Prototype Selection Method for Nearest Neighbors Algorithm

Chunlin Chen
Department of Electrical and Computer Engineering
University of California, San Diego
chc126@ucsd.edu

Abstract

One major limitation of the nearest neighbors algorithm (NN) lies in its data inefficiency and computational intensity, especially when it comes across large training sets. Therefore, it is of great significance to select a small but representative subset from the original training set to form a prototype that can be utilized for nearest neighbor classification, without loss of classification accuracy. In this paper, we reviewed and implemented several classical prototype selection methods, including k-Means clustering, condensed nearest neighbors (CNN), edited nearest neighbors (ENN), and tested and compared their performances on the `MNIST` dataset.

1 High-level Description of Prototype Selection Methods

We choose three different methods to select prototypes. k-Means clustering(MacQueen, 1967) partitions the training set into clusters and gather the centroids of clusters as the prototype. Condensed nearest neighbors (CNN)(Hart, 1968) maintains a store set by proceeding through the training set and dropping redundant samples. Edited nearest neighbors (ENN)(Wilson, 1972) cleans the training set by removing samples close to the decision boundary, it removes observations from classes when any or most of its closest neighbors are from a different class. The baseline method is just randomly selecting a subset of the training set as the prototype.

2 Pseudocode of Selected Methods

Let T be the training set and M be the desired cardinality of the prototype set.

Algorithm 1 Prototype selection based on k-Means clustering

Initialization: M, prototype $\leftarrow []$
for all label $y_i \in \{0, 1, ..., 9\}$ **do**
  `KMeans`(# of clusters $\leftarrow \frac{M}{10}$)
  `KMeans.fit`($\{x \mid \text{label}(x) = y_i\}$)
  centroids $\leftarrow$ `KMeans.cluster_centers_`
  $\text{prototype}_i \leftarrow \underset{\{x|\text{label}(x)=y_i\}}{argmin} \|\text{centroids} - x\|$
  prototype.append(prototype_i)
end for

Algorithm 2 Edited Nearest Neighbors

Initialization: M, prototype $\leftarrow []$
while |prototype| $< M$ **do**
  `shuffle`(T)
  for all $x_i \in T$ **do**
    $x_{nn} = \underset{x \in T \setminus x_i}{argmin} \|x_i - x\|$
    if label(x_{nn}) = label(x) **then**
      prototype.append(x_{nn})
    end if
  end for
end while

3 Experimental Results

We tested our selected methods on the `MNIST` dataset at $M = 500, 1000, 2000, 5000, 10000$. For

Algorithm 3 Condensed Nearest Neighbors

Initialization: M, prototype $\leftarrow []$, grabbag $\leftarrow []$
while |prototype| $< M$ **do**
  shuffle(T)
  prototype.append(x_0)
  for all $x_i \in T, i = 1, ..., n$ **do**
    $x_{nn} = \underset{x \in \text{prototype}}{argmin} \|x_i - x\|$
    if label(x_{nn}) $\neq$ label(x) **then**
      prototype.append(x_{nn})
    else
      grabbag.append(x_{nn})
    end if
  end for
  for all $x_i \in$ grabbag **do**
    $x_{nn} = \underset{x \in \text{prototype}}{argmin} \|x_i - x\|$
    if label(x_{nn}) $\neq$ label(x) **then**
      prototype.append(x_{nn})
    end if
  end for
end while

methods involving randomness, i.e. the baseline method, ENN, and CNN, we ran 5 experiments for each method at each M and took the average. We use two metrics for the evalutaion of the performances of different methods, the error rate and runtime. The results are shown in Figure 1 and Figure 2

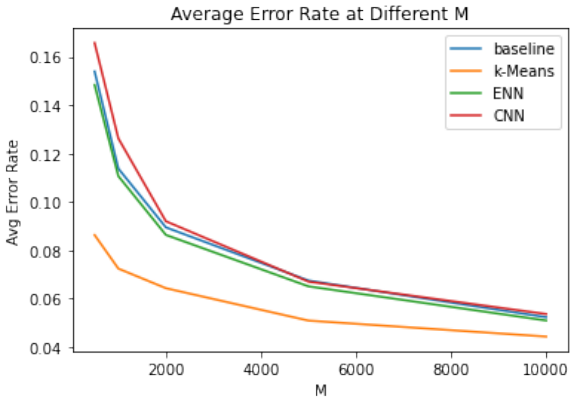


Figure 1: Average Error Rate at Different Values of M.

In addition, we compute the confidence intervals of error rates for methods with randomness under 95% confidence level. We assume the error rates follow a Student's t distribution. The confidence

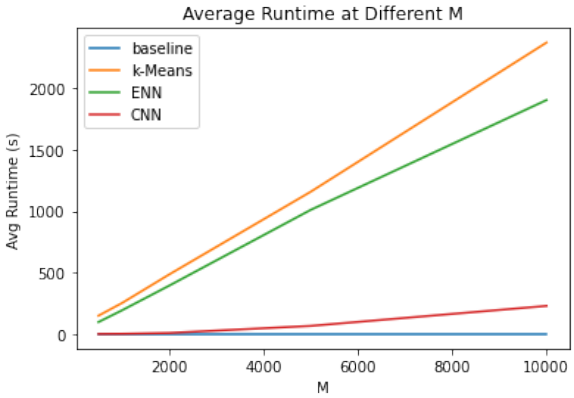


Figure 2: Average Runtime at Different Values of M.

interval should be calculated by:

$$\bar{X} \pm t_{p,n-1} \cdot \frac{S}{\sqrt{n}}, t_{p,n-1} = F_{n-1}^{-1}(\frac{1+p}{2}) \quad (1)$$

, where $p = 0.95$, F_{n-1}^{-1} is the probability point function of Student's t distribution with $n-1$ degrees of freedom, $\bar{X}$ is the sample mean, S is the sample standard deviation and $n = 5$. The results are shown in Table 1-3.

M	**Lower Bound**	**Upper Bound**
500	0.13912541	0.16875459
1000	0.11014883	0.11737117
2000	0.08723524	0.09168476
5000	0.06450015	0.07041985
10000	0.0508521	0.0537879

Table 1: Confidence Intervals of Error Rates of **Baseline** at Different Values of M under 95% Confidence Level

M	**Lower Bound**	**Upper Bound**
500	0.14315001	0.15360999
1000	0.10602034	0.11525966
2000	0.08388967	0.08879033
5000	0.06260339	0.06751661
10000	0.04931612	0.05256388

Table 2: Confidence Intervals of Error Rates of **ENN** at Different Values of M under 95% Confidence Level

4 Critical Evaluation

According to the results illustrated in last section, we can find that in general, as the value of M increases, i.e. the size of the prototype set becomes larger, the error rates of all methods show a significant downward trend and gradually reach convergence. The k-Means based method has a clear

M	**Lower Bound**	**Upper Bound**
500	0.1645219	0.1669581
1000	0.12434615	0.12813385
2000	0.0897184	0.0942416
5000	0.06564539	0.06839461
10000	0.05289005	0.05434995

Table 3: Confidence Intervals of Error Rates of **CNN** at Different Values of M under 95% Confidence Level

improvement over random selection, while ENN only shows a tiny edge over baseline. CNN is even worse than baseline when M is small and the gap becomes negligible when M is decently large. k-Means is an efficient clustering algorithm, therefore the prototype selected from the centroids of its clusters are highly representative, and the classification accuracy becomes better as the number of clusters increases.

While the results from ENN and CNN can be explained by our implementation of these methods, since in the original algorithm description, they both should be proceeded over the entire training set to find a really representative subset. However, in our implementation, we stop the procedure when the cardinality of the prototype reaches M, which for sure will have a worse result since the algorithms haven't learned complete patterns of the training set. Nonetheless, if we implement the original algorithm, the cardinality of the prototype is still a sticky issue, since we cannot control it, it's dependent on the inherent characteristics of the training data. In the worst case, the prototype could just be the original set.

In terms of runtime, all methods are beaten by the baseline since they all require considerable amount of loops. CNN is faster because it conducts nearest neighbor rule between one sample and a relatively small store set. In future, we would like to try other dimensionality reduction techniques, such as PCA, or proved efficient feature extractors, such as convolutional neural networks and transformers, to improve prototype selection performances.

References

Peter Hart. 1968. The condensed nearest neighbor rule (corresp.). *IEEE transactions on information theory*, 14(3):515–516.

James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA.

Dennis L Wilson. 1972. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 408–421.

```
import  torch
import  torchvision
import  torchvision.datasets  as  datasets
from  torchvision.transforms  import  ToTensor
import  numpy  as  np
import  matplotlib.pyplot  as  plt
from  sklearn.neighbors  import  KNeighborsClassifier
from  sklearn.cluster  import  KMeans
from  sklearn.metrics  import  pairwise_distances_argmin_min
from  collections  import  defaultdict
import  random
import  time
from  scipy.stats  import  t
import  math


#  Data  preprocessing
trainset  =  datasets.MNIST(root='./data',  train=True,  download=True,  transform=ToTensor())
testset  =  datasets.MNIST(root='./data',  train=False,  download=True,  transform=ToTensor())
train_data  =  trainset.data
train_labels  =  trainset.targets
train_data  =  train_data.view([len(train_data),  28*28])
train_data  =  train_data  /  255.0
test_data  =  testset.data
test_labels  =  testset.targets
test_data  =  testset.data.view([len(test_data),  28*28])
test_data  =  test_data  /  255.0
train_data  =  np.array(train_data)
train_labels  =  np.array(train_labels)
test_data  =  np.array(test_data)
test_labels  =  np.array(test_labels)


def  kNNclassifier(xTrain,  yTrain,  xTest=test_data,  yTest=test_labels):
    model  =  KNeighborsClassifier(n_neighbors=1)
    model.fit(xTrain,  yTrain)
    accuracy  =  model.score(xTest,  yTest)

    return  accuracy


def  baselineMethod(xTrain,  yTrain,  M):
    indices  =  np.random.randint(0,  xTrain.shape[0],  size=M)

    return  indices


def  kMeansPrototype(xTrain,  yTrain,  M):
    indices  =  []
    classes  =  np.unique(yTrain)
    num_clusters  =  int(M  /  len(classes))
    x_indices  =  defaultdict(list)
    for  index  in  range(len(yTrain)):
        x_indices[yTrain[index]].append(index)
    for  label  in  x_indices.keys():
        centroids  =  KMeans(n_clusters=num_clusters).fit(xTrain[x_indices[label]]).cluster_centers_
        prototype_indices,  _  =  pairwise_distances_argmin_min(centroids,  xTrain[x_indices[label]])
        for  i  in  range(num_clusters):
            prototype_indices[i]  =  x_indices[label][prototype_indices[i]]
        indices.append(prototype_indices)
    indices  =  np.concatenate(indices).ravel()

    return  indices


def  EditedNN(xTrain,  yTrain,  M):
    indices  =  []
    x_index  =  np.random.permutation(len(yTrain))
    for  index  in  x_index:
        if  len(indices)  >=  M:
            return  indices
        dist  =  np.linalg.norm(xTrain  -  xTrain[index],  ord=2,  axis=1)
        nn  =  np.argsort(dist,  axis=0)[1]
        if  yTrain[nn]  ==  yTrain[index]:
            indices.append(index)

    return  indices
```

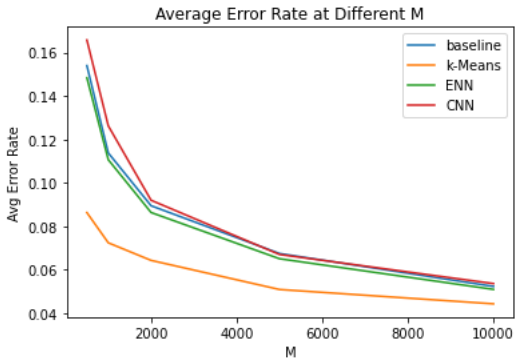
```
def CNN(xTrain, yTrain, M):
    indices = []
    grabbag = []
    x_index = np.random.permutation(len(yTrain))
    indices.append(x_index[0])
    for index in range(1, len(x_index)):
        if len(indices) >= M:
            return indices
        dist = np.linalg.norm(xTrain[indices] - xTrain[index], ord=2, axis=1)
        nn = np.argsort(dist, axis=0)[0]
        if yTrain[nn] != yTrain[index]:
            indices.append(index)
        else:
            grabbag.append(index)

    for index in grabbag:
        if len(indices) >= M:
            return indices
        dist = np.linalg.norm(xTrain[indices] - xTrain[index], ord=2, axis=1)
        nn = np.argsort(dist, axis=0)[0]
        if yTrain[nn] != yTrain[index]:
            indices.append(index)

    return indices


def run():
    iters = 5
    subset_size = [500, 1000, 2000, 5000, 10000]
    models = ['baseline', 'kmeans', 'enn', 'cnn']
    err = np.zeros((5, 4, 5))
    runtime = np.zeros((5, 4, 5))
    for M_index, M in enumerate(subset_size):
        for model_index, model in enumerate(models):
            if model == 'kmeans':
                start = time.time()
                proto_indices = kMeansPrototype(train_data, train_labels, M)
                end = time.time()
                runtime[M_index][model_index][:] = end - start
                acc = kNNclassifier(train_data[proto_indices], train_labels[proto_indices])
                err[M_index][model_index][:] = 1 - acc
            else:
                for iter in range(iters):
                    start = time.time()
                    if model == 'baseline':
                        proto_indices = baselineMethod(train_data, train_labels, M)
                    elif model == 'enn':
                        proto_indices = EditedNN(train_data, train_labels, M)
                    elif model == 'cnn':
                        proto_indices = CNN(train_data, train_labels, M)
                    end = time.time()
                    runtime[M_index][model_index][iter] = end - start
                    acc = kNNclassifier(train_data[proto_indices], train_labels[proto_indices])
                    err[M_index][model_index][iter] = 1 - acc
    np.save('error.npy', err)
    np.save('runtime.npy', runtime)
    return err, runtime


run()


err = np.load('error.npy')
runtime = np.load('runtime.npy')
err_avg, runtime_avg = np.mean(err, axis=2), np.mean(runtime, axis=2)
err_avg, runtime_avg

      (array([[0.15394, 0.0863 , 0.14838, 0.16574],
              [0.11376, 0.0724 , 0.11064, 0.12624],
              [0.08946, 0.0643 , 0.08634, 0.09198],
              [0.06746, 0.0509 , 0.06506, 0.06702],
              [0.05232, 0.0443 , 0.05094, 0.05362]]),
       array([[0.00000000e+00, 1.49568158e+02, 9.77819602e+01, 3.89502430e-01],
              [0.00000000e+00, 2.52976594e+02, 1.93612186e+02, 2.09773755e+00],
              [1.99270248e-04, 4.85420623e+02, 3.94287601e+02, 9.12717113e+00],
              [0.00000000e+00, 1.15862122e+03, 1.01106121e+03, 6.65758853e+01],
              [2.01416016e-04, 2.37312594e+03, 1.90511121e+03, 2.29231033e+02]]))
```

```
M = [500, 1000, 2000, 5000, 10000]
err_b = np.take(err_avg, 0, axis=1)
err_k = np.take(err_avg, 1, axis=1)
err_e = np.take(err_avg, 2, axis=1)
err_c = np.take(err_avg, 3, axis=1)
err_b, err_k, err_e, err_c
```

```
(array([0.15394, 0.11376, 0.08946, 0.06746, 0.05232]),
 array([0.0863, 0.0724, 0.0643, 0.0509, 0.0443]),
 array([0.14838, 0.11064, 0.08634, 0.06506, 0.05094]),
 array([0.16574, 0.12624, 0.09198, 0.06702, 0.05362]))
```

```
plt.title('Average Error Rate at Different M')
plt.plot(M, err_b, label='baseline')
plt.plot(M, err_k, label='k-Means')
plt.plot(M, err_e, label='ENN')
plt.plot(M, err_c, label='CNN')
plt.legend()
plt.xlabel('M')
plt.ylabel('Avg Error Rate')
```

```
Text(0, 0.5, 'Avg Error Rate')
```

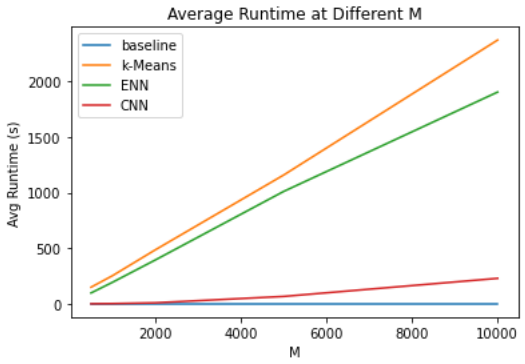



```
rt_b = np.take(runtime_avg, 0, axis=1)
rt_k = np.take(runtime_avg, 1, axis=1)
rt_e = np.take(runtime_avg, 2, axis=1)
rt_c = np.take(runtime_avg, 3, axis=1)
rt_b, rt_k, rt_e, rt_c
```

```
(array([0.        , 0.        , 0.00019927, 0.        , 0.00020142]),
 array([ 149.56815791,  252.97659421,  485.4206233 , 1158.62122011,
        2373.12593985]),
 array([  97.7819602 ,  193.61218629,  394.28760066, 1011.06121492,
        1905.11121449]),
 array([  0.38950243,   2.09773755,   9.12717113,  66.5758853 ,
        229.2310329 ]))
```

```
plt.title('Average Runtime at Different M')
plt.plot(M, rt_b, label='baseline')
plt.plot(M, rt_k, label='k-Means')
plt.plot(M, rt_e, label='ENN')
plt.plot(M, rt_c, label='CNN')
plt.legend()
plt.xlabel('M')
plt.ylabel('Avg Runtime (s)')
```

```
Text(0, 0.5, 'Avg Runtime (s)')
```




```
df = 5 - 1
t_value = t.ppf((1+0.95)/2, df)
np.std(err, axis=2)
err_avg-t_value/math.sqrt(5)*np.std(err, axis=2), err_avg+t_value/math.sqrt(5)*np.std(err, axis=2)
```

```
(array([[0.13912541, 0.0863    , 0.14315001, 0.1645219 ],
        [0.11014883, 0.0724    , 0.10602034, 0.12434615],
        [0.08723524, 0.0643    , 0.08388967, 0.0897184 ],
        [0.06450015, 0.0509    , 0.06260339, 0.06564539],
        [0.0508521 , 0.0443    , 0.04931612, 0.05289005]]),
 array([[0.16875459, 0.0863    , 0.15360999, 0.1669581 ],
        [0.11737117, 0.0724    , 0.11525966, 0.12813385],
        [0.09168476, 0.0643    , 0.08879033, 0.0942416 ],
        [0.07041985, 0.0509    , 0.06751661, 0.06839461],
        [0.0537879 , 0.0443    , 0.05256388, 0.05434995]]))
```