N-Tuple Classifier for Devanagri Digits

Prepare data

Load the dataset into a variable called data. Here data is assumed to have records as rows and the feature values as columns. The final column is the target class in numerical form. In our example, this translates to having 20,000 rows and 1024+1 columns.

```
load deva_final.mat data;
```

Check if the data is loaded properly by printing out a section of the data along with it's attributes.

```
whos data;
```

```
  Name          Size                 Bytes  Class     Attributes

  data      20000x1025            164000000  double
```

```
disp(data(100:115,800:810));
```

```
     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     1     1     1     1     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     1     1     1     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     0     1     1     1     0
```

Define the number of classes K.

```
K = 10;
```

Quantize data into L levels: 0,1,...,L-1

Decide the number of levels L. As a simpler and more prevalent use-case for image-related sets, we assume that all features will be quantized into the same number of levels.

```
L = 2;
```

Choose the type of quantization. At the moment we go with simple quantization with equal divisions. Such a quantization has been assigned type '1'.

```
type = 1;
```

In the event of equal divisions, the quantization algorithm needs the minimum and maximum feature values. In our case they are 0 and 255. These minimum and maximum values are passed as a vector into a variable d_range.

```
d_range = [0 1];
```

Then we call the quantization function to assign the correct quantized values corresponding to each entry in the data matrix.

```
[data,level_vals] = n_quantize(data,d_range,L,type);
```

Then we check the portion of data to see if it has been successfully quantized.

```
disp(data(100:115,800:810)); disp(level_vals);
```

```
     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     1     1     1     1     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     1     1     1     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     1     1     0     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     0     1     1     1     0

     0     1     1
```

Additionally the levels used by the quantization algorithm are stored in level_vals.

```
disp(level_vals);
```

```
     0     1     1
```

Train-Test-Crossval Splitting

As described earlier, typical workflow in most classification algorithms involves splitting the dataset into three portions. In the basic implementation we only use the training and testing sets.

```
train_perc = 0.75;
test_perc = 0.25;
crossval_perc = 0;
[train,test,crossval] = n_ttc_split(data,train_perc,test_perc,crossval_perc);
```

We can check if the splitting was indeed according to the proportion specificed:

```
whos train test crossval;
```

```
  Name              Size                    Bytes  Class     Attributes

  crossval          0x1025                      0  double
  test           5000x1025               41000000  double
  train         15000x1025              123000000  double
```

Create tuples

Decide the length (N) of each tuple. In this example we assume that all tuples are of equal size. Although mathematically we are allowed to have tuples of varying sizes, we stick to uniform sizes for simplicity in implementation.

```
N = 10;
```

Decide the number of tuples.

```
T = 100;
```

Decide the number of features, or just let the algorithm figure out on it's own using the size of variable data.

```
F = size(data,2)-1
```

```
 F = 1024
```

Create tuples using the previously declared variables.

```
tuples = n_make_tuples(N,T,F);
```

Check if the tuples have been created properly.

```
whos tuples;
```

```
  Name          Size             Bytes  Class     Attributes

  tuples      100x10              8000  double
```

Print out a portion to check the values.

```
disp(tuples(3:13,1:7));
```

```
         724          84         425         887         291          49         363
         186         670         647         518         212         467         173
         725         649         414         323         274         782         641
         123         177         839          29         929         958         367
         979          24         514         698         762          12         969
          35         610         883         494         684         912         182
         629         738         178        1013         112         175         480
```

```
    82     589     216     597     336     948       9
    83     821     737       5     125     916     651
    14     643     578     793     429     581     916
   342     219      38     354     338     374     631
```

Create memory structure

To create a memory structure with all the K x T tables, as mentioned earlier we need to be careful of how the values in these table would be combined. The starting value for the table (val), if the product rule will be used, should be non-zero. We choose to go with 1. We will later how this value affects the results.

```
val = 1;
```

We create the memory structure using the previously supplied information. The variable mem is a 1x100 cell array, with each cell having 10 cells. Each of these 10 cells has 1024 values.

```
mem = n_create_memory(N,T,L,K,val); whos mem;
```

```
  Name      Size              Bytes  Class    Attributes

  mem       1x100           8315200  cell
```

Check values in mem. Printing out first ten values of the table corresponding to tuple t = 4, and class k = 8. They should be all asigned equal to the value specified in the variable val.

```
disp(mem{4}{8}(1:10));
```

```
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
```

Tabulate probability values (Training)

The training function loops through all the observations and tuples to tabulate the values for our memory tables. The tic-toc function is used to keep track of the training time.

```
tic; mem = n_tuple_train(train,tuples,mem,N,T,L,K); toc;
```

```
Tuple number: 1
Tuple number: 2
Tuple number: 3
Tuple number: 4
Tuple number: 5
Tuple number: 6
Tuple number: 7
Tuple number: 8
Tuple number: 9
```

```
Tuple number: 10
Tuple number: 11
Tuple number: 12
Tuple number: 13
Tuple number: 14
Tuple number: 15
Tuple number: 16
Tuple number: 17
Tuple number: 18
Tuple number: 19
Tuple number: 20
Tuple number: 21
Tuple number: 22
Tuple number: 23
Tuple number: 24
Tuple number: 25
Tuple number: 26
Tuple number: 27
Tuple number: 28
Tuple number: 29
Tuple number: 30
Tuple number: 31
Tuple number: 32
Tuple number: 33
Tuple number: 34
Tuple number: 35
Tuple number: 36
Tuple number: 37
Tuple number: 38
Tuple number: 39
Tuple number: 40
Tuple number: 41
Tuple number: 42
Tuple number: 43
Tuple number: 44
Tuple number: 45
Tuple number: 46
Tuple number: 47
Tuple number: 48
Tuple number: 49
Tuple number: 50
Tuple number: 51
Tuple number: 52
Tuple number: 53
Tuple number: 54
Tuple number: 55
Tuple number: 56
Tuple number: 57
Tuple number: 58
Tuple number: 59
Tuple number: 60
Tuple number: 61
Tuple number: 62
Tuple number: 63
Tuple number: 64
Tuple number: 65
Tuple number: 66
Tuple number: 67
Tuple number: 68
Tuple number: 69
Tuple number: 70
Tuple number: 71
Tuple number: 72
Tuple number: 73
Tuple number: 74
```

```
Tuple number: 75
Tuple number: 76
Tuple number: 77
Tuple number: 78
Tuple number: 79
Tuple number: 80
Tuple number: 81
Tuple number: 82
Tuple number: 83
Tuple number: 84
Tuple number: 85
Tuple number: 86
Tuple number: 87
Tuple number: 88
Tuple number: 89
Tuple number: 90
Tuple number: 91
Tuple number: 92
Tuple number: 93
Tuple number: 94
Tuple number: 95
Tuple number: 96
Tuple number: 97
Tuple number: 98
Tuple number: 99
Tuple number: 100
Elapsed time is 14.976091 seconds.
```

Check values in mem. Printing out first ten values of the table corresponding to tuple t = 4, and class k = 8. They should now reflect the conditional probabilities of finding the tuple t=4 in an observation belonging to class k = 8.

```
disp(mem{4}{8}(1:10));
```

```
    0.0073
    0.0079
    0.0112
    0.0013
    0.0145
    0.0462
    0.0073
    0.0086
    0.0020
    0.0040
```

Adjust values, cross-validation, combining classifiers, etc

This part of the code will typically deal with modifying the original N-Tuple scheme using the cross-validation data. Since this is a basic implementation, we move on to the testing procedure directly for the time being.

Test the classifer

While testing the N-Tuple classifier, one of the most important decisions to be made is the strategy for combination of scores. We use the variable rule (as 1 or 2) to decide the combination rule. The sum rule with these values of N and T gives ~82% accuracy, while the product rule performs brilliantly with the Devanagri set ~92% accuracy.

```
rule = 2;
```

Calculate the scores for the test set.

```
test_scores = n_tuple_test(test,tuples,mem,rule,N,T,L,K);
```

Check the test_scores for the first observation in the test set.

```
disp(test_scores(1,:)');
```

```
  1.0e-143 *

    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.4627
    0.0000
    0.0000
```

Predict the final class of the test data using a majority vote approach.

```
predictions = zeros(size(test,1),1);
    for m = 1:size(test,1)
        [~,predictions(m)] = max(test_scores(m,:));
    end
```

Let us check the predictions for the first 5 obervations in the test set.

```
disp(predictions(1:5));
```

```
     8
     8
     3
     8
     7
```

Using the predictions for the testing data, we calculate the accuracy of our N-Tuple classifier.

```
accuracy = 100*sum(predictions==test(:,end))/size(test,1);
```

Printing the accuracy:

```
disp(['Accuracy = ' num2str(accuracy) ' %']);
```

```
 Accuracy = 93.1 %
```