Java Implementation of AMCE Algorithm

```
// AMCE Algorithm Implementation (Java)

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import org.apache.spark.ml.classification.*;
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator;
import org.apache.spark.ml.feature.VectorAssembler;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public class AMCE {

    // Configuration
    private static final long MONITORING_INTERVAL = 1000; // Milliseconds
    private static final long MON_WIN_DUR = 5000; // Milliseconds (example)

    // Data Structures
    private Map<String, Object> trainedBCs = new ConcurrentHashMap<>(); // Placeholder for trained base
classifiers
    private Object metaLearner; // Placeholder for meta-learner (ANN)
    private Map<String, Double> weights = new ConcurrentHashMap<>();
    private List<Map<String, Double>> performanceHistory = new ArrayList<>();
    private List<Map<String, Double>> trainingSet; // Placeholder for training data
    private List<Map<String, Double>> realTimeFeatures; // Placeholder for real-time features
    private List<Double> trueLabels; // Placeholder for True labels from training.

    // Placeholder for Alert related methods
    private void SendAlert(Map<String, Object> alert) {
        System.out.println("Alert sent: " + alert);
    }

    private Map<String, Object> NewAlert(String srcCtrlID, String tgtNetSeg, int atkSev, long ts,
List<String> extFeats) {
        Map<String, Object> alert = new HashMap<>();
        alert.put("srcCtrlID", srcCtrlID);
        alert.put("tgtNetSeg", tgtNetSeg);
        alert.put("atkSev", atkSev);
        alert.put("ts", ts);
        alert.put("extFeats", extFeats);
        return alert;
    }

    // Placeholder for data preprocessing
    private List<Map<String, Double>> PreprocessData(List<Map<String, Double>> data) {
        // Implement data preprocessing logic here
        return data; // Return the preprocessed data
    }

    // Placeholder for base classifier training
    private Map<String, Object> TrainBCs(List<Map<String, Double>> trainingData) {
        // Implement training of base classifiers (KNN, DT, RF, SVM, XGBoost)
        // Store trained models in trainedBCs map
        Map<String, Object> trainedModels = new ConcurrentHashMap<>();
        trainedModels.put("KNN", "Placeholder KNN Model");
        trainedModels.put("DT", "Placeholder DT Model");
```

```
        trainedModels.put("RF", "Placeholder RF Model");
        trainedModels.put("SVM", "Placeholder SVM Model");
        trainedModels.put("XGBoost", "Placeholder XGBoost Model");
        return trainedModels;
    }

    // Placeholder for meta-learner training
    private void TrainMetaLearner(List<Map<String, Double>> bcPredictions, List<Map<String, Double>>
confidenceScores, List<Double> labels) {
        // Implement training of the meta-learner (ANN)
        metaLearner = "Placeholder Meta-Learner";
    }

    // Placeholder for generating predictions and confidence scores
    private Map<String, Object[]> GeneratePredictionsAndConfidences(Map<String, Object> trainedModels,
List<Map<String, Double>> data) {
        Map<String, Object[]> results = new ConcurrentHashMap<>();
        for (String modelName : trainedModels.keySet()) {
            results.put(modelName, new Object[]{"Placeholder Prediction", 0.8}); // Placeholder
prediction and confidence
        }
        return results;
    }

    // Placeholder for getting confidence from base classifiers
    private Object[] GetConfidence(Object bc, List<Map<String, Double>> features) {
        // Implement logic to get prediction and confidence from a base classifier
        return new Object[]{"Placeholder Prediction", 0.8}; // Placeholder prediction and confidence
    }

    // Placeholder for adjusting weights
    private Map<String, Double> AdjustWeights(Map<String, Double> weights, List<Map<String, Double>>
performanceHistory) {
        // Implement logic to adjust weights based on performance history
        for (String modelName : weights.keySet()) {
            weights.put(modelName, 1.0); // Placeholder: Set all weights to 1.0
        }
        return weights;
    }

    // Placeholder for evaluating performance
    private Map<String, Double> Evaluate(String finalPrediction, List<Double> trueLabels) {
        // Implement evaluation logic (Accuracy, Precision, Sensitivity, Balanced F1 Score, TNR)
        Map<String, Double> metrics = new HashMap<>();
        metrics.put("accuracy", 0.9);
        metrics.put("precision", 0.9);
        metrics.put("sensitivity", 0.9);
        metrics.put("f1", 0.9);
        metrics.put("tnr", 0.9);
        return metrics;
    }

    // Placeholder for initializing weights
    private Map<String, Double> InitializeWeights(Map<String, Object> trainedBCs) {
        Map<String, Double> initialWeights = new ConcurrentHashMap<>();
        for (String modelName : trainedBCs.keySet()) {
            initialWeights.put(modelName, 1.0);
        }
        return initialWeights;
    }

    // Placeholder for initializing performance history
```

```
    private List<Map<String, Double>> InitializePerformanceHistory(Map<String, Object> trainedBCs) {
        return new ArrayList<>();
    }

    // Placeholder for getting real-time features
    private List<Map<String, Double>> GetRealTimeFeatures() {
        // Implement logic to retrieve real-time features from Module 3
        return realTimeFeatures;
    }

    // Placeholder for meta-learner prediction
    private String MetaLearnerPredict(Map<String, String> bcPredictions, Map<String, Double>
confidenceScores, Map<String, Double> weights) {
        // Implement meta-learner prediction logic
        return "Normal"; // Placeholder: Predict "Normal" or "Attack"
    }

    // Placeholder for updating performance history
    private void UpdatePerformanceHistory(List<Map<String, Double>> performanceHistory, Map<String,
Double> performance) {
        performanceHistory.add(performance);
    }

    // Training Phase
    public void train(List<Map<String, Double>> trainingData, List<Double> labels) {
        this.trainingSet = trainingData;
        this.trueLabels = labels;
        List<Map<String, Double>> preprocessedData = PreprocessData(trainingData);
        trainedBCs = TrainBCs(preprocessedData);
        Map<String, Object[]> predictionsAndConfidences = GeneratePredictionsAndConfidences(trainedBCs,
preprocessedData);

        List<Map<String, Double>> bcPredictions = new ArrayList<>();
        List<Map<String, Double>> confidenceScores = new ArrayList<>();
        List<Double> trueLabels = new ArrayList<>();

        for(int i = 0; i < trainingData.size(); i++){
            Map<String, Double> bcPredictionRow = new HashMap<>();
            Map<String, Double> confidenceScoreRow = new HashMap<>();
            for(String modelName : trainedBCs.keySet()){
                bcPredictionRow.put(modelName,
predictionsAndConfidences.get(modelName)[0].equals("Attack") ? 1.0 : 0.0);
                confidenceScoreRow.put(modelName, (Double) predictionsAndConfidences.get(modelName)[1]);
            }
            bcPredictions.add(bcPredictionRow);
            confidenceScores.add(confidenceScoreRow);
            trueLabels.add(this.trueLabels.get(i));
        }
        TrainMetaLearner(bcPredictions, confidenceScores, trueLabels);
    }

    // Detection Phase
    public void detect() {
        weights = InitializeWeights(trainedBCs);
        performanceHistory = InitializePerformanceHistory(trainedBCs);

        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
        scheduler.scheduleAtFixedRate(() -> {
            realTimeFeatures = GetRealTimeFeatures();
            if (realTimeFeatures != null && !realTimeFeatures.isEmpty()) {
                List<Map<String, Double>> preprocessedFeatures = PreprocessData(realTimeFeatures);
                Map<String, String> bcPredictions = new HashMap<>();
```

```
                Map<String, Double> confidenceScores = new HashMap<>();

                for (String modelName : trainedBCs.keySet()) {
                    Object[] result = GetConfidence(trainedBCs.get(modelName), preprocessedFeatures);
 bcPredictions.put(modelName, (String) result[0]);
                    confidenceScores.put(modelName, (Double) result[1]);
                }

                String finalPrediction = MetaLearnerPredict(bcPredictions, confidenceScores, weights);

                if (System.currentTimeMillis() % MON_WIN_DUR == 0) {
                    Map<String, Double> performance = Evaluate(finalPrediction, trueLabels); //
 Placeholder true labels
                    UpdatePerformanceHistory(performanceHistory, performance);
                    weights = AdjustWeights(weights, performanceHistory);
                }

                if (finalPrediction.equals("Attack")) {
                    SendAlert(NewAlert("LDMC", "Network", 3, System.currentTimeMillis(), new
 ArrayList<>())); // Placeholder alert
                    try {
                        Thread.sleep(MONITORING_INTERVAL);
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
            }
        }, 0, MONITORING_INTERVAL, TimeUnit.MILLISECONDS);
    }

    // Example Usage (Replace with your actual controller integration)
    public static void main(String[] args) {
        AMCE amce = new AMCE();

        // Placeholder Training Data (Replace with your actual data)
        List<Map<String, Double>> trainingData = new ArrayList<>();
        List<Double> trainingLabels = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            Map<String, Double> dataPoint = new HashMap<>();
            dataPoint.put("feature1", Math.random());
            dataPoint.put("feature2", Math.random());
            trainingData.add(dataPoint);
            trainingLabels.add(Math.random() < 0.5 ? 0.0 : 1.0); // 0.0 for normal, 1.0 for attack
        }

        amce.train(trainingData, trainingLabels);
        amce.detect();

        // Keep the main thread alive (or use a proper controller integration)
        try {
            Thread.currentThread().join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Explanation:

1. **Imports:**
   - Imports necessary Java utility classes.
   - Placeholder imports for machine learning libraries.
2. **Configuration:**
   - MONITORING_INTERVAL and MON_WIN_DUR are defined as constants.
3. **Data Structures:**
   - trainedBCs: Stores trained base classifiers.
   - metaLearner: Stores the trained meta-learner (ANN).
   - weights: Stores weights assigned to base classifiers.
   - performanceHistory: Stores performance history.
   - trainingSet: Stores training data.
   - realTimeFeatures: Stores real-time features.
   - trueLabels: Stores true labels.
4. **Placeholder Methods:**
   - SendAlert(): Sends an alert message to the controller.
   - NewAlert(): Creates a new alert.
   - PreprocessData(): Preprocesses data.
   - TrainBCs(): Trains base classifiers.
   - TrainMetaLearner(): Trains the meta-learner.
   - GeneratePredictionsAndConfidences(): Generates predictions and confidence scores.
   - GetConfidence(): Gets prediction and confidence from a base classifier.
   - AdjustWeights(): Adjusts weights.
   - Evaluate(): Evaluates performance.
   - InitializeWeights(): Initializes weights.
   - InitializePerformanceHistory(): Initializes performance history.
   - GetRealTimeFeatures(): Gets real-time features.
   - MetaLearnerPredict(): Meta-learner prediction.
   - UpdatePerformanceHistory(): Updates performance history.
5. **Training Phase (train()):**
   - Preprocesses training data.
   - Trains base classifiers.
   - Generates predictions and confidence scores.
   - Trains the meta-learner.
6. **Detection Phase (detect()):**
   - Initializes weights and performance history.
   - Uses a ScheduledExecutorService to execute the detection logic periodically.
   - Gets real-time features.
   - Preprocesses features.
   - Gets predictions and confidence scores from base classifiers.
   - Gets final prediction from the meta-learner.
   - Evaluates performance and adjusts weights periodically.
   - Sends an alert if an attack is detected.
7. **Example Usage (main()):**
   - Creates an AMCE instance.
   - Generates placeholder training data.
   - Trains the model.
   - Starts the detection phase.
   - Keeps the main thread alive.

Deployment Instructions:

1. **Controller Integration:**
    - Integrate the AMCE class into your Java-based controller (Beacon, Floodlight, OpenDaylight, ONOS).
    - Use the controller's APIs to retrieve real-time features from P4-enabled switches.
    - Implement the alert mechanism to send alerts to the appropriate modules.
2. **Machine Learning Libraries:**
    - Include the necessary machine learning libraries (e.g., Spark MLlib, Weka, or custom implementations) in your controller project.
    - Replace the placeholder machine learning logic with your actual implementations.
3. **Training Data:**
    - Prepare your training data (e.g., CICIoT2023, CICIoMT2024).
    - Load the training data into the AMCE application.
4. **Controller Startup:**
    - Start your controller.
    - The AMCE application will start its training and detection phases.
5. **Testing:**
    - Generate SYN flood traffic in your SD-IoT network.
    - Monitor the controller for alerts.
    - Verify the accuracy of the attack detection.
6. **Real-Time Data Input:**
    - Connect the P4 switches that contain the data to the controller.
7. **Model Persistence:**
    - Implement model persistence to save the trained base classifiers and meta-learner. This will prevent retraining every time the controller restarts.
8. **Error Handling:**
    - Add error handling to your code to catch exceptions.
9. **Performance Tuning:**
    - Tune the AMCE algorithm parameters (e.g., monitoring interval, thresholds, learning rates) to optimize performance.
10. **Controller Communication:**
    - Implement the necessary communication with the controller.