Optical Character Recognition

Introduction

Having no experience related to our today's subject, I had to figure out what kind of model I should use at first. Few hours spent on search through the internet I have decided to use Neural Network, as it says that there are many different approaches to optical character recognition problem, but one of the most common and popular approach is based on neural networks. Besides that, previously I had short survey related to Hopfield Neural Network, and thought that it can help me a bit if I use Neural Network for this problem.

Having no experience with Python and MATLAB, I decided to code in Java. One of the most popular machine learning libraries for Java is Weka, but I didn't like it because of less examples and descriptions. I found a book by Jeff Heaton (about 500 pages), which explains pretty well, and I decided to stick to the library (encog) he talked about in the book.

Modules and their functionalities

Neural Network

First model to consider is NN.
Package nn:
Classes NN_Model

Constructors and Description
NN_Model (int hiddenNeurons, int layersNumber)

The input and output neurons numbers are specified in Config class, and these are 400 and 26 respectively.
The main idea is that we should first load the data, and normalize it. Next step is to prepare training step and then train a network to recognize pattern from the training set. In the training step we teach the network to respond with desired output for a specified input. For this purpose each training sample is represented by two components: possible input and desired network's output for the input. After the training step is done, we can give our test examples to the network and the network will form an output, from which we can resolve a pattern type presented to the network.

Note, the desired network's output in training sample has the following form:
Let's take letter 'C' – [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
So, as an testing output we are getting probability distribution and looking for the highest one, and by its position in the output array, we can say which letter it has to be.

NN Model Results

Below are provided the test result of NN tested with:
Train size: 3549
Test size 3563

hiddenNeurons: 200, layersNumber: 1, errorThreshold < 0.03	-	Accuracy: 36.79%
hiddenNeurons: 200, layersNumber: 1, errorThreshold < 0.02	-	STOPPED at #400
hiddenNeurons: 200, layersNumber: 1, errorThreshold < 0.027	-	Accuracy: 42.88%
hiddenNeurons: 400, layersNumber: 1, errorThreshold < 0.027	-	Accuracy: 39.18%
hiddenNeurons: 100, layersNumber: 1, errorThreshold < 0.027	-	Accuracy: 38.98%
hiddenNeurons: 100, layersNumber: 2, errorThreshold < 0.027	-	STOPPED at #400
hiddenNeurons: 1000, layersNumber: 1, errorThreshold < 0.027	-	Accuracy: 37.15%
hiddenNeurons: 300, layersNumber: 1, errorThreshold < 0.027	-	Accuracy: 39.37%

Note, all the results related to the neural network is not going to be the same as you run the second time due to weights reset and training.

I have started with error threshold less than 3%, and as it works but the result was not so good, I have changed it to 2%. However, 400 iterations it run, but didn't give me error value less than 0.026. Thus, I have continued with error threshold less than 2.7%.

From the results we can say that hidden neurons 300 and 400 does not make almost any difference. Network with 1000 hidden neurons performs even slightly bad and takes much more time to train.
Anyway, here the best one is 200 hidden neurons and only 1 layer, which gives 42.77 accuracy. Because this is not so good result, I haven't even tried with feature extraction, as later we will see that my feature extraction methods are taking out too much useful information and without it I'm getting better results.

Note, all the objects for this results are created in nn.Main class and commented, so that you can easily test it by running above mentioned class.
Because of my machine power limitations I couldn't experiment with more layers and more hidden neurons, as the CPU gets to 100% (i3 CPU M350 2.27GHz, RAM 4GB).

Support Vector Machine

Next model to consider is SVM.
Package svm:
Classes SVM_Model

Constructors and Description
SVM_Model (boolean featureExtraction, int theInputCount)
SVM_Model (boolean featureExtraction, int theInputCount, enum KernelType)

Again, the main idea is the same as above, but the desired network's output in training sample has another form. Due to encog library does not support probability distribution for SVM model, the desired network's output in training sample has to be just one value:
Let's take letter 'C' – [2]
So, there is no probability distribution, thus the testing output is in a form of letter's index in the array of alphabet.

SVM Model Results

Below are provided the test result of SVM tested with:
Train size: 3549
Test size 3563

Normalization, KernelType: RBF	-	Accuracy 72.10%
Feature Extraction, KernelType: RBF	-	Accuracy 38.67%
Normalization, KernelType: Poly	-	Accuracy 70.41%
Normalization, KernelType: Linear	-	Accuracy 18.15%
Normalization, KernelType: Sigmoid	-	Accuracy 5.69%
Normalization, KernelType: Precomputed	-	Accuracy 5.95%

From the results we can say that the default kernel type works for me as top one. Kernel type Poly in the second place, and the difference is not big.

Feature extraction interface and classes are in featureExtraction package, and constructed in the way that easily can be added another ones or deleted whenever needed. All the new feature extraction classes should implement FeMethod interface, and can be used while creating featureExtraction object.

As we can see from the results, my feature extraction is not so good, as without feature extraction and only with normalization I'm getting better result. Perhaps, I'm removing to much useful information. This is due to the fact that Java does not support good libraries for image processing. All the libraries that I have tried so far has certain type requirements for reading the

data. However, I needed to provide 1 or 2 dimensional array rather than read from file.
In fact, there is not much existing developer tools for image processing in Java.

I have implemented Horizontal Celled Projection and Vertical Celled Projection just because these ones are easy to understand and implement for person who is far from image processing.

Note, all the objects for this results are created in svm.Main class and commented, so that you can easily test it by running above mentioned class.

K Nearest Neighbors

The last model to consider is KNN.
Package knn:

Classes KNN_Model

Constructors and Description
KNN_Model (boolean featureExtraction)

Encog library doesn't provide this model or just I couldn't find, and I tried to shift to Weka for this model. The KNN model that Weka provides takes an input Instances class object, but for creating this object you should read you instances from ARFF file (Attribute Relation File Format). In other words it was another pain, and due to simplicity of this model, I decided to implement it by my own. As a result I've got kind of brute force logic in my method.

KNN Model Results

Below are provided the test result of KNN tested with:
Train size: 3549
Test size 3563

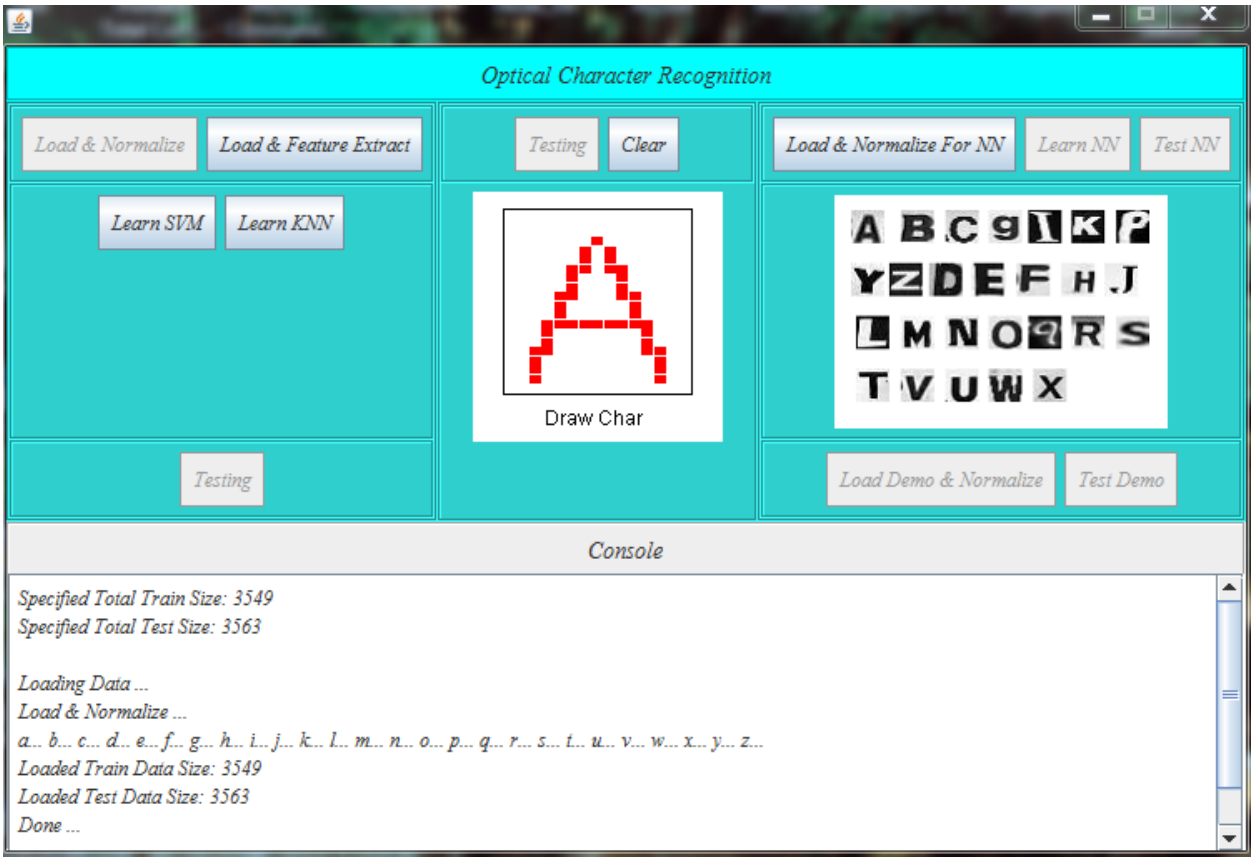
Normalization	-	Accuracy 65.84%
Feature Extraction	-	Accuracy 66.34%

Here, we can see that my feature extraction methods instead of decreasing the accuracy like in SVM example, they are increasing a little bit.

GUI

Package gui

To combine all the results, I have created GUI interface, which consists of 4 parts.



Left panel is used to load the data and normalize or load data and extract feature, then Learn SVM or KNN, and then test the learned network.

The middle panel provides window for drawing some character and then testing it with learned network. As you can see, testing button is not active here, unless you have trained network.

The right panel provides button for loading and normalizing for Neural Network. As it finishes, the 'LearnNN' button is activated and used to train the neural network. After this step the 'Test NN' can be used to test the testing set.
Also, the right panel provides button for loading and normalizing the picture with letters. The demo picture size that I have created from my data is 140 x 200. So, whenever I'm loading the image, I'm getting 21.600 small images as a test set. The button is activated only if you have already trained neural network. So, you can train neural network, then load the demo image and test it.

As I mentioned above, due to machine power limitations, I couldn't experiment more with different amount of hidden neurons and by adding more hidden layers.
Note, I have used neural network for 'demo picture', because with svm as a result of testing, I'm getting predicted results as a single number rather than probability distribution.

The problem is that encog library does not support probability distribution for svm model. I have spent a lot of time and have made sure that there is no such option. Here, again I tried to switch to weka, but again the same problem: model takes Instance class as an input, which you can create from reading ARFF file.

Due to this problem, here I've used neural network although the accuracy is not so good. And because the results for demo picture is not so accurate, I could not draw frame for each character.

Below are provided the test result of demo picture tested with NN:
PROB_THRESHOLD = 98.5% - Found 55 letters
PROB_THRESHOLD = 99% - Found 17 letters

PROB _THRESHOLD is used to filter and throw the results that have less probability.

Note, all the results related to the neural network is not going to be the same as you run the second time due to weight reset and training.

Finally, the last part of GUI interface provides console with all system logs.

Note, the objects for this results are created in gui.Main class and commented, so that you can easily test it by running above mentioned class.

Benefit of Proposed System

If we overcome the drawbacks, the benefit of the proposed system is that it would support multiple functionalities.

And one of the most important benefit is the experience I've got while trying to make it work ☺

Drawback of the System

Feature Extraction:
As I mentioned above, having no experience in image processing and implementing this project in Java, which does not support it that much, limited the abilities to have a good feature vector, thus raw data gives more accurate results.

Demo picture:
Because of I couldn't get the probability distribution using SVM model, I've used NN model, which accuracy is low comparing with first one, and it's not possible to filter more accurately the small images that do not continue the entire character, thus drawing frame for each found

letter is meaningless. At first, needs to create more accurate neural network.

Draw character:
This is additional feature, I've decided to add. This model is based on binary values. However, my training data is grayscale images, thus the accuracy is bad. It mostly recognizes the characters 'A', 'R', 'O' and few more. Perhaps, needs either have black and white pictures, or convert the image pixels into 1 and 0. This part is more like Hopfield Neural Network for pattern recognition where the pattern is vector of 1 and 0.

Future Enhancements

1. Needs to implement new feature extraction method, such as Sobel, or find good library which could take array as an input and return array (JAVA). As I mentioned above, the feature extraction methods I have created are removing to much useful information.

2. Solve the computer power limitation problem, create more accurate neural network. Thus, filter the images based on that probability more accurately to get those that contain character and draw a frame.

3. Create another network model for the 'draw character' (Hopfield neural Network maybe).