Enhancing One Nearest Neighbor Classification Efficiency through Prototype Selection: A Study on the MNIST Dataset

Allen Zou

February 1, 2024

Abstract

Nearest neighbor classification often poses computational challenges, especially with large datasets. This paper explores the application of established prototype selection methodologies to enhance the efficiency of nearest neighbor classification. Leveraging insights from previous work, including DBSCAN clustering [1] and Active Learning [4], the approach focuses on selecting a representative subset of prototypes from the training set. The parameter M represents the number of prototypes selected from the training set. A concise pseudocode for the prototype selection algorithm is provided, and extensive experiments are conducted on the MNIST dataset using varying values of M, including $M = 10000, 5000, 1000, 500, 100$ [2]. Evaluation results are presented to assess the performance of the proposed method.

1 Introduction

Nearest neighbor classification is a fundamental technique in machine learning, but its computational complexity grows with the size of the training set. Prototype selection offers a solution by identifying a representative subset of the training data for classification. This paper explores a few prototype selection strategies and evaluates their impact on classification accuracy compared to the baseline of uniform random sampling using the MNIST dataset [2]. It presents pseudocode for the proposed algorithm, describes the experimental setup, and analyzes the obtained results.

2 Prototype Selection Strategies

Prototype selection methods aim to reduce the computational complexity of nearest neighbor classification by identifying a subset of the training data that captures its essential characteristics. In this section, we describe three established prototype selection strategies: Uniform Random Sampling, DBSCAN-based Selection, and Active Learning. The parameter M represents the number of prototypes selected from the training set.

2.1 Random Sampling

Uniform Random sampling is a straightforward prototype selection strategy that involves selecting a subset of prototypes randomly from the training set. This method serves as our baseline approach for comparison with more sophisticated strategies. By randomly choosing prototypes, we aim to establish a performance benchmark against which other strategies will be evaluated.

Algorithm 1 Random Sampling

$N \leftarrow$ Length of X_{train}
$indices \leftarrow$ Randomly choose M from $[0, N)$
$prototypes \leftarrow$ Sample at $indices$ from X_{train}
$labels \leftarrow$ Select labels at $indices$ from y_{train}
return $prototypes, labels$

2.2 DBSCAN-based Selection

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm [1] implemented in the scikit-learn library [3]. In prototype selection, DBSCAN identifies dense regions in the feature space and selects prototypes from core points within these regions, excluding outliers. This strategy focuses on capturing meaningful structures in the data by emphasizing densely populated regions.

For our implementation, eps and $min\_samples$ is to 20 and 10, respectively. The eps parameter defines the maximum distance between two samples for them to be considered as

Algorithm 2 DBSCAN Prototype Selection

$core\_points \leftarrow$ Apply DBSCAN(X_{train})
$indices \leftarrow$ Randomly Select M $core\_points$
$prototypes \leftarrow$ Sample at $indices$ from X_{train}
$labels \leftarrow$ Select labels at $indices$ from y_{train}
return $prototypes, labels$

in the same neighborhood, while $min\_samples$ specifies the minimum number of samples in a neighborhood for a point to be considered as a core point. These parameter values were chosen empirically to ensure that the algorithm captures clusters effectively while excluding noisy points.

2.3 Active Learning

Active Learning is a dynamic prototype selection strategy that leverages uncertainty in the classifier's predictions. In our implementation, we utilize logistic regression as the base classifier, which is available in the scikit-learn library [3]. The algorithm selects prototypes based on the uncertainty associated with each sample, emphasizing instances where the classifier exhibits higher uncertainty.

Algorithm 3 Active Learning Prototype Selection

Train LR classifier C on $X_{\text{train}}, y_{\text{train}}$
$prob \leftarrow$ predicted probabilities from C
$uncertainty \leftarrow 1 - \max(P, axis = 1)$
$indices \leftarrow \text{argsort}(uncertainty)[-M:]$
$prototypes \leftarrow$ Sample at $indices$ from X_{train}
$labels \leftarrow$ Select labels at $indices$ from y_{train}
return $prototypes, labels$

3 Experimental Setup

We conducted experiments to evaluate the performance of prototype selection strategies in one nearest neighbor classification using the MNIST dataset. These experiments aim to compare the classification accuracy and computational efficiency of three prototype selection methods: Random Sampling, DBSCAN-based Selection, and Active Learning.

3.1 Dataset

We used the MNIST dataset, a benchmark dataset in machine learning, consisting of 28x28 pixel grayscale images of handwritten digits ranging from 0 to 9. The dataset comprises 70,000 samples, divided into 60,000 training images and 10,000 test images. Each image is associated with a corresponding label indicating the digit it represents.

3.2 Evaluation Metrics

We evaluated the prototype selection methods based on the following metrics:

- **Accuracy**: classification accuracy over multiple experiments.
- **Training Time**: time taken to select prototypes (including training time if relevant).
- **Test Time**: time taken to classify test samples.

3.3 Experimental Procedure

We conducted experiments for each prototype selection method using different values of M, representing the number of prototypes selected from the training set. For each combination of method and M, we performed 10 experiments to account for randomness and obtain estimates of the evaluation metrics.

It's important to note that while increasing the number of experiments generally provides more robust results, the computational demands grow proportionally. While 10 experiments provide insights into the performance of the prototype selection methods, conducting additional experiments could further enhance the analysis. Extensive computational resources required for additional experiments should be considered in the future.

4 Experimental Results

We present the experimental results of evaluating prototype selection methods using the MNIST dataset. Table 1 provides a summary of the mean accuracy and mean time metrics across various values of M. Figures 1, 2, and 3 illustrate the relationship between these metrics and the number of prototypes selected. In these figures, each dot represents the mean value of the corresponding metric, and the shadings around the dots indicate one standard deviation from the mean.

Table 1: Experimental results for prototype selection methods

	Mean Accuracy			Mean Test Time (s)			Mean Training Time (s)		
Prototypes	**Random**	**DBSCAN**	**AL**	**Random**	**DBSCAN**	**AL**	**Random**	**DBSCAN**	**AL**
100	0.639	0.647	0.298	0.171	0.231	0.189	0.001	43.599	351.509
500	0.792	0.796	0.433	0.253	0.270	0.243	0.002	43.491	329.956
1000	0.833	0.836	0.512	0.325	0.307	0.320	0.003	36.377	315.747
5000	0.892	0.893	0.723	1.108	0.985	0.961	0.011	37.132	304.187
10000	0.913	0.910	0.844	2.057	1.790	1.741	0.030	36.170	295.461

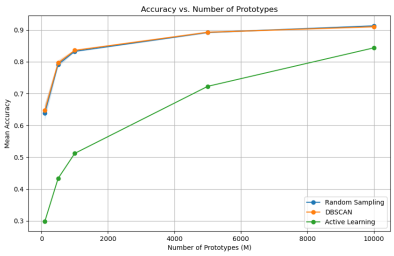

Figure 1: Mean accuracy vs. number of prototypes

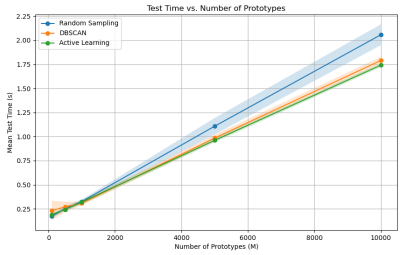

Figure 2: Mean test time vs. number of prototypes

5 Discussion

The experimental results, as outlined in Table 1, offer insights into the behavior of prototype selection methods and their impact on nearest neighbor classification using the MNIST dataset.

5.1 Accuracy Analysis

The mean accuracy results reveal unexpected patterns among the various prototype selection methods. Particularly, the accuracy of Active Learning appears to be lower compared to Random Sampling and DBSCAN. A possible explanation for this discrepancy is the intricate nature of handwritten digits in MNIST. However, it's important to note that these results could be influenced by specific parameters or aspects of the Logistic Regression implementation, warranting further investigation.

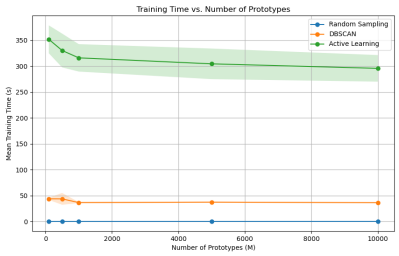

Figure 3: Mean training time vs. number of prototypes

5.2 Test Time Comparison

Comparing the mean test times, DBSCAN and Active Learning exhibit no significant difference between their test times. On the contrary, Random Sampling, despite being a simpler strategy, shows slightly higher mean test times. This result could be attributed to the random nature of the sampling process.

5.3 Training Time Comparison

The mean training times for the prototype selection methods follow the expected trend of increasing from Random Sampling to DBSCAN and then to Active Learning. However, the training time for Random Sampling is significantly higher than that of the other two prototype selection methods. Particularly noteworthy is the disparity between Active Learning and the other methods. Given that Active Learning also underperformed noticeably compared to DBSCAN and Random Sampling in terms of test time, this may necessitate a closer examination of the codebase.

5.4 Number of Prototypes

As the number of prototypes M increases, we observe an improvement in test accuracy, with the

curve approaching an asymptote for larger M values. Meanwhile, the test time exhibits a linear increase, directly proportional to the size of the selected prototype set. Notably, the training time remains relatively stable across different M values, indicating that the training process for the nearest neighbor classifier does not significantly vary with the number of prototypes.

6 Conclusion

Our exploration of prototype selection methodologies reveals insights into enhancing one nearest neighbor classification efficiency. Through experimentation on the MNIST dataset, we compared the performance of prototype selection methods against uniform random sampling.

The accuracy analysis revealed unexpected trends among prototype selection methods, with Active Learning exhibiting lower accuracy compared to Random Sampling and DBSCAN. Additionally, the comparison of test times showed no significant difference between DBSCAN and Active Learning, while Random Sampling exhibited slightly higher mean test times. Furthermore, the underperformance of Logistic Regression in the Active Learning prototype sampling method prompts further investigation. Lastly, as the number of prototypes M increased, test accuracy improved, with test time increasing linearly, while training time remained relatively stable.

Future work could focus on refining parameter settings, exploring alternate selection strategies, and evaluating the approach across diverse datasets to establish its broader applicability.

References

[1] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, 1996.

[2] Hojjatk. Mnist dataset. https://www.kaggle.com/datasets/hojjatk/mnist-dataset, n.d.

[3] Scikit-learn Contributors. Scikit-learn documentation, n.d.

[4] B. Settles. Active learning literature survey. Technical Report 1648, University of Wisconsin-Madison, Department of Computer Sciences, 2010.

```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
```

```
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()
```

```
# 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)
```