TDT4173 Assignment 5

Optical Character Recognition

Iver Jordal

# Overview

Programming language: Python

Machine learning library: scikit-learn

Image processing: scikit-image and Pillow

## How to install

See install.md

## How to run

|  |  |
| --- | --- |
| python preprocess.py | Load data set, preprocess images, split data set, augment training data, write HDF5 file |
| python training.py | Load the HDF5 file, train a classifier, store the classifier to a file, run the classifier on the test set, print an accuracy score |
| python detection.py | Load the trained classifier, run character detection on the images inside the “detection-tests” folder, write images with rectangles and characters overlayed |
| python test\_classifier.py | Pick some arbitrary images from the data set, print them to the console and attempt to classify them |
| python test\_preprocessing.py | Pick some arbitrary images from the data set, preprocess them and create variations of them (as in data augmentation), store them in the “tmp” folder for you to inspect |

# Preprocessing

## Method 1: Noise reduction and edge detection

Seeing that the images were noisy and some were inverted (white on black vs. black on white), I thought it would be a good idea to apply some image processing that would yield appproximately the same image for a noise-free image and a noisy, inverted image.

### Noise reduction: Bilateral filter

A bilateral filter is an edge-preserving and noise reducing filter. It averages pixels based on their spatial closeness and radiometric similarity [0]

The good thing about this filter is that while it smoothes out noise, it does not blur edges.

### Edge detection: Sobel filter

This filter uses a combination of 3x3 kernels that are convolved with the image. This approximates the gradient of the image intensity.

### Result

I quickly found that noise reduction and edge detection wasn’t a very good idea. The noise reduction alone seemingly didn’t affect the predictive performance at all, while applying sobel edge detection actually hurt predictive performance.



*Figure 1: original, noise reduction, noise reduction + sobel filter*

## Method 2: Contrast stretching, rescale values

Because some images had low contrast, I thought it would be a good idea to kind of normalize those images. Also, I decided to rescale all intensity values from [0, 255] to [0, 1]. This is how the contrast stretching works:

Find the 15th and the 85th percentile to obtain the dark and the bright threshold, respectively. Then rescale the intensity values so that these two intensity thresholds become black and white, respectively. This means that

* The 15 % darkest pixels become completely black
* The 15 % brightest pixels become completely white
* The contrast is enhanced, so there is a better separation between background and foreground

### Result

Given that I use a random forest classifier with 30 trees and no data augmentation:

|  |  |
| --- | --- |
| Without contrast stretching | 62.7 % accuracy |
| With contrast stretching | **64.8 % accuracy** |

# 

*Figure 2: original, contrast stretched*

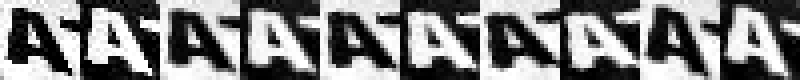
## Data augmentation: Rotation, inverting

I tried augmenting the training set by creating combinations of rotated (-10, -5, 5 and 10 degrees) and inverted images. This makes sense because some of the images in the data set are inverted, and orientation is not always perfect.

## Results

The data augmentation method significantly improves accuracy. Given that I use preprocessing method 2 (contrast stretching) and a random forest classifier with 30 trees:

|  |  |
| --- | --- |
| Without data augmentation | 64.8 % accuracy |
| With data augmentation | **72.2 % accuracy** |

  
Figure 3: Leftmost image is preprocessed (contrast is stretched). The other images are various combinations of rotation and inversion.

## Techniques I want to try

If I had to do more work on this project, I’d try Histogram of Oriented Gradients (HOG) and Scale-invariant feature transform (SIFT). Those are algorithms that can describe local features in images.

# Models

When I first looked at the dataset, I initially thought that a support vector machine would work well. But then I started doing lots of data augmentation, which makes the data set too large for the support vector machine.

## K nearest neighbours

The K nearest neighbours technique is based on finding the K images that are most similar to the image you are trying to classify. Those the most common class among those K images becomes the predicted class. This model is easy to understand, and it yields good accuracy. Although the training time is low, it takes relatively long to classify an image with this technique. The reason is that for each classification the algorithm iterates over the entire training set to find the K most similar images.

|  |  |
| --- | --- |
| 3 nearest neighbours | 78.7 % accuracy |
| 5 nearest neighbours | **79.2 % accuracy** |
| 10 nearest neighbours | 78.8 % accuracy |

## Ensemble methods

Ensemble methods are based on training multiple weak classifiers (decision trees) and combining their predictions into a single prediction. These are good because

* The training is fairly quick (many threads can be run in parallel)
* They can output probabilities for each class
* Classification execution time and accuracy is pretty good
* I tried random forest and extremely randomized trees (both with n\_estimators set to 30). I quickly found that the latter was better. I also found that increasing n\_estimators improved accuracy.

|  |  |
| --- | --- |
| Random forest, n\_estimators=30 | 72.2 % accuracy |
| Extremely randomized trees, n\_estimators=30 | 76.6 % accuracy |
| Extremely randomized trees, n\_estimators=150 | **80.7 % accuracy** |

Sadly, due to a bug in scikit-learn, I was unable to store this best classifier with joblib [1]. I had to use cPickle instead, and could only store smaller classifiers (i.e. with fewer estimators). Consequently I’ll use the extremely randomized trees with n\_estimators=30 for the character detection task.

## Additional models

I could try running the data through an artificial neural network. Perhaps a *convolutional* neural network. I’ve heard that those easily get over 90 % accuracy on this dataset.

## Evaluation

Both K nearest neighbours and extremely randomized trees worked well. The latter is best because it yields more accurate probabilities (they are based on 150 estimators) and has better accuracy on the test set. How their performance are measured: predict class for each image in the test set. The percentage of correctly classified images represents the accuracy of the classifier.

# Character Detection

The weakest component of my OCR system is the character detection. The classifier assumes that there is a character on the image and the output probabilities sum to 1. The preprocessing also enhances any low-contrast content that may be just noise or weak shadow gradient. These are both problems when using the classifier for character detection/recognition.

In the character detection problem the assumption “there is a character on the image” does not make sense, as many images will not have characters in them. As I see it there are several ways to fix this:

1. Add a class that is “not a character” and train the classifier on that. This requires extra training data. For example one could create a few thousand images of perlin noise with varying roughness.

2. Different kind of preprocessing and feature engineering. For example the amount of contrast and the amount of sharp edges on the image could be useful for checking if there’s a character or not.

3. Train n independent classifiers where the output probabilities are independent of each other (i.e. they do not necessarily sum to 1). That way each probability represents for example “what’s the probability that the image contains an A” rather than “what’s the probability that the character on the image is an A”.

~~Anyway, assuming I won’t fix all of this, here’s an evaluation of the character detection system. First, I created a few simple images that contain character images from the data set. I also added some perlin noise on one of them. The OCR is actually pretty good at finding and recognizing the characters correctly.~~

~~It’s kind of hacked together based on the classifier. The classifier that actually assumes that the image that it’s trying to classify is an image of a character. Therefore it might not work well on unseen kinds of images. The probabilities that the model outputs sum to 1, and they are therefore not absolute probabilities. There’s actually a difference between “classify this image” and “does this image contain a character, and if yes, which one”. In the latter we need to start thinking of other performance measures than accuracy. Two common measures are precision and recall. There’s a tradeoff between precision and recall. Good precision: High confidence, but may not detect all characters. Good recall: Will detect most characters, but also yield some false positives.~~

# Lessons learned

* Edge detection is not so useful, especially when the characters are bold
* Data augmentation is a good idea, but makes the training take longer
* Scikit-image is a handy library for common image processing methods
* Scikit-learn has a bug with a bad workaround [1]
* Extremely randomized trees is indeed better than random forest
* OCR is hard

# References

0: <http://scikit-image.org/docs/dev/auto_examples/plot_denoise.html>

1: <https://github.com/scikit-learn/scikit-learn/issues/6650>