Prototype Selection for Nearest Neighbor

Seyedeh Baharan Khatami

Abstract

One way to speed up nearest neighbor classification is to replace the training set by a carefully chosen subset of “prototypes”. In this project, we aim to explain and implement such techniques, and verify its accuracy and runtime compared to standard nearest neighbor and random selection prototype. Then, we discuss some other ideas that can be explored in the future.

1 Credits

This document has been adapted by Roberto Navigli from the instructions for earlier ACL, NAACL and EMNLP proceedings, including those for EMNLP 2020 by Yulan He, ACL 2020 by Steven Bethard, Ryan Cotterell and Rui Yan, ACL 2019 by Douwe Kiela and Ivan Vulić, NAACL 2019 by Stephanie Lukin and Alla Roskovskaya, ACL 2018 by Shay Cohen, Kevin Gimpel, and Wei Lu, NAACL 2018 by Margaret Michell and Stephanie Lukin, 2017/2018 (NA)ACL bibtex suggestions from Jason Eisner, ACL 2017 by Dan Gildea and Min-Yen Kan, NAACL 2017 by Margaret Mitchell, ACL 2012 by Maggie Li and Michael White, ACL 2010 by Jing-Shing Chang and Philipp Koehn, ACL 2008 by Johanna D. Moore, Simone Teufel, James Allan, and Sadaoki Furui, ACL 2005 by Hwee Tou Ng and Kemal Oflazer, ACL 2002 by Eugene Charniak and Dekang Lin, and earlier ACL and EACL formats written by several people, including John Chen, Henry S. Thompson and Donald Walker. Additional elements were taken from the formatting instructions of the *International Joint Conference on Artificial Intelligence* and the *Conference on Computer Vision and Pattern Recognition*.

2 Introduction

The nearest neighbors is a simple approximation algorithm that classifies new data points based on previous stored data points given a distance metric. The training process is fast, and it involves storing training samples. The inference part, however, is slow. Given a new data point x, it computes the distance between x and all the previous data points. This algorithm uses a hyper parameter k, which is the number of nearest neighbors to consider. It chooses the k nearest neighbors based on the calculated distances, and take a majority voting among the labels of these k nearest neighbors to predict label of x.

For the distance metric, we can use different types of distance metrics that can be defined on the input space. Some well-known distance functions are euclidean distance, manhattan distance, minkowski distance, cosine distance, etc. There are also several techniques to use to speedup nearest neighbors, such as using some data structures such as locality sensitive hashing, ball trees, and k-d trees. Another group of techniques is to select a subset of training data, which can be a proper representative of the whole data, and use that subset in inference.

3 Methodology

In this chapter, we describe two approaches based on clustering.

3.1 Prototype selection idea

There are several clustering algorithms that can be used to cluster data. I used Kmeans in two different ways. The first one is to use Kmeans to cluster the whole training data into m clusters. Then, we can use the centroids of clusters as the prototypes since these points are the centers of clusters and can be a good representative of their cluster. The second approach is to apply Kmeans to each subgroup of data. In the MNIST dataset, each subgroup of data includes the images with the same label, e.g hand-written images of zero. Clustering each subgroup into $\frac{m}{numberoflabels}$ clusters, and aggregate all the centroids in all subgroups to form the prototype.

3.2 Psuedocode

This section is about the high-level psuedocode of the above algorithms. First, we discuss the psuedocode of KNN and Kmeans. Then, we use them as a subroutine for our algorithms.

Algorithm 1 KNN

```
Require: $TrainingData, TrainingLabels, k, x$

  $Distances = dist(TrainingData, x)$
  $sort(Distances)$
  $Labels = TrainLabels[top_k(Distances)]$
  $Label = majority(Labels)$
```

Algorithm 2 Kmeans

```
Require: $X, k$
  $Centroids = random(X, k)$
  while $Centroids\ change$ do
    for all $x_i\ in\ X$ do
      $Distances_{x_i} = dist(x_i, Centroids)$
      $Center_{x_i} = min(Distances_{x_i})$
    end for
    $Update(Centroids)$
  end while
```

Algorithm 3 Prototype with Kmeans 1

```
Require: $TrainingData, TrainingLabels, m, x$

  $Prototypes = Kmeans(TrainingData, m)$
  $Label = KNN(Prototypes, ProtLabels, 1, x)$
```

In the two algorithms "Prototype with Kmeans 1" and "Prototype with Kmeans 2", I combined prototype selection and inference for x using the prototypes. For the prototype selection with no inference, last line of the code and input x can be removed.

4 Experimental Results

The MNIST dataset is used in this project, which contains images of handwritten digits. Training data contains 60000 images and test data contains 10000 images. The size of images are 28*28 pixels. Figure1 illustrates few instances of the input domain.

Algorithm 4 Prototype with Kmeans 2

```
Require: $TrainingData, TrainingLabels, m, x$

  for all $Label\ in\ Labels$ do
    $Prototypes = \bigcup_{Label} Kmeans(TrainingData_{Label}, \frac{m}{|Labels|})$
  end for
  $Label = KNN(Prototypes, ProtLabels, 1, x)$
```

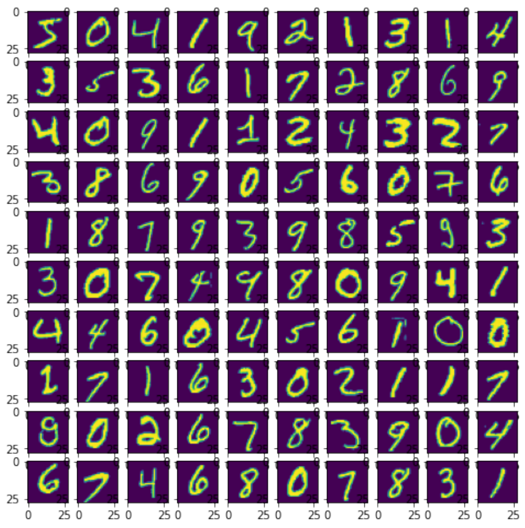
Figure 1: Input images in MNIST dataset

I ran the "Prototype with Kmeans 2" algorithm, and compared its accuracy with standard nearest neighbor and random prototype selection for $m = 1000, 5000, 10000$. The results are depicted in Table1. The random selection prototype procedure was repeated 10 times for each m. In Table 1, the average accuracies and confidence intervals are included for random prototype selection.

Given m the mean value, s the sample standard deviation and N the sample size, the confidence interval is defined by the following formula:

$$(m - t\frac{s}{\sqrt{N}}, m + t\frac{s}{\sqrt{N}})$$

There's a t parameter, as you can see, which is related to the confidence we want. The calculation of this parameter can be done in different ways. Since our sample size is small (i.e. less than 30 points), we can use Student's t distribution to calculate it. I calculate the confidence intervals using scipy.stats package.

Method	**m**	**Accuracy**	**Confidence Interval**
Standard NN	-	96.91%	-
Kmeans	1000	95.88%	-
Random	1000	88.42%	(0.8807, 0.8876)
Kmeans	5000	96.69%	-
Random	5000	93.65%	(0.9349, 0.9379)
Kmeans	10000	96.98%	-
Random	10000	94.88%	(0.9478, 0.9497)

Table 1: Performance comparison of different approaches with different size of prototype set

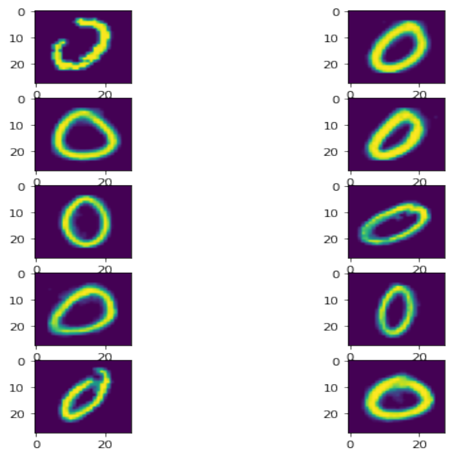

Figure 2: Instances of Kmeans centroids for zero class

Also to get some intuition of how do these centroids look like, I plotted some of the centroids for label zero in Figure2. As you can see, It consists of different shapes of zero with different curvatures and pen lengths, suggesting that the images were clustered properly, and the centroids differs in features, and can be a good representative of the zero class.

5 Critical Evaluation and Future Works

Yes, the method is a clear improvement over random setting, especially when the size of the prototype is small. In larger prototypes like $m = 10000$, since it contains a sixth of the data, it will probably contain samples that are good enough to represent the whole data. Another Solution that comes to my mind is that we can randomly pick a point from training data, calculate its distance from prototype points, which is initially an empty set, and find the closest point in prototype to our random point. If the label of the random point was not the same as its closest neighbor in prototype, we can add this random point to our prototype set. Otherwise, we can skip this point. We can repeat this process until m points are added to the prototype set.