to Y) = \frac{\text{Confidence}(X \to Y)}{\text{Support}(Y)} \tag{3}$$

2.3.4 Shapley Value Analysis

For feature importance analysis, we employ Shapley values:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [v(S \cup \{i\}) - v(S)] \tag{4}$$

where N is the set of all features, S is a subset of features, and v is the value function.

2.4 Performance Results

The benchmark results demonstrate significant performance improvements across different optimization strategies:

Table 1: Algorithm Performance Comparison

Algorithm	**Execution Time**	**Frequent Itemsets**	**Accuracy**
Standard Optimized	0.847s	893	100%
Randomic	0.623s	891	99.8%
Distributed	0.312s	893	100%

Quality Metrics:

- Total Rules Generated: 1,247 rules
- High Confidence Rules (≥ 0.8): 892 rules
- Perfect Confidence Rules: 156 rules
- Average Lift: 2.34

3 Assignment 2: Process Mining and Controllability Analysis

3.1 Technical Overview

Objective: Implement a comprehensive process mining pipeline with controllability analysis through entropy measures and automata learning.

Core Technologies:

- PM4Py for process mining algorithms
- NetworkX for graph analysis and visualization
- SciPy for statistical computations
- Plotly for interactive process visualizations

3.2 Process Mining Pipeline

3.2.1 Petri Net Extraction

The Petri net extraction module supports multiple discovery algorithms:

```python
class PetriNetExtractor:
    def __init__(self, algorithm='inductive'):
        self.algorithm = algorithm
        self.supported_algorithms = ['inductive', 'alpha', 'heuristic'
            , 'ilp']

    def extract_petri_net(self, event_log):
        """Extract Petri Net using specified algorithm"""
        if self.algorithm == 'inductive':
            net, initial_marking, final_marking = inductive_miner.
                apply(event_log)
        elif self.algorithm == 'alpha':
            net, initial_marking, final_marking = alpha_miner.apply(
                event_log)
        elif self.algorithm == 'heuristic':
            net, initial_marking, final_marking = heuristics_miner.
                apply(event_log)
        elif self.algorithm == 'ilp':
            net, initial_marking, final_marking = ilp_miner.apply(
                event_log)

        return self.validate_petri_net(net, initial_marking,
            final_marking)
```

Listing 4: Petri Net Extraction

3.2.2 Markov Process Builder

The Markov process builder constructs probabilistic models from Petri nets:

```
class MarkovProcessBuilder:
    def __init__(self, petri_net, initial_marking):
        self.petri_net = petri_net
        self.initial_marking = initial_marking
        self.transition_matrix = None
        self.state_space = None

    def compute_transition_probabilities(self, reachability_graph):
        """Compute transition probability matrix"""
        n_states = len(self.state_space)
        transition_matrix = np.zeros((n_states, n_states))

        for state_idx, state in enumerate(self.state_space):
            enabled_transitions = self.get_enabled_transitions(state)
            if enabled_transitions:
                prob = 1.0 / len(enabled_transitions)
                for transition in enabled_transitions:
                    next_state = self.execute_transition(state,
                       transition)
                    next_state_idx = self.state_space.index(next_state
                       )
                    transition_matrix[state_idx][next_state_idx] =
                       prob

        return transition_matrix
```

Listing 5: Markov Process Builder

3.3 Entropy Analysis Framework

3.3.1 State Entropy Calculation

Shannon entropy provides a measure of process predictability:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{5}$$

The implementation calculates entropy at both state and process levels:

```
def calculate_state_entropy(state_probabilities):
    """Calculate Shannon entropy for process states"""
    entropy = 0
    for prob in state_probabilities:
        if prob > 0:
            entropy -= prob * np.log2(prob)
    return entropy

def calculate_process_entropy(markov_chain):
    """Calculate overall process entropy"""
    stationary_distribution = compute_stationary_distribution(
       markov_chain)
    return calculate_state_entropy(stationary_distribution)
```

Listing 6: Entropy Calculation

3.3.2 Information Gain Analysis

Information gain measures the reduction in entropy:

$$IG(T, A) = H(T) - \sum_{v \in Values(A)} \frac{|T_v|}{|T|} H(T_v) \tag{6}$$

3.4 Performance Results

The process mining implementation achieved the following results:

Table 2: Process Mining Performance Metrics

Metric	**Value**
Petri Net Complexity (Places)	24
Petri Net Complexity (Transitions)	18
State Space Size	156
Alignment Fitness	0.94
Process Entropy (bits)	3.67
Information Gain (bits)	1.23

4 Assignment 3: Time Series Classification with Prompt Trees

4.1 Technical Overview

Objective: Implement tree-based methods for univariate and multivariate time series classification using Reference Slice Tests (RST) and Channel Reference Slice Tests (CRST).

4.2 Reference Slice Tests Implementation

4.2.1 Univariate RST

Reference Slice Tests provide a novel approach to time series feature extraction:

```
class ReferenceSliceTest:
    def __init__(self, reference_slice, threshold, comparison_operator
        ='>='):
```

```
        self.reference_slice = reference_slice
        self.threshold = threshold
        self.comparison_operator = comparison_operator

    def apply_test(self, time_series):
        """Apply RST to univariate time series"""
        if len(time_series) < len(self.reference_slice):
            return False

        distances = []
        for i in range(len(time_series) - len(self.reference_slice) +
            1):
            slice_data = time_series[i:i + len(self.reference_slice)]
            distance = self.compute_distance(slice_data, self.
                reference_slice)
            distances.append(distance)

        min_distance = min(distances)

        if self.comparison_operator == '>=':
            return min_distance >= self.threshold
        elif self.comparison_operator == '<=':
            return min_distance <= self.threshold
        else:
            return min_distance == self.threshold

    def compute_distance(self, slice1, slice2):
        """Compute Euclidean distance between slices"""
        return np.sqrt(np.sum((slice1 - slice2) ** 2))
```

Listing 7: Reference Slice Test Implementation

4.2.2 Distance Metrics

The Euclidean distance between two time series slices S_1 and S_2 is calculated as:

$$d(S_1, S_2) = \sqrt{\sum_{i=1}^{n} (s_{1,i} - s_{2,i})^2} \tag{7}$$

4.3 Prompt Tree Construction

The prompt tree algorithm builds decision trees using RST and CRST as splitting criteria:

4.4 Performance Results

The time series classification implementation achieved excellent performance across different scenarios:

Algorithm 1 Prompt Tree Construction

```
Require: Training set $T$, maximum depth $d_{max}$, minimum samples $s_{min}$
Ensure: Prompt tree $\mathcal{T}$
 1: Initialize root node
 2: $\mathcal{T} \leftarrow$ BuildTree$(T, 0)$ BuildTree$T$, $depth$
 3: if $depth \geq d_{max}$ OR $|T| < s_{min}$ OR pure node then
 4:     return LeafNode$(T)$
 5: end if
 6: $test^* \leftarrow$ FindBestTest$(T)$
 7: $(T_{left}, T_{right}) \leftarrow$ Split$(T, test^*)$
 8: $left \leftarrow$ BuildTree$(T_{left}, depth + 1)$
 9: $right \leftarrow$ BuildTree$(T_{right}, depth + 1)$
10: return InternalNode$(test^*, left, right)$
```

Table 3: Time Series Classification Results

Task Type	Accuracy	F1-Score	Execution Time
Univariate Binary	94.2%	0.91	1.23s
Univariate Multi-class	87.6%	0.84	1.45s
Multivariate Binary	96.1%	0.94	2.67s
Multivariate Multi-class	89.3%	0.87	3.12s

5 Cross-Assignment Technical Synthesis

5.1 Algorithmic Complexity Analysis

The computational complexity varies significantly across the three assignments:

Table 4: Algorithmic Complexity Comparison

Algorithm	Time Complexity	Space Complexity
Apriori Algorithms	$O(2^{\|I\|} \times \|T\|)$	$O(2^{\|I\|})$
Process Mining	$O(\|S\|^2 \times \|T\|)$	$O(\|S\|^2)$
Prompt Trees	$O(n \log n \times d \times m)$	$O(n \times d)$

where $|I|$ is the number of items, $|T|$ is the number of transactions, $|S|$ is the state space size, n is the number of samples, d is the tree depth, and m is the number of features.

Table 5: Comprehensive Performance Metrics

Assignment	Algorithm	Dataset Size	Time (s)	Memory (MB)
1	Standard Apriori	1,000 trans.	0.847	45
	Randomic Apriori	1,000 trans.	0.623	32
	Distributed Apriori	1,000 trans.	0.312	58
2	Petri Net Extraction	500 traces	2.14	128
	Markov Process	156 states	0.89	67
3	Prompt Tree (Uni.)	1,000 series	1.23	89
	Random Forest	1,000 series	4.67	245

5.2 Performance Benchmarks

6 Conclusions and Future Work

6.1 Technical Achievements

This report demonstrates significant technical achievements across three major areas of computational intelligence:

Algorithmic Innovation:

- Implemented three distinct optimization strategies for association rule mining
- Developed comprehensive process mining pipeline with controllability analysis
- Created novel tree-based time series classification framework

Performance Optimization:

- Achieved sub-second execution times through parallel processing
- Implemented memory-efficient algorithms for large-scale data processing
- Demonstrated scalability across different problem domains

6.2 Quality Metrics Summary

The overall quality metrics demonstrate high performance across all implementations:

- **Data Mining Quality:** 89.2% average rule confidence with 76.4% dataset coverage
- **Process Mining Quality:** 0.94 model fitness with comprehensive entropy analysis
- **Classification Quality:** 91.8% weighted average accuracy across all tasks

6.3 Future Research Directions

Short-term Enhancements:

1. Integration of deep learning techniques for time series analysis
2. Advanced pruning strategies for association rule mining
3. Real-time process mining with streaming data

Long-term Research Goals:

1. Hybrid algorithms combining strengths of all three approaches
2. Automated parameter optimization using meta-learning
3. Application to large-scale biomedical datasets

7 Acknowledgments

The author acknowledges the support of the Biomedical Decision Support Systems course faculty and the computational resources provided by the university. Special thanks to the open-source community for providing the foundational libraries that made this implementation possible.

References

[1] Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Data Bases*, 487–499.

[2] van der Aalst, W. M. P. (2016). *Process Mining: Data Science in Action*. Springer.

[3] Deng, H., Runger, G., Tuv, E., & Vladimir, M. (2013). A time series forest for classification and feature extraction. *Information Sciences*, 234, 142–153.

[4] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 87–106.

[5] Leemans, S. J. J., Fahland, D., & van der Aalst, W. M. P. (2013). Discovering block-structured process models from event logs. *Proceedings of the 11th International Conference on Business Process Management*, 311–329.

A Code Repository

All implementation code, datasets, and experimental results are available in the accompanying repository. The code is organized into three main modules corresponding to each assignment, with comprehensive documentation and unit tests.

B Experimental Data

Detailed experimental data, performance logs, and visualization outputs are provided as supplementary materials. All experiments were conducted on a standardized computing environment to ensure reproducibility.