Prototype Selection for Nearest Neighbor

Yousef Hawas
yhawas@ucsd.edu

Abstract

This report contains a description of the algorithm used for prototype selection for the Nearest Neighbors algorithm. Furthermore, the algorithm is compared to a baseline prototype selection (uniform random selection). Both algorithms are tested on the MNIST digit dataset.

1 Introduction and Algorithm Description

The Nearest Neighbor algorithm in its simplest form suffers from a high time complexity that scales linearly with the size of the training data. In order to counteract this effect, prototype selection is used. Prototype selection entails picking a subset of data points from the training set. This helps to minimize the size of the training set along with the time complexity of prediction. The proposed algorithm for prototype selection is as follows. The algorithm is first provided with the labelled training set as well as a number **M** that corresponds to the number of prototypes to select. Then the *average* data points for each class in the training set is computed, and the prototypes selected are the points closest to their class's *average* data point. The algorithm also ensures a balanced training set by having the same number of data points per class. The motivation behind this algorithm is to protect against outliers within every class that lie far away from the class's average and cause wrong predictions.

2 Algorithm Pseudocode

The psuedocode for the algorithm is shown in the **Algorithm 1** figure. The algorithm takes in the labelled training set and the desired number of prototypes, **M**. The parameter M is divisible by 10 so as to ensure a balanced dataset. It then returns a prototype set of size M and its corresponding labels.

Algorithm 1: Prototype selection algorithm

```
foreach point in datapoints do
    Insert point into dictionary with
      corresponding label as key
end
foreach class label in dictionary do
    Calculate average data point for class
      label
end
foreach point in datapoints do
    Calculate Euclidean Distance between
      point and it's class average
end
Sort the distances in ascending order
foreach index in sorted distance indices do
    Point = Training Set [index]
    Class = Training Labels [index]
    if Number of class points in prototype
      set < M/10 then
        Add point to prototype set
        Increment Number of class points in
          prototype set
    end
    if Total Number of points in prototype
      set = M then
        return Prototype Set
    end
end
```

3 Experimental Results

In order to evaluate the algorithms performance, it was tested on the MNIST digit dataset. The experiment involved feeding the algorithm different values of M and comparing it to uniform random selection of M points from the dataset. In order to properly evaluate the performance of uniform random selection a confidence interval of 95% was calculated. First, random selection for a given value of M was ran 10 times in order to get 10 different values of the prediction accuracy. The mean was then calculated using Equation 1.

$$\bar{Acc} = \frac{Acc_1 + Acc_2 + ... + Acc_{10}}{10} \tag{1}$$

Following the calculation of the mean, the standard deviation of the 10 experiments was calculated using Equation 2.

$$\sigma = \sqrt{\frac{\sum (Acc_i - \bar{Acc})^2}{n}} \tag{2}$$

Finally, a confidence interval of 95% was calculated using Equation 3 and the calculated mean and standard deviation.

$$CI = \bar{Acc} \pm \frac{(1.96 \times \sigma)}{\sqrt{n}} \tag{3}$$

The results on the MNIST test data for uniform random selection are presented in the following table.

M	**Accuracy (95% confidence)**
10	0.3531 ± 0.0303
20	0.4548 ± 0.0319
50	0.6168 ± 0.0188
100	0.7182 ± 0.0101
200	0.7894 ± 0.0083
500	0.8471 ± 0.004
1000	0.8844 ± 0.0022
2000	0.9128 ± 0.0011
5000	0.9364 ± 0.0015
10000	0.9486 ± 0.0008
20000	0.9578 ± 0.0007

As for the prototype selection algorithm, it is completely deterministic. Therefore the experiment for the algorithm is only run once per value of M, and there is no need for confidence intervals. The results on the MNIST test data for the algorithm are presented in the following table.

M	**Accuracy**
10	0.4393
20	0.5186
50	0.6909
100	0.7511
200	0.7955
500	0.8539
1000	0.8915
2000	0.9144
5000	0.9377
10000	0.9508
20000	0.9592

4 Critical Evaluation

As seen in the tables provided in the previous section, the proposed prototype selection algorithm manages to outperform the uniform random selection in a 95% confidence interval for almost all values of M.

4.1 Small M

For small values of M, there are multiple reasons as to why it outperforms uniform random selection drastically. One probable reason is that the proposed algorithm guarantees a balanced training set, whereas random selection might leave classes out. Furthermore, the algorithm chooses the points based on the intuition that the data points closest to the class's average are good representations of that class. Random selection, on the other hand, might very well choose images that we would consider to be outliers of the class.

4.2 Larger M

As for larger values of M, we find that the difference in performance is very slight. This is because as M increases, it becomes increasingly likely that we have a relatively even distribution in our set of prototypes. Furthermore, the random selection will sample more data points that are representative of the classes, combating the effect of outliers.

However, the presence of outliers is still possible for random selection, whereas the proposed algorithm will always add the closest values to the average. This could explain the slight improvement shown in the results.

4.3 Further Improvement

While the results show that the proposed selection algorithm performs well compared to random selection, there is always room for improvement. The algorithm is based on the assumption that the points closest to the average of the class are the best representation for that class. This is a somewhat rigid way to think of the classification problem. It can be argued that these points are ’easy’ examples and do not paint the picture of the whole class. Perhaps a future improvement would be to incorporate the ’difficult’ data points that lie further away from the average. A better algorithm would strike a balance in the prototype set between ’easy’ points and ’difficult’ points, allowing for a more general representation of the classes within the prototype set, and coverage of more data points within every class. However, extra care would also need to be taken so as not to plague the prototype set with outliers.

In [59]:
```
import idx2numpy
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import random
from collections import defaultdict
```

In [5]:
```
X_train = idx2numpy.convert_from_file('./train-images-idx3-ubyte')
Y_train = idx2numpy.convert_from_file('./train-labels-idx1-ubyte')
print(X_train.shape)
print(Y_train.shape)

X_test = idx2numpy.convert_from_file('./t10k-images-idx3-ubyte')
Y_test = idx2numpy.convert_from_file('./t10k-labels-idx1-ubyte')
print(X_test.shape)
print(Y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

In [10]:
```
X_train = X_train.reshape(60000, 28*28)
X_test = X_test.reshape(10000, 28*28)
```

Using entire Training Set

In [11]:
```
nn_classifier = KNeighborsClassifier(n_neighbors=1)
nn_classifier.fit(X_train, Y_train)
```

Out[11]:
```
KNeighborsClassifier(n_neighbors=1)
```

In [15]:
```
def compute_accuracy(preds, ys):
    correct = [pred == y for pred,y in zip(preds, ys)]
    return sum(correct)/len(correct)
```

In [17]:
```
test_preds = nn_classifier.predict(X_test)
print("Test Accuracy: " , compute_accuracy(test_preds, Y_test))
```

```
Test Accuracy:  0.9691
```

Random Prototype Selection

In [240…
```
def random_prototype_selection(train_images, train_labels, M=1000):
    sample_indices = random.sample(range(len(train_images)), M)
    sample_images = train_images[sample_indices]
    sample_labels = train_labels[sample_indices]
    return sample_images, sample_labels
```

In [241…

```
import math
```

In [313…

```
def perform_experiment(train_images, train_labels, M=1000):
    accuracies = []
    for i in range(10):
        X_train_M, Y_train_M = random_prototype_selection(X_train, Y_train, M)
        nn_M = KNeighborsClassifier(n_neighbors=1)
        nn_M.fit(X_train_M, Y_train_M)
        test_preds = nn_M.predict(X_test)
        acc = compute_accuracy(test_preds, Y_test)
        accuracies.append(acc)
    mean = sum(accuracies)/len(accuracies)
    std_dev = math.sqrt(sum([(acc - mean)**2 for acc in accuracies])
                        /(len(accuracies)))
    ci = 1.96*std_dev/math.sqrt(len(accuracies))
    return mean, ci
```

Prototype Selection Algorithm

In [314…

```
def compute_distance(a,b):
    return np.linalg.norm(a-b)
```

In [315…

```
def prototype_selection(train_images, train_labels, M):
    average_dict = {}
    label_image_dict = defaultdict(list)
    for j in range(len(train_images)):
        label = train_labels[j]
        label_image_dict[label].append(train_images[j])

    for label in label_image_dict:
        average_dict[label] = np.average(label_image_dict[label], axis=0)

    distances = []
    for j in range(len(train_images)):
        label = train_labels[j]
        average = average_dict[label]
        dist = compute_distance(average, train_images[j])
        distances.append(dist)

    sorted_indices = np.array(distances).argsort()
    sorted_distances = np.array(distances)[sorted_indices]
    sorted_images = train_images[sorted_indices]
    sorted_labels = train_labels[sorted_indices]

    cnt_per_class = defaultdict(int)
    total_cnt = 0
    return_images = []
    return_labels = []
    for i in sorted_indices:
        label = sorted_labels[i]
        if cnt_per_class[label] < M//10:
            return_images.append(sorted_images[i])
            return_labels.append(sorted_labels[i])
            cnt_per_class[label] += 1
            total_cnt+=1
        if total_cnt == M:
            break

    return return_images, return_labels
```

Accuracy Comparison

In [316…

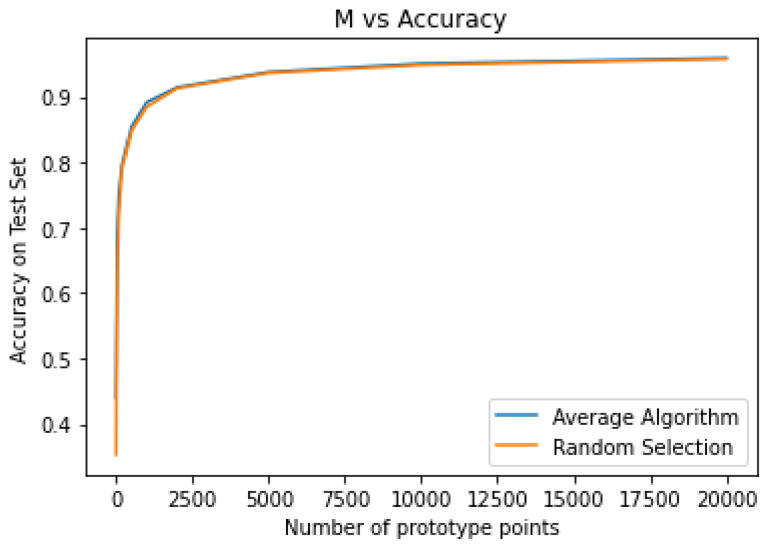
```
import matplotlib.pyplot as plt
```

In [322…

```
M_values = [10,20,50,100,200,500,1000,2000,5000,10000,20000]
accuracies = []
random_accuracies=[]
confidence_intervals = []
for M in M_values:
    new_X_train, new_Y_train = prototype_selection(X_train, Y_train, M)
    nn_prototyped = KNeighborsClassifier(n_neighbors=1)
    nn_prototyped.fit(new_X_train, new_Y_train)
    test_preds = nn_prototyped.predict(X_test)
    accuracy_on_M = compute_accuracy(test_preds, Y_test)
    rand_acc, ci = perform_experiment(X_train, Y_train, M)
    accuracies.append(accuracy_on_M)
    confidence_intervals.append(ci)
    random_accuracies.append(rand_acc)
```

In [318…

```
plt.plot(M_values, accuracies, label = 'Average Algorithm')
plt.plot(M_values, random_accuracies, label='Random Selection')
plt.legend()
plt.xlabel('Number of prototype points')
plt.ylabel('Accuracy on Test Set')
plt.title('M vs Accuracy')
plt.show()
```



In [319…

```
print(random_accuracies, confidence_intervals)
```

```
[0.35307, 0.45481999999999995, 0.6168, 0.7181899999999999, 0.78936, 0.84708,
0.8843799999999999, 0.91275, 0.93635, 0.94858, 0.9578200000000001] [0.03033059
374849097, 0.031875443301952675, 0.018833386966767295, 0.010108639907722494,
0.00825177225473412, 0.003967322756721454, 0.002228167447926651, 0.00105412655
7866742, 0.0014733282322686944, 0.0008392828939040839, 0.0006680121675538576]
```

In [320…

```
print(accuracies)
```

```
[0.4393, 0.5186, 0.6909, 0.7511, 0.7955, 0.8539, 0.8915, 0.9144, 0.9377, 0.950
8, 0.9592]
```

In [ ]: