Python Implementation of AMCE Algorithm

```
# AMCE Algorithm Implementation (Python)

import time
import threading
import queue
import random  # Placeholder for data generation
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
import xgboost as xgb
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

class AMCE:
    def __init__(self, mon_win_dur=5, monitoring_interval=1):
        self.mon_win_dur = mon_win_dur
        self.monitoring_interval = monitoring_interval
        self.trained_bcs = {}  # Placeholder for trained base classifiers
        self.meta_learner = None  # Placeholder for meta-learner (ANN)
        self.weights = {}
        self.performance_history = []
        self.training_set = []  # Placeholder for training data
        self.real_time_features = queue.Queue()  # Queue for real-time features
        self.true_labels = []  # Placeholder for True labels from training.

    def send_alert(self, alert):
        print(f"Alert sent: {alert}")

    def new_alert(self, src_ctrl_id, tgt_net_seg, atk_sev, ts, ext_feats):
        return {
            "srcCtrlID": src_ctrl_id,
            "tgtNetSeg": tgt_net_seg,
            "atkSev": atk_sev,
            "ts": ts,
            "extFeats": ext_feats,
        }

    def preprocess_data(self, data):
        # Implement data preprocessing logic here
        return data  # Return the preprocessed data

    def train_bcs(self, training_data):
        # Implement training of base classifiers (KNN, DT, RF, SVM, XGBoost)
        # Store trained models in trained_bcs dictionary
        trained_models = {
            "KNN": "Placeholder KNN Model",
            "DT": "Placeholder DT Model",
            "RF": "Placeholder RF Model",
            "SVM": "Placeholder SVM Model",
            "XGBoost": "Placeholder XGBoost Model",
        }
        return trained_models

    def train_meta_learner(self, bc_predictions, confidence_scores, labels):
        # Implement training of the meta-learner (ANN)
        self.meta_learner = "Placeholder Meta-Learner"

    def generate_predictions_and_confidences(self, trained_models, data):
```

```
        results = {}
        for model_name in trained_models:
            results[model_name] = ("Placeholder Prediction", 0.8)  # Placeholder prediction and
confidence
        return results

    def get_confidence(self, bc, features):
        # Implement logic to get prediction and confidence from a base classifier
        return ("Placeholder Prediction", 0.8)  # Placeholder prediction and confidence

    def adjust_weights(self, weights, performance_history):
        # Implement logic to adjust weights based on performance history
        for model_name in weights:
            weights[model_name] = 1.0  # Placeholder: Set all weights to 1.0
        return weights

    def evaluate(self, final_prediction, true_labels):
        # Implement evaluation logic (Accuracy, Precision, Sensitivity, Balanced F1 Score, TNR)
        metrics = {
            "accuracy": 0.9,
            "precision": 0.9,
            "sensitivity": 0.9,
            "f1": 0.9,
            "tnr": 0.9,
        }
        return metrics

    def initialize_weights(self, trained_bcs):
        initial_weights = {}
        for model_name in trained_bcs:
            initial_weights[model_name] = 1.0
        return initial_weights

    def initialize_performance_history(self, trained_bcs):
        return []

    def get_real_time_features(self):
        try:
            return self.real_time_features.get(timeout=1) # get with timeout
        except queue.Empty:
            return None

    def meta_learner_predict(self, bc_predictions, confidence_scores, weights):
        # Implement meta-learner prediction logic
        return "Normal"  # Placeholder: Predict "Normal" or "Attack"

    def update_performance_history(self, performance_history, performance):
        performance_history.append(performance)

    def train(self, training_data, labels):
        self.training_set = training_data
        self.true_labels = labels
        preprocessed_data = self.preprocess_data(training_data)
        self.trained_bcs = self.train_bcs(preprocessed_data)
        predictions_and_confidences = self.generate_predictions_and_confidences(self.trained_bcs,
preprocessed_data)

        bc_predictions = []
        confidence_scores = []
        for i in range(len(training_data)):
            bc_prediction_row = {}
            confidence_score_row = {}
```

```
            for model_name in self.trained_bcs:
                bc_prediction_row[model_name] = 1.0 if predictions_and_confidences[model_name][0] ==
"Attack" else 0.0
                confidence_score_row[model_name] = predictions_and_confidences[model_name][1]
            bc_predictions.append(bc_prediction_row)
            confidence_scores.append(confidence_score_row)

        self.train_meta_learner(bc_predictions, confidence_scores, labels)

    def detect(self):
        self.weights = self.initialize_weights(self.trained_bcs)
        self.performance_history = self.initialize_performance_history(self.trained_bcs)

        def detection_loop():
            while True:
                features = self.get_real_time_features()
                if features:
                    preprocessed_features = self.preprocess_data(features)
                    bc_predictions = {}
                    confidence_scores = {}

                    for model_name in self.trained_bcs:
                        prediction, confidence = self.get_confidence(self.trained_bcs[model_name],
preprocessed_features)
                        bc_predictions[model_name] = prediction
                        confidence_scores[model_name] = confidence

                    final_prediction = self.meta_learner_predict(bc_predictions, confidence_scores,
self.weights)

                    if time.time() % self.mon_win_dur == 0:
                        performance = self.evaluate(final_prediction, self.true_labels)  # Placeholder
true labels
                        self.update_performance_history(self.performance_history, performance)
                        self.weights = self.adjust_weights(self.weights, self.performance_history)

                    if final_prediction == "Attack":
                        self.send_alert(self.new_alert("LDMC", "Network", 3, time.time(), []))  #
Placeholder alert
                        time.sleep(self.monitoring_interval)
                else:
                    time.sleep(0.1) # Check again soon if no data.

        threading.Thread(target=detection_loop).start()

# Example Usage (Replace with your actual controller integration)
if __name__ == "__main__":
    amce = AMCE()

    # Placeholder Training Data (Replace with your actual data)
    training_data = []
    training_labels = []
    for _ in range(100):
        data_point = {"feature1": random.random(), "feature2": random.random()}
        training_data.append(data_point)
        training_labels.append(0.0 if random.random() < 0.5 else 1.0)  # 0.0 for normal, 1.0 for attack

    amce.train(training_data, training_labels)
    amce.detect()

    # Placeholder for real-time feature injection
    def feature_injector():
```

```
        while True:
            features = [{"feature1": random.random(), "feature2": random.random()}]
            amce.real_time_features.put(features)
            time.sleep(0.5)

    threading.Thread(target=feature_injector).start()

    # Keep the main thread alive (or use a proper controller integration)
    while True:
       time.sleep(1)
```

Explanation of the Python Code:

1. **Imports:**
    - Imports necessary Python modules.
    - Placeholder imports for machine learning libraries.
2. **AMCE Class:**
    - Initializes configuration and data structures.
3. **Placeholder Methods:**
    - send_alert(): Sends an alert message.
    - new_alert(): Creates a new alert dictionary.
    - preprocess_data(): Preprocesses data.
    - train_bcs(): Trains base classifiers.
    - train_meta_learner(): Trains the meta-learner.
    - generate_predictions_and_confidences(): Generates predictions and confidence scores.
    - get_confidence(): Gets prediction and confidence from a base classifier.
    - adjust_weights(): Adjusts weights.
    - evaluate(): Evaluates performance.
    - initialize_weights(): Initializes weights.
    - initialize_performance_history(): Initializes performance history.
    - get_real_time_features(): Retrieves real-time features from the queue.
    - meta_learner_predict(): Meta-learner prediction.
    - update_performance_history(): Updates performance history.
4. **Training Phase (train()):**
    - Stores training data and labels.
    - Preprocesses data.
    - Trains base classifiers.
    - Generates predictions and confidence scores.
    - Trains the meta-learner.
5. **Detection Phase (detect()):**
    - Initializes weights and performance history.
    - Defines a detection_loop() function that runs in a separate thread.
    - The loop continuously retrieves real-time features from the queue.
    - 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.
    - The detection_loop() is started in a new thread.

6. **Example Usage (if __name__ == "__main__":)**
    - Creates an AMCE instance.
    - Generates placeholder training data.
    - Trains the model.
    - Starts the detection phase.
    - Defines a feature_injector() function to simulate real-time feature injection.
    - Starts the feature_injector() in a new thread.
    - Keeps the main thread alive.

Deployment Instructions:

1. **Controller Integration:**
    - Integrate the AMCE class into your Python-based controller (Ryu or POX).
    - 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:**
    - Install the necessary machine learning libraries (e.g., scikit-learn, XGBoost) in your controller environment.
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. Libraries such as pickle can be used for this.
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.
11. **Feature Queue:**
    - The code uses a queue to pass features. Make sure the controller and P4 switch data transfer methods are correctly placing data into this queue.