Abstract

In this project, we develop and implement a reinforcement learning-based framework for online active learning, inspired by the CBeAL (Contextual Bandit-Based Ensemble Active Learning) methodology. Our approach addresses the challenge of balancing exploration and exploitation during sample selection in streaming scenarios, particularly within industrial cyber-physical systems. To this end, we construct a diverse committee of active learning agents, including max-min distance, high-dimensional density, margin-based, and reinforced exploitation strategies. These agents are integrated via an EXP4.P ensemble controller enhanced with an EWMA-based flipping mechanism to ensure dynamic adaptability and prevent agent dominance. We apply this framework to the UCI Spambase dataset and train a two-layer MLP classifier to evaluate decision-making efficiency. Experimental results demonstrate that a compact agent ensemble can achieve high accuracy (93.49%) and F1 score (91.74%) while labeling significantly fewer samples than full supervision. Our findings validate the effectiveness of contextual bandit-based selection and lay the foundation for future deployment in real-world data stream environments.

Contents

1. Introduction to Dataset

The UCI Spambase dataset[1] is a well-known benchmark dataset widely used for experiments in spam email detection. It was donated by Mark Hopkins, Erik Reeber, George Forman, and Jaap Suermondt from Hewlett-Packard Labs and is hosted by the UCI Machine Learning Repository. This dataset is designed to support research on binary classification problems, specifically distinguishing between spam and non-spam (ham) emails based on extracted textual features.

The dataset consists of 4,601 email instances, each described by 57 continuous input features and 1 binary target variable. The features represent word frequencies, character frequencies, and statistical measures related to capital letters usage. The target variable is a binary label where: (1) 1 indicates that the email is classified as spam, and (2) 0 indicates a non-spam (legitimate) email.

1.1. Feature Description

Word Frequency Attributes (48 features): These features represent the percentage occurrence of specific words in the email text. For example, attributes like word_freq_make, word_freq_address, and word_freq_free measure the frequency of common words often found in spam emails.

Table 1: Overview of Spambase Dataset Features

Feature Group	**Count**	**Example Features**	**Description**	**Data Type**
Word Frequencies	48	`word_freq_make`, `word_freq_free`, `word_freq_business`	Percentage occurrence of specific words in the email text	Continuous (0–100)
Character Frequencies	6	`char_freq_;`, `char_freq_!`, `char_freq_$`	Frequency of specific characters used in emails, such as punctuation or special symbols	Continuous (0–100)
Capital Run Lengths	3	`capital_run_length_average`, `capital_run_length_longest`	Statistics on sequences of capital letters, including average length, longest run, and total count	Continuous (≥ 0)
Target Label	1	`spam`	Binary class label indicating whether the email is spam (1) or not (0)	Binary (0 or 1)

Character Frequency Attributes (6 features): These capture the percentage of characters such as ;, (, [, !, $, and # in the text, which are often used in promotional or malicious emails.

Capital Run Length Attributes (3 features): (1) capital_run_length_average: the average length of uninterrupted sequences of capital letters. (2) capital_run_length_longest: the length of the longest such sequence. (3) capital_run_length_total: the total number of capital letters in the email.

These attributes are derived using pre-processing scripts that tokenize the emails and compute statistical metrics on token distributions. The design reflects linguistic patterns typical of spam messages, such as excessive use of certain keywords or stylistic markers (e.g., all-caps text or punctuation-heavy sentences).

In order to better understand this dataset, we performed a visual data analysis on some representative features in the data. These histograms shown in figure 1 use real Spambase data to show the distribution of the following features: `word_freq_make`: This word appears less frequently in most emails, with only a few appearing more frequently; `word_freq_free`: This word is usually associated with spam and also shows a similar right-skewed distribution; `char_freq_!`: The frequency of exclamation marks increases significantly in some emails, reflecting the exaggerated style common in spam; `capital_run_length_average`: The average continuous capital length varies widely, and some emails contain long capital paragraphs, which is one of the typical characteristics of spam.

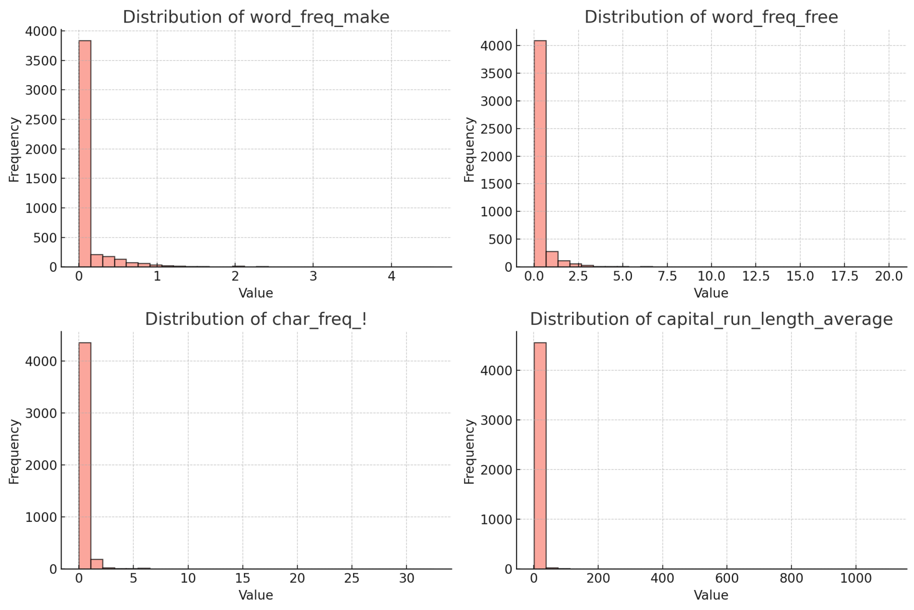

Figure 1: Analysis of selected features

For `word_freq_make`, most samples are concentrated around 0, indicating that "make" hardly appears in most emails, and only a few samples have a high frequency of occurrence. Although "make" is a common verb, it may appear more frequently in advertisements or spam emails with sentences such as "make money fast". Therefore, extreme values may be potential signs of spam.

For `word_freq_free`,most samples have values close to 0, but there is a long right tail, indicating that the word appears frequently in some emails. This feature is one of the strong indicator words in spam identification. "Free" is common in topics such as promotions, gifts, and winning prizes, so emails with high values of this feature are more likely to be spam. The right-skewed distribution clearly reveals its concentration on a certain type of email (spam).

For `char_freq_!`, although the frequency of exclamation marks is low in most emails, the right tail is long, which means that some emails use a lot of exclamation marks.Exclamation marks are frequently used to express emotions or emphasize, and are especially abused in spam (for example: "Buy now!", "Limited time offer!"). High values of this feature usually correspond to content with highly promotional or fraudulent language.

For `capital_run_length_average`, the value fluctuates widely, and the average length of capital letters in some emails is significantly higher than that in other emails. Capital letters are often used in spam to attract attention, such as "FREE", "WINNER", "URGENT". Therefore, the concentrated use of capital letters is another typical feature of spam. High values of this feature may correspond to long capital segments in the email title or body.

1.2. Use Cases and Applications

The Spambase dataset has been extensively used in academic research and coursework to:

(1) Benchmark classification algorithms (e.g., Naive Bayes, SVM, Decision Trees).

(2) Explore feature selection and dimensionality reduction techniques.

(3) Test text mining and natural language processing (NLP) pipelines.

(4) Study imbalanced classification, although this dataset is relatively balanced (roughly 39.4% of instances are spam).

Because it captures real-world patterns in email communications while remaining relatively small and easy to process, it remains a standard testbed for developing and evaluating spam filters and learning models.

1.3. Limitations

Although the dataset is valuable for experimentation, it is important to note: The emails are represented by engineered features rather than raw text, which limits the ability to apply modern deep learning or embedding-based NLP models directly. What's more, the dataset was collected in the 1990s, so the spam patterns may differ significantly from those in modern email systems.

2. Significant Progress

In our project on reinforcement learning for balancing exploration and exploitation in active learning, we have selected and implemented the core methodology proposed in the paper *Ensemble Active Learning by Contextual Bandits for AI Incubation in Manufacturing (CBeAL)*[2]. This section outlines the significant progress achieved so far, including the theoretical grounding, algorithmic implementation, and initial validation using the Spambase dataset.

The CBeAL framework addresses the central challenge of active learning in online scenarios: determining when to acquire new labeled data while balancing the need to explore underrepresented regions and exploit known boundaries. It formulates this trade-off using contextual multi-armed bandits, where each "arm" corresponds to a decision about whether or not to label an incoming sample. The novelty of CBeAL lies in its dynamic ensemble of active learning agents, each designed to promote either exploration or exploitation. The decision to acquire a label is guided by EXP4.P with an EWMA-based flipping mechanism to ensure adaptability over time.

2.1. Active Learning Agents

We implemented the following agents described in the CBeAL framework:

Max-Min Distance Based Agent (ALAgent_X_MMd_B): This agent computes the distance of a new sample x_t to the samples in a sliding window W, and selects it with probability:

$$p_t = \frac{\min_{x_i \in W} \|x_t - x_i\|}{\max_{x_j \in W} \min_{k \neq j} \|x_j - x_k\|}$$

This strategy encourages uniform space exploration by favoring samples far from existing labeled data.

High-Dimensional Density Agent (ALAgent_X_hD_B): This agent estimates local sparsity by comparing the distance from the new sample to the maximum pairwise distances in W:

$$p_t = \min\left(\frac{\text{count}(\|x_t - x_i\| > \tau)}{L \cdot \tau}, 1.0\right)$$

It aims to sample from low-density regions that may uncover new cluster structures.

Margin-Based Agent (ALAgent_X_M_B): This agent targets exploitation by selecting samples close to the decision boundary, which are more likely to improve classification accuracy. A sample is selected if its prediction margin $m = p - 0.5$ is lower than a dynamic threshold, which is updated over time:

$$\theta_{t+1} = \theta_t \cdot (1 \pm s)$$

Here, s is a shrinking/expanding factor depending on whether the last sampled margin was informative.

Reinforced Exploitation Agent (RAL_B): This agent adjusts a certainty threshold θ_t dynamically using reinforcement learning principles. When a prediction is incorrect and the sample is acquired, θ_t is slightly increased to make the agent more selective; if correct, it is reduced:

$$\theta_{t+1} = \min \left\{ \theta_t \left(1 + \eta \cdot \left(1 - 2 \frac{r_t}{\rho_-} \right) \right), 1 \right\}$$

This learning mechanism allows the agent to self-tune its focus on uncertain regions near the classification boundary.

2.2. EXP4.P-EWMA Ensemble Controller

The controller, implemented as `Exp4P_EN_SWAP`, combines agent outputs $\xi_t^i \in \mathbb{R}^2$ into an overall decision:

$$P_t = (1 - K p_{\min}) \cdot \sum_{i=1}^{N} \alpha_{i,t} \cdot \xi_t^i + p_{\min}$$

where $\alpha_{i,t}$ represents the decision power of agent i. This soft voting system adapts over time based on reward feedback.

After each iteration, agent weights $\alpha_{i,t}$ are updated:

$$\alpha_{i,t+1} = \alpha_{i,t} \cdot \exp \left(\frac{p_{\min}}{2} \cdot \left(\hat{g}_{i,t} + \hat{v}_{i,t} \sqrt{\frac{\ln N}{KT}} \right) \right)$$

To prevent premature dominance by one agent (e.g., an overly confident exploitation agent), a control mechanism based on exponentially weighted moving average (EWMA) is used. When the standardized weight $\alpha_{s,i,t}$ exceeds upper or lower bounds, a weight flipping occurs:

$$\alpha_{i,t+1} \leftarrow 2\mu - \alpha_{i,t+1}$$

This forces exploration and preserves diversity in agent behavior.

2.3. Reward Function

The reward signal is set as follows:

$$r_t = \begin{cases} \rho^+, & \hat{y}_t \neq y_t \\ \rho^-, & \hat{y}_t = y_t \end{cases}$$

It gives positive feedback (+1) when a selected sample would have been misclassified, and negative feedback (-1) otherwise, effectively promoting informative sample acquisition. In our implementation, $\rho^+ = 2$ and $\rho^- = -1$. A positive reward $\rho^+ = 2$ is assigned when the model would misclassify the sample without annotation. A negative reward $\rho^- = -1$ penalizes uninformative acquisitions. This design encourages the controller to prioritize samples that meaningfully improve model performance.

2.4. Data Pipeline and Experimental Setup

We used the UCI Spambase dataset to simulate a binary classification task with streaming data:

- **Preprocessing:** We utilize standardization using StandardScaler. And we stratified train-test split (80-20). In addition, we downsampling majority class to ensure class balance.
- **Model:** Two-layer MLP with ReLU activation and sigmoid output:

```
self.model = nn.Sequential(
    nn.Linear(input_dim, hidden_dims[0]),
    nn.ReLU(),
    nn.Linear(hidden_dims[0], hidden_dims[1]),
    nn.ReLU(),
    nn.Linear(hidden_dims[1], 1),
    nn.Sigmoid()
)
```

- **Training:** When we do the model training process, we optimized the model with Adam and early stopping.What's more, we tuned the hyperparameters via stratified k-fold cross-validation ($k = 3$) and we used F1 score as the objective

2.5. Initial Validation and Integration

- collects per-agent binary decision arrays. The function `get_agent_outputs` computes the decision output of each agent over a dataset, given the model predictions and margins.
- The ensemble fusion is done via `exp4p_fusion`, which performs:

$$\text{final\_decision} = \left(\frac{\sum_i \alpha_i d_i}{\sum_i \alpha_i} > 0.5 \right)$$

  It integrates the decisions using reward-weighted voting.
- Rewards are assigned based on `reward_fn`, comparing predicted labels and ground-truths.

3. Simulation Results

In Table2 we report the number of selected samples, accuracy and F1 score for three different agent-fusion strategies (Agent$_2$, Agent$_4$, Agent$_6$) and the baseline MLP classifier. Agent$_2$ fuses two active-learning agents (ALAgent_X_MMd_B + ALAgent_X_hD_B), Agent$_4$ adds two more (ALAgent_X_M_B + RAL_B), and Agent$_6$ integrates all six committee members (the three AL agents plus RAL_B, RAL_B_EXP4P, Exp4P_EN_SWAP). The baseline uses a two-hidden-layer MLP (128$\rightarrow$64 units, ReLU activations, sigmoid output) trained on the full dataset.

From the results in Table 2, Agent$_2$ achieves the highest sample efficiency, reaching 93.49% accuracy and 91.74% F1 with only 1,324 samples (45% of full dataset). Agent$_4$ and Agent$_6$ require more labels (1,939 and 2,103 samples) but do not substantially outperform Agent$_2$, indicating diminishing returns from adding additional agents. The baseline MLP trained on all 2,900 samples attains 92.51% accuracy and 90.86% F1, which is outperformed by Agent$_2$ and matched closely by Agent$_6$ with fewer labels. This demonstrates that a carefully chosen small committee can yield superior performance while significantly reducing labeling effort.

Table 2: Performance Comparison of Agent Fusion Strategies and Baseline MLP

Method	# Samples	Accuracy	F1 Score
Agent_2 (*al1* + *al2*)	1324	0.9349	0.9174
Agent_4 (*al1* + *al2* + *al3* + *rl1*)	1939	0.9240	0.9059
Agent_6 (*al1* + *al2* + *al3* + *rl1* + *rl2* + *rl3*)	2103	0.9316	0.9159
Baseline ($\text{MLP}_{128,64}$)	2900	0.9251	0.9086

4. Next Work Plan

We have successfully reimplemented the CBeAL framework and demonstrated its core mechanisms in a synthetic scenario. The next steps include applying the system to more realistic streaming datasets and quantitatively analyzing the cumulative regret and accuracy improvement over baselines. We also plan to conduct ablation studies to assess each agent's contribution and explore dynamic budget constraints in the annotation process.

References

[1] Reeber Erik Forman George Hopkins, Mark and Jaap Suermondt. Spambase. UCI Machine Learning Repository, 1999. DOI: https://doi.org/10.24432/C53G6X.

[2] Yifan Zeng, Xiaowei Chen, and Ruixin Jin. Ensemble active learning by contextual bandits for ai incubation in manufacturing. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 15(1):1–26, 2023.

A. Appendix A: Supplementary Material

Simulation code for this submitted draft.

```
# 0. Import libraries
# ==========================
import pandas as pd
import numpy as np
import random
from copy import deepcopy
from scipy.stats import entropy
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.utils import resample
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, ParameterGri
from sklearn.metrics import f1_score, accuracy_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# ==========================
# 0.5 Hyperparameters & thresholds
# ==========================
DENSITY_QUANTILE     = 50
MARGIN_QUANTILE      = 15
RL_CONF_THRESHOLD    = 0.6
EWMA_MEAN            = 0.5
EXP4P_ROUNDS         = 10
EXPLORATION_RATE     = 0.15
EPOCHS               = 100
PATIENCE             = 10
CV_FOLDS             = 3
HYPERPARAMS          = {
    'hidden_dims': [(64, 32)],
    'lr': [1e-3]
}

# ==========================
# 1. Define Agent Classes
# ============================
class ALAgent_X_MMd_B:
    """MaxMin distance AL agent."""
    def __init__(self, DIM, window_size, density_threshold, budget=None):
        self._q = DIM
        self._Size = window_size
        self._d_threshod = density_threshold
        self._budget = budget
        self._n_seen = 0
        self._n_acquired_samples = 0
        self._W = []
        self._probs = []

    def NNlocalDensity(self, x_t):
        if self._n_seen < 2:
            self._W.append(x_t)
            return True
        W_array = np.vstack(self._W)
        D_mat = euclidean_distances(W_array)
        np.fill_diagonal(D_mat, np.nan)
        min_dists = np.nanmin(D_mat, axis=1)
        u = x_t.flatten()
        dists = np.linalg.norm(W_array - u, axis=1)
        prob = np.min(dists) / np.max(min_dists)
        prob = min(prob, 1.0)
        self._probs.append(prob)
        decision = bool(np.random.binomial(1, prob))
        # update sliding window
        if len(self._W) < self._Size:
            self._W.append(x_t)
        else:
            self._W.pop(0)
            self._W.append(x_t)
        return decision

    def get_agent_decision(self, x_t, certainty=None, margin=None):
        dec = self.NNlocalDensity(x_t.reshape(1, -1))
        self._n_seen += 1
        if dec:
            self._n_acquired_samples += 1
        return [1.0 - float(dec), float(dec)]


class ALAgent_X_hD_B:
    """ High Dimensional density AL agent."""
    def __init__(self, DIM, window_size, density_threshold, budget=None):
        self._q = DIM
        self._Size = window_size
        self._d_threshod = density_threshold
        self._budget = budget
        self._n_seen = 0
        self._n_acquired_samples = 0
        self._W = []

    def NNlocalDensity(self, x_t):
        if self._n_seen < 2:
            self._W.append(x_t)
            return True
        W_array = np.vstack(self._W)
        D_mat = euclidean_distances(W_array)
        np.fill_diagonal(D_mat, np.nan)
        max_dists = np.nanmax(D_mat, axis=1)
        u = x_t.flatten()
        dists = np.linalg.norm(W_array - u, axis=1)
        count = np.sum(dists > max_dists * self._d_threshod)
        prob = min(count / (len(self._W) * self._d_threshod), 1.0)
        decision = bool(np.random.binomial(1, prob))
        if len(self._W) < self._Size:
            self._W.append(x_t)
        else:
            self._W.pop(0)
            self._W.append(x_t)
        return decision

    def get_agent_decision(self, x_t, certainty=None, margin=None):
        dec = self.NNlocalDensity(x_t.reshape(1, -1))
        self._n_seen += 1
        if dec:
            self._n_acquired_samples += 1
        return [1.0 - float(dec), float(dec)]


class ALAgent_X_M_B:
    """Margin based AL agent."""
    def __init__(self, adjust_s, window_size, threshold_margin, budget=None):
        self._s = adjust_s
        self._Size = window_size
        self._threshold_margin = threshold_margin
        self._budget = budget
        self._n_seen = 0
        self._n_acquired_samples = 0
        self._W = []

    def NNlocalDensity(self, x_t):
        if self._n_seen < 2:
            self._W.append(x_t)
            return True
        W_array = np.vstack(self._W)
        D_mat = euclidean_distances(W_array)
        np.fill_diagonal(D_mat, np.nan)
        min_dists = np.nanmin(D_mat, axis=1)
        u = x_t.flatten()
        dists = np.linalg.norm(W_array - u, axis=1)
        count = np.sum(dists < min_dists)
        return bool(count > 0)

    def get_agent_decision(self, x_t, certainty, margin):
        self._n_seen += 1
        dec = False
        if self.NNlocalDensity(x_t):
            random_margin = np.random.normal(0, 1) * self._threshold_margin
            dec = (margin < random_margin)
            if dec:
                self._threshold_margin *= (1 - self._s)
                self._n_acquired_samples += 1
            else:
                self._threshold_margin *= (1 + self._s)
        if len(self._W) < self._Size:
            self._W.append(x_t)
        else:
            self._W.pop(0)
            self._W.append(x_t)
        return [1.0 - float(dec), float(dec)]


class RAL_B:
    """Basic RAL with uncertainty +    greedy   ."""
    def __init__(self, threshold_uncertainty, threshold_greedy, eta, budget=None):
        self._threshold_uncertainty = threshold_uncertainty
        self._threshold_greedy = threshold_greedy
        self._eta = eta
        self._alpha = [1.0]
        self._n_seen = 0
        self._n_acquired_samples = 0
        self.label_decision = False

    def ask_committee(self, x_certainty):
        committee_decision = (x_certainty < self._threshold_uncertainty)
        return committee_decision, [committee_decision]

    def should_label(self, committee_decision):
        self._n_seen += 1
        if random.random() < self._threshold_greedy:
            self.label_decision = True
        else:
            self.label_decision = bool(committee_decision)
        if self.label_decision:
            self._n_acquired_samples += 1

    def get_agent_decision(self, x_t, certainty, margin=None):
        comm, _ = self.ask_committee(certainty)
        self.should_label(comm)
        return [1.0 - float(self.label_decision), float(self.label_decision)]


class RAL_B_EXP4P(RAL_B):
    """RAL with EXP4.P weight updates."""
    def __init__(self, threshold_uncertainty, eta, reward, penalty, mode):
        super().__init__(threshold_uncertainty, 0.0, eta)
        self._reward = reward
        self._penalty = penalty
        self._mode = mode



class Exp4P_EN_SWAP:
    """EXP4.P with EWMA and swap mechanism."""
    def __init__(self, committee_agents, max_T, delta, p_min, reward, penalty,
                 budget=None, budget_mode='soft', greedy_threshold=0.01,
                 mode='ewma', ewma_k=5, ewma_Lambda=0.3, ewma_mu=0.5):
        self._committee_agents   = committee_agents
        self._K                  = 2
        self._N                  = len(committee_agents)
        self._delta              = delta
        self._reward             = reward
        self._penalty            = penalty
        self._budget_mode        = budget_mode
        self._threshold_greedy   = greedy_threshold
        self._p_min              = p_min or np.sqrt(np.log(self._N)/(self._K*max_T
        w0 = np.ones(self._N)
        self._model = {
            'w':     [w0.tolist()],
            'w_std': [(w0 / w0.sum()).tolist()]
        }
        self._all_decision = []
        self._exp_decision = []
        self.label_decision = False
        self._mode           = mode
        self._ewma_k         = ewma_k
        self._ewma_Lambda    = ewma_Lambda
        self._ewma_mu        = ewma_mu
        self._E_t             = []


    def ask_committee(self, x_t, certainty, margin):
        decs = [
            agent.get_agent_decision(x_t, certainty, margin)[1]
            for agent in self._committee_agents
        ]
        self._all_decision.append(decs)
        w_std = np.array(self._model['w_std'][-1])
        probs = []
        for action in [0, 1]:
            weighted = [
                w_std[i] * ((1 - action) + action * decs[i])
                for i in range(self._N)
            ]
            p = (1 - self._K * self._p_min) * sum(weighted) + self._p_min
            probs.append(p)
        probs = np.array(probs)
        probs /= probs.sum()
        self._exp_decision.append(probs.tolist())
        if random.random() < self._threshold_greedy:
            choice = 1
        else:
            choice = int(np.argmax(probs))
        self.label_decision = bool(choice)
        return self.label_decision, probs.tolist(), decs


    def get_agent_decision(self, x_t, certainty, margin=None):
        dec, _, _ = self.ask_committee(x_t, certainty, margin)
        return [1.0 - float(dec), float(dec)]


# ==========================
# 2. Data Preprocessing (spambase dataset)
# ==========================
column_names = [f'feature_{i}' for i in range(57)] + ['label']
df = pd.read_csv(r"D:\zhuomian\10-paper\spambase\spambase.data",
                 header=None, names=column_names)
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split (stratified)
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)

# Downsample majority class to balance
Xy = np.hstack((X_train, y_train.reshape(-1,1)))
majority = Xy[y_train == 0]
minority = Xy[y_train == 1]
maj_down = resample(majority, replace=False, n_samples=len(minority), random_state=
balanced = np.vstack((minority, maj_down))
np.random.shuffle(balanced)
X_train_balanced = balanced[:, :-1]
y_train_balanced = balanced[:, -1]

# Prepare test loader
X_test_tensor  = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor  = torch.tensor(y_test, dtype=torch.float32)
test_dataset    = TensorDataset(X_test_tensor, y_test_tensor)
test_loader     = DataLoader(test_dataset, batch_size=64, shuffle=False)


# ==========================
# 3. Define MLP Classifier
# ==========================
class MLPClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dims=(128,64)):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, hidden_dims[0]),
            nn.ReLU(),
            nn.Linear(hidden_dims[0], hidden_dims[1]),
            nn.ReLU(),
            nn.Linear(hidden_dims[1], 1),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.model(x)


# ==========================
# 4. Instantiate Agents
# ==========================
input_dim      = X_train_balanced.shape[1]
window_size    = 50
density_thresh = DENSITY_QUANTILE / 100.0
margin_thresh  = MARGIN_QUANTILE / 100.0
adjust_s       = 0.1
```

```
al1 = ALAgent_X_MMd_B(input_dim, window_size, density_thresh)
al2 = ALAgent_X_hD_B(input_dim, window_size, density_thresh)
al3 = ALAgent_X_M_B(adjust_s, window_size, margin_thresh)

rl1 = RAL_B(RL_CONF_THRESHOLD, EXPLORATION_RATE, eta=1.0)
rl2 = RAL_B_EXP4P(RL_CONF_THRESHOLD, eta=1.0, reward=1, penalty=-1, mode='const')
rl3 = Exp4P_EN_SWAP(
    committee_agents=[al1, al2, al3],
    max_T=X_train_balanced.shape[0],
    delta=0.1,
    p_min=None,
    reward=1,
    penalty=-1,
    greedy_threshold=EXPLORATION_RATE,
    ewma_mu=EWMA_MEAN
)

agents = [al1, al2, al3, rl1, rl2, rl3]


# ===========================
# 5. Helper to collect agent decisions
# ===========================
def get_agent_outputs(X, model, agents):
    probs   = torch.sigmoid(model(torch.tensor(X, dtype=torch.float32))).detach().n
    margins = np.abs(probs - 0.5)
    outputs = []
    for agent in agents:
        decisions = []
        for x, p, m in zip(X, probs, margins):
            dec = agent.get_agent_decision(x.reshape(1,-1), p, m)
            if isinstance(dec, (list, np.ndarray)):
                dec = int(dec[1] >= 0.5)
            decisions.append(dec)
        outputs.append(np.array(decisions))
    return outputs


# ===========================
# 6. EXP4.P EWMA Fusion & Reward
# ===========================
def exp4p_fusion(agent_outputs, reward_fn, y_true, y_pred):
    N         = len(agent_outputs)
    alpha     = np.ones(N)
    decisions = np.vstack(agent_outputs)
    for _ in range(EXP4P_ROUNDS):
        explore_mask = (np.random.rand(*decisions.shape) < EXPLORATION_RATE)
        decisions_r  = np.where(explore_mask, 1 - decisions, decisions)
        fused_r      = (alpha @ decisions_r / alpha.sum()) >= 0.5
        rewards      = reward_fn(y_true, y_pred, fused_r)
        for i in range(N):
            alpha[i] *= np.exp(rewards * decisions_r[i])
        alpha /= alpha.sum()
    final = (alpha @ decisions / alpha.sum()) >= 0.5
    return final.astype(int)

def reward_fn(y_true, y_pred, selected):
    wrong = (y_true != y_pred)
    return (wrong & selected)*2.0 + (~wrong & selected)*(-1.0)



# ===========================
# 7. Training utilities
# ===========================
def train_model(model, train_loader, val_loader, lr, epochs, patience):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.BCELoss()
    best_loss = float('inf')
    wait      = 0
    best_w    = None
    for ep in range(epochs):
        model.train()
        for Xb, yb in train_loader:
            optimizer.zero_grad()
            loss = criterion(model(Xb).squeeze(), yb)
            loss.backward(); optimizer.step()
        model.eval()
        losses = []
        with torch.no_grad():
            for Xv, yv in val_loader:
                losses.append(criterion(model(Xv).squeeze(), yv).item())
        val_loss = np.mean(losses)
        if val_loss + 1e-4 < best_loss:
            best_loss, wait, best_w = val_loss, 0, model.state_dict()
        else:
            wait += 1
            if wait >= patience:
                break
    model.load_state_dict(best_w)
    return model

def hyperparameter_search(X, y):
    skf = StratifiedKFold(n_splits=CV_FOLDS, shuffle=True, random_state=42)
    best_score, best_params = -1, None
    for params in ParameterGrid(HYPERPARAMS):
        scores = []
        for tr_idx, va_idx in skf.split(X, y):
            X_tr, X_va = X[tr_idx], X[va_idx]
            y_tr, y_va = y[tr_idx], y[va_idx]
            tr_ds = TensorDataset(torch.tensor(X_tr, dtype=torch.float32),
                                  torch.tensor(y_tr, dtype=torch.float32))
            va_ds = TensorDataset(torch.tensor(X_va, dtype=torch.float32),
                                  torch.tensor(y_va, dtype=torch.float32))
            tr_ld = DataLoader(tr_ds, batch_size=64, shuffle=True)
            va_ld = DataLoader(va_ds, batch_size=64, shuffle=False)
            mdl = MLPClassifier(X.shape[1], hidden_dims=params['hidden_dims'])
            mdl = train_model(mdl, tr_ld, va_ld, params['lr'], EPOCHS, PATIENCE)
            preds, labs = [], []
            mdl.eval()
            with torch.no_grad():
                for Xb, yb in va_ld:
                    out = (mdl(Xb).squeeze().numpy() >= 0.5).astype(int)
                    preds.extend(out.tolist()); labs.extend(yb.numpy().astype(int).
            scores.append(f1_score(labs, preds))
        avg = np.mean(scores)
        if avg > best_score:
            best_score, best_params = avg, params
    return best_params

def train_and_eval(X_tr_full, y_tr_full):
    best_p = hyperparameter_search(X_tr_full, y_tr_full)
    X_tr, X_va, y_tr, y_va = train_test_split(
        X_tr_full, y_tr_full, test_size=0.2, random_state=42, stratify=y_tr_full
    )
    tr_ds = TensorDataset(torch.tensor(X_tr, dtype=torch.float32),
                          torch.tensor(y_tr, dtype=torch.float32))
    va_ds = TensorDataset(torch.tensor(X_va, dtype=torch.float32),
                          torch.tensor(y_va, dtype=torch.float32))
    tr_ld = DataLoader(tr_ds, batch_size=64, shuffle=True)
    va_ld = DataLoader(va_ds, batch_size=64, shuffle=False)
    mdl = MLPClassifier(X_tr_full.shape[1], hidden_dims=best_p['hidden_dims'])
    mdl = train_model(mdl, tr_ld, va_ld, best_p['lr'], EPOCHS, PATIENCE)
    preds, labs = [], []
    mdl.eval()
    with torch.no_grad():
        for Xb, yb in test_loader:
            out = (mdl(Xb).squeeze().numpy() >= 0.5).astype(int)
            preds.extend(out.tolist()); labs.extend(yb.numpy().astype(int).tolist()
    return accuracy_score(labs, preds), f1_score(labs, preds)


# ==============================
# 8. Run Experiments
# ==========================
initial_model = MLPClassifier(input_dim)
_ = train_and_eval(X_train_balanced, y_train_balanced)

agent_outputs = get_agent_outputs(X_train_balanced, initial_model, agents)

results = {}
for k in [2, 4, 6]:
    idx      = list(range(k))
    selected = np.any([agent_outputs[i] for i in idx], axis=0)
    X_sub    = X_train_balanced[selected]
    y_sub    = y_train_balanced[selected]
    acc, f1  = train_and_eval(X_sub, y_sub)
    results[f"Agent_{k}"] = (len(X_sub), acc, f1)

# Baseline on full balanced set
acc, f1 = train_and_eval(X_train_balanced, y_train_balanced)
results['Baseline'] = (len(X_train_balanced), acc, f1)

for name, (n, acc, f1) in results.items():
    print(f"{name}: Samples={n}, Accuracy={acc:.4f}, F1={f1:.4f}")
```