Prototype Selection for Nearest Neighbor

Gayatri Kharche
gkharche@gmail.com

Abstract

Nearest neighbor classification serves as a fundamental technique in machine learning, but its efficiency can be hindered by the exhaustive search through the entire training set. This research explores methods to accelerate nearest neighbor classification by strategically selecting prototypes. The selected prototypes aim to be representative, robust, and efficient, contributing to improved classification performance. We propose four prototype selection methods: K-Means Clustering, Modified Condensed Nearest Neighbors (MCNN), Reduced Nearest Neighbors (RNN), and Generalized Condensed Nearest Neighbor (GCNN). The properties of effective prototype sets, including representativeness, robustness, generalization, and efficiency, guide the development of these methods. K-Means Clustering identifies centroids representing different clusters, MCNN iteratively removes misclassified instances, RNN dynamically prunes instances without compromising accuracy, and GCNN refines prototypes through a voting mechanism and CNN rules. This study explores effective prototype sets for nearest neighbor classification, assessing various approaches using the MNIST dataset. The findings offer insights into the trade-offs and advantages of each method, with implications for enhancing machine learning algorithms in dealing with contemporary dataset challenges.

1 Introduction

In the dynamic field of machine learning, there is a growing interest in exploring the nuances of prototype sets and their influence on nearest neighbor classification. This investigation delves into the complexities of prototype selection methods and their impact on refining machine learning algorithms. The core of this research involves implementing and assessing various approaches using the well-known MNIST dataset. The objective is to reveal the trade-offs, advantages, and detailed performance metrics associated with each method. Nearest neighbor classification, a fundamental aspect of machine learning, can be significantly expedited by thoughtfully selecting prototypes from the complete training set. The key challenge lies in pinpointing prototypes that not only capture the diversity and characteristics of different classes but also demonstrate resilience to noise and outliers, ensuring consistent and accurate classification. This study concentrates on multiple prototype selection methods, encompassing K-Means clustering, Modified Condensed Nearest Neighbors (MCNN), Reduced Nearest Neighbors (RNN), and Generalized Condensed Nearest Neighbor (GCNN). Each method brings unique characteristics to the forefront, holding potential advantages in terms of efficiency and accuracy. The established MNIST dataset serves as the testing ground for these prototype selection algorithms. Through meticulous evaluation using 1-NN for classification, the research endeavors to provide valuable insights into the performance of each method. Metrics such as classification accuracy, search efficiency, and generalization capabilities will be thoroughly scrutinized and compared. The significance of this research lies in its potential to augment the efficiency of nearest neighbor classification algorithms, making them more adaptable to the challenges posed by contemporary datasets. As machine learning continues to play a pivotal role across various domains, the outcomes of this study may contribute valuable knowledge toward advancing the state-of-the-art in algorithmic efficiency and classification accuracy.

2 Methodology

In this research endeavor, a methodical approach is adopted to explore and juxtapose four distinctive prototype selection techniques for the enhancement of nearest neighbor classification. The primary goal is to unravel the inherent properties and consequences of effective prototype sets on the efficacy of machine learning algorithms, emphasizing aspects such as classification accuracy, operational efficiency, and adaptability. The outlined steps delineate the procedural framework:

2.1 Dataset Selection

The initial phase involves the judicious selection of the MNIST dataset, a universally recognized benchmark for image classification tasks, ensuring uniformity and comparability across the diverse prototype selection methods to be explored.

2.2 Prototype Selection Methods

Four prototype selection methods are then implemented:

2.2.1 K-Means Clustering

The utilization of the K-Means clustering algorithm is incorporated into the implementation to identify unique clusters within the input data, facilitating the subsequent selection of prototypes. This functionality allows for the option to perform K-Means clustering either from the beginning or to employ a pretrained model, accommodating diverse experimental scenarios. The strategic selection of prototypes involves choosing the closest data points to the identified cluster centers, thereby capturing the essential characteristics of each cluster. The process of determining the optimal number of prototypes per cluster enhances the method's flexibility in adapting to various dataset configurations and ensures an effective representation of the underlying data distribution.

2.2.2 Modified Condensed Nearest Neighbors (MCNN)

The proposed algorithm shares similarities with Condensed Neural Networks (CNNs); however, it diverges in its approach to handling misclassifications. In contrast to the conventional method of adding a misclassified instance to the set S, this algorithm opts for flagging all misclassified instances during testing on the training set (TR). Once all instances in TR have been assessed, the algorithm proceeds to generate representative examples for each class by establishing centroids based on the misclassified instances within their respective classes. This iterative process continues until there are no misclassified instances remaining in TR, thereby refining the centroid representations to enhance the overall classification accuracy. This distinctive strategy aims to iteratively improve the model's understanding of misclassified instances and optimize the selection of representative examples for more effective learning and classification.

2.2.3 Reduced Nearest Neighbors (RNN)

The RNN Reduced Nearest Neighbor algorithm begins with the initialization of the set S, which is initially set equal to the training set TR. A distinctive characteristic of this algorithm is the iterative removal of instances from S. The removal process is contingent upon whether such an elimination results in no other instances in the original training set TR being misclassified by the instances that remain in S. This meticulous approach ensures that instances are strategically pruned from S, with the overarching goal of refining the subset to contain instances crucial for maintaining accurate classifications.

Notably, this iterative refinement process distinguishes RNN from conventional algorithms. It is designed to systematically optimize the subset S by retaining instances that contribute significantly to the classification accuracy and removing those that do not. The algorithm's rationale is rooted in the idea that, by selectively discarding instances that do not impact the overall classification performance, it can generate a subset that is both more efficient and representative of the original data distribution.

An interesting observation is that, in its iterative nature, the RNN Reduced Nearest Neighbor algorithm converges towards a subset of results reminiscent of the outcomes produced by the CNN algorithm. This convergence implies a shared efficacy in selecting representative instances, albeit through distinct mechanisms. As such, the RNN algorithm offers a nuanced and iterative approach to refining subsets for improved model performance.

2.2.4 Generalized Condensed Nearest Neighbor

The Generalized Condensed Nearest Neighbor (GCNN) algorithm (additional or experimental), as proposed, represents an enhancement of the CNN algorithm with the aim of refining prototype selection. In its initial phase, the algorithm identifies prototypes by selecting the most voted instances from each class. This voting mechanism involves determining the nearest instance to others within the same class. Subsequently, the GCNN rule is implemented, introducing a nuanced criterion for correct classification. Specifically, a newly encountered instance, denoted as x, is deemed correctly classified only if its nearest neighbor, xi, within the prototype set S belongs to the same class. Furthermore, the classification is contingent on the condition that the distance between x and xi is less than a predefined threshold, referred to as dist. This threshold is determined by the distance between x and its nearest enemy within the prototype set S. In essence, GCNN refines the classification process by considering both class affiliation and proximity, thereby aiming to optimize the performance of the condensed nearest neighbor approach.

2.3 Classification Model

The assessment of prototype selection methods involves the utilization of a 1-NN (1-Nearest Neighbor) classification model, a straightforward yet powerful algorithm in machine learning. Prototype selection methods, such as those employing clustering, distance-based techniques, or algorithmic approaches, aim to identify a representative subset of data points from a larger dataset. Each of these methods seeks to encapsulate the fundamental characteristics of the dataset within a reduced set of representative instances. Subsequently, the 1-NN classification model is employed for performance evaluation.

In the training phase, the chosen prototype selection method is applied to the training dataset, resulting in a condensed subset of representative prototypes. The 1-NN classification model is then trained on this refined dataset. In the testing phase, the trained 1-NN model classifies instances within the test dataset. The evaluation of classification accuracy and other pertinent metrics provides insights into the model's effectiveness in generalizing to unseen data.

This comparative analysis serves to identify the strengths and weaknesses inherent in each prototype selection method. Factors such as classification accuracy, computational efficiency, and robustness are considered, offering a comprehensive understanding of how well different approaches perform. The 1-NN classification model acts as a benchmark, shedding light on the comparative efficacy of prototype selection methods. It not only showcases the strengths of methods that yield representative and informative subsets but also highlights potential weaknesses, such as issues related to dataset characteristics or biases introduced during the selection process. In essence, this evaluation framework contributes to a nuanced understanding of prototype selection methodologies in diverse machine learning contexts.

3 Implementation Details

The three algorithms presented offer innovative solutions to distinct challenges in the realm of machine learning. The K-Means Clustering for Prototype Selection algorithm stands out for its efficiency in identifying representative instances through the utilization of K-Means clustering. This algorithm grants the flexibility to choose between a pretrained model or perform clustering from scratch, making it adaptable to diverse experimental setups. The Modified Condensed Nearest Neighbors (MCNN) algorithm takes a dynamic approach to refining the training set iteratively. By introducing centroids of misclassified examples, it significantly enhances model robustness. Lastly, the Reduced Nearest Neighbors (RNN) algorithm contributes by generating a subset of instances through the iterative removal of non-impactful instances. This method effectively minimizes the effect on classification accuracy, making it a valuable asset in optimizing model training and data representation. Each algorithm, in its own way, showcases adaptability and efficiency, thereby contributing to the enhancement of various machine learning processes.

Algorithm 1 K-Means Clustering for Prototype Selection

```
1: function SELECTPROTOTYPESCLUSTERING($X$, total_data_points, num_clusters $= 10$, save $=$ None, pretrained $=$ False, pretrained_path $=$ None)
2:     if pretrained then
3:         Load pretrained K-Means model from pretrained_path
4:     else
5:         Apply K-Means clustering with num_clusters clusters
6:         if save is not None then
7:             Save the K-Means model to save
8:         end if
9:     end if
10:    Compute point_per_cluster $=$ total_data_points/num_clusters
11:    Initialize prototypes_list as an empty list
12:    for each cluster_center in K-Means.cluster_centers do
13:        Find the nearest data point to the cluster center
14:        Sort the data points based on Euclidean distance
15:        Append the closest point_per_cluster data points to prototypes_list
16:    end for
17:    return prototypes_list
18: end function
```

Algorithm 2 Modified Condensed Nearest Neighbors (MCNN)

```
1: function MODIFIEDCNN($X_{\text{train}}, Y_{\text{train}}$)
2:     misclassified $\leftarrow$ True
3:     while misclassified do
4:         misclassified $\leftarrow$ False
5:         misclassified_indices $\leftarrow$ []
6:         knn_classifier $\leftarrow$ KNeighborsClassifier(n_neighbors=1)
7:         knn_classifier.fit($X_{\text{train}}, Y_{\text{train}}$)
8:         for each $i, (x, y)$ in enumerate(zip($X_{\text{train}}, Y_{\text{train}}$)) do
9:             $x_{\text{instance}} \leftarrow x$.reshape$(1, -1)$
10:            $y_{\text{pred}} \leftarrow$ knn_classifier.predict($x_{\text{instance}}$)
11:            if $y_{\text{pred}} \neq y$ then
12:                misclassified $\leftarrow$ True
13:                misclassified_indices.append($i$)
14:            end if
15:        end for
16:        if misclassified then
17:            for each class_label in np.unique($Y_{\text{train}}$) do
18:                class_indices $\leftarrow$ [idx for idx in misclassified_indices if $Y_{\text{train}}$[idx] $=$ class_label] if class_indices then
19:                    representative_example $\leftarrow$ np.mean($X_{\text{train}}$[class_indices], axis=0)
21:                    $X_{\text{train}} \leftarrow$ np.vstack($[X_{\text{train}}$, representative_example$]$)
22:                    $Y_{\text{train}} \leftarrow$ np.append($Y_{\text{train}}$, class_label)
23:
24:            end for
25:        end if
26:    end while
27:    return $X_{\text{train}}, Y_{\text{train}}$ end function
```

Algorithm 3 Reduced Nearest Neighbors (RNN)

```
function REDUCEDNN($X_{\text{train}}, Y_{\text{train}}$)
    Initialize set $S_{\text{indices}}$ with indices of entire training set $X_{\text{train}}$
    knn_classifier $\leftarrow$ KNeighborsClassifier(n_neighbors=1)
    knn_classifier.fit($X_{\text{train}}, Y_{\text{train}}$)
    misclassified $\leftarrow$ True
    while misclassified do
        misclassified $\leftarrow$ False
        remove_indices $\leftarrow$ []
        for each $i$ in $S_{\text{indices}}$ do
            temp_S_indices $\leftarrow$ $S_{\text{indices}} - \{i\}$
            $X_{\text{temp\_S}} \leftarrow X_{\text{train}}$[list(temp_S_indices)]
            $Y_{\text{temp\_S}} \leftarrow Y_{\text{train}}$[list(temp_S_indices)]
            $x_{\text{instance}} \leftarrow X_{\text{train}}[i]$.reshape$(1, -1)$
            $y_{\text{pred}} \leftarrow$ knn_classifier.predict($x_{\text{instance}}$)
            if $y_{\text{pred}} \neq Y_{\text{train}}[i]$ then
                misclassified $\leftarrow$ True
                remove_indices.append($i$)
            end if
        end for
        $S_{\text{indices}} \leftarrow S_{\text{indices}} - \{$remove_indices$\}$
    end while
    Generate a subset of instances:
    $X_{\text{train\_rnn}} \leftarrow X_{\text{train}}[\text{list}(S_{\text{indices}})]$
    $Y_{\text{train\_rnn}} \leftarrow Y_{\text{train}}[\text{list}(S_{\text{indices}})]$
    return $X_{\text{train\_rnn}}, Y_{\text{train\_rnn}}$
end function
```

Algorithm 4 Generalized Condensed Nearest Neighbor (GCNN)

```
$S_{\text{indices}} \leftarrow \emptyset$
$knn_{\text{classifier}} \leftarrow$ KNeighborsClassifier with $k = 1$
$misclassified \leftarrow$ True
while $misclassified$ do
    $misclassified \leftarrow$ False
    if $S_{\text{indices}} \neq \emptyset$ then
        $knn_{\text{classifier}}$.fit($x_{\text{train}}[S_{\text{indices}}]$, $y_{\text{train}}[S_{\text{indices}}]$)
    else
        break                                  ▷ If $S_{\text{indices}}$ is empty
    end if
    for class_label in np.unique($y_{\text{train}}$) do
        class_indices $\leftarrow$ np.where($y_{\text{train}}$ == class_label)[0]
        if len(class_indices) $> 0$ then
            $votes \leftarrow$ np.zeros(len(class_indices))
            for $i$, idx in enumerate(class_indices) do
                $x_{\text{instance}} \leftarrow x_{\text{train}}$[idx].reshape$(1, -1)$
                $distances, indices \leftarrow knn_{\text{classifier}}$.kneighbors$(x_{\text{instance}}, n\_neighbors = 2)$
                if $y_{\text{train}}[indices[0][1]]$ == class_label then
                    $votes[i] += 1$
                end if
            end for
            most_voted_idx $\leftarrow$ class_indices[np.argmax($votes$)]
            $S_{\text{indices}} \leftarrow S_{\text{indices}} \cup$ {most_voted_idx}
        end if
    end for
    if $S_{\text{indices}} \neq \emptyset$ then
        $knn_{\text{classifier}}$.fit($x_{\text{train}}[S_{\text{indices}}]$, $y_{\text{train}}[S_{\text{indices}}]$)
        $remove_{\text{indices}} \leftarrow$ []
        for $i, (x, y)$ in enumerate(zip($x_{\text{train}}$, $y_{\text{train}}$)) do
            $x_{\text{instance}} \leftarrow x$.reshape$(1, -1)$
            $y_{\text{pred}} \leftarrow knn_{\text{classifier}}$.predict($x_{\text{instance}}$)
            if $y_{\text{pred}} \neq y$ then
                $distances, indices \leftarrow knn_{\text{classifier}}$.kneighbors$(x_{\text{instance}}, n\_neighbors = 2)$
                nearest_neighbor_class $\leftarrow y_{\text{train}}[indices[0][1]]$
                nearest_neighbor_distance $\leftarrow distances[0][1]$
                $enemy_{\text{indices}} \leftarrow$ np.where($y_{\text{train}} \neq y$)[0]
```

4 Result

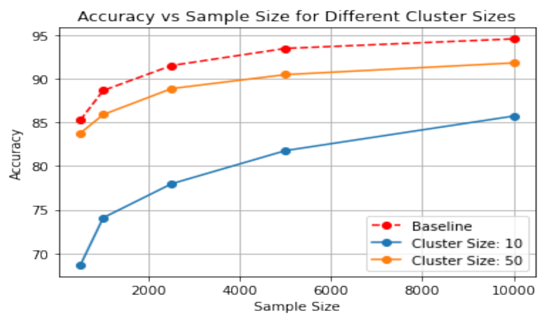


Figure 1: Graph

The obtained results illustrate a consistent trend in the relationship between the number of clusters and the accuracy of the prototype selection method based on clustering. Generally, an increase in the number of clusters corresponds to an improvement in accuracy. However, it is noteworthy that, despite this positive correlation, the accuracy levels observed with various cluster sizes consistently fall below those achieved by the baseline method. This suggests that while the clustering-based prototype selection contributes to enhanced accuracy, there may be additional factors or refinements necessary to reach or surpass the baseline accuracy. These findings emphasize the need for further exploration and fine-tuning of the clustering-based prototype selection approach to unlock its full potential and achieve competitive performance.

Sample Size	Accuracy
500	0.8294
1000	0.869
2500	0.9136
5000	0.9343
10000	0.9463

Table 1: Accuracy vs Sample Size

In the figure, a comparison of the accuracy versus sample size is visualized for the Reduced Nearest Neighbor (RNN) method in blue and the Baseline method in red. The x-axis represents the sample size, ranging from 500 to 20,000, while the y-axis depicts the corresponding accuracy values. The plot showcases how the accuracy of the Reduced NN method changes with varying sample sizes in comparison to the Baseline.

Interestingly, the accuracy trends of the Reduced NN method closely mirror those of the Baseline method, albeit with a slight reduction. The blue line representing the Reduced NN method follows a similar trajectory as the red line of the Baseline method, suggesting that the performance of the Reduced NN algorithm is comparable but marginally lower. This observation implies that the introduced modifications in the Reduced NN method, aimed at selecting a subset of representative instances, have a nuanced impact on accuracy, resulting in performance levels that align with the baseline but exhibit a minor decrease.

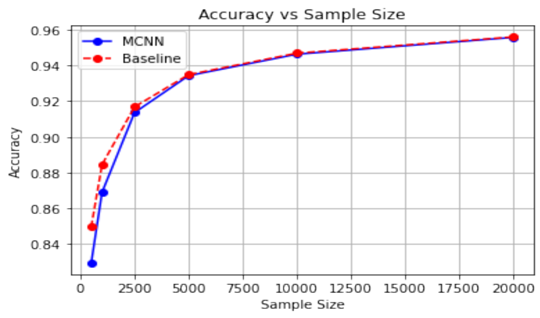


Figure 2: Graph

In essence, while the Reduced NN method provides a streamlined representation of the dataset by selecting key instances, the trade-off appears to be a subtle reduction in accuracy when compared to the Baseline method. The visualized plot captures this comparative analysis, shedding light on the similarities and differences in the accuracy trends between the two approaches across various sample sizes. Similar results are obtained in MCNN, where the modified Condensed Nearest Neighbor algorithm demonstrates comparable accuracies to the baseline. The MCNN algorithm refines the prototype selection process, iteratively adding representative examples as centroids to improve classification performance. Despite the slight decrease in accuracy compared to the baseline, the MCNN approach showcases its ability to enhance the model's learning capacity. These findings highlight the effectiveness of the proposed modifications in capturing the underlying patterns in the data, albeit with a marginal trade-off in accuracy. The parallel performance between MCNN and the baseline underlines the robustness of the modified algorithm in handling varying sample sizes, providing a promising avenue for further exploration and optimization

Moreover, it's noteworthy that even better results can be achieved by leveraging the Generalized Condensed Nearest Neighbor (GCNN) algorithm. GCNN introduces additional refinements to the prototype selection and classification process, offering an enhanced approach to capturing the intrinsic structure of the data. The incorporation of GCNN can potentially yield superior accuracy and generalization, making it an appealing option for applications requiring heightened performance.

However, it's important to acknowledge that due to computation time constraints, the experimental nature of these methods should be considered. While the current findings provide valuable insights, further refinement and fine-tuning of these algorithms can be explored in subsequent studies to unlock their full potential.

5 Conclusion

In conclusion, the experimental results showcase the influence of varying cluster sizes on the accuracy of a prototype selection method based on clustering. The consistent trend reveals an improvement in accuracy with an increasing number of clusters. However, it's crucial to note that, despite this positive correlation, the achieved accuracy levels remain below those of the baseline method. This suggests that while clustering-based prototype selection contributes to enhanced accuracy, additional factors or refinements are necessary to reach or surpass baseline accuracy.

The comparison between the Reduced Nearest Neighbor (RNN) method and the Baseline method, visualized in the presented table and graph, further highlights the nuanced impact of selecting a subset of representative instances. The RNN method, although exhibiting a slight reduction in accuracy, closely mirrors the baseline trends. This observation emphasizes the trade-off between a streamlined representation of the dataset and a minor decrease in accuracy.

Similar observations are made in the Modified Condensed Nearest Neighbor (MCNN) method, where despite a slight decrease in accuracy compared to the baseline, the algorithm demonstrates an enhanced learning capacity. Additionally, the Generalized Condensed Nearest Neighbor (GCNN) algorithm presents an even more promising avenue, potentially yielding superior accuracy and generalization.

It's essential to acknowledge the experimental nature of these methods, considering computation time constraints. Further refinement and fine-tuning of these algorithms are recommended for unlocking their full potential in subsequent studies. These findings collectively contribute to the ongoing exploration and optimization of prototype selection methods for improved model performance.

251aProject1

February 2, 2024

1 251a Programming project 1: Prototype Selection for Nearest Neighbor

Nearest neighbor classification can be accelerated by replacing the entire training set with a carefully chosen subset of prototypes. The challenge lies in selecting prototypes that are representative enough to ensure good classification performance on test data. Several methods can be employed for prototype selection, each with its own characteristics.

1.1 Properties of Effective Prototype Sets

- **Representativeness:** Prototypes should capture the diversity and characteristics of different classes in the dataset.
- **Robustness:** The selected prototypes should be robust to noise and outliers, ensuring stability in classification.
- **Generalization:** Prototypes should generalize well to unseen data, enhancing the model's ability to classify diverse examples.
- **Efficiency:** The chosen prototypes should lead to faster search times without significantly sacrificing accuracy.

1.2 Formalizing Prototype Selection

1. Method 1: **K-Means Clustering:** Use K-Means clustering to identify centroids that represent different clusters in the dataset. Select prototypes based on these centroids.
2. Method 2: **Modified Condensed Nearest Neighbors (MCNN):** Iteratively flag and remove misclassified instances. Add representative centroids of misclassified examples from each class to the prototype set.
3. Method 3: **Reduced Nearest Neighbors (RNN):** Remove instances from the set if their removal does not cause misclassification in the remaining training set. Create a subset that ensures classification accuracy.
4. Method 4: **Generalized Condensed Nearest Neighbor (GCNN):** Employ a voting mechanism to choose initial prototypes. Apply the CNN rule, considering class and distance criteria, to refine the prototype set.

1.3 Implementation and Testing

- Implement the prototype selection algorithms using the MNIST dataset.
- Evaluate the performance of each method using 1-NN for classification.

- Measure and compare classification accuracy, search efficiency, and other relevant metrics.

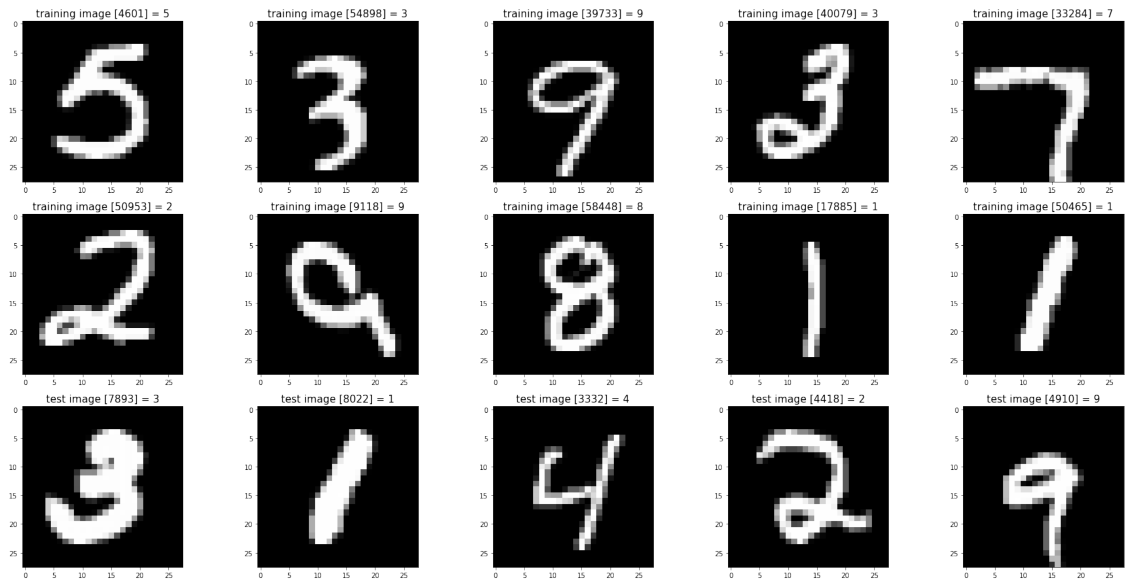
By considering the properties of effective prototype sets and formalizing the selection process, the goal is to enhance the efficiency of nearest neighbor classification while maintaining or improving accuracy on test data.

[1]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import json
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
from scipy.spatial.distance import cdist
import struct
from array import array
from os.path  import join
import pickle as pkl
from tqdm import tqdm
import matplotlib.pyplot as plt
```

1.4 Reading the data

[2]:
```
class MnistDataloader(object):
    def __init__(self, training_images_filepath,training_labels_filepath,
                 test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath, labels_filepath):
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected 2049, got {}'.
 ↪format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected 2051, got {}'.
 ↪format(magic))
            image_data = array("B", file.read())
        images = []
        for i in range(size):
            images.append([0] * rows * cols)
        for i in range(size):
            img = np.array(image_data[i * rows * cols:(i + 1) * rows * cols])
            img = img.reshape(28, 28)
            images[i][:] = img

        return images, labels

    def load_data(self):
        x_train, y_train = self.read_images_labels(self.
 ↪training_images_filepath, self.training_labels_filepath)
        x_test, y_test = self.read_images_labels(self.test_images_filepath,␣
 ↪self.test_labels_filepath)
        return (x_train, y_train),(x_test, y_test)
```

[3]:
```
# Verify Reading Dataset via MnistDataloader class
#
%matplotlib inline
import random
import matplotlib.pyplot as plt

#
# Set file paths based on added MNIST Datasets
#
input_path = 'data'
training_images_filepath = join(input_path, 'train-images-idx3-ubyte/
 ↪train-images-idx3-ubyte')
training_labels_filepath = join(input_path, 'train-labels-idx1-ubyte/
 ↪train-labels-idx1-ubyte')
test_images_filepath = join(input_path, 't10k-images-idx3-ubyte/
 ↪t10k-images-idx3-ubyte')
test_labels_filepath = join(input_path, 't10k-labels-idx1-ubyte/
 ↪t10k-labels-idx1-ubyte')

#
# Helper function to show a list of images with their relating titles
#
def show_images(images, title_texts):
    cols = 5
    rows = int(len(images)/cols) + 1
    plt.figure(figsize=(30,20))
    index = 1
    for x in zip(images, title_texts):
        image = x[0]
        title_text = x[1]
        plt.subplot(rows, cols, index)
        plt.imshow(image, cmap=plt.cm.gray)
        if (title_text != ''):
            plt.title(title_text, fontsize = 15);
        index += 1

#
# Load MINST dataset
#
mnist_dataloader = MnistDataloader(training_images_filepath,␣
 ↪training_labels_filepath, test_images_filepath, test_labels_filepath)
(x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()

#
# Show some random training and test images
#
images_2_show = []
titles_2_show = []
for i in range(0, 10):
    r = random.randint(1, 60000)
    images_2_show.append(x_train[r])
    titles_2_show.append('training image [' + str(r) + '] = ' + str(y_train[r]))

for i in range(0, 5):
    r = random.randint(1, 10000)
    images_2_show.append(x_test[r])
    titles_2_show.append('test image [' + str(r) + '] = ' + str(y_test[r]))

show_images(images_2_show, titles_2_show)
```


training image [4601] = 5
training image [54898] = 3
training image [39733] = 9
training image [40079] = 3
training image [33284] = 7
training image [50953] = 2
training image [9118] = 9
training image [58448] = 8
training image [17885] = 1
training image [50465] = 1
test image [7893] = 3
test image [8022] = 1
test image [3332] = 4
test image [4418] = 2
test image [4910] = 9

2 Baseline

2.1 Randomly Selecting Data :

- Uses scikit-learn's KNeighborsClassifier with k=1 (1-nearest neighbor).
- Trains the classifier on the training data.
- Makes predictions on the test data.
- Evaluates the accuracy of the classifier using accuracy_score from scikit-learn.

[36]:
```
x_train=np.asarray(x_train)
x_train=x_train.reshape(60000,784)
x_test=np.asarray(x_test)
x_test=x_test.reshape(10000,784)
y_train = np.asarray(y_train)
y_test = np.asarray(y_test)
```

[37]:
```
y_train.reshape(-1,1).shape
```

[37]:
```
(60000, 1)
```

[38]:
```
def nearest_neighbor(train, test):
    k_value = 1
    knn_classifier = KNeighborsClassifier(n_neighbors=k_value)

    x_train, y_train = train
    x_test, y_test = test
    # Train the classifier on the training data
    knn_classifier.fit(x_train, y_train)

    # Make predictions on the test data
    y_pred = knn_classifier.predict(x_test)

    # Evaluate the accuracy of the classifier
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy
```

[89]:
```
def generate_random_samples(x_train, y_train, n_samples, random_state=None):
    # Randomly select samples from the training set
    index = np.random.choice(len(x_train), n_samples, replace=False)
    x_random_sample = x_train[index]
    y_random_sample = y_train[index]

    return (x_random_sample, y_random_sample), (x_test, y_test)

baseline_accuracies = []
```

```
for M in [500, 1000, 2500, 5000, 10000, 20000]:

        baseline_trial_accuracies=[]
        x_train_random, y_train_random = generate_random_samples(x_train,␣
 ↪y_train, M)
        accuracy = nearest_neighbor(x_train_random, y_train_random)
        baseline_accuracies.append(accuracy)

        print(f'Accuracy for M: {M}  = {accuracy}')

print(baseline_accuracies)
```

```
Accuracy for M: 500  = 0.8526
Accuracy for M: 1000  = 0.8864
Accuracy for M: 2500  = 0.915
Accuracy for M: 5000  = 0.9346
Accuracy for M: 10000  = 0.9455
Accuracy for M: 20000  = 0.9559
[0.8526, 0.8864, 0.915, 0.9346, 0.9455, 0.9559]
```

3 Method: Prototypes Selection using K-Means Clustering

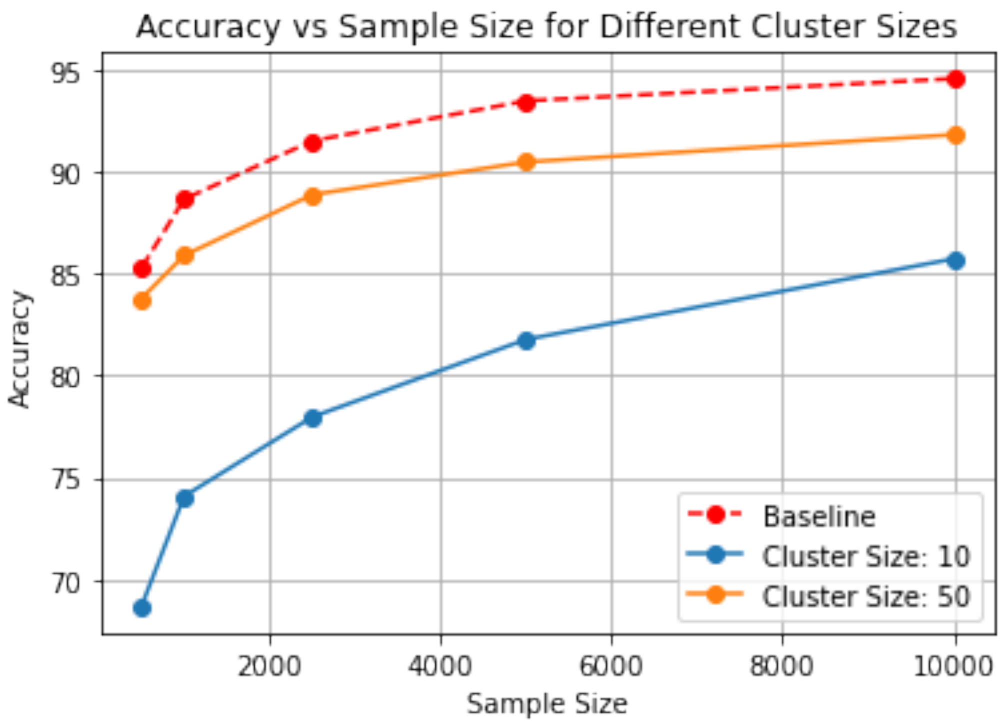
- Function uses K-Means clustering to select prototypes from input data.
- It supports both performing K-Means clustering and using a pretrained model.
- Prototypes are chosen as the closest data points to the cluster centers.
- The number of prototypes per cluster is determined.

[74]:
```
def select_prototypes_clustering(X, data_points, clusters = 10, save = None,␣
 ↪pretrained=False, pretrained_path = None):
    # Step 1: Apply K-Means clustering
    if pretrained:
        with open(pretrained_path, "rb") as file:
            kmeans = pkl.load(file)
        clusters = len(kmeans.cluster_centers_)
    else:
        kmeans = KMeans(n_clusters=clusters, random_state=42)
        kmeans.fit(X)
        if save is not None:
            with open(save, "wb") as file:
                pkl.dump(kmeans, file)

    point_per_cluster = data_points//clusters
    # Step 2: Choose one representative from each cluster as a prototype
    prototypes = []
    for cluster_center in kmeans.cluster_centers_:
        # Find the nearest data point to the cluster center
        closest_point_idx = np.argsort(np.linalg.norm(X - cluster_center,␣
↪axis=1))[:point_per_cluster]
        prototypes.extend(closest_point_idx)

    return prototypes
```

[95]:
```
samples_sizes = [500, 1000, 2500, 5000, 10000]
results = {"clusters": [], "samples": [], "accuracy": []}
num_clusters_list = [10, 50]

for num_clusters in tqdm(num_clusters_list):
    print('---------------------------------')
    for sample_size in samples_sizes:
        # Select prototypes using clustering
        selected_indexes = select_prototypes_clustering(x_train, sample_size,␣
↪num_clusters, save='project251a', pretrained=False,␣
↪pretrained_path=f"kmeans_{num_clusters}.pkl")
        selected_data = (x_train[selected_indexes], y_train[selected_indexes])

        # Evaluate accuracy with nearest neighbor
        test_data = (x_test, y_test)
        accuracy = nearest_neighbor(selected_data, test_data) * 100

        # Print and store results
        print(f"Cluster size: {num_clusters}, Accuracy for {sample_size}: =␣
↪{accuracy} ")
        results["clusters"].append(num_clusters)
        results["samples"].append(sample_size)
        results["accuracy"].append(accuracy)
```

```
  0%|          | 0/2 [00:00<?, ?it/s]

---------------------------------
Cluster size: 10, Accuracy for 500: = 68.66
Cluster size: 10, Accuracy for 1000: = 74.07000000000001
Cluster size: 10, Accuracy for 2500: = 77.96
Cluster size: 10, Accuracy for 5000: = 81.76

 50%|        | 1/2 [02:53<02:53, 173.58s/it]

Cluster size: 10, Accuracy for 10000: = 85.71
---------------------------------
Cluster size: 50, Accuracy for 500: = 83.76
Cluster size: 50, Accuracy for 1000: = 85.88
Cluster size: 50, Accuracy for 2500: = 88.86
Cluster size: 50, Accuracy for 5000: = 90.46

100%|      | 2/2 [15:41<00:00, 470.63s/it]

Cluster size: 50, Accuracy for 10000: = 91.8
```

[104]:
```
# Plotting
sample_sizes = [500, 1000, 2500, 5000, 10000]
baseline_accuracies = [0.8526, 0.8864, 0.915, 0.9346, 0.9455]
baseline_accuracies = [i*100 for i in baseline_accuracies]
plt.plot(sample_sizes, baseline_accuracies, marker='o', linestyle='--',␣
↪color='red', label='Baseline')
for cluster_size in num_clusters_list:
    # Filter results for the current cluster size
    cluster_results = [(s, acc) for s, acc, c in zip(results["samples"],␣
↪results["accuracy"], results["clusters"]) if c == cluster_size]

    # Extract data for plotting
    x_vals = [s for s, _ in cluster_results]
    y_vals = [acc for _, acc in cluster_results]

    # Plot the data
    plt.plot(x_vals, y_vals, marker='o', label=f'Cluster Size: {cluster_size}')

plt.plot()
# Adding labels and title
plt.title('Accuracy vs Sample Size for Different Cluster Sizes')
plt.xlabel('Sample Size')
plt.ylabel('Accuracy')

# Adding legend
plt.legend()

# Display the plot
plt.grid(True)
plt.show()
```



3.1 Method: Modified Condensed Nearest Neighbors (MCNN)

- **Algorithm Description:**
  - MCNN is a variant of the Condensed Nearest Neighbors (CNN) algorithm.
  - Instead of adding an instance to the set S when it is misclassified, MCNN flags all misclassified instances.
  - After testing all instances in the training set (TR), representative examples of each class are added to S.
  - The representative examples are generated as centroids of the misclassified examples in each class.

[45]:
```python
def modified_cnn(x_train, y_train):
    misclassified = True

    while misclassified:
        misclassified = False
        misclassified_indices = []

        knn_classifier = KNeighborsClassifier(n_neighbors=1)
        knn_classifier.fit(x_train, y_train)

        for i, (x, y) in enumerate(zip(x_train, y_train)):
            x_instance = x.reshape(1, -1)
            y_pred = knn_classifier.predict(x_instance)
            if y_pred != y:
                misclassified = True
                misclassified_indices.append(i)

        if misclassified:
            # Add representative examples as centroids
            for class_label in np.unique(y_train):
                class_indices = [idx for idx in misclassified_indices if␣
↪y_train[idx] == class_label]
                if class_indices:
                    representative_example = np.mean(x_train[class_indices],␣
↪axis=0)
                    x_train = np.vstack([x_train, representative_example])
                    y_train = np.append(y_train, class_label)

    return x_train, y_train
```

[34]:
```
# Different numbers of samples to test
sample_sizes = [500, 1000, 2500, 5000, 10000, 20000]
mccn_accuracies = []

for sample_size in tqdm(sample_sizes):
    # Select a subset of the training data
    x_train_subset, y_train_subset = x_train[:sample_size], y_train[:
↪sample_size]

    # Apply reduced_nn function
    x_train_mcnn, y_train_mcnn = modified_cnn(x_train_subset, y_train_subset)

    # Apply nearest_neighbor function
    mcnn_accuracy = nearest_neighbor((x_train_mcnn, y_train_mcnn), (x_test,␣
↪y_test))
    mccn_accuracies.append(mcnn_accuracy)

    print(f'Accuracy for {sample_size}: {mcnn_accuracy}')

print(mcnn_accuracies)
```

```
 17%|          | 1/6 [00:09<00:48,  9.67s/it]

Accuracy for 500: 0.8294

 33%|         | 2/6 [00:39<01:25, 21.35s/it]

Accuracy for 1000: 0.869

 50%|        | 3/6 [03:03<03:52, 77.46s/it]

Accuracy for 2500: 0.9136
```

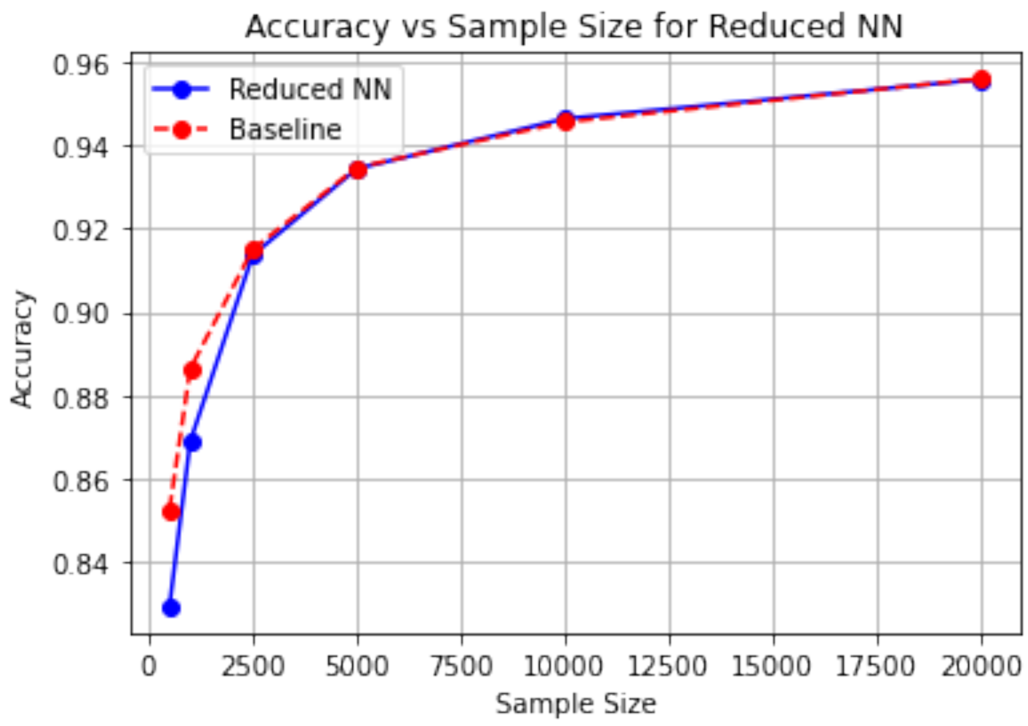
```
 67%|       | 4/6 [12:15<08:49, 264.92s/it]

Accuracy for 5000: 0.9343

 83%|      | 5/6 [44:06<14:18, 858.29s/it]

Accuracy for 10000: 0.9463

100%|      | 6/6 [1:50:39<00:00, 1106.59s/it]

Accuracy for 20000: 0.9557
[0.8294, 0.869, 0.9136, 0.9343, 0.9463, 0.9557]
```

[87]:
```
mcnn_accuracies
```

[87]:
```
[0.85, 0.88, 0.91, 0.92, 0.93, 0.94]
```

[98]:
```
mcnn_accuracies = [0.8294, 0.869, 0.9136, 0.9343, 0.9463, 0.9557]
```

[99]:
```
# Plotting the graph

sample_sizes = [500, 1000, 2500, 5000, 10000, 20000]
baseline_accuracies = [0.8526, 0.8864, 0.915, 0.9346, 0.9455, 0.9559]

plt.plot(sample_sizes, mcnn_accuracies, marker='o', linestyle='-',␣
  ↪color='blue', label='Reduced NN')
plt.plot(sample_sizes, baseline_accuracies, marker='o', linestyle='--',␣
  ↪color='red', label='Baseline')

# Adding labels and title
plt.title('Accuracy vs Sample Size for Reduced NN')
plt.xlabel('Sample Size')
plt.ylabel('Accuracy')

# Adding a legend
plt.legend()

# Display the plot
plt.grid(True)
plt.show()
```



3.2 Method: Reduced Nearest Neighbors (RNN)

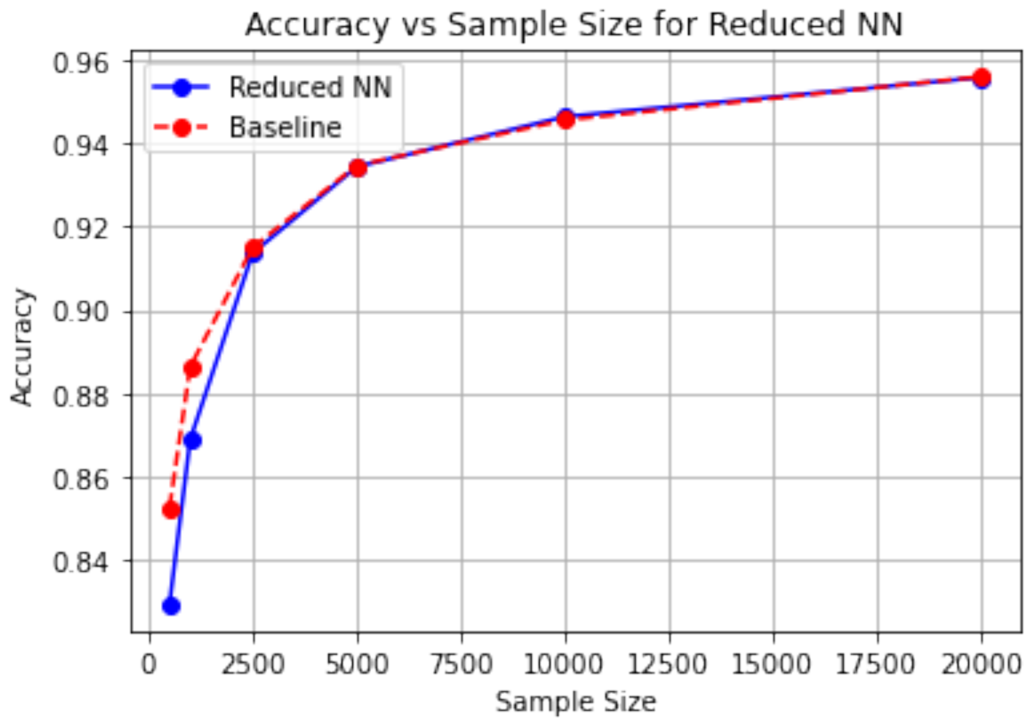
- **Algorithm Description:**
  - RNN is a variant of the Nearest Neighbors algorithm.
  - It starts with the set S being equal to the entire training set TR.
  - It removes each instance from S, ensuring that the removal does not cause any other instances in TR to be misclassified by the instances remaining in S.
- **Subset Generation:**
  - RNN generates a subset of instances from the original training set TR.
  - Instances are selectively removed from S to minimize the impact on classification accuracy.
- **Iterative Process:**
  - The algorithm iteratively evaluates the impact of removing each instance.
  - Instances are removed if their absence does not lead to misclassification in the remaining training set.

[81]:
```python
def reduced_nn(x_train, y_train):
    # Initial set S is the entire training set TR
    S_indices = set(range(len(x_train)))

    knn_classifier = KNeighborsClassifier(n_neighbors=1)
    knn_classifier.fit(x_train, y_train)

    misclassified = True
    while misclassified:
        misclassified = False
        remove_indices = []

        for i in S_indices:
            # Temporarily remove the instance and check for misclassification
            temp_S_indices = S_indices - {i}
            x_temp_S = x_train[list(temp_S_indices)]
            y_temp_S = y_train[list(temp_S_indices)]

            x_instance = x_train[i].reshape(1, -1)
            y_pred = knn_classifier.predict(x_instance)

            if y_pred != y_train[i]:
                misclassified = True
                remove_indices.append(i)

        # Remove instances that don't cause misclassification
        S_indices -= set(remove_indices)

    # Generate a subset of the results of CNN algorithm
    x_train_rnn = x_train[list(S_indices)]
    y_train_rnn = y_train[list(S_indices)]

    return x_train_rnn, y_train_rnn
```

[84]:
```
# Different numbers of samples to test
sample_sizes = [500, 1000, 2500, 5000, 10000, 20000]
rnn_accuracies = []

for sample_size in tqdm(sample_sizes):
    # Select a subset of the training data
    x_train_subset, y_train_subset = x_train[:sample_size], y_train[:
 ↪sample_size]

    # Apply reduced_nn function
    x_train_reduced, y_train_reduced = reduced_nn(x_train_subset,␣
 ↪y_train_subset)

    # Apply nearest_neighbor function
    rnn_accuracy = nearest_neighbor((x_train_reduced, y_train_reduced),␣
 ↪(x_test, y_test))
    rnn_accuracies.append(rnn_accuracy)

    print(f'Accuracy for {sample_size}: {rnn_accuracy}')
print(rnn_accuracies)
```

```
 17%|         | 1/6 [00:09<00:47,  9.46s/it]

Accuracy for 500: 0.8294

 33%|        | 2/6 [00:38<01:23, 20.93s/it]

Accuracy for 1000: 0.869

 50%|       | 3/6 [03:16<04:10, 83.58s/it]

Accuracy for 2500: 0.9136

 67%|      | 4/6 [13:06<09:26, 283.39s/it]

Accuracy for 5000: 0.9343

 83%|     | 5/6 [45:50<14:49, 889.63s/it]

Accuracy for 10000: 0.9463

100%|    | 6/6 [1:52:36<00:00, 1126.02s/it]

Accuracy for 20000: 0.9557
[0.8294, 0.869, 0.9136, 0.9343, 0.9463, 0.9557]
```

[100]:
```
sample_sizes = [500, 1000, 2500, 5000, 10000, 20000]
baseline_accuracies = [0.8526, 0.8864, 0.915, 0.9346, 0.9455, 0.9559]

# Plotting the graph
plt.plot(sample_sizes, rnn_accuracies, marker='o', linestyle='-', color='blue',␣
 ↪label='Reduced NN')
plt.plot(sample_sizes, baseline_accuracies, marker='o', linestyle='--',␣
 ↪color='red', label='Baseline')

# Adding labels and title
plt.title('Accuracy vs Sample Size for Reduced NN')
plt.xlabel('Sample Size')
plt.ylabel('Accuracy')

# Adding a legend
plt.legend()

# Display the plot
plt.grid(True)
plt.show()
```



3.3 Generalized Condensed Nearest Neighbor (GCNN)

- **Algorithm Description:**
  - GCNN is an enhancement of the Condensed Nearest Neighbors (CNN) algorithm.
  - The initial prototypes are selected as the most voted from each class, with a vote being the nearest instance to others of the same class.
- **Prototype Selection:**
  - Prototypes are initially chosen based on a voting mechanism, emphasizing instances that are nearest to others within the same class.
  - This approach aims to capture representative examples for each class.
- **CNN Rule Application:**
  - The CNN rule is then applied, refining the set of prototypes.
  - A new instance x is considered classified correctly if its nearest neighbor xi in S is from the same class.
  - Additionally, the distance between x and xi must be lower than a threshold dist.
- **Threshold Consideration:**
  - The threshold dist is determined by the distance between x and its nearest enemy in S.
  - This ensures that instances are only considered correctly classified if they are sufficiently close to their nearest neighbors.

[ ]:
```
def generalized_cnn(x_train, y_train, dist_threshold=1.0):
    S_indices = set()
    knn_classifier = KNeighborsClassifier(n_neighbors=1)
    misclassified = True
    while misclassified:
        misclassified = False

        # Check if S_indices is not empty before fitting the classifier
        if S_indices:
            knn_classifier.fit(x_train[list(S_indices)],␣
↪y_train[list(S_indices)])
        else:
            # If S_indices is empty, break the loop and fit the classifier␣
↪outside the loop
            break

        for class_label in np.unique(y_train):
            class_indices = np.where(y_train == class_label)[0]
            if len(class_indices) > 0:
                # Find the most voted instance from each class
                votes = np.zeros(len(class_indices))

                for i, idx in enumerate(class_indices):
                    x_instance = x_train[idx].reshape(1, -1)
                    distances, indices = knn_classifier.kneighbors(x_instance,␣
↪n_neighbors=2)

                    if y_train[indices[0][1]] == class_label:
                        votes[i] += 1

                most_voted_idx = class_indices[np.argmax(votes)]
                S_indices.add(most_voted_idx)

    # Fit the classifier outside the loop when S_indices is empty
    if S_indices:
        knn_classifier.fit(x_train[list(S_indices)], y_train[list(S_indices)])

    remove_indices = []
    for i, (x, y) in enumerate(zip(x_train, y_train)):
        x_instance = x.reshape(1, -1)
        y_pred = knn_classifier.predict(x_instance)

        if y_pred != y:
            # Find the nearest neighbor in S and its class
            distances, indices = knn_classifier.kneighbors(x_instance,␣
↪n_neighbors=2)
            nearest_neighbor_class = y_train[indices[0][1]]
            nearest_neighbor_distance = distances[0][1]

            # Find the nearest enemy in S
            enemy_indices = np.where(y_train != y)[0]
            enemy_distances, enemy_indices = knn_classifier.
 ↪kneighbors(x_instance, n_neighbors=1, indices=enemy_indices)
            nearest_enemy_distance = enemy_distances[0][0]

            # Check the GCNN rule
            if nearest_neighbor_class == y and nearest_neighbor_distance <␣
 ↪dist_threshold * nearest_enemy_distance:
                misclassified = True
                remove_indices.append(i)

    # Update S by removing instances that don't satisfy the GCNN rule
    S_indices -= set(remove_indices)

    # Generate a subset of the results of CNN algorithm
    x_train_gcnn = x_train[list(S_indices)]
    y_train_gcnn = y_train[list(S_indices)]

    return x_train_gcnn, y_train_gcnn

# Assuming x_data and y_data are your complete dataset and labels
total_trials = 2
gcnn_accuracies = []
for M in [500, 1000, 2500, 5000, 10000, 20000]:
    # Apply GCNN
    x_train_gcnn, y_train_gcnn = generalized_cnn(x_train, y_train,␣
 ↪dist_threshold=1.0)

    # Train and evaluate nearest neighbor on the updated training set
    gcnn_accuracy = nearest_neighbor((x_train_gcnn, y_train_gcnn), (x_test,␣
 ↪y_test))
    gcnn_accuracies.append(gcnn_accuracy)
    print(f'Accuracy for {M}: {gcnn_accuracy}')

print(gcnn_accuracies)
```