Use K-Means for prototype selection

Shuo Xu
Department of Computer Science and Engineering
University of California San Diego
s3xu@ucsd.edu

Abstract

In this project, we introduced a prototype selection strategy using k-means algorithm. We firstly use PCA to decrease the dimensionality of the dataset. The algorithm picks the same number of k-centroids for each class, all of which are combined to get the prototype. The schema overnumbers the random selection baseline by 5.5% over the MNIST dataset.

1 Idea Description

In my prototype selection method, the data is firstly decomposed in dimension by PCA algorithm. Then K-means is applied to each class, respectively. Finally, the k-centroids from each class are assigned with their original labels and are combined as prototypes of the whole dataset.

2 pseudocode

Algorithm 1 Prototype selection algorithm

```
1:  procedure PROTOTYPESELECTION(M, train_dataset)
2:      $x\_prototypes \leftarrow [\,]$
3:      $y\_prototypes \leftarrow [\,]$
4:      $i \leftarrow 0$
5:      $k = M/10$
6:      for i < 10 do
7:          k_centroids = KMeans(k, data of class $i$ in train_dataset)
8:          $x\_prototypes \leftarrow x\_prototypes + k\_centroids$
9:          $y\_prototypes \leftarrow y\_prototypes + k\ copies\ of\ i$
10:         $i \leftarrow i + 1.$
        return $x\_prototypes, y\_prototypes$
```

3 Experimental Result

3.1 Configuration

The experiment is carried out on the complete MNIST dataset with 50000 and 10000 for training and testing, separately.

PCA algorithm is applied over raw dataset before prototype selection. According to the performance on validation dataset, I choosed 25 as the number of principle components, which not only preserved enough information to achieve satisfying accuracy in prototype selection, but also greatly shortened the calculation time.

In the k-means algorithm, each class contributes the same amount of prototypes to the final result. This is implemented by distributing total number of prototype, i.e. M, to each of the 10 classes, which means $k = M/10$.

To reduce the negative effect of randomness, for each value of M, 10 repeated train-and-test cycles are applied over a certain type algorithm (either baseline or my algorithm). The averaged accuracy as well as the standard deviation (std) over test set are reported.

3.2 Performance

The performance of baseline and k-means algorithm are compared in this section.

Table 1: Averaged accuracy on test dataset

M	1000	2500	5000	7500	10000	15000	20000	25000	30000	35000	40000	45000	50000
Baseline	0.9002	0.9272	0.9416	0.9481	0.9535	0.9581	0.9613	0.9633	0.9656	0.966	0.9681	0.9676	0.9692
K-Means	0.952	0.9617	0.9648	0.9672	0.9668	0.9675	0.9684	0.9681	0.9692	0.9686	0.9695	0.9694	0.9702

Table 2: Standard deviation(std) on test dataset

M	1000	2500	5000	7500	10000	15000	20000	25000	30000	35000	40000	45000	50000
Baseline	0.0026	0.0018	0.0022	0.0013	0.001	0.0011	0.0006	0.0016	0.0014	0.0011	0.0007	0.0011	0.0009
K-Means	0.0	2.2e-16	1.1e-16	0.0	0.0	1.1e-16	0.0	1.1e-16	0.0	1.1e-16	0.0	1.1e-16	1.1e-16

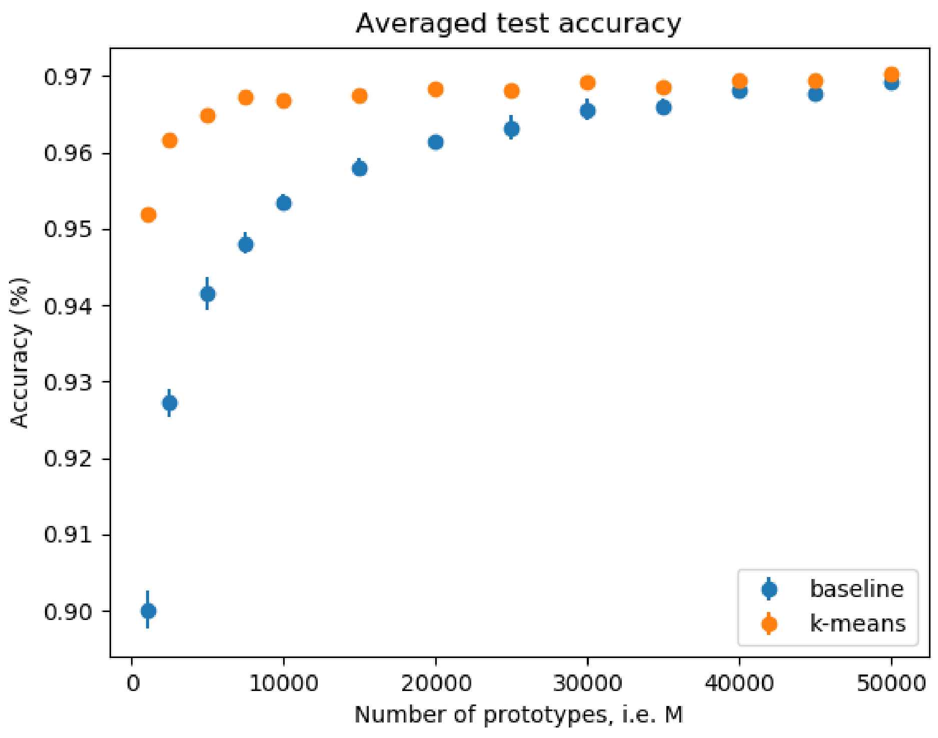

Figure 1: The averaged accuracy over different number of prototypes

4 Critical evaluation

From the table and figure in section 3, it's obvious k-means method always perform better than over random selection. This advantage is clearly obvious when $M \in [0, 25000]$. However, the superiority decreases as M increases, which suggest the performance of random selection is improving as more prototype are selected.

Some possible further scope of this method aims to be more robust to boundary case, where there may be some overlap between classes. In my algorithm, the method is select k-centroids in each class, without competing with centroids of other classes. And each class is equally regarded in terms of their contribution to the final prototypes set, this might not be wise because not all classes needs equal prototype to distinguish itself from the others. Intuitively speaking, the ones on the "outer side" of dataset space would have less possible overlap with other classes, and thus needs less prototype to seperate themselves from the rest. But for those locate in the midst of the dataset space, these classes need to be given more prototypes so as to sever them from the others.

The method I may want to try out is described as below. The class of each centroid is assigned as such - count all the number of samples belongs to the centroid, the class with most samples is assigned as the centroid's class.

Algorithm 2 Improved prototype selection algorithm

```
1: procedure ImprovedPrototypeSelection(M, train_dataset)
2:     M_centroids = KMeans(M, data of class $i$ in train_dataset)
3:     $y\_prototypes \leftarrow [\ ]$
4:     $i \leftarrow 0$.
5:     for i < M do
6:         $label \leftarrow the\ majority\ class\ in\ M\_centroids_i$
7:         $y\_prototypes_i \leftarrow label$
8:         $i \leftarrow i + 1$.
       return $M\_centroids, y\_prototypes$
```

References

[1] Jia Li. Prototype Methods: K-Means, *STAT557: Data Mining*, Department of Statistics, The Pennsylvania State University.

Appendix Python Implementation

```
from keras.datasets import mnist
from collections import defaultdict

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from multiprocessing import Pool, TimeoutError

def convert2dict(x_dataset, y_dataset):
    datadict = defaultdict(list)
    for x, y in zip(x_dataset, y_dataset):
        datadict[y].append(x)
    for key in datadict:
        datadict[key] = np.array(datadict[key])
    return datadict

def load_mnist():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.reshape((x_train.shape[0], -1))
    x_test = x_test.reshape((x_test.shape[0], -1))
    return x_train, y_train, x_test, y_test

x_train, y_train, x_test, y_test = load_mnist()

def visualize(pixels, label):
    pixels = np.array(pixels, dtype='uint8')

    # Reshape the array into 28 x 28 array (2-dimensional array)
    pixels = pixels.reshape((28, 28))

    # Plot
    plt.title('Label is {label}'.format(label=label))
    plt.imshow(pixels, cmap='gray')
    plt.show()

def test(x_prototype, y_prototype, x_test, y_test):
    one_nn = KNeighborsClassifier(n_neighbors=1)
    one_nn.fit(x_prototype, y_prototype)
    return one_nn.score(x_test, y_test)

def baseline(M, x_train, y_train, x_test, y_test):
    rand_idx = np.random.permutation(len(x_train))[:M]
    x_prototype = x_train[rand_idx]
    y_prototype = y_train[rand_idx]
    return x_prototype, y_prototype

def schema(R, x_train, y_train):

    # K-means
    train_dic = convert2dict(x_train, y_train)
    prototypes, labels = [], []
    for digit in train_dic:
        print(1)
        kmeans = KMeans(n_clusters=R, random_state=0).fit(train_dic[
digit])
        prototypes.append(kmeans.cluster_centers_)
        labels.extend([digit] * R)
    return np.vstack(prototypes), np.array(labels)

def train_and_test(M):
```

```python
    baseline_result = defaultdict(list)
    ps_result = defaultdict(list)
    for _ in range(10):
        R = M // 10

        x_prototype_base, y_prototype_base = baseline(M, x_train, y_train, x_test, y_test)
        accuracy_base = test(x_prototype_base, y_prototype_base, x_test, y_test)
        baseline_result[M].append(accuracy_base)
        print("Base accuracy = {}".format(accuracy_base))

        x_prototype_schema1, y_prototype_schema1 = schema(R, x_train, y_train)
        accuracy_schema= test(x_prototype_schema1, y_prototype_schema1, x_test, y_test)
        ps_result[M].append(accuracy_schema)
        print("Schema accuracy = {}".format(accuracy_schema))

    np.savez('results-{}'.format(M), baseline_result=baseline_result, ps_result=ps_result)

# Perform PCA
pca = PCA(n_components=25)
pca.fit(x_train)
x_train = pca.transform(x_train)
x_test = pca.transform(x_test)

Ms = [1000, 2500, 5000, 7500, 10000, 15000, 25000, 35000, 45000, 50000]

if __name__ == '__main__':
    pool = Pool(processes=6)
    pool.map(train_and_test, Ms)
```