Project 1 Write-up

Haotian Xu
hax030@ucsd.edu

Abstract

In this project, we investigate the efficacy of prototype selection methods in enhancing the speed of nearest neighbor classification. Specifically, we implement and compare three distinct approaches: Random Selection, K-means clustering, and Condensed Nearest Neighbor (CNN). Our evaluation focuses on their performance relative to classification accuracy and computational efficiency when applied to the MNIST dataset. This comparative analysis aims to determine the most effective technique for reducing computational demands while maintaining or improving the accuracy of the nearest neighbor classifier.

1 Introduction

The Nearest Neighbor (NN) method is celebrated for its effectiveness in certain machine learning tasks. However, its computational intensity emerges as a primary limitation, particularly with large datasets. One strategy to expedite NN is through prototype selection, which involves undersampling the training set. In this project, we examine and implement three prototype selection methods: Random Selection, K-means clustering, and Condensed Nearest Neighbor (CNN) (Hart, 1968). Random Selection, serving as a baseline, reduces the dataset size by randomly selecting a subset of samples. K-means segments data into k clusters and employs the centroids as representative prototypes. CNN aims to preserve the most informative instances while eliminating redundancies.

For our empirical study, we utilize the MNIST dataset (Deng, 2012), a renowned benchmark in machine learning for handwritten digit recognition. We employ a 1-NN classifier on the undersampled training set to evaluate the performance of our prototype selection methods. Our assessment spans three tiers of sample sizes: 10,000, 5,000, and 1,000. For each tier, we compare the classification accuracy and running time of the different methods, aiming to discern the most efficient approach in balancing computational speed and accuracy.

2 Algorithm

The MNIST dataset is partitioned into a training set comprising 60,000 samples and a test set encompassing 10,000 samples. We employ each prototype selection method to undersample the training set within sizes of 10,000, 5,000, or 1,000 samples. In subsequent sections, we explain each method in detail and accompany the explanations with corresponding pseudocode.

2.1 Random Selection

This method is straightforward yet potent. We randomly select a subset from the training set and utilize this subset to train the 1-NN classifier. To mitigate the influence of outliers induced by randomness, the process is repeated multiple times using different random seeds. Subsequently, we compute the confidence intervals to ensure the robustness of our results. We use the following formula to compute the 95% confidence interval:

$$E = 1.96 \times \left(\frac{\sigma}{\sqrt{n}} \right) \tag{1}$$

$$CI = \mu \pm E \tag{2}$$

Where E is the margin of error for 95% confidence interval, σ is the standard deviation of the accuracy, n is the number of measures we have done, CI is the 95% confidence interval, and μ is the mean accuracy.

Below is the pseudocode corresponding to the Random Selection method:

Method Name: Random Selection
Input:

- X_{train}: Training features

- y_{train}: Training labels
- M: Desired number of samples in the subset
- *seed*: Seed for random number generator

Output:

- A subset of X_{train} of size M
- Corresponding labels from y_{train}

Method:

1. Initialize $S = \emptyset$ (the selected subset of samples)
2. Initialize $L = \emptyset$ (the labels corresponding to the selected subset)
3. Set random seed to ensure reproducibility: *seed*(*seed*)
4. Generate a list of M unique random indices I from the range $[0, length(X_{train}) - 1]$
5. For each index $i \in I$:
   - Add $X_{train}[i]$ to S
   - Add $y_{train}[i]$ to L
6. Return S and L

2.2 K-means

K-means clustering method can cluster some centroids from the original dataset which are representative. However, directly applying this method to a training set with 60,000 samples is computationally intensive. We make some adjustments to speed up this method.

We divide the problem into ten subtasks corresponding to ten labels, and each one will apply K-means method to a subset of a certain label with a smaller size. We merge each result to form the undersample training set. Since K-means has randomness, we also repeat the process with different random seeds.

Below is the pseudocode:
Method Name: K-means Prototype Selection
Input:

- X_{train}: Training features
- y_{train}: Training labels
- M: Total desired number of prototypes

Output:

- $P_{centers}$: Prototype centers
- P_{labels}: Labels corresponding to the prototype centers

Method:

1. Initialize $P_{centers} = \emptyset$ (array to hold prototype centers)
2. Initialize $P_{labels} = \emptyset$ (list to hold labels of prototype centers)
3. Calculate $M_{per\_class} = \frac{M}{10}$ (number of prototypes per class)
4. For each label in range(10):
   (a) Extract indices of class samples: $I_{class} = \{i | y_{train}[i] = label\}$
   (b) Extract class samples: $S_{class} = X_{train}[I_{class}]$
   (c) Initialize KMeans with M_{per_class} centers
   (d) Fit KMeans on S_{class}
   (e) Retrieve cluster centers: $C_{class} = KMeans.cluster\_centers\_$
   (f) Append C_{class} to $P_{centers}$
   (g) Extend P_{labels} with current label
5. Return $P_{centers}$, P_{labels}

2.3 CNN

The idea of CNN is to keep the representatives and remove redundancy. We first have an initialized S. For each element in the training set T, if the classifier trained on S fails to predict the element, add that element to S. We keep progressing with this process until we reach the end or we meet the sample size limit, and return S as the undersampled subset. Since this method cannot exactly control the sample size, we set a limit to stop the method. We use 1-NN as the classifier in the method.

Below is the pseudocode:
Method Name: CNN
Input:

- X_{train}: Training features
- y_{train}: Training labels
- M_{limit}: Maximum number of prototypes

Output:

Method	M	Accuracy	95% Confidence Interval	Actual Sample Size	Running Time (seconds)
Random Selection	10000	0.9492	±0.0016	10000	0.0541
	5000	0.9343	±0.0011	5000	0.0296
	1000	0.8800	±0.0034	1000	0.0070
K-means	10000	0.9721	±0.0017	10000	782.7324
	5000	0.9696	±0.0021	5000	449.0552
	1000	0.9589	±0.0021	1000	168.5232
CNN	10000	0.9346	-	5066	365.7314
	5000	0.9345	-	5000	342.4237
	1000	0.8851	-	1000	13.7789

Table 1: Performance metrics for Random Selection, K-means, and CNN methods.

- $P_{prototypes}$: Prototype features
- P_{labels}: Labels corresponding to the prototypes

Method:

1. Initialize $P_{prototypes}$ with the first sample of X_{train}
2. Initialize P_{labels} with the label of the first sample of y_{train}
3. Initialize a 1-NN classifier
4. For each sample x and label y in X_{train} and y_{train}:
   (a) If the length of $P_{prototypes}$ is greater than or equal to M_{limit}, break the loop.
   (b) Train the 1-NN classifier with $P_{prototypes}$ and P_{labels}
   (c) Predict the label of x using the 1-NN classifier
   (d) If the predicted label does not match y, add x to $P_{prototypes}$ and y to P_{labels}
5. Return $P_{prototypes}$ and P_{labels}

3 Experimental Evaluation

Our empirical findings, encompassing sample size, test accuracy, and average running time, are detailed in Table 1. Notably, since the exact sample size of CNN cannot be predetermined, the actual number of samples utilized by CNN is also specified. For methods incorporating elements of randomness, their 95% confidence intervals are presented to quantify the reliability of the accuracy measurements.

The K-means method exhibits superior accuracy compared to the other techniques, consistently outperforming Random Selection across all sample sizes. This notable distinction in performance underscores its effectiveness. However, it's also the most time-consuming approach among those tested. Future improvements should focus on enhancing computational efficiency without compromising the high level of accuracy achieved by this method.

In contrast, while our CNN method demonstrates quicker execution times compared to K-means, its accuracy does not surpass that of Random Selection. This observation could be related to the MNIST dataset, which is well-formatted, and a random subset is still as representative as a carefully selected subset from CNN. Moreover, CNN lacks parameters to precisely control the resulting sample size, leading to a selection of only 5,066 samples under a limit of 10,000 in our experiments. Future research could explore enhancements or alternatives to CNN, such as RNN (Gates, 1972) and FCNN (Angiulli, 2007), aiming for increased efficiency and accuracy.

4 Conclusion

We have investigated and implemented Random Selection, K-means, and CNN methods for prototype selection. Our experiments find that K-means method has superior performance in predicting accuracy compared to the Random Selection method, but requires a long computational time. CNN is less computationally demanding than K-means but does not achieve significantly higher accuracy than the Random Selection does.

References

F. Angiulli. 2007. Fast nearest neighbor condensation for large data sets classification. *IEEE Transactions on Knowledge and Data Engineering*, 19(11):1450–1464.

L. Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142.

G.W. Gates. 1972. The reduced nearest neighbor rule. *IEEE Transactions on Information Theory*, 18(3):431–433.

P. E. Hart. 1968. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14(3):515–516.

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, accuracy_score
import numpy as np
import time


mnist = fetch_openml('mnist_784')
index_number= np.random.permutation(70000)
x1,y1=mnist.data.loc[index_number],mnist.target.loc[index_number]
x1.reset_index(drop=True,inplace=True)
y1.reset_index(drop=True,inplace=True)
x1 = x1/255.0
X_train , X_test = x1[:60000].to_numpy(), x1[60000:].to_numpy()
y_train , y_test = y1[:60000].to_numpy(), y1[60000:].to_numpy()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968: FutureWarning: The default value of `parser` will change f
  warn(
```

```
def random_prototype_selection(X_train, y_train, M, seed):
    np.random.seed(seed)
    random_indices = np.random.choice(X_train.shape[0], size=M, replace=False)
    return X_train[random_indices, :], y_train[random_indices]

M_values = [10000, 5000, 1000]
num_experiments = 5

results = {}
times = {}
for M in M_values:
    temp_time = []
    accuracies = []
    for seed in range(num_experiments):
        start_time = time.time()
        X_train_prototypes, y_train_prototypes = random_prototype_selection(X_train, y_train, M, seed)
        end_time = time.time()
        temp_time.append(end_time - start_time)
        knn_classifier = KNeighborsClassifier(n_neighbors=1)
        knn_classifier.fit(X_train_prototypes, y_train_prototypes)
        y_pred = knn_classifier.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        accuracies.append(accuracy)

    mean_accuracy = np.mean(accuracies)
    std_deviation = np.std(accuracies)
    confidence_interval = 1.96 * (std_deviation / np.sqrt(num_experiments))
    results[M] = (mean_accuracy, confidence_interval)
    times[M] = np.mean(temp_time)
for M, (mean_accuracy, confidence_interval) in results.items():
    print(f"M={M}: Mean Accuracy: {mean_accuracy:.4f}, 95% Confidence Interval: ±{confidence_interval:.4f}, Running time: {times[M]:.4f}
```

```
M=10000: Mean Accuracy: 0.9492, 95% Confidence Interval: ±0.0016, Running time: 0.0541
M=5000: Mean Accuracy: 0.9343, 95% Confidence Interval: ±0.0011, Running time: 0.0296
M=1000: Mean Accuracy: 0.8800, 95% Confidence Interval: ±0.0034, Running time: 0.0070
```

```
from sklearn.cluster import KMeans

M_values = [10000, 5000, 1000]
num_experiments = 2
results = {}
times = {}

for M in M_values:
    accuracies = []
    execution_times = []
    M_per_class = M // 10
    for experiment in range(num_experiments):
        prototype_centers = np.empty((0, X_train.shape[1]))
        prototype_labels = []
        num_per_class = M // 10  # Number of prototypes per class
        start_time = time.time()
        for label in range(10):
            class_indices = np.where(y_train == str(label))[0]
            class_samples = X_train[class_indices]
            kmeans = KMeans(n_clusters=M_per_class, init='k-means++', random_state=experiment)
            kmeans.fit(class_samples)
            class_prototypes = kmeans.cluster_centers_
            prototype_centers = np.vstack([prototype_centers, class_prototypes])
            prototype_labels.extend([label] * M_per_class)
        end_time = time.time()
        knn_classifier = KNeighborsClassifier(n_neighbors=1)
        knn_classifier.fit(prototype_centers, prototype_labels)
        y_pred = knn_classifier.predict(X_test)
        y_test_int = y_test.astype(int)
        accuracy = accuracy_score(y_test_int, y_pred)
        accuracies.append(accuracy)
        execution_times.append(end_time - start_time)

    mean_accuracy = np.mean(accuracies)
    std_deviation = np.std(accuracies)
    mean_time = np.mean(execution_times)

    confidence_interval = 1.96 * (std_deviation / np.sqrt(num_experiments))

    # Store the results
    results[M] = (mean_accuracy, confidence_interval, mean_time)

# Print the results with error bars and running time
for M, (mean_accuracy, confidence_interval, mean_time) in results.items():
    print(f"M={M}: Mean Accuracy: {mean_accuracy:.4f}, 95% Confidence Interval: ±{confidence_interval:.4f}, Running Time: {mean_time:.4
```

```
    warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
M=10000: Mean Accuracy: 0.9721, 95% Confidence Interval: ±0.0017, Running Time: 782.7324 seconds
M=5000: Mean Accuracy: 0.9696, 95% Confidence Interval: ±0.0021, Running Time: 449.0552 seconds
M=1000: Mean Accuracy: 0.9589, 95% Confidence Interval: ±0.0021, Running Time: 168.5232 seconds
```

```python
def condensed_nearest_neighbor(X_train, y_train, M_limit):
    prototypes = np.array([X_train[0]])
    prototype_labels = np.array([y_train[0]])
    knn_classifier = KNeighborsClassifier(n_neighbors=1)

    for x, y in zip(X_train, y_train):
        if len(prototypes) >= M_limit:
            break
        knn_classifier.fit(prototypes, prototype_labels)
        y_pred = knn_classifier.predict([x])[0]
        if y_pred != y:
            prototypes = np.vstack([prototypes, [x]])
            prototype_labels = np.append(prototype_labels, y)

    return prototypes, prototype_labels
results = []
M_values = [10000, 5000, 1000]
execution_times = []
sample_size = []
for M in M_values:
    start_time = time.time()
    X_train_prototypes, y_train_prototypes = condensed_nearest_neighbor(X_train, y_train, M)
    end_time = time.time()
    knn_classifier = KNeighborsClassifier(n_neighbors=1)
    knn_classifier.fit(X_train_prototypes, y_train_prototypes)
    y_pred = knn_classifier.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    results.append(accuracy)
    execution_times.append(end_time - start_time)
    sample_size.append(len(X_train_prototypes))

for i,m in enumerate(M_values):
  print(f"M={m}: Accuracy: {results[i]:.4f}, Actual Sample Size: {sample_size[i]}, Running Time: {execution_times[i]:.4f} seconds")
```

```
M=10000: Accuracy: 0.9346, Actual Sample Size: 5066 Running Time: 365.7314 seconds
M=5000: Accuracy: 0.9345, Actual Sample Size: 5000 Running Time: 342.4237 seconds
M=1000: Accuracy: 0.8851, Actual Sample Size: 1000 Running Time: 13.7789 seconds
```