Prototype Selection for Nearest Neighbour Classification

Abstract

The k-nearest neighbour classification algorithm classifies a data point in accordance with it's closest neighbours in the training dataset. Although with increasing training data size the accuracy of the prediction increases, the computational efficiency decreases due to a larger number of distance calculations. There thus arises a necessity to select a subset of prototypes that best represents the available data.

1 Introduction

To study methods of efficient prototype selection for nearest neighbour classification through the course of this paper, we use the MNIST dataset. This dataset contains 60,000 training samples and 10,000 test samples. With this data we explore prototype selection algorithms that given a size parameter M, return a representative prototype set of M samples. We begin by generating baselines through random sampling and then implement algorithms like class-wise k-means clustering, Learning Vector Quantization and Hart Condensation. We evaluate the performance of each of the techniques by computing the accuracy of 1-NN classification on the test dataset. Finally, we perform a comparative analysis of the models and evaluate the efficiency improvement vs accuracy trade off on the best performing model.

2 Prototype Selection Algorithms

2.1 Random Selection (Baseline)

To evaluate if the designed prototype selection algorithms are performing better than the naive approach i.e. random selection, we define the baselines. For a required prototype set of size M, we randomly sample M data points from the original training data. We perform this sampling for a number of iterations and at each iteration, we compute the evaluation metrics (accuracy). The average metrics for a sample size M are then used to judge the credibility of the algorithms outlined below.

To account for class imbalance due to such random selection, we further implement a sampling procedure where we sample M/c data points from the training data for each distinct label (c is the number of distinct labels or classes). Similar to the previous approach, we obtain average metrics over multiple iterations and these metrics serve as the baseline.

2.2 K-Means Clustering

In this approach, we consider the data clusters for each label separately and perform prototype selection on the individual clusters using the k-means algorithm. The intuition behind this approach is that if we were to select k points that represent a cluster, the set of k-centroids of this cluster is the set that best summarizes the underlying data and hence is the best representative. We therefore perform k-means clustering on each of the c subsets (c is the number of distinct labels or classes) where k is set to M/c (M is the desired prototype size). Since we are computing the centroids of the subsets of training data, this method returns a set of virtual points (data points not present in the original data) as prototypes.

2.3 Learning Vector Quantization

In this method (designed in reference to Kohonen (1995)), we begin with a prototype set of M data points sampled uniformly across all distinct labels in the training dataset. In the learning step, we randomly sample points from the training data and predict its label with respect to the prototype set using the 1-nearest neighbor classifier. If the predicted label is correct, we move the nearest neighbor in the prototype set towards the sampled point. Otherwise, we move the nearest neighbor in the prototype set away from the sampled point. The distance by which the prototype is moved is determined by a learning rate which decreases gradually in each iteration to ensure convergence. Since, we are moving the prototypes from the original dataset in the real value space, the resulting points in the prototype set need not belong to the training set. Hence, a set of virtual points is generated as the representative of the training set.The pseudo code

is outlined in **Algorithm 1**

Algorithm 1 Learning Vector Quantization

Select M/c prototypes for each of the c distinct labels by randomly sampling from training data

For $iteration$ in max_iter:

- Sample a data point x_i from training set randomly and compute its nearest neighbour nn_p in prototypes.
- If label(x_i) = label(nn_p) update nn_p as:

$$nn_p = nn_p - \eta(x_i - nn_p)$$

- Else update x_i as:

$$nn_p = nn_p + \eta(x_i - nn_p)$$

- Decrease the learning rate parameter by a constant η_o.

We also experiment with a slightly modified version of the algorithm outlined above. In this case, when the randomly sampled data point is correctly classified, we do not move the nearest prototype. We only update the nearest prototype when it wrongly classifies a new data point.

2.4 Condensed Nearest Neighbours

In condensation (designed in reference to Hart (1968)), we begin with a small set of randomly sampled prototypes and incrementally construct the prototype set of a desired size. We evaluate each training point and determine if it contributes to the classification. If a training data point is correctly classified by the prototype set at that instant using the 1-nearest neighbor classifier, we do not add the data point to the prototype set. In case of a misclassification, we add it to the prototype set. However, in this approach we do not have a control of the size of the representative set. In order to handle this, we stop adding new prototypes once required prototype size is achieved. Alternatively, in a scenario where the set has a size less than M, we randomly add samples from training data till the required subset size is achieved. In this method since we are only selecting the prototypes from the original data, virtual points are not created and resulting prototype set is a subset of the training data. The pseudo code is outlined in **Algorithm 2**.

Algorithm 2 Condensed Nearest Neighbours

Select N prototypes by randomly sampling from training data ($N << M$)

For x_i in training_data:

- Classify x_i using 1-NN w.r.t protypes.
- If x_i is incorrectly classified, add x_i to prototypes.
- If prototype size $\geq M$: $break$.

While prototype size $< M$:

- Randomly sample training set and add to prototypes.

3 Experimental Results

We first compute the evaluation metrics on the entire dataset to get a picture of the performance tradeoff when scaling down to a representative prototype set. The metrics on entire dataset are indicated as *Original training set* in Table 1.

To generate baselines, random sampling has been performed over multiple iterations on the entire data followed by the sampling from data corresponding to each label as outlined in section 2.1. The average performance metrics across these iterations have been reported in Table 1. Although for high values of desired prototype size M the accuracy of random sampling is very high, the performance rapidly degrades for lower values of M. These values will serve a benchmark to evaluate the other algorithms going forward.

Data type	Accuracy	N samples
Original training set	96.91	60000
Random Sampling	94.76	10000
Random Sampling	93.49	5000
Random Sampling	88.58	1000

Table 1: Accuracy on original dataset and randomly sampled prototype sets (baselines)

In the k-means clustering, we generate M/c centroids representative of the each of the c classes in the training data. The collection of these centroids is then the prototype set. The accuracy metrics for various prototype sizes are reported in Tables 2,3 and 4. For the sample size reduction from $M = 60000$ to $M = 10000$, we observe that even

Algorithm	Accuracy
K-Means	96.85
LVQ(a)	95.72
LVQ(b)	94.84
Condensation(a)	96.76
Condensation(b)	95.89

Table 2: Accuracy for M = 10000

though the training data is scaled down by 6-fold, the loss is accuracy is negligible. Additionally for the case where $M = 5000$ as well, a significant reduction in performance is not observed. For all sample sizes, this method performs significantly better than random sampling.

In Learning Vector Quantization, we initialize the required number of prototypes randomly from the training set and consequently make adjustments to these representatives in accordance with whether or not they classify new training points correctly. The magnitude of these adjustments depends on the learning rate which is reduced in each iteration. Two approaches have been tested for this method - the first one (LVQ(a)) where the update is performed both in case of correct and incorrect classification and second (LVQ(b)) where update is performed only on incorrect classification. Experiments with varying values of initial learning rates (from 0.1 to 0.5) and the shrinking parameter have been performed and the results for best learning rate in each case are reported in Tables 2,3 and 4. Learning vector quantization performs random sampling for all prototype set sizes which is indicative of an underlying pattern being learnt by the algorithm. However, K-means still outperforms this method for all values of M (potential reasons analyzed in section 4).

Algorithm	Accuracy
K-Means	96.69
LVQ(a)	95.37
LVQ(b)	95.43
Condensation(a)	95.94
Condensation(b)	95.24

Table 3: Accuracy for M = 5000

In the Hart condensation method as outlined in section 2.4, we incrementally build the prototype set. During this process, we do not have a control on the prototype set size and hence the set size is adjusted after the condensation. In performing this process on the original training set, we observe that the condensation scales down the dataset to M = 5003 data points in a single iteration. Then two approaches were followed to adjust the set size. In approach labelled Condensation(a), random samples were added/removed from the prototype set to achieve the required M. In Condensation(b), new centroids from the training data were added to increases set size and prototypes were replaced by their centroids to decrease the size. For the case where $M = 5000$, the condensed data set has been assumed as the prototype set without any modification. The accuracies for both approaches have been reported in Tables 2,3 and 4. It is observed that the algorithm performs better than random sampling for all values of M. Further comparative analysis with other methods is discussed in section 4.

Algorithm	Accuracy
K-Means	95.8
LVQ(a)	93.55
LVQ(b)	92.74
Condensation(a)	92.18
Condensation(b)	91.61

Table 4: Accuracy for M = 1000

The performance of the methods outlined is visualized in Figure 1.

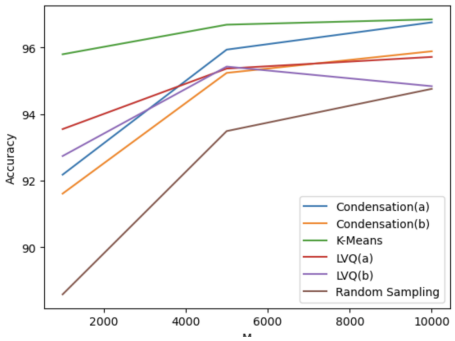

Figure 1: Acuuracy of prototype selection methods

4 Critical Evaluation

From our experimental results, we observe that the prototype set returned by the k-means clustering (on the subset of training data corresponding to each label) returns the best representative set for all values of M. This higher performance can be attributed to the fact that these prototypes are virtual points (not present in the original dataset) and are centroids that summarize the underlying information in the data with minimum possible loss. We

observe that upon scaling down the training data size by 6 times using this method, the decrease in accuracy is merely 0.06%. The time complexity of the 1-NN algorithm is directly proportional to number of data points in training dataset. Hence for a 6 fold improvement in time of computation, the trade-off is 0.06% and for a 12 fold improvement in time of computation, the trade-off is 0.22%. Therefore, using k-means clustering method to compute prototypes gives us efficient solution with the best performance metrics.

In the case of Learning Vector Quantization, although the method performs significantly better than random sampling, the K-means selection still outperforms this method. This could be due to the reason that the random initialization of the sample points by LVQ does not capture the essence of the data as well their centroids. We also observe that LVQ(a) where update happens both in case of correct and incorrect classification during learning performs better than LVQ(b) where update happens only during misclassification.

There is further scope for experimentation and improvement in prototype selection by LVQ especially in the strategy for initialization as well as controlling the learning rate parameter. A more meaningful initialization that encapsulates the data better than random selection can be worked upon. In a brief experiment, centroids were chosen as the initial prototypes. However this has not performed as well as the random initialization. This could be because centroids provide an efficient set of prototypes as seen in k-means clustering and moving them around by adjusting them at each learning step might be detrimental. In the case of learning parameter, we have performed a linear shrinkage in each iteration. Various other trajectories of the learning rate parameter can be studied upon for optimal convergence scenarios.

Similar to LVQ, although Hart condensation method performs better than the random sampling it does not outperform the k-means solution. This could again be attributed to the random initialization or the addition and removal of random samples during scaling up and down to the required size as we have no control over the prototype set size during condensation. The performance is significantly inferior as compared to the k-means method for values of M that are less than the condensed set size because the points that contribute to classification (as determined during condensation) are excluded or replaced during the scaling down process.

References

P. Hart. 1968. The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516.

Teuvo Kohonen. 1995. *Learning Vector Quantization*, pages 175–189. Springer Berlin Heidelberg, Berlin, Heidelberg.

In [141…
```
import idx2numpy
import numpy as np
from sklearn.cluster import KMeans
from sklearn.neighbors import KNeighborsClassifier
import random
from collections import Counter
from sklearn.metrics import precision_recall_fscore_support
```

In [29]:
```
from imblearn.under_sampling import CondensedNearestNeighbour
```

In [2]:
```
x_train = idx2numpy.convert_from_file('train-images.idx3-ubyte')
y_train = idx2numpy.convert_from_file('train-labels.idx1-ubyte')
```

In [3]:
```
x_test = idx2numpy.convert_from_file('t10k-images.idx3-ubyte')
y_test = idx2numpy.convert_from_file('t10k-labels.idx1-ubyte')
```

In [4]:
```
x_train.shape
```

Out[4]:
```
(60000, 28, 28)
```

In [5]:
```
y_train.shape
```

Out[5]:
```
(60000,)
```

In [6]:
```
x_test.shape
```

Out[6]:
```
(10000, 28, 28)
```

In [7]:
```
y_test.shape
```

Out[7]:
```
(10000,)
```

In [8]:
```
x_train=[i.flatten() for i in list(x_train)]
x_test=[i.flatten() for i in list(x_test)]
```

Full dataset training

In [137…
```
model = KNeighborsClassifier(n_neighbors=1)
model.fit(x_train, y_train)
```

Out[137]:
```
▾ KNeighborsClassifier
KNeighborsClassifier(n_neighbors=1)
```

In [138…
```
model.score(x_test,y_test)
```

Out[138]:
```
0.9691
```

In [139…
```
y_pred = model.predict(x_test)
```

In [143…
```
precision_recall_fscore_support(y_test,y_pred,average='weighted')
```

Out[143]:
```
(0.9692024021594029, 0.9691, 0.9690691238431085, None)
```

In [131…
```
Counter(y_train)
```

Out[131]:
```
Counter({1: 6742,
         7: 6265,
         3: 6131,
         2: 5958,
         9: 5949,
         0: 5923,
         6: 5918,
         8: 5851,
         4: 5842,
         5: 5421})
```

Random Samples as prototypes

In [151…
```
ls = list(zip(x_train,y_train))
def knn_random_sample(n_samples):
    xy_rand = random.sample(ls,n_samples)
    xr = [i[0] for i in xy_rand]
    yr = [i[1] for i in xy_rand]
    #print(Counter(yr))
    model = KNeighborsClassifier(n_neighbors=1)
    model.fit(xr, yr)
    acc = model.score(x_test,y_test)
    #y_pred = model.predict(x_test)
    #p = precision_recall_fscore_support(y_test,y_pred,average='weighted')
    return acc#,p
```

In [304…
```
a = 0
for i in range(15):
    a += knn_random_sample(10000)
print(a/15)
```

```
0.9476533333333331
```

In [116…
```
for i in range(15):
    a = knn_random_sample(5000)
    print(a)
```

```
0.9363
0.9327
0.9368
0.9327
0.9331
0.9335
0.9361
0.9345
0.9353
0.9362
0.9325
0.9348
0.9359
0.9371
0.9352
```

In [117…
```
for i in range(15):
    a = knn_random_sample(1000)
    print(a)
```

```
0.8929
0.8894
0.8856
0.8805
0.8862
0.8816
0.8835
0.8888
```

```
0.8813
0.8868
0.8943
0.8794
0.8886
0.8901
0.8842
```

In [16]:
```
Counter(y_train)
```

Out[16]:
```
Counter({1: 6742,
         7: 6265,
         3: 6131,
         2: 5958,
         9: 5949,
         0: 5923,
         6: 5918,
         8: 5851,
         4: 5842,
         5: 5421})
```

In [126…
```
def knn_random_sample2(n_samples):
    pro = []
    pro_label = []
    k = int(n_samples/10)

    for i in range(10):
        print("Getting "+ str(k) + " centers for label "+str(i))
        xy = [j for j in ls if j[1]==i]
        xl = [j[0] for j in xy]
        #print(xl)
        #yl = [i[1] for i in xy]
#         kmeans = KMeans(n_clusters=k,random_state=0, n_init="auto").fit(xl)
#         pro.extend(kmeans.cluster_centers_)
        pro.extend(random.sample(xl,k))
        pro_label.extend([i]*k)
    return pro, pro_label
```

In [165…
```
pro, pro_label = knn_random_sample2(5000)
pro_zip = list(zip(pro,pro_label))
random.shuffle(pro_zip)
print("Training dataset size is "+str(len(pro_zip)))
xr = [i[0] for i in pro_zip]
yr = [i[1] for i in pro_zip]
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xr, yr)
acc = model.score(x_test,y_test)
acc
```

```
Getting 500 centers for label 0
Getting 500 centers for label 1
Getting 500 centers for label 2
Getting 500 centers for label 3
Getting 500 centers for label 4
Getting 500 centers for label 5
Getting 500 centers for label 6
Getting 500 centers for label 7
Getting 500 centers for label 8
Getting 500 centers for label 9
Training dataset size is 5000
```

Out[165]:
```
0.938
```

Approach 1

In [167…
```
def get_prototypes1(k):
    pro = []
    pro_label = []

    for i in range(10):
        print("Getting "+ str(k) + " centers for label "+str(i))
        xy = [j for j in ls if j[1]==i]
        xl = [j[0] for j in xy]
        #yl = [i[1] for i in xy]
        kmeans = KMeans(n_clusters=k,random_state=0, n_init="auto").fit(xl)
        pro.extend(kmeans.cluster_centers_)
        pro_label.extend([i]*k)
    return pro, pro_label
```

In [169…
```
M = 10000
pro, pro_label = get_prototypes1(int(M/10))
pro_zip = list(zip(pro,pro_label))
random.shuffle(pro_zip)
print("Training dataset size is "+str(len(pro_zip)))
xr = [i[0] for i in pro_zip]
yr = [i[1] for i in pro_zip]
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xr, yr)
acc = model.score(x_test,y_test)
acc
```

```
Getting 1000 centers for label 0
Getting 1000 centers for label 1
Getting 1000 centers for label 2
Getting 1000 centers for label 3
Getting 1000 centers for label 4
Getting 1000 centers for label 5
Getting 1000 centers for label 6
Getting 1000 centers for label 7
Getting 1000 centers for label 8
Getting 1000 centers for label 9
Training dataset size is 10000
```

Out[169]:
```
0.9685
```

In [170…
```
M = 5000
pro, pro_label = get_prototypes1(int(M/10))
pro_zip = list(zip(pro,pro_label))
random.shuffle(pro_zip)
print("Training dataset size is "+str(len(pro_zip)))
xr = [i[0] for i in pro_zip]
yr = [i[1] for i in pro_zip]
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xr, yr)
acc = model.score(x_test,y_test)
acc
```

```
Getting 500 centers for label 0
Getting 500 centers for label 1
Getting 500 centers for label 2
Getting 500 centers for label 3
Getting 500 centers for label 4
Getting 500 centers for label 5
Getting 500 centers for label 6
Getting 500 centers for label 7
Getting 500 centers for label 8
Getting 500 centers for label 9
Training dataset size is 5000
```

Out[170]:
```
0.9669
```

In [171…
```
M = 1000
pro, pro_label = get_prototypes1(int(M/10))
pro_zip = list(zip(pro,pro_label))
random.shuffle(pro_zip)
print("Training dataset size is "+str(len(pro_zip)))
xr = [i[0] for i in pro_zip]
yr = [i[1] for i in pro_zip]
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xr, yr)
acc = model.score(x_test,y_test)
acc
```

```
Getting 100 centers for label 0
Getting 100 centers for label 1
Getting 100 centers for label 2
Getting 100 centers for label 3
Getting 100 centers for label 4
Getting 100 centers for label 5
Getting 100 centers for label 6
Getting 100 centers for label 7
Getting 100 centers for label 8
Getting 100 centers for label 9
Training dataset size is 1000
```

Out[171]:
```
0.959
```

Approach 2 - LVQ

In [272…
```
prototypes = []
for i in range(10):
    xy = [j for j in ls if j[1]==i]
    prototypes.extend(random.sample(xy,1000))
len(prototypes)
```

Out[272]:
```
10000
```

In [273…
```
#prototypes = pro_zip
xp = [i[0] for i in prototypes]
yp = [i[1] for i in prototypes]
lr = 0.15
lri = lr
max_iter = 2000
```

In [274…
```
for i in range(max_iter):
    rs = random.randint(0, len(ls)-1)
    x = ls[rs][0]
    label = ls[rs][1]
    model = KNeighborsClassifier(n_neighbors=1)
    model.fit(xp, yp)
    nn = model.kneighbors([x],return_distance = False)[0][0]
    nn_label = yp[nn]
#     print(nn_label)
#     print(label)
    if(label != nn_label):
        ## Move protype away
#         print("moving away")
        xp[nn] = xp[nn] - lr*(x-xp[nn])
    else:
        ## Move prototype closer
#         print("moving closer")
        xp[nn] = xp[nn] + lr*(x-xp[nn])

#     print(x_train[i])
```

```
#     print(xp[nn])
    lr = lr - lri/max_iter
```

In [303…
```
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xp, yp)
acc = model.score(x_test,y_test)
acc
```

```
0.9578
```

Approach 3 CNN

In [276…
```
def cnn_protoype(M):
    random.shuffle(ls)
    pr = random.sample(ls,1000)#[ls[0]]
    for i in range(len(ls)):
        #print(i)
        if(len(pr)>=M): break
        #pred = predict(ls[i][0])

        xp = [i[0] for i in pr]
        yp = [i[1] for i in pr]
        model = KNeighborsClassifier(n_neighbors=1)
        model.fit(xp, yp)
        nn = model.kneighbors([ls[i][0]],return_distance = False)[0][0]
        nn_label = yp[nn]

        if(nn_label != ls[i][1]):
            pr.append(ls[i])
    return pr
```

In [302…
```
ls_cnnp = cnn_protoype(10000)
len(ls_cnnp)
```

```
5003
```

In [279…
```
def get_prototypes1(k,yp):
    pro = []
    pro_label = []

    for i in range(10):
        c = k-Counter(yp)[i]
        if(c<=0): continue
        print("Getting "+ str(c) + " centers for label "+str(i))
        xy = [j for j in ls if j[1]==i]
        xl = [j[0] for j in xy]
        #yl = [i[1] for i in xy]
        kmeans = KMeans(n_clusters=c,random_state=0, n_init="auto").fit(xl)
        pro.extend(kmeans.cluster_centers_)
        pro_label.extend([i]*c)
    return pro, pro_label
```

In [281…
```
pro, pro_label = get_prototypes1(1000,yp)
xp.extend(pro)
yp.extend(pro_label)
model = KNeighborsClassifier(n_neighbors=1)
model.fit(xp, yp)
acc = model.score(x_test,y_test)
acc
```

```
Getting 734 centers for label 0
Getting 798 centers for label 1
Getting 431 centers for label 2
Getting 306 centers for label 3
```

```
Getting 416 centers for label 4
Getting 387 centers for label 5
Getting 665 centers for label 6
Getting 516 centers for label 7
Getting 152 centers for label 8
Getting 286 centers for label 9
```

Out[281]: 0.9676