Project 1 — Prototype Selection for Nearest Neighbor

Zhiqiao Gong
z6gong@ucsd.edu

Abstract

In this project, I compared Random, K-Means, and Condensed Nearest Neighbor (CNN) prototype selection methods with different prototype sizes from 100 to 10000. I evaluated their performance based on test error and computational runtime, statistically analyzed the results, and calculated average test errors with 95% confidence intervals. The analysis provides insights into the trade-offs between accuracy, computational efficiency, and prototype size in nearest-neighbor classification.

1 Implementation Details

1.1 Data Loading

MNIST is a dataset of handwritten digits. I imported it using a function to process IDX files. The read_index function opened each file in binary, read the header info to determine data type and dimensions, and loaded data into NumPy arrays for efficient manipulation.

The image data was then reshaped from its original flat format of 28x28 pixels into a 784-dimensional vector for machine learning algorithms. Labels were loaded from their respective files and converted to an integer data type for consistency in classification tasks.

No further normalization or preprocessing was applied to preserve raw pixel values as features for the prototype selection algorithms and ensure compatibility with the K-Nearest Neighbors algorithm.

1.2 Prototype Selection Methods

Three distinct prototype selection strategies were implemented and evaluated:

1.2.1 Random Selection

This approach was used as a baseline. It involved selecting a random subset of the MNIST training dataset. The selection process was simple and involved using a function that randomly chose M indices from the training data, ensuring that there were no duplicates. The selected data points along with their corresponding labels formed the prototype set.

1.2.2 K-Means Clustering

K-Means clustering(Hartigan and Wong, 1979) is an iterative algorithm that partitions the dataset into K distinct, non-overlapping subsets (clusters). It minimizes the within-cluster sum of squares, i.e., the variance within each cluster. The process aims to find cluster centers (centroids) that are representative of certain regions of the data.

For each centroid, assign a label based on the majority label of points in the corresponding cluster. This ensures that each prototype (centroid) has a class label for classification.

To calculate the new centroids:

$$C_i = \frac{1}{|S_i|} \sum_{x \in S_i} x$$

where C_i represents the centroid of cluster i, S_i is the set of points in cluster i, and x is a data point in S_i.

Algorithm 1 K-Means Clustering for Prototype Selection

Require: $X \neq \emptyset$ (dataset), $K > 0$ (number of clusters)
Ensure: Cluster centroids C representing prototypes
1: Initialize C with K random points from X
2: **repeat**
3: Assign each point $x \in X$ to the nearest centroid in C
4: Update each centroid $c \in C$ to be the mean of its assigned points
5: **until** centroids C do not change
6: **return** C

One of the main challenges with the k-means algorithm is that it can potentially converge to a local minimum, which may not be the most optimal solution. To mitigate this issue, the algorithm is often run multiple times with different initializations. Choosing the appropriate number of clusters (k) is another critical factor that can significantly impact the results. Typically, the value of k is determined based on domain knowledge or techniques such as the elbow method.

1.2.3 Condensed Nearest Neighbor (CNN)

The Condensed Nearest Neighbor (CNN) algorithm(Hart, 1968) is an instance-based learning technique used to reduce the size of the training dataset for the k-nearest neighbor classifier. The key idea behind CNN is to create a condensed set of the training data such that the 1-NN classifier's performance on the training set is as close as possible to its performance using the full training set.

The CNN algorithm works iteratively. It starts by initializing the condensed set with one instance from the training set. Then, for each instance in the training data, it is classified using the current condensed set. If an instance is classified incorrectly, it is added to the condensed set. This process continues until all instances in the training set are classified correctly by the condensed set, or no more instances can be added.

In the 1-NN classification within CNN:

$$label(x) = argmin_{p \in P} distance(x, p)$$

where $label(x)$ is the predicted label for a data point x, P is the set of current prototypes, and $distance(x, p)$ is the distance metric (e.g., Euclidean distance).

CNN has a key advantage in that it helps to reduce the size of the training data, which in turn can lead to faster classification times. However, it is important to note that the order in which instances are presented to the algorithm can have an impact on the final condensed set. As a result, it may be necessary to go through multiple passes or randomly shuffle the training data to achieve more stable and consistent results.

Algorithm 2 Condensed Nearest Neighbor (CNN) for Prototype Selection

Require: $X \neq \emptyset$ (dataset with features), $Y \neq \emptyset$ (dataset with labels), $M > 0$ (desired number of prototypes)
Ensure: Prototype subset P and labels L

```
P ← ∅
L ← ∅
Add first element of X to P and its label to L
for i ← 2 to |X| do
    Classify x_i using 1-NN with P
    if x_i is misclassified then
        Add x_i to P and its label to L
    end if
    if |P| = M then
        break
    end if
end for
return P, L
```

2 Experimental Results

To evaluate the effectiveness of different prototype selection strategies, I measured their impact on the classification performance of a k-Nearest Neighbors (k-NN) classifier. The primary metric used was the test error, which is calculated as the proportion of misclassified test set samples. The accuracy score was subtracted from one to obtain the test error.

To ensure a thorough evaluation, I conducted multiple experiments for each prototype selection method, especially those involving randomness (such as Random Selection and K-Means), in order to account for any variability in the results. I recorded the test error for each run and calculated the average test error across all runs to reduce the impact of outliers or anomalies.

Statistical reliability was measured using confidence intervals. I calculated 95% confidence intervals for mean test error using a formula.

The 95% confidence interval for the mean test error is given by the formula:

$$CI = \bar{x} \pm z \times \frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the sample mean of the test errors, z is the z-score corresponding to the 95% confidence level, s is the sample standard deviation of the test errors, and n is the number of experiments or trials.

The error bar plots visually represent the confidence intervals, providing an intuitive understanding of the methods' performance consistency. This comparative analysis reveals the relationship between computational efficiency and classification accuracy for different prototype sizes.

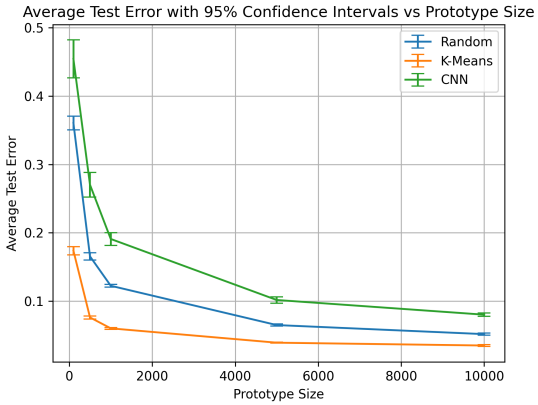

Figure 1: Average Test Error with 95% Confidence Intervals vs Prototype Size

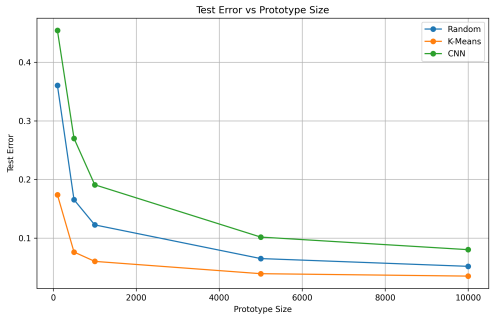

Figure 2: Test Error vs Prototype Size

The plot of Average Test Error with 95% Confidence Intervals vs Prototype Size (Figure 1) shows that while all methods show a decrease in test error with an increase in prototype size, the CNN method consistently shows the highest test error. The K-Means method shows significant improvement as the prototype size increases, achieving notably lower error rates for larger sizes. The Random method, while starting off with higher error rates for small prototype sizes, improves and outperforms CNN across all prototype sizes. The error bars for the 95% confidence intervals indicate consistent results across multiple runs, particularly for larger prototype sizes.

Based on the plot of Test Error vs Prototype Size (Figure 2), it can be inferred that CNN has the highest test error, followed by Random and K-Means. The CNN method exhibits the highest error, which is particularly noticeable at smaller prototype sizes. This suggests that it might not be very effective in selecting representative prototypes. As the number of prototypes increases, the difference between the methods becomes less significant, with K-Means

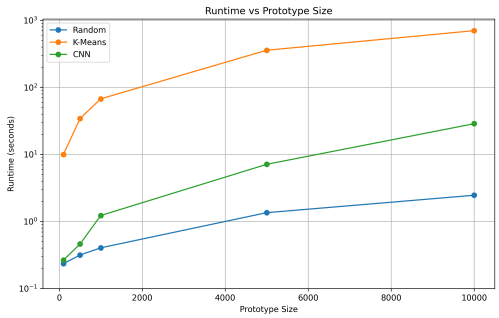

Figure 3: Runtime vs Prototype Size

showing the lowest test error at larger prototype sizes.

Runtime performance was measured for each method to gain insights into computational demands for prototype selection and subsequent classification, which is crucial for applications where prediction speed is a significant concern.

The plot of Runtime against Prototype Size (Figure 3) demonstrates that the Random method is the most computationally efficient method, as it has the lowest runtime across all prototype sizes. On the other hand, the K-Means method has a higher computational cost, particularly at larger prototype sizes. The CNN method's runtime increases at a slower pace, but it remains more time-consuming than Random selection.

3 Evaluation

In this project, I evaluated prototype selection methods for the MNIST dataset to improve the efficiency of the nearest neighbor classifier. My methods were Random selection, K-Means, and Condensed Nearest Neighbor (CNN).

The results indicated that K-Means generally offered a clear improvement over Random selection in terms of test error, particularly as the prototype size increased. CNN, while computationally less demanding, did not perform as well in reducing the test error. The Random selection method, surprisingly, maintained a competitive balance between runtime efficiency and error rate.

To better select prototypes for MNIST, advanced clustering techniques like DBSCAN or spectral clustering can be used. I can also preprocess the data with feature extraction methods such as PCA or t-SNE to reduce the number of prototypes needed and improve the efficacy of the nearest neighbor classifier.

Using ensemble methods that combine prototype selection strategies, active learning technique, and adaptive prototyping strategy can also optimize the prototype set. Active learning can focus training on informative data points, while adaptive prototyping can yield a balanced set of prototypes. Exploring these strategies can improve prototype-based nearest neighbor classification beyond random selection's baseline performance.

The next steps could involve testing prototype selection methods on different datasets, applying more sophisticated clustering techniques such as DBSCAN or hierarchical clustering, and experimenting with advanced machine learning models that may be more sensitive to prototype selection strategies.

References

Peter Hart. 1968. The condensed nearest neighbor rule (corresp.). *IEEE transactions on information theory*, 14(3):515–516.

John A Hartigan and Manchek A Wong. 1979. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108.

```
#!/usr/bin/env python
# coding: utf-8

# In[20]:


import numpy as np
import matplotlib.pyplot as plt
import struct
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier, NearestNeighbors
from sklearn.metrics import accuracy_score
import time
import matplotlib.pyplot as plt
from scipy.stats import norm


# In[2]:


def read_file(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)


# In[107]:


# Load Dataset
train_images_file = 'archive/train-images.idx3-ubyte'
train_labels_file = 'archive/train-labels.idx1-ubyte'
test_images_file = 'archive/t10k-images.idx3-ubyte'
test_labels_file = 'archive/t10k-labels.idx1-ubyte'

train_images = read_file(train_images_file)
train_labels = read_file(train_labels_file)
test_images = read_file(test_images_file)
test_labels = read_file(test_labels_file)


# In[44]:


# Random Prototype Selection
def Random_prototypes(X, y, M):
    indices = np.random.choice(len(X), size=M, replace=False)
    return X[indices], y[indices]


# In[85]:


# K means Prototype Selection
def Kmeans_prototypes(X, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, random_state=None, n_init='auto').fit(X)
    prototypes = kmeans.cluster_centers_
    return prototypes

def Assign_centroids(X, y, centroids):
    kmeans = KMeans(n_clusters=len(centroids), init=centroids, n_init=1, max_iter=1)
    kmeans.fit(X)
    labels = np.zeros(len(centroids))

    for i in range(len(centroids)):
        labels[i] = np.argmax(np.bincount(y[kmeans.labels_ == i]))

    return labels
```

```
# In[91]:


# CNN Prototype Selection
def CNN_prototypes(X, y, M):
    indices = np.random.permutation(len(X))
    X_shuffled = X[indices]
    y_shuffled = y[indices]
    prototypes = [X_shuffled[0]]
    labels = [y_shuffled[0]]
    nn = NearestNeighbors(n_neighbors=1)

    for i in range(1, M):
        nn.fit(prototypes, labels)
        nearest = nn.kneighbors([X_shuffled[i]], 1, return_distance=False)
        nearest_label = labels[nearest[0][0]]
        if y_shuffled[i] != nearest_label:
            prototypes.append(X_shuffled[i])
            labels.append(y_shuffled[i])

    return np.array(prototypes), np.array(labels)


# In[92]:


# Evaluate Test Error
def Evaluate_test_error(prototypes, prototype_labels, X_test, y_test):
    knn = KNeighborsClassifier()
    knn.fit(prototypes, prototype_labels)
    y_pred = knn.predict(X_test)
    test_error = 1 - accuracy_score(y_test, y_pred)
    return test_error


# In[96]:


prototype_sizes = [100, 500, 1000, 5000, 10000]
methods = ['Random', 'K-Means', 'CNN']
results = {}
runtime_results = {}
num_experiments = 5
error_results = {method: {size: [] for size in prototype_sizes} for method in methods}


# In[97]:


for size in prototype_sizes:
    for method in methods:
        for e in range(num_experiments):
            start_time = time.time()
            if method == 'Random':
                prototypes, labels = Random_prototypes(train_images.reshape(-1, 784),
train_labels, size)
            elif method == 'K-Means':
                prototypes = Kmeans_prototypes(train_images.reshape(-1, 784), size)
                labels = Assign_centroids(train_images.reshape(-1, 784), train_labels,
prototypes)
            elif method == 'CNN':
                prototypes, labels = CNN_prototypes(train_images.reshape(-1, 784),
train_labels, size)

            test_error = Evaluate_test_error(prototypes, labels, test_images.reshape(-1,
784), test_labels)
            error_results[method][size].append(test_error)
```

```
            # results[(method, size)] = test_error
            end_time = time.time()
            runtime = end_time - start_time
            runtime_results[(method, size)] = runtime
            print(e,method,size,test_error)

avg_errors = {method: [] for method in methods}
std_errors = {method: [] for method in methods}
for method in methods:
    for size in prototype_sizes:
        m = np.mean(error_results[method][size])
        s = np.std(error_results[method][size])
        avg_errors[method].append(m)
        std_errors[method].append(s)
        results[method, size] = m
results


# In[108]:


std_errors


# In[105]:


# Plot Test Error with Confidence Intervals
confidence = 0.95
z_score = norm.ppf(1 - (1 - confidence) / 2)  # Z-score for 95% confidence

ci_errors = {method: [] for method in methods}

for method in methods:
    for size in prototype_sizes:
        sample_mean = np.mean(error_results[method][size])
        sample_std = np.std(error_results[method][size])
        margin_error = z_score * (sample_std / np.sqrt(num_experiments))
        ci_errors[method].append(margin_error)
for method in methods:
    avg_error = [np.mean(error_results[method][size]) for size in prototype_sizes]
    ci_error = ci_errors[method]
    plt.errorbar(prototype_sizes, avg_error, yerr=ci_error, label=method, capsize=5)

plt.xlabel('Prototype Size')
plt.ylabel('Average Test Error')
plt.title('Average Test Error with 95% Confidence Intervals vs Prototype Size')
plt.legend()
plt.grid(True)

plt.savefig('AverageTestError95ConfidenceIntervals.png', format='png', dpi=300)
plt.show()


# In[106]:


# Plot Test Error
plt.figure(figsize=(10, 6))

for method in methods:
    sizes = []
    errors = []
    for size in prototype_sizes:
        sizes.append(size)
        errors.append(results[(method, size)])
    plt.plot(sizes, errors, marker='o', label=method)

plt.xlabel('Prototype Size')
```

```
plt.ylabel('Test Error')
plt.title('Test Error vs Prototype Size')
plt.legend()
plt.grid(True)
plt.savefig('TestError.png', format='png', dpi=300)
plt.show()

# Plot Runtime
plt.figure(figsize=(10, 6))

for method in methods:
    sizes = []
    runtimes = []
    for size in prototype_sizes:
        sizes.append(size)
        runtimes.append(runtime_results[(method, size)])
    plt.plot(sizes, runtimes, marker='o', label=method)

plt.xlabel('Prototype Size')
plt.ylabel('Runtime (seconds)')
plt.title('Runtime vs Prototype Size')
plt.yscale('log')
plt.ylim(ymin=0.1)
plt.legend()
plt.grid(True)
plt.savefig('Runtime.png', format='png', dpi=300)
plt.show()
```