ASSIGNMENT 5

TDT4173 - Machine Learning and Case-Based Reasoning

Written by:

Thomas Gautvedt
Håkon Ødegård Løvdal

Spring 2016

NTNU

Norwegian University of Science and Technology

1 System overview

Our Optical Character Recognition (hereby denoted OCR) is created using a modular structure. In order to make it customizable for multiple problems and setups, it is easy to change elements like; (1) pre-processing techniques, (2) models and/or (3) the data-set loader. The OCR-object accepts inputs for all of these elements, all of which implements an abstract class to ensure demanded functionality. We make use of the following libraries in our system:

scikit-image

`skimage` is used to load images and to do data augmentation like adding random noise to all images.

scikit-learn

`sklearn` is used to provide implementations of models used for classification. Due to it's simplicity and efficiency it proved a good tool for our system.

numpy

`numpy` is used for all lists and matrices, since they are much more compact than Python lists in terms of memory. `numpy` is also considered a fundamental tools in scientific computing.

This approach proved to be very useful when we started severe tweaking of parameters and testing of the system. It became easy to change the models for classification and append different pre-processing techniques.

All the default parameters of the system is set in such a way that when all the requirements found in `requirements.txt` are installed, the code could be executed by typing `python3 main.py` in the terminal. Also remember to add all character-folders to the `data/chars74k-lite`-folder.

The first time the OCR is run, the data set and classifier will be generated and pickled to a binary, gZipped file. In order to do a classification to an image you must provide at least one image in the sub-folder `data/images`. The output files will also be stored here after each run as a *gif* with an animation of each individual classification, as well as a static image with all the windows applied. For your convenience the `data` folder consists of example images to use (move it into `data/images`). In order to change the pre-processing techniques, provide a new technique in the `pre_processing` list in `main.py`

2 Data augmentation

The provided data set consists only of 7112 images, which proved too small in order to achieve a good performance. In order to increase our performance, we chose to perform data augmentation on the data set. This both increased the size of the set, but also increased the performance of the OCR with approximately 15%. The techniques we used in this approach were: adding images with noise and shifting images one pixel in three directions.

For the noise approach we used three different types of noise: *salt and pepper*, *poisson* and *gaussian-distribution*. We initially started out with only gaussian-distribution, but the performance did not increase enough for us to be satisfied with the result. Testing all of the noise function for

	None	Move	Noise	Noise + Move
Prediction	approx. 71%	approx. 79%	approx. 80%	approx. 85%

Table 1: Comparing different data augmentation methods with kNN model

them self did not give any significant result. But adding all of the three noise types increased the performance significantly. It went from approximately 80% to 85%.

The last data augmentation technique we implemented were image shifting. This technique were purely implemented in order to increase the size of the data set. We shift/roll the images one pixel in three directions.

3 Feature Engineering

In order for our system to discriminate between the different classes/characters we tested different methods to engineer good features. Below we describe three of the methods that proved somewhat efficient.

3.1 Image Type Conversion

Our image type conversion uses the grayscale values and transforms them into binary values $\{0, 1\}$. The original grayscale values ranges from 0 to 255. To translate these values into their binary counterpart, we apply the following function:

$$f(x) = \begin{cases} \lfloor \frac{value}{255} \rfloor, & \text{if } x \leq 128 \\ \lceil \frac{value}{255} \rceil, & \text{otherwise} \end{cases}$$

3.2 Normalizing

We also tried applying normalization of the grayscale values to the images. As mentioned before, grayscale values ranges from 0 to 255. Simply diving the grayscale value on 255 gives us the normalized value for that individual pixel. The normalization has the benefit that it is easier to detect variance in the data set.

3.3 TV denoising

We tried to make use of a technique from signal processing called total variation (hereby denoted TV) denoising. This is a commonly used technique in image processing. We did not implement this functionallity our self, but rather we used a method in the *scikit-image*-library called `denoise_tv_chambolle`.

The strength in using this technique is that it removes noise in the original image, but preserves important details like edges.

4 Models

For our OCR, we tested many of the classifiers available in the `sklearn` library. Below are two of the classifiers we decided to tweak further.

4.1 k-Nearest Neighbours

The first model we decided to experiment with was the k-nearest neighbour algorithm (hereby denoted kNN). The main reasoning for this were that kNN is considered one of the simplest machine learning algorithms. We thought this were a good place to start when developing our OCR. kNN is *non-parametric* algorithm. This means that it does not make any assumptions about the data. Also kNN is a *lazy algorithm*, which means that it does not use the training data to do any generalization. The (positive) effect of this is that the training is done very fast. This we experienced early on in the development, when we added data augmentation to data set.

A disadvantage with kNN is the fact that it does not know which attribute that is the most important. kNN only computes the distance between data points using a distance metric, where every attribute is weighted equally in the total distance.

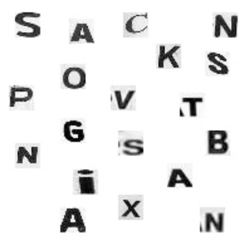
Another disadvantage of kNN we experienced early on, is that the prediction done in kNN performs slow. This is a pay-off for having a lightning fast training. If the training set is large, and no precautions are made, like using KD-trees, hashing etc. We experienced that even though the prediction took quite some time, the classification were correct surprisingly often.

For a problem like OCR and a data set like the 20×20 pixels Chars74k data set, we initially though that a classifier like kNN would have a decent performance. We mainly though this because of how the letters in the data set were very similar in both shape and nature, which works well with how kNN calculates its distances.

4.2 Decision Tree

The second model we tried were decision tree learning, and more precisely classification trees. Decision tree has a model which maps observations to conclusions about the item's target value. While decision trees are effective, they have the disadvantage that they may produce overly complicated trees that do not generalise data very well. For tasks such as text recognition, overfitting will lead to poor performance because the model is not able to detect similarity well.

During testing, decision trees never performed better than kNN, no matter how much we tweaked the parameters. The gap between the two models were at times significant too. See figure 1 for a visualization on how the two models compared against each other.


(a) Original image

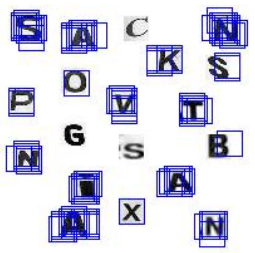
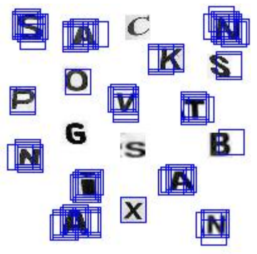

(b) Classified characters with kNN(k=21)

Figure 2: Image before and after classification

5 Evaluation


(a) kNN Output (k=21)

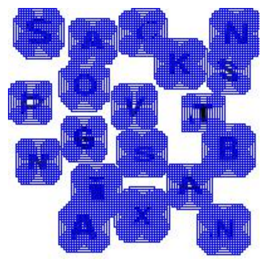
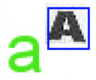
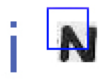

(b) Decision Tree Output (max_depth=20)

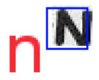
Figure 1: Comparison between kNN and Decison Tree

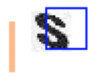
When testing our system, we ended up splitting the data set with the typical split 80/20, using 80% as the training set and 20% as the test set. We would like to point out that this split is easily configurable in the `Chars74KLoader`-configuration.

Initially we started out by using a simple sum-function in order to evaluate the correctness of the classifier. Later on we discovered built-in functions in *scikit-learn* that included much more thorough statistics (including precision and recall). In addition to this, we found a function for creating a confusion matrix for the test.


(a) P=1.0%


(b) P=0.95%


(c) P=1.0%


(d) P=1.0%

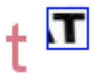

(e) P=1.0%

Figure 3: Good and bad classifications

Our final configuration ended up using Image Type Conversion kNN(k=21). See figure 2 for a visualization on how a run of the classifier turned out when applying the classifier using the sliding window technique. In figure 3 you can see three good classifications and two bad classifications.

6 Components

Our OCR successfully locates the various letters in the images because we only ignore the parts of the image that is very empty. Even sections that contains less than 30% pixels that are not 100% white are evaluated by the kNN model. The “false positives” you can see in figure 1b are derived from a kNN that is not able to correctly classify letters with a reasonable probability. As depicted in figure 3d, the kNN model predicts that the s is in fact an l with a 100% certainty, is which obviously wrong.

7 Lessons learned

During the project we encountered some problems. When implementing the Image Type Conversion, we learned later on that the conversion formula we had created did not work. Due to some images being more gray than black, our assumption that the mean value of the image had to be greater than and equal to 128 did not hold. This made the system not being able to correctly classify some gray images. An attempt to address this issue were made, but due to time limitations it did not succeed.

The structure of the system is a thing we are happy about. The structure made it easy to configure different aspects of the system for easy and fast testing.