```python
# 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
```

```python
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
```

```python
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")
```

```python
# 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()
```

```python
# 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)
```