Neural Network & Genetic Algorithm: Input and Output Encoding

KANGJIE YIN

College of Engineering and Computer Science,
Australia National University

Abstract. When training a neural network (NN) [1], we prefer unencrypted data because we do not want data encoded by other previous users. In some situation, for a single attribute, there may be too many instances of one class and very few of others. Firstly, we implement a basic neural network and use it as the fitness function to apply Genetic Algorithm (GA) [2], and GA plays a role in selecting input features. Secondly, we apply input and output encoding technique to improve performance. Finally, the result of our neural network will be compared to that of others' experiment.

Introduction

To train a neural network better, the representation of raw data matters. But what should be done in pre-process and encode stage to improve the performance of the neural work on the data? Thus, the result of neural network with basically pre-processed data will be compared to that of neural network with encoded data using a particular method. The decisions we made will be based on the analysis of raw data and a classification problem will be solved based on chosen data set. After the neural network is trained, GA will help us to select features to find relatively best combination of input attributes.

The raw data is from mushroom records drawn from the Audubon society field guide to north America mushrooms [3]. This data set includes 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species will only be edible or poisonous, that is, two classes totally. There are 8124 instances in this data set and each instance has 22 attributes. Only in attribute "stalk-root" data missing exists. All data is nominal instead of numeric. This data set is appropriate to do classification considering its large number of instances and encrypted data.

Neural Network

Data Preparation

To solve this classification problem, a neural network will be implemented. But before the implementation, all data will be pre-processed and encoded. The first column of this data set is target column which represents the class of a mushroom, and 22 attribute columns follow. Firstly, we move the first column to the last. This avoids mistaking the index of attributes and makes it easier to operate data. Secondly, in all 8124 instances, there are 2480 instances with missing data in column “stalk-root”. Given that all data is nominal, so it is hard to fill missing data through substitution methods. Here we do complete case analysis by using only rows with all the values and all instances with missing data will be dropped. Thirdly, because the input of a neural network should be numeric, we need to encode the data. Take the first attribute ‘cap-shape’ as an example, there are 5 categories in it which are ‘bell=b’, ‘conical=c’, ‘convex=x’, ‘flat=f’, ‘knobbed=k’ and ‘sunken=s’ respectively and they are all recorded as the right part of equal mark. Then, we encode these categories with 0, 1, 2, 3 and 4 in sequence and this makes inputs numeric. This encoding technique will be applied on all attributes.

Implementation

The neural network will be implemented in PyTorch. Firstly, we define several hyper parameters (number of inputs, number of classes, training epochs, and learning rate) for this neural network:

```
input_neurons = n_features  #equals to 22 here
hidden_neurons = 35
output_neurons = 2
learning_rate = 0.001
num_epochs = 1000
```

Then we define a customised neural network structure and apply a linear transformation from input layer to hidden layer and then to output layer. In forward function, we define a process of performing forward passing and get output data and predicted values after it is done:

```
class TwoLayerNet(torch.nn.Module):
   def __init__(self, n_input, n_hidden, n_output):
      super(TwoLayerNet, self).__init__()
      self.hidden = torch.nn.Linear(n_input, n_hidden)
      self.out = torch.nn.Linear(n_hidden, n_output)

   def forward(self, x):
      h_input = self.hidden(x)
      h_output = F.sigmoid(h_input)
      y_pred = self.out(h_output)
      return y_pred
```

Genetic Algorithm

Implementation

Genetic Algorithm is implemented in PyTorch. We use the trained neural network as the fitness function, and the result of this function is the accuracy of the neural network. Our aim is to select the neural network with relatively largest accuracy and its feature combination. We will initialise a few identical chromosomes and each chromosome represents one combination of features. For example, chromosome [1,0,1,0,0,1] means only the first, third and sixth feature will be trained by neural network. Several hyper parameters are defined:

```
DNA_SIZE = n_features    # number of bits in DNA
POP_SIZE = 20           # population size
CROSS_RATE = 0.8         # DNA crossover probability
MUTATION_RATE = 0.01    # mutation probability
N_GENERATIONS = 40       # generation size
```

Then we define non-zero fitness function, population select function, gene crossover function and mutation function. Population select function is based on fitness value, and population with higher fitness value has higher chance to be selected:

```
def get_fitness(prediction):
    return prediction + 1e-3 - np.min(prediction)
def select(pop, fitness):
    idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE,
    replace=True, p=fitness/fitness.sum())
    return pop[idx]
def crossover(parent, pop):
    if np.random.rand() < CROSS_RATE:
        # randomly select another individual from population
        i = np.random.randint(0, POP_SIZE, size=1)
        # choose crossover points(bits)
        cross_points = np.random.randint(0, 2,
        size=DNA_SIZE).astype(np.bool)
        # produce one child
        parent[cross_points] = pop[i, cross_points]
    return parent
def mutate(child):
    for point in range(DNA_SIZE):
        if np.random.rand() < MUTATION_RATE:
        child[point] = 1 if child[point] == 0 else 0
    return child
```

Evaluation

The data set will be divided into two parts: 80% of it will be used to do training and 20% is for testing. After all chromosomes are initialized, we build a neural network for each chromosome to do training and testing, and we will use the testing accuracy to do evaluation. Compared to training accuracy, testing accuracy is more suitable to represent a neural network's ability to do classification when meets new dataset.

Under all hyper parameters set above, this neural is trained and tested with the pre-processed data. Table 1 shows the results of accuracy of five independent tests.

Test Number	Chromosome	Accuracy

Test 1	00100111000001001011110	88.44%
Test 2	11001010000011001011110	89.82%
Test 3	00010011101011110010110	87.97%
Test 4	10001100011000001110100	90.80%
Test 5	10001010011100100000111	90.43%

Table 1

The chromosome in the last generation will be taken as the most fitness DNA and its corresponding accuracy indicates the performance of our NN & GA technique. It can be calculated from Table 1 that the average accuracy is 89.49%.

Method

To improve the of performance of NN & GA, an input and output encoding technique will be implemented [2]. Based on the analysis of raw data, encoding decisions will be made.

Data Analysis (Examples)

Figure 1 shows the frequency of "spore-print-color" and this feature has nine types: black = 'k', brown = 'n', buff = 'b', chocolate = 'h', gray = 'g', green = 'r', orange = 'o
, pink = 'p', purple = 'u', red = 'e', white = 'w', yellow = 'y'. The data is nominal value and there is no particular distribution, three types ('k', 'n' and 'h') are common and others are rare. Patterns with output 1 (encoding of 'n') similar to those with output 3 (encoding of 'h') may result in wrong output 2 (encoding of 'b').

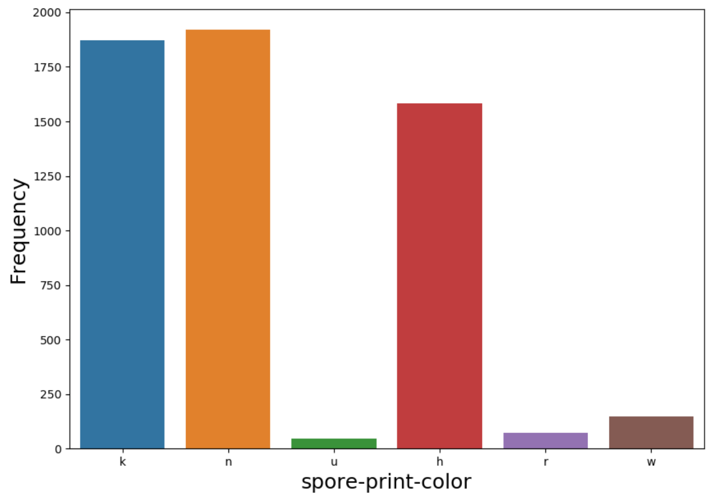

Figure 1

It can be seen from Figure 2 that although feature veil-type should have had two types (partial = 'p' and universe = 'u'), only 'p' exists among all instances. That is, this attribute is redundant.

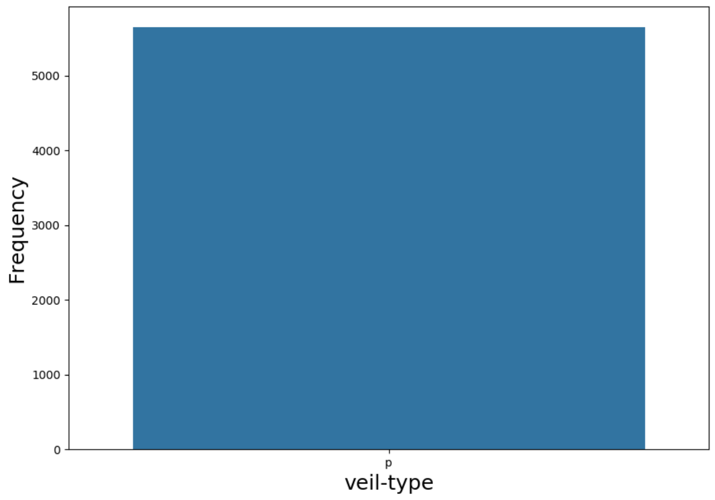

Figure 2

Attribute ‘population’ has six types: ‘a’ = abundant, ‘c’ = clustered, ‘n’ = numerous, ‘s’ = scattered, ‘v’ = several and ‘n’ = solitary. Although it is nominal, we can still say “abundant” > “clustered” > “numerous” > “scattered” > “several” > “solitary” according to the actual meanings of these words. Figure 3 shows the frequency of attribute population

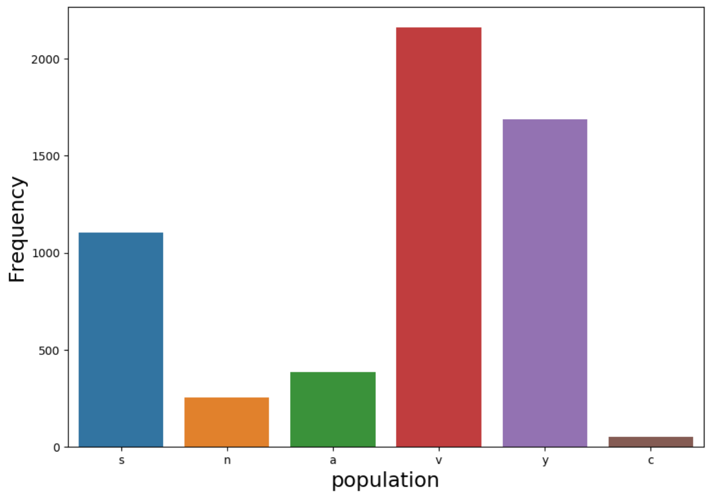

Figure 3

For attribute cap-shape, the distribution of it is similar with that of spore-print-color. There are two common types which are convex = ’x’ and flat = ’f’ respectively. Other types are rare.

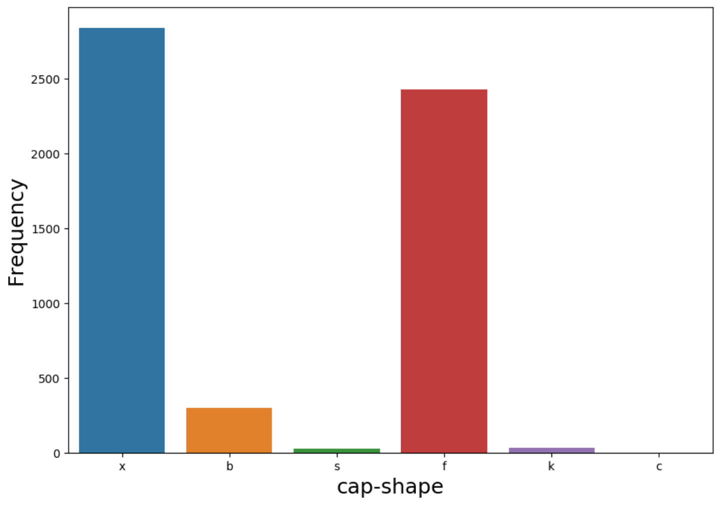

Figure 4

Encoding Decisions

For attribute spore-print-color, consider that four types are common and others are rare, a single continuously value input is not appropriate. We represent this attribute with four inputs. Therefore, these three common types will be distinguished from others. The input vector these inputs would be sparse and information may be lost in some degree. Table 2 indicates each type with five inputs, and all unmarked activations are 0.1.

VALUE	G1	G2	G3	G4
k		0.9		
n			0.9	
u	0.9			
h				0.9
w	0.9			
r	0.9			
o	0.9			
y	0.9			
b	0.9			

Table 2

For attribute veil-type, because it only has one value ‘p’ among all instances, it contributes nothing when we train or test the neural network with data set having this attribute. Therefore, we decide to drop this column.

For attribute population, we encode all types of it as Table 3 shows. If we want to measure the weight of ‘abundant’ and that of ‘solitary’, ‘abundant’ should have larger numeric value than that of ‘solitary’. Because ‘abundant’ means more than ‘solitary’ considering meanings of these two words. To avoid ‘not exist’ situation (‘solitary’ > ‘not exist’), we encode attribute population from 1 to 6 instead of starting from 0.

VALUE	ENCODING
solitary	1
several	2
scattered	3
numerous	4
clustered	5
abundant	6

Table 3

For attribute cap-shape, we use the same encoding method of spore-print-color to process it. Table 4 shows the encoding in detail and all unmarked activation are 0.1.

VALUE	G1	G2	G3
b	0.9		
c	0.9		
x		0.9	
f			0.9
k	0.9		
s	0.9		

Table 4

Here is example code of encoding attribute 'cap-shape':

```
data.insert(1,'01',dataf[0])
data.insert(2,'02',dataf[0])
data[0].replace(['b','c','x','f','k','s'],[0.9,0.9,0.1,0.1,0.9,0.
9],inplace=True)
data[1].replace(['b','c','x','f','k','s'],[0.1,0.1,0.9,0.1,0.1,0.
1],inplace=True)
data[2].replace(['b','c','x','f','k','s'],[0.1,0.1,0.1,0.9,0.1,0.
1],inplace=True)
```

Output Encoding

It is difficult for the neural network to do classification since the output vectors are sparse. We introduce equilateral encoding [4] to encode outputs. Equilateral encoding represents n possibilities with n-1 units, which is 1 less than 1-of-n encoding. Each set of units should have an equal Euclidean distance from the others. This requirement can guarantee that all wrong choosing will have the same error weight. For example, if choosing class 3 is correct, then choosing class 2 and choosing class 4 will have the same error weight. Table 4 is generated by equilateral encoding based on our data set. As the Table 5 indicates, 2 possibilities are represented by 1 unit (1 less unit than 1-of-n encoding). Because there are only two classes, the Euclidean distance must be the same.

Category	Unit 1
Edible	0
Poisonous	1

Table 5

Calculating Euclidean distance between the output vector and above values, and optimise the category which has the minimum distance with output vector:

$$\text{Output category} = \text{MIN (distance } (U_i)) \qquad (1)$$

$$\text{Where } distance(U_i) = \sqrt{\sum_{j=1}^{1} (y_j - u_{j_i})^2}$$

$$\text{Where } U_i = (u_{j_e}, u_{j_p})$$

Results and Discussion

After the technique is implemented, the number of columns of the encoded data set becomes 26 from 22. We still use 80% of the data set to do training and use rest of it to do testing. Table 6 shows the result of testing, and these five tests are independent.

Test Number	Chromosome	Accuracy
Test 1	01101100011000101001110100	66.24%
Test 2	10101110000011101111110111	69.61%
Test 3	01111111101100101001010011	67.94%
Test 4	00111100000110011110010100	71.00%
Test 5	01111011010000101101101001	65.92%

Figure 6

We can observe from the result that this technique does not help a lot to our NN & GA and even results in worse classify ability of recognizing poisonous mushrooms. The reason of this is likely to be output having only two classes, thus equilateral encoding does not perform well or even has negative effects. Another reason may be that because there are 26 input attributes, it is impossible to explore all 2^{26} chromosomes. Therefore, initialising population plays significant role in selecting features. If the selected features in initialising step do not contribute a lot, the result after several generations may not be satisfactory enough.

There is another experiment conducted in 1997 [5], which used this mushroom data set as well. The method is to simplify the neural network to guarantee clear logical functions performed by the network. It transfers multi-layered perceptron (MLP) to a logical network smoothly and use structural learning with forgetting (SLF) approach to obtain a single rule:

edible if odor = (almond $\vee$ anise $\vee$ none) $\wedge$ spore-print-color = $\neg$green

Only two attributes and four antecedents are used. This rule results in 48 errors and 99.41% accuracy, which is much more ideal than our NN & GA technique. The rules obtained by the algorithm is ordered, starting from most often used rules to rules that handle only a few cases. In this way, it is easier and faster to find the optimized rule.

Conclusion and Future Work

We have implemented input and output encoding technique to try to improve our neural network, but it still did not perform well. Consider that all input data are nominal, more effort needs to make to find the best encoding methods of data suitably for the network. Missing data needs to fill appropriately instead of being dropped directly, because dropping may lead to the change of distribution of some attributions. Furthermore, increasing the generation size of GA may let chromosomes have more possibility to mutate and become more appropriate for neural network, but it will be time consuming relatively. Additionally, pattern reduction techniques can be applied to avoid losing too much information.

Reference

1. Schalkoff, R. J. (1997). *Artificial neural networks* (Vol. 1). New York: McGraw-Hill.
2. Houck, C. R., Joines, J., & Kay, M. G. (1995). A genetic algorithm for function optimization: a Matlab implementation. *Ncsu-ie tr*, *95*(09), 1-10.
3. Schlimmer, J. (1981). Mushroom records drawn from The Audubon Society field guide to north American mushrooms. *GH Lincoff (Pres), New York*.
4. Bustos, R. A., & Gedeon, T. D. (1995). Decrypting Neural Network Data: A GIS Case Study. In *Artificial Neural Nets and Genetic Algorithms* (pp. 231-234). Springer, Vienna.
5. Duch, W., Adamczak, R., & Grabczewski, K. (1997). Extraction of crisp logical rules using constrained backpropagation networks.