# File: project\_1.py

import numpy as np

import pandas as pd

from tqdm import tqdm

import time

import joblib

from sklearn.datasets import fetch\_openml

from sklearn.model\_selection import train\_test\_split

from sklearn.utils import check\_random\_state

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import DBSCAN

from sklearn.linear\_model import LogisticRegression

from sklearn.neighbors import KNeighborsClassifier

SEED = 42

# Load or fetch the MNIST dataset with a progress bar

def load\_mnist():

    try:

        # Try to load the dataset from the local file if it exists

        mnist = joblib.load('mnist\_dataset.joblib')

    except FileNotFoundError:

        # If the file doesn't exist, fetch the dataset from OpenML

        print('Downloading MNIST...')

        mnist = fetch\_openml(name='mnist\_784', version=1, cache=True, parser='auto')

        # Save the dataset locally

        joblib.dump(mnist, 'mnist\_dataset.joblib')

    return mnist

# Load the MNIST dataset

mnist = load\_mnist()

# Split the data into features and labels

X, y = mnist.data, mnist.target.astype(int)

# Standardize the features

scaler = StandardScaler()

X\_standardized = scaler.fit\_transform(X)

# Split the dataset into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_standardized, y, test\_size=0.2, random\_state=SEED)

# Prototype Selection Methods

def random\_sampling(X\_train, y\_train, M):

    """

    Randomly select M samples from the training set.

    Parameters:

    - X\_train: Features of the training set (NumPy array or Pandas DataFrame)

    - y\_train: Labels of the training set (NumPy array or Pandas DataFrame)

    - M: Number of prototypes to select

    Returns:

    - selected\_prototypes: Randomly selected prototypes

    """

    # Convert Pandas DataFrame to NumPy array if needed

    X\_train = X\_train.values if isinstance(X\_train, pd.DataFrame) else X\_train

    y\_train = y\_train.values if isinstance(y\_train, pd.Series) else y\_train

    # Convert y\_train to a NumPy array

    y\_train\_np = np.array(y\_train)

    # Get the indices of randomly selected prototypes

    selected\_indices = np.random.choice(len(X\_train), size=M, replace=False)

    # Extract the corresponding samples

    selected\_prototypes = X\_train[selected\_indices]

    selected\_labels = y\_train\_np[selected\_indices]

    return selected\_prototypes, selected\_labels

def dbscan\_prototype\_selection(X\_train, y\_train, M, eps=20, min\_samples=10, random\_state=None):

    """

    Use DBSCAN for prototype selection.

    Parameters:

    - X\_train: Features of the training set (NumPy array or Pandas DataFrame)

    - y\_train: Labels of the training set (NumPy array or Pandas DataFrame)

    - M: Number of prototypes to select

    - eps: The maximum distance between two samples for one to be considered in the neighborhood of the other

    - min\_samples: The number of samples (or total weight) in a neighborhood for a point to be considered as a core point

    - random\_state: Seed for reproducibility

    Returns:

    - selected\_prototypes: Prototypes selected using DBSCAN

    """

    # Convert Pandas DataFrame to NumPy array if needed

    X\_train = X\_train.values if isinstance(X\_train, pd.DataFrame) else X\_train

    y\_train = y\_train.values if isinstance(y\_train, pd.Series) else y\_train

    # Apply DBSCAN for clustering

    dbscan = DBSCAN(eps=eps, min\_samples=min\_samples)

    labels = dbscan.fit\_predict(X\_train)

    # Identify core points (excluding outliers)

    core\_points\_indices = np.where(labels != -1)[0]

    if len(core\_points\_indices) == 0:

        raise ValueError("No core points found. Adjust parameters (eps, min\_samples) or provide more data.")

    # Randomly select M prototypes from core points using random\_state

    random\_state = check\_random\_state(random\_state)

    selected\_indices = random\_state.choice(core\_points\_indices, size=min(M, len(core\_points\_indices)), replace=False)

    # Extract the corresponding samples using NumPy array indexing

    selected\_prototypes = X\_train[selected\_indices, :]

    selected\_labels = y\_train[selected\_indices]

    return selected\_prototypes, selected\_labels

def active\_learning\_prototype\_selection(X\_train, y\_train, M, max\_iter=1000, random\_state=None):

    """

    Use Active Learning for prototype selection.

    Parameters:

    - X\_train: Features of the training set (NumPy array or Pandas DataFrame)

    - y\_train: Labels of the training set (NumPy array or Pandas DataFrame)

    - M: Number of prototypes to select

    - max\_iter: Maximum number of iterations for Logistic Regression

    - model\_filename: File name for saving/loading the model

    - random\_state: Seed for reproducibility

    Returns:

    - selected\_prototypes: Prototypes selected using Active Learning

    """

    # Convert Pandas DataFrame to NumPy array if needed

    X\_train = X\_train.values if isinstance(X\_train, pd.DataFrame) else X\_train

    y\_train = y\_train.values if isinstance(y\_train, pd.Series) else y\_train

    # Train a new classifier (don't save the classifier because we need

    # to record the training time for the report)

    classifier = LogisticRegression(max\_iter=max\_iter, random\_state=check\_random\_state(random\_state))

    classifier.fit(X\_train, y\_train)

    # Get predicted probabilities for each class

    predicted\_probs = classifier.predict\_proba(X\_train)

    # Calculate uncertainty as the maximum probability across classes

    uncertainty = 1 - np.max(predicted\_probs, axis=1)

    # Select M prototypes with the highest uncertainty

    selected\_indices = np.argsort(uncertainty)[-M:]

    # Extract the corresponding samples

    selected\_prototypes = X\_train[selected\_indices]

    selected\_labels = y\_train[selected\_indices]

    return selected\_prototypes, selected\_labels

# Code to run experiments

# Define the range of M values to experiment with

M\_values = [100, 500, 1000, 5000, 10000]

# Define the prototype selection methods

prototype\_methods = [

    # ("Random Sampling", random\_sampling),

    # ("DBSCAN", dbscan\_prototype\_selection),

    ("Active Learning", active\_learning\_prototype\_selection)

]

# Create a 1-KNN classifier

classifier = KNeighborsClassifier(n\_neighbors=1)

# Initialize the results dictionary with empty lists for each metric

results = {

    "Method": [],

    "M": [],

    "Mean Accuracy": [],

    "Std Accuracy": [],

    "Mean Training Time (s)": [],

    "Std Training Time (s)": [],

    "Mean Test Time (s)": [],

    "Std Test Time (s)": []

}

# Number of experiments to run for each method and M value

num\_experiments = 10

# Create a dictionary to store results for each experiment

experiment\_results = []

# Run experiments

for method\_name, method\_func in prototype\_methods:

    print(f"\nRunning experiments for {method\_name}...\n")

    for M in M\_values:

        print(f"Number of Prototypes (M): {M}")

        # Lists to store results for each experiment

        accuracies = []

        training\_times = []

        test\_times = []

        for \_ in range(num\_experiments):

            # Timer for training time

            start\_time = time.time()

            # Prototype selection

            prototypes, labels = method\_func(X\_train, y\_train, M)

            # Train KNN classifier on the selected prototypes

            classifier.fit(prototypes, labels)

            # Record training time

            training\_times.append(time.time() - start\_time)

            # Timer for test time

            start\_time = time.time()

            # Evaluate on the test set

            y\_pred = classifier.predict(X\_test)

            # Record test time

            test\_times.append(time.time() - start\_time)

            # Compute accuracy

            accuracies.append(accuracy\_score(y\_test, y\_pred))

        # Compute statistics across experiments

        mean\_accuracy = np.mean(accuracies)

        std\_accuracy = np.std(accuracies)

        mean\_training\_time = np.mean(training\_times)

        std\_training\_time = np.std(training\_times)

        mean\_test\_time = np.mean(test\_times)

        std\_test\_time = np.std(test\_times)

        # Record results

        results["Method"].append(method\_name)

        results["M"].append(M)

        results["Mean Accuracy"].append(mean\_accuracy)

        results["Std Accuracy"].append(std\_accuracy)

        results["Mean Training Time (s)"].append(mean\_training\_time)

        results["Std Training Time (s)"].append(std\_training\_time)

        results["Mean Test Time (s)"].append(mean\_test\_time)

        results["Std Test Time (s)"].append(std\_test\_time)

        print(f"Mean Accuracy: {mean\_accuracy:.4f} (±{std\_accuracy:.4f})")

        print(f"Mean Training Time: {mean\_training\_time:.4f} seconds (±{std\_training\_time:.4f})")

        print(f"Mean Test Time: {mean\_test\_time:.4f} seconds (±{std\_test\_time:.4f})\n")

# Convert results to a DataFrame for easier analysis and visualization

results\_df = pd.DataFrame(results)

# Save results to a CSV file

results\_df.to\_csv("prototype\_selection\_results.csv", index=False)

# Display the results

print(results\_df)

print("Results saved to prototype\_selection\_results.csv")

# File: visualization.py

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

# Load the results from the CSV file

results\_df = pd.read\_csv("results/prototype\_selection\_results.csv")

# Visualize Accuracy vs. Number of Prototypes (M)

plt.figure(figsize=(10, 6))

for method in results\_df['Method'].unique():

    method\_data = results\_df[results\_df['Method'] == method]

    plt.plot(method\_data["M"], method\_data["Mean Accuracy"], marker="o", label=method)

    plt.fill\_between(method\_data["M"],

                     method\_data["Mean Accuracy"] - method\_data["Std Accuracy"],

                     method\_data["Mean Accuracy"] + method\_data["Std Accuracy"],

                     alpha=0.2)

plt.title("Accuracy vs. Number of Prototypes")

plt.xlabel("Number of Prototypes (M)")

plt.ylabel("Mean Accuracy")

plt.grid(True)

plt.legend()

plt.savefig("accuracy\_vs\_prototypes.png")  # Save the figure

plt.show()

# Visualize Training Time vs. Number of Prototypes (M)

plt.figure(figsize=(10, 6))

for method in results\_df['Method'].unique():

    method\_data = results\_df[results\_df['Method'] == method]

    plt.plot(method\_data["M"], method\_data["Mean Training Time (s)"], marker="o", label=method)

    plt.fill\_between(method\_data["M"],

                     method\_data["Mean Training Time (s)"] - method\_data["Std Training Time (s)"],

                     method\_data["Mean Training Time (s)"] + method\_data["Std Training Time (s)"],

                     alpha=0.2)

plt.title("Training Time vs. Number of Prototypes")

plt.xlabel("Number of Prototypes (M)")

plt.ylabel("Mean Training Time (s)")

plt.grid(True)

plt.legend()

plt.savefig("training\_time\_vs\_prototypes.png")  # Save the figure

plt.show()

# Visualize Test Time vs. Number of Prototypes (M)

plt.figure(figsize=(10, 6))

for method in results\_df['Method'].unique():

    method\_data = results\_df[results\_df['Method'] == method]

    plt.plot(method\_data["M"], method\_data["Mean Test Time (s)"], marker="o", label=method)

    plt.fill\_between(method\_data["M"],

                     method\_data["Mean Test Time (s)"] - method\_data["Std Test Time (s)"],

                     method\_data["Mean Test Time (s)"] + method\_data["Std Test Time (s)"],

                     alpha=0.2)

plt.title("Test Time vs. Number of Prototypes")

plt.xlabel("Number of Prototypes (M)")

plt.ylabel("Mean Test Time (s)")

plt.grid(True)

plt.legend()

plt.savefig("test\_time\_vs\_prototypes.png")  # Save the figure

plt.show()

# Table of Mean Accuracy and Mean Training Time for each Method and M

summary\_table = results\_df.pivot\_table(index="M", columns="Method", values=["Mean Accuracy", "Mean Training Time (s)", "Mean Test Time (s)"])

summary\_table.to\_csv("summary\_table.csv")

print("Summary Table:")

print(summary\_table)