Optimizations on Simple Classification using Binary Data

https://arxiv.org/abs/1707.01945

HE Jiaxin

Summary of Paper

As binary or one-bit, representations of data arise naturally in many areas and are appealing in both hardware implementations and algorithm design. In this work[1], the researchers studied the problem of data classification from binary data obtained from the sign pattern of low-dimensional projections and proposed a framework with low computation and resource costs. This paper illustrates the utility of the proposed approach through stylized and realistic numerical experiments, and provides a theoretical analysis for a simple case. In conclusion, the paper claims that this supervised classification algorithm is also relevant to analyzing similar multi-level-type algorithms, and future directions of this work can be the utilization of the algorithm for more complicated data geometries, identifying settings where real-valued measurements may be worth the additional complexity, analyzing geometries with non-uniform densities of data, as well as a generalized theory for high dimensional data belonging to many classes and utilizing multiple levels within the algorithm. They believe that this method will be able to extend nicely into other applications such as hierarchical clustering and classification as well as detection problems.

Question 1: Condition for Asymptotic Behaviour

This paper proved the conditions under which the probability that the algorithms classify the binary data correctly converge to 1 as the number of features approaches infinity.
The proof was done under a 2 dimension scenario with 2 classes. A 2-dimensional data can be represented as a vector on a 2-dimensional coordinate plot. All vectors in class 1 form a cone on the plane, the angle of the cone is A1. Similarly, we can measure the angle of the class 2 A2. The distance between the two cones is defined as A12.

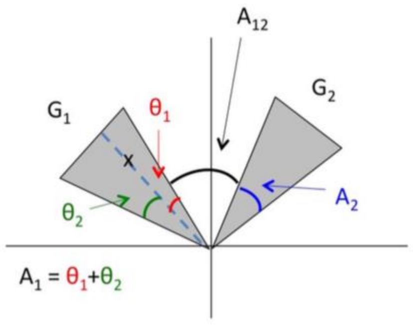

Figure 1. Angular representation of two classes of two dimension data

It proved 2 scenarios under which the probability converge: 1) A12 ≥ A1 and 2A12 ≥ π − 2A1; 2) A1 + A12 > 0.58π and A12 + 3 /4A1 ≤ 1/2 π, where A1 is the angle of class 1, and A12 is the distance between two classes in angular representation. From the proof, it can be observed that the probability does not converge to 1 when A1 +A12 > 0.58π. This shows the limitation of the algorithm. A question that we would like to ask is that how to make the classification valid when A1 +A12 > 0.58π?
First of all, a simulation could be done to produce a plot of the probability versus m when the condition satisfies A1 +A12 > 0.58π. From the plot, we could observe whether the probability approaches zero or converge to a value smaller than 1. The former result suggests that the method is no longer valid. The latter indicates that the accuracy decrease when A1 +A12 > 0.58π.

Question 2: Geographical Information Loss

Another potential issue we observed from this paper is the geographical information loss if the algorithm is applied to 2-dimensional data input, for example, the MNIST digit images and YaleB facial recognition input used in this paper.

The algorithm itself transforms the 2-dimensional input via signum function for its binary form and only randomly selected binary data are used as a training example. By this way, the algorithm has ignored the geographic information that each input preserved. For example, the relative positions between curly top and straight bottom of digit “2”.

This issue has been reflected in the model accuracy of the MNIST dataset. As illustrated in the paper, the classification accuracy reaches slightly better than 90% (Figure 9). Based on current research results, a simple Neuron Network can reach accuracy of more than 95%. Moreover, most of training libraries nowadays, such as tensorflow, is able to archive 98% accuracy on testing data in using the same dataset.

There are few ways in current research to preserve the geographical information such as applying deep Neuron Networks to let algorithm learn the feature adaptively. However, this could lead astray from what the authors intend to archive in this paper, which is a simple and cost efficient algorithm to for binary data.

As authors also stated in this paper, the algorithm does not necessarily concern with geometry preservation in low-dimensional space (assuming 2-dimensional space is considered as low dimension). Moreover, as the focus of the paper is on computation and resource costs efficient method, it is reasonable to benchmark the algorithm in different aspect with commonly used algorithm such as simple Neuron Networks or Supporting Vector Machine.

Question 3: Compare with other useful classic models

Firstly, we notice that this paper only illustrates some results of itself performance but doesn’t mention the advantages of this model compared with other useful classic models.

We choose the MNIST as our original dataset. And considering the mix-up degree, all optimizing experiment is based on the digits ‘0’ and ‘5’, because these two classes are less likely to be understandable. Here, on the basis of implementation of this binary data model, we compare the performance with SVM, Random Forest, and CNN in aspects of accuracy and training time.

Model	Binary Data	SVM	Random Forest	CNN
Accuracy	0.87	0.94	0.97	0.92
Training time	0.3382	0.0339	0.0872	5.1250

Table 1 Accuracy and training time with SVM, RF, CNN. For binary data: m=100, l=2. For SVM: kernel=’RBF’, gamma=’scale’. For Random Forest: estimators=10, cv=5. For CNN: batch_size=60, epoch=100, n_convolution_layer=2, n_pooling_layer=2, nb_classes=2.

From the perspective of accuracy, we found that Random Forest and SVM have better accuracy in classifying classes. However, we also need to realize that these two models are almost fixed given a dataset, while Binary Data and CNN can architect more complex construction to improve their performance.
The comparison of training time shows a great advantage for the non-iterative method. Models like CNN which use gradient descent, usually need an iterative process and take a long time to seek for a local minimum solution. But it is obvious that SVM and Random Forest still spend less time on training data than Binary Data Model. There are 2 main reasons for this phenomenon: 1. Despite iterative process, Binary Model will spend a number of time fitting layers' patterns. 2. Traditional models like SVM and Random Forest have been packaged into the existed library, by which means we can use straight API command to generate model and train datasets. It is much faster than we implement an original model and train it. We positively believe that if there already exists a package of Binary Data Model, its training time will be similar to Random Forest, even SVM, because this model doesn't use the ensemble technique and straightly return the results.

Question 4: What is the best layer for the model

The paper provides us a simple but efficient method for classification. We wonder whether this model can construct with multi-layers to obtain a more accurate result. However, as the number of layers increases, there are two problems inevitable: parameter overloading, and overfitting.
For the first problem, we need to acknowledge that this model only uses a limited time to train datasets in shallow layers and has a satisfying recognition result. When we choose the model with more layers, the number of identified sign patterns (2n) grows rapidly. There exists a problem of exponential explosion of parameter. In the practical training, when the number of layers goes up to 6, CPU is not quantified for this model, and the computer needs to have a large memory to store intermediate variables. But from our current result, multi-layer does make a contribution to improving accuracy. The error ratio is 13.3, 13.61, 12.45, 12.05, 11.0 respective to layers 1,2,3,4,5. We can see an obvious decline in the error ratio.
On the other hand, overfitting is also a problem which we have to consider with multi-layer Binary Data Model. Although the error ratio decreases with more layers, the possibility of overfitting increases. To solve the overfitting of Binary Data Model, we can use early stopping and regularization, but not a dropout. The reason is similar to our ensemble idea. During the fitting procedure, r (l, I, t, g) has contained an idea of dropout. With the restriction of hardware, currently, we can only do these researches for this paper. Future directions of this work include solving overfitting problems and choose a proper number of layers for classification.

Future Work to Optimize The Model

We think out two potential aspects trying to improve the model's performance. Constructing a deeper model which is similar to the neural network. The model uses an A matrix to change the original dimension of dataset and try to find a linear-classifiable space to solve classification problem. It is an automatic idea to create multi-kernel and use a nonlinear function to deepen this model for each kernel. According to this idea, we can add certain numbers of the kernel before generating a final Q matrix. Each kernel is generated as A, to have independent identically distributed standard Gaussian entries. And after inner product of the kernel and input, we use a sigmoid matrix here to transform the elements

into -1 and 1. Because this model need not iterate, there does not exist a risk of vanishing gradient problem with using a sigmoid function.
However, the result is contrary to what we thought. From the figure, we can obviously see that with the increase of the number of kernels, the error ratio become bigger and closer to 50%, which is equal to the result of randomly guessing without any information. But it is reasonable because the kernel in this paper is generated from Gaussian distribution, and after several inner products, the information contained in the original dataset is gradually washed away.

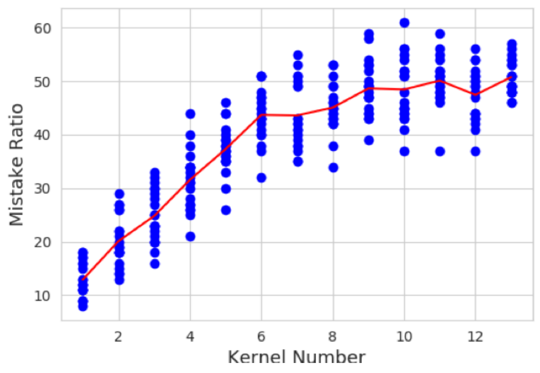

Figure 2. Trend for mistake ratio with increasing of kernel number

Secondly, we try to optimize the model from the ensemble idea. Like the bagging method, we collect a subset of Q matrix to train the dataset, as a weak model. And then, recur this process for a certain time (we set). And finally, we combine these weak models together as a strong model. For the final definition of class, we use the 'vote' method, choosing the most frequent class in weak models. For this Random Forest kind-of method, the results for error ratio is {14.0, 14.6, 14.2, 13.5}. it seems that the list of result shows a slight improvement for classification, but from a statistical test, we cannot reject the null hypothesis H0: the ensemble method doesn't improve the model. the reason is possible that r (l, I, t, g) is calculated from the number of sign patterns, which is similar to 'vote' idea.

Conclusion

This supervised classification algorithm provided might be helpful in analyzing multi-level type algorithms. However, we suspect the geographical information loss will limit the performance of this algorithm, which has been reflected in the model accuracy of the MNIST dataset. Meanwhile, the comparison between this algorithm and the traditional machine learning methods implies that traditional methods provided slightly higher accuracy while CNN takes significantly more time than the others. Thus we may believe that the new algorithm provided is acceptable in terms of accuracy, while most of its advantages come from simplifying the process and shortening the training time. Moreover, overfitting might be an issue of this algorithm resulting by improper layer selection.

References

[1] Deanna Needell, Rayan Saab, and Tina Woolf. Simple Classification using Binary Data. Journal of Machine Learning Research 19 (2018) 1-30

[2] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. IEEE Intelligent Systems and their Applications, 13(4):18–28, 1998.

Appendix

Below is the part of code for Question 4: What's the best layers for the model. And to find out more code for other aspects we try to optimize, such as the method of constructing deeper and wider models , please fork us in GitHub: https://github.com/HisonEdc/BinaryDataClassification

```
import numpy as np
import pandas as pd
import itertools
import random

# set global variable
M = 100
P = 50
L = 6
RUNTIMES = 5

# get MNIST dataset
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('.../MNIST_data', one_hot=True)

# count the sample number of each datasets
train_nums = mnist.train.num_examples
validation_nums = mnist.validation.num_examples
test_nums = mnist.test.num_examples

# get train_data,validation_data,test_data
train_data = mnist.train.images      # (55000, 784)
val_data = mnist.validation.images   # (5000, 784)
test_data = mnist.test.images        # (10000, 784)

# get each dataset's label, which is [0,1,...,0,0], 1 by 10 matrix
train_labels = mnist.train.labels      #(55000,10)
val_labels = mnist.validation.labels   #(5000,10)
test_labels = mnist.test.labels        #(10000,10)

# for this code, we simply realize the model of this paper
# get model dataset class0-class9 from train_data
train_class0 = train_data[np.where(train_labels[:, 0] == 1)]   # (5444, 784)
train_class1 = train_data[np.where(train_labels[:, 5] == 1)]   # (6179, 784)

# considering only 2 digit classes: 0 and 1, each for 100 samples
# randomly get model dataset class0 and class1 from train_data
sub_train_class0 = train_class0[:P, :].T   # (784, 100)
```

```
sub_train_class1 = train_class1[:P, :].T   # (784, 100)

mistake_list = []
for runtime in range(RUNTIMES):
   # generate kernel matrix A (m by n), which n = 784, and here we choose m
= 50
   # take A to have independent identically distributed standard Gaussian
entries
   A = np.random.randn(M, 784)

   # inner product of A and X, then use sigmoid function to identify all
elements as 0 or 1
   Q_class0 = np.dot(A, sub_train_class0)  # (50, 100)
   Q_class1 = np.dot(A, sub_train_class1)  # (50, 100)

   Q_class0 = np.where(Q_class0 >= 0, 1, -1)
   Q_class1 = np.where(Q_class1 >= 0, 1, -1)

   Q = [Q_class0, Q_class1]

   """
   here, we start to construct L layers, each layer has M sets,
   we use combination to random choose m set for each layer.
   """
   # generate L layers combinations in a list
   layer = []
   for i in range(L):
       all_combination_layer = []
       for i in itertools.combinations(range(M), i+1):
           all_combination_layer.append(i)
       layer.append(random.sample(all_combination_layer, M))

   """
   # the next step we need to do is to calculate r(l, i, t, g)
   # this index has 4 dimensions, we need to calculate r for different layers,
selection sets, sign patterns and class
   # as different layers have different numbers of sign pattern, so we divide
r into the layers'number(2) sets
   r_layer1 = np.zeros((m, 2, 2))   # the parameter is 'i' sets, 't' sign
patterns, and 'g' classes
   r_layer2 = np.zeros((m, 4, 2))

   # p is the number of training points from 'i'-th set, 't'-th sign pattern,
and 'g'-th class
   p_layer1 = np.zeros((m, 2, 2))
   p_layer2 = np.zeros((m, 4, 2))
   """
   # collect all r_layers and p_layers in 2 dicts
   dict_r_layers = {'r_layers': None}
   dict_p_layers = {'p_layers': None}
   for lth in range(L):
       dict_r_layers[lth] = np.zeros((M, np.power(2, lth + 1), 2))
       dict_p_layers[lth] = np.zeros((M, np.power(2, lth + 1), 2))
   # firstly, generate a enumeration dict which contains all possible pattern
of l-th layer
   def enumeration(length):
       enum_list = []
       fds([], length, enum_list)
       return enum_list

   def fds(sub_list, length, enum_list):
       if len(sub_list) == length:
           enum_list.append(sub_list)
           return
       for i in [-1, 1]:
           fds(sub_list + [i], length, enum_list)

   enum_dict = {'enum_dict': None}
   for lth in range(L):
       enum_dict[lth] = enumeration(lth + 1)

   # then, we started to fill the l layers' value r
   for lth in range(L):
       # get l-th layer's enumeration list
       enum_list = enum_dict[lth]

       # fill r_layer1 matrix
       def count_numbers_layer(set, pattern, clas):
           lambda_layer_set = Q[clas][list(layer[lth][set])].T
           count = 1
           for i in range(len(lambda_layer_set)):
               if (lambda_layer_set[i] == enum_list[pattern]).all():
                   count += 1
           return count

       for i in range(M):
           for t in range(np.power(2, lth + 1)):
               for g in range(2):
                   dict_p_layers[lth][i,  t,  g]  =  count_numbers_layer(i,  t,
g)

       for i in range(M):
           for t in range(np.power(2, lth + 1)):
               for g in range(2):
                   minus = []
                   for j in range(2):
                       minus.append(abs(dict_p_layers[lth][i,   t,   g]   -
dict_p_layers[lth][i, t, j]))
                   dict_r_layers[lth][i, t, g] = dict_p_layers[lth][i, t, g]
* sum(minus) / np.power(sum(dict_p_layers[lth][i, t]), 2)

   # classification: testing
   # for this example, we classify 50 images per digit class
   test_class0 = test_data[np.where(test_labels[:, 0] == 1)]    # (980, 784)
   test_class1 = test_data[np.where(test_labels[:, 5] == 1)]   # (1135, 784)
   test_class2 = test_data[np.where(test_labels[:, 2] == 1)]   # (1032, 784)

   # considering only 2 digit classes: 0 and 1, each for 100 samples
   # randomly get model dataset class0 and class1 from train_data
   sub_test_class0 = test_class0[:50, :].T   # (784, 50)
   sub_test_class1 = test_class1[:50, :].T   # (784, 50)

   # inner product of A and X, then use sigmoid function to identify all
elements as 0 or 1
   Q_test_class0 = np.dot(A, sub_test_class0)  # (50, 50)
   Q_test_class1 = np.dot(A, sub_test_class1)  # (50, 50)

   Q_test_class0 = np.where(Q_test_class0 >= 0, 1, -1)
   Q_test_class1 = np.where(Q_test_class1 >= 0, 1, -1)

   Q_test = np.hstack((Q_test_class0, Q_test_class1))  # (50,100)

   test_predict = np.zeros((100, 2))
   for k in range(Q_test.shape[1]):
       sample = Q_test[:, k]
       for i in range(len(Q_test)):
           for lth in range(L):
               # get l-th layer's enumeration list
               enum_list = enum_dict[lth]
               # accumulate b value of l-th layer
               t_index = enum_list.index(list(sample[list(layer[lth][i])]))
               test_predict[k] += dict_r_layers[lth][i, t_index]

   test_predict_class = np.argmax(test_predict, axis=1)

   true_class = [0] * 50 + [1] * 50
   mistake_count = 0
   for i in range(100):
       if test_predict_class[i] != true_class[i]:
           mistake_count += 1
   mistake_list.append(mistake_count)
print('the list of mistakes in the recursive process is {}, average mistakes
per loop is {}'.format(mistake_list, sum(mistake_list) / RUNTIMES))
```