Prototype Selection for Nearest Neighbor

Nidhi Dhamnani

A59012902
CSE 251A: ML - Learning Algorithms
ndhamnani@ucsd.edu

Keywords: Prototyping, 1-NN, K-means

Abstract. The goal of this project is to propose an algorithm to select a subset of training data points called 'prototypes' to improve the performance of nearest neighbours.

[1] Motivation

Nearest neighbors algorithm (1-NN) is a non-parametric classification method where the output is a class membership. In 1-NN the object is assigned the same class as that of it's nearest neighbor. The algorithm relies on distance metric for classification.

Some of the drawbacks of Nearest Neighbour are:

- Requires high memory due to the need to store all of the training data
- Given that it stores all of the training data, it can be computationally expensive
- Sensitive to the scale of the data and noise
- With large data, the prediction stage might be slow

To overcome some of these challenges, we need to do data reduction which is a technique used for working with huge data sets. Usually, only some of the data points are needed for accurate classification. Those data are called the prototypes and can be found as follows:

- Select the class-outliers, that is, training data that are classified incorrectly by 1-NN
- Separate the rest of the data into two sets: (i) the prototypes that are used for the classification decisions and (ii) the absorbed points that can be correctly classified by 1-NN using prototypes. The absorbed points can then be removed from the training set

In this project, I have proposed one of the approaches to do prototyping for 1-NN.

[2] Dataset

The MNIST database is a large dataset consisting of handwritten digits. It has a training set of 60,000 samples and a test set of 10,000 samples. It is a subset of a larger set available from NIST. The data is size-normalized and centered in a fixed-size image. The original black and white images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

[3] Description

The proposed prototype algorithm is based on k-means clustering. K-means clustering is a method for finding clusters and cluster centers in a set of unlabeled data. It aims to partition training data into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. The initial cluster centers are randomly selected. I used euclidean distance as a measure of closeness between two data points for both k-means and 1-NN algorithm.

Prototype Algorithm

The data is pre-processed to convert the image into meaningful features. Each image of size 28*28 pixels is converted into a flattened array of size 784 features for both train and test samples. These feature vectors are used to train the NN model. The steps followed in prototyping algorithm are:

- Segregated the data for each class label to ensure we take equal number of data points from each class in the prototype.
- For datapoints in each of the class label
  - Trained the k-means classifier where the number of clusters is equal to the total number of points in prototype (M) divided by the number of unique classes in the dataset.
  - Calculated the nearest point to the centroid of each cluster in the trained k-means and added the nearest point in the prototype dataset
  - Therefore, we will have M/C (M=Size of prototype, C=Number of class labels) points from each class. Each of the selected data point is closest to one of the M/C centroids of the trained k-means algorithm
- Trained the nearest neighbour classifier on this newly created prototyped dataset

Pseudo Code

Require: M (Number of prototype points), X_train (Training data), y_train (Training labels), T (Number of training samples)

// Initializing Data Structures
$C = Set(y\_train)$
$N = Len(C)$
$points\_per\_class = int(M/N)$
$prototyped\_X\_train = []$
$prototyped\_y\_train = []$
$class\_labelled\_data = \{\}$ //dictionary having key (k) as class label and value as list of data points
//having k as the label
for $i\ in\ \{0, 1, ...(T-1)\}$ **do**
  $class\_labelled\_data[y\_train[i]].append(X\_train[i])$
end for
for $class\_label\ in\ C$ **do**
  $train\_data = class\_labelled\_data[class\_label]$
  $kmeans = KMeans(n\_clusters = points\_per\_class).fit(train\_data)$
  $closest\_points = get\_closest\_points(kmeans.cluster\_centers, data)$ // List of data points
  // where i^{th} datapoint is the closest point to i^{th} cluster centroid. Len(closest_points) = M/N

for $cp\ in\ closest\_points$ **do**
$prototyped\_X\_train.append(cp)$
$prototyped\_y\_train.append(class\_label)$
end for
end for
$nn\_prototype = KNeighborsClassifier(n\_neighbors = 1)$
$nn\_prototype = nn\_prototype.fit(prototyped\_X\_train, prototyped\_y\_train)$
return $prototyped\_X\_train, prototyped\_y\_train$

Proof of Correctness

The proposed algorithm is based on k-means which is an unsupervised learning algorithm. K-means was chosen to select prototype because of it's property to find clusters of points which are close to each other. The assumption is that the points which are close to each other forming a cluster have similar features and therefore, we can replace the cluster with just a single point. This single point selection is based on nearest distance from the cluster centroid. Conversely, by not considering the points far from centroid, we remove the datapoints which are near the boundary of the cluster. Hence, we remove the points which are most likely to be misclassified by 1-NN. Therefore, the proposed model satisfies the conditions described in [Section 1] to design a prototype model.

Random Selection Algorithm

The steps followed to choose the random datapoints are as follows:

- Created a set called $random\_indices\_set$ which stores the indices of points to be included in the training set
- While the number of elements in $random\_indices\_set$ are less than M
  - Randomly select an integer in the range [0, 60000) and add it to $random\_indices\_set$
- For each v in $random\_indices\_set$, add X_train[v] into the randomly created training dataset

For each M, performed the random selection 25 times to obtain better randomly distributed points and get a better estimate of the performance.

[5] Experimental Results

The experiments are performed over different values of M ranging from 1000 to 12500.

As shown in Fig. 1, the proposed prototype model beats the random selection in terms of accuracy for all the values of M. The results for prototype model are based on two iterations with random initializations for k-means cluster centroids. I couldn't run prototype model for more iterations due to the amount of time taken in training k-means. However, the results for random model are averaged over 25 iterations for each M.

Fig. 2 shows the confidence interval for random sampling.

The formula used for calculating confidence interval is:

$CI = accuracy\_mean \pm (confidence\_level * accuracy\_std\_deviation)/sqrt(n\_samples)$

Here for each M, $n\_samples$ is 25, $accuracy\_mean$ is the mean of accuracies of 25 iterations, $confidence\_level$ is 95% and $accuracy\_std\_deviation$ is the standard deviation of accuracies of 25 iterations.

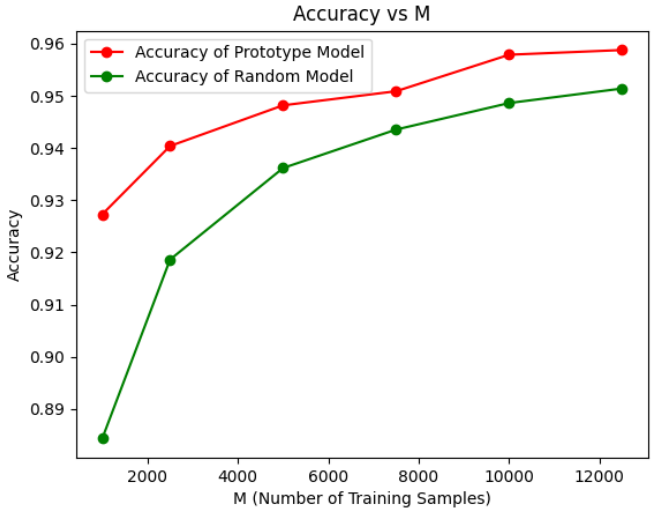

Fig. 1: Accuracy vs No. of Training Samples in Prototype

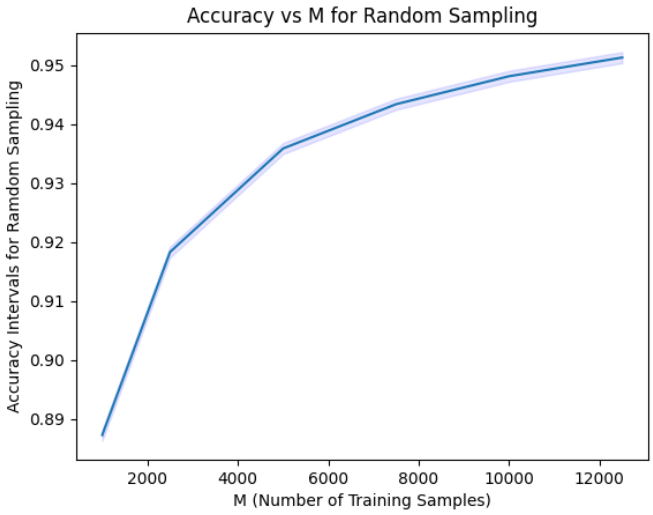

Fig. 2: Confidence Interval vs No. of Training Samples for Random Selection

[6] Conclusions

Prototyping is a good approach to reduce the training data size which in-turn reduces the test time for 1-NN. However, the proposed model comes with a couple of tradeoffs. Due to the high dimensions, most of the features are 0 and the data is sparse. As we know k-means suffers from the curse of dimensionality, therefore, there is a lot of scope for improvement.

In terms of future work, we can perform dimentionality reduction on feature set and test the performance. We can also evaluate how will the model perform in case of imbalanced dataset where we do not or can not sample equal data points from each class. In this case, we can try an approach to take the points closest to the centroids ignoring the class balancing. It would be interesting to see how the model performs in such cases. We can also try and look for an approach which takes relatively less time for prototype selection. K-means suffers from the similar problems as that of K-NN and therefore, the proposed prototype selection algorithm is computationally intensive for a large dataset.

References

1. http://yann.lecun.com/exdb/mnist/
2. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
3. https://dl.acm.org/doi/abs/10.1145/3033288.3033301
4. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
6. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise_distances_argmin_min.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html
8. https://www.stat.auckland.ac.nz/~yee/784/files/ch13PrototypeMethods.pdf
9. https://en.wikipedia.org/wiki/K-means_clustering
10. https://en.wikipedia.org/wiki/Confidence_interval

project1_code

January 28, 2022

[10]:
```
from keras.datasets import mnist
from matplotlib import pyplot
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪accuracy_score
from sklearn.cluster import KMeans
import numpy as np
import random
from sklearn.utils import shuffle
from sklearn.metrics import pairwise_distances_argmin_min
import matplotlib.pyplot as plt
```

[2]:
```
(train_X, train_y), (test_X, test_y) = mnist.load_data()
```

[3]:
```
train_X_flatten = []
for row in train_X:
    train_X_flatten.append(row.flatten())

test_X_flatten = []
for row in test_X:
    test_X_flatten.append(row.flatten())
```

[4]:
```
def separate_data(X, y):
    labelled_data = {key: [] for key in range(10)}
    for i in range(len(X)):
        labelled_data[y[i]].append(X[i])
    return labelled_data
```

[5]:
```
labelled_data_train = separate_data(train_X_flatten, train_y)
```

[6]:
```
M_values = [1000, 2500, 5000, 7500, 10000, 12500]
```

[8]:
```
accuracies_proto = []
for M in M_values:
    points_per_class = int(M/10)
    prototyped_X_train = []
    prototyped_y_train = []
```

```
    for i in range(10):
        data = labelled_data_train[i]
        kmeans = KMeans(n_clusters=points_per_class).fit(data)
        closest, _ = pairwise_distances_argmin_min(kmeans.cluster_centers_, data)
        for j in closest:
            prototyped_X_train.append(data[j])
            prototyped_y_train.append(i)

    prototyped_X_train, prototyped_y_train = shuffle(prototyped_X_train,␣
↪prototyped_y_train)
    print(len(prototyped_X_train), len(prototyped_y_train))
    nn_prototype = KNeighborsClassifier(n_neighbors=1).fit(prototyped_X_train,␣
↪prototyped_y_train)
    y_pred = nn_prototype.predict(test_X_flatten)
    acc = accuracy_score(test_y, y_pred)
    print('Accuracy:', acc)
    accuracies_proto.append(acc)
```

```
1000 1000
Accuracy: 0.9263
2500 2500
Accuracy: 0.9418
5000 5000
Accuracy: 0.9502
7500 7500
Accuracy: 0.9517
10000 10000
Accuracy: 0.9558
12500 12500
Accuracy: 0.9557
```

[9]:
```
accuracies_random = []

for M in M_values:
    accuracy_random = 0
    num_trials = 25
    for i in range(num_trials):
        random_training_index = set()
        while (len(random_training_index)<M):
            random_training_index.add(random.randint(0, 60000-1))

        random_X_train = []
        random_y_train = []
        for index in random_training_index:
            random_X_train.append(train_X_flatten[index])
            random_y_train.append(train_y[index])
```

```
        nn_random = KNeighborsClassifier(n_neighbors=1).fit(random_X_train,␣
↪random_y_train)
        y_pred = nn_random.predict(test_X_flatten)
        a_r = accuracy_score(test_y, y_pred)
        accuracy_random += a_r

    print('Accuracy:', accuracy_random/num_trials)
    accuracies_random.append(accuracy_random/num_trials)
```

```
Accuracy: 0.8844319999999999
Accuracy: 0.917268
Accuracy: 0.9353239999999999
Accuracy: 0.9438280000000002
Accuracy: 0.9484239999999999
Accuracy: 0.9513079999999998
```

[12]:
```
plt.plot(M_values, accuracies_proto, color='r', label='Accuracy of Prototype␣
↪Model', marker = 'o')
plt.plot(M_values, accuracies_random, color='g', label='Accuracy of Random␣
↪Model', marker = 'o')
plt.xlabel('M (Number of Training Samples)')
plt.ylabel('Accuracy')
plt.title('Accuracy vs M')
plt.legend()
plt.show()
```