DELFT UNIVERSITY OF TECHNOLOGY

PATTERN RECOGNITION, FINAL REPORT

Drawing a line

The not so trivial task of pattern recognition

A. Kosiorek, D. Snijders, R. Verschuuren
4746651, 4744098, 4712986

supervised by
Dr David TAX

February 2, 2018

Contents

1 Introduction

This report contains the results of the final assignment for the course IN4085 Pattern Recognition 2017/2018 at Delft University of Technology. For the purpose of this final assignment, a case is introduced in section 1.1, the remaining sections of the report explain our methods used and obtained results with the goal of correctly classifying handwritten digits.

1.1 Case

Assume your group plays the role of a pattern recognition consultancy company. Your company is faced with an assignment by a client company working on automatic bank cheque processing applications. This client wants you to research the possibility of using pattern recognition techniques to classify individual digits in bank account numbers and the monetary amount.

They are interested in two scenarios: (i) the pattern recognition system is trained once, and then applied in the field; (ii) the pattern recognition system is trained for each batch of cheques to be processed.
In pattern recognition terms, Scenario 1 means the amount of training data available is large, whereas for Scenario 2 it is much smaller. More concrete:

1. Scenario 1: Large. 1000 Images per class were used and the goal is to achieve at least 95% accuracy on a test dataset.
2. Scenario 2: Small. Only 10 images per class were used and the goal is to achieve at least 75 % accuracy on a test dataset.

For training purposes (and for educational matters evaluation purposes) the NIST dataset is used which is a database of handwritten digits produced by the National Institute of Standards and Technology. This dataset consists of 2800 images for each of the 10 classes: "0", "1", . . . , "9". These images were extracted from forms filled out by volunteers, thresholded in black-and-white, and segmented into 128×128 pixels.

1.2 Background

We mainly used the theory as provided in the course IN4085 Pattern Recognition and from the book Pattern Recognition [1] used alongside the course. Furthermore, documentation from the scikit-learn pages proved very helpful as well[1].

[1]http://scikit-learn.org/stable

2 Method

Before going into more details on the specific pattern recognition techniques applied, this section first gives an overview of our high-level research method and elaborate on the implemented methods used to obtain our results. A brief explanation of the implementation is also given.

2.1 Method Overview

At a very high level what we did to reach our conclusions was to a large extend based on trial and error: We started out with a variety of classifiers we learned about, starting from most basic ones to more complex. We ran them with default settings, analyzed the results, and tried some of them again with altered settings that we predicted could improve the results. These predictions were based on the theory we learned and observed trends as well as sometimes just random guesses. Later we applied various preprocessing techniques, to obtain better results.

In order to speed up this process we tried to make use of a modular approach in our python implementation, which when it comes to the pattern recognition flow roughly had the following order:

1. Loading the dataset and splitting it between training and testing data
2. Pre-process the data, consisting of:
    - 2.1. Distorting the training data to increase the amount of training data
    - 2.2. Converting the data to a (dis)similarity representation
    - 2.3. Dimensionality reduction
3. Training the classifier on the resulting dataset
4. Testing the accuracy of the classifier on the test set
5. Outputing the results to the terminal as well as a log file

The shape of the dataset as well as the pre-processing to apply can be changed at wish, all of these steps are further detailed in later sections.

2.2 Implementation

While it was hinted to use MatLab for the actual implementation, we decided to make use of Python. Our main reasons for this is that Python is free and open source as well as a popular general programming language and often applied in the field of data-science. The main drawback of choosing Python was the created need of exporting the dataset from MatLab to a .mat file, which can be read by the SciPy libraries[2]. Aside from this, we were successfully able to replace the matlab classifier functions

[2] `https://www.scipy.org/`

with classifiers from the sklearn[3] python module. Understanding the functions used to implement a certain classifier can be easily seen from the `cls_*` files, where the '*' indicates the classifier.

For a more in depth guide or help using the software see appendix A.

3 Preprocessing

In order to improve the performance of classification, prepossessing to the initial dataset is applied.

3.1 Distortions

Since properly training a classifier often relies on large amount of data available, a viable tactic to increase accuracy can be to artificially enlarge the available training data. To do so we made use of 2 distortions, techniques that artificially generate 'new' training samples based on the available real training data.

The first distortion technique we implemented is what we called the *shift* distortion. The shift distortion takes an original sample in pixel representation and moves the entire 'picture' 1 pixel to the the left, right ,up and down. While the structure of the image remains the same, the actual feature (pixel) values do change. This effictively increases the size of the data five fold compared to not applying the distortion.

The second distortion technique we implemented is what we called a *grow* distortion, although it actually combines a grow and shrink step. In the first, grow, step it iterates over all pixels and sets the value of it's 4 cross-like neighbours (left, right, up ,down) to 1. Since all pixels with value 1 remain 1 and some pixels with value 0 become 1, this grows the image. What follows is a shrink step that again iterates over all the pixels yet this time checks for each pixel (pivot) if its surrounding 8 (left, left-up, up, right-up, etc.) are set to 1, and if so leaves the pivot pixel at 1. If any of these 8 neighbours are not 1, then the pivot pixel is set to 0. Opposite from the grow step this leaves all the 0's be and sets some of the 1 to 0.

Since the grow step only adds 4 pixels, while the shrink step requires 8 pixels, compared to the original the overall image erodes a bit and gets a bit more smooth. Additionally, lone pixels and noise get's reduced or even completely removed. Our implementation provided 1 'eroded' sample for each original sample, doubling the data size when applied.

Both distortions can be applied simultaniously, providing a 10 times increase of the available (training) data.

[3]`http://scikit-learn.org`

3.2 Dimensionality reduction

As images are of size: $28x28$, we obtain 784 feature vector. Each component of the vector is a value of 0 or 1. There are some regions of every image having the same value (e.g. left upper corner is always white). Those regions doesn't bring any information about the classes and they can be considered as 'useless' features, or even noise. What worse can happen is when classifier puts too much emphasis on this noise, resulting in overfitting. Therefore, some sort of dimensionalty reduction can be applied to boost the classification performance and as well reduce the time complexity of training process.

The approach we decided on is Principal Component Analysis (PCA) - a well-known feature extraction method. A PCA build-in function from *sklearn.decomposition* was used. The *n_components* (amount of components) is a crucial parameter it takes, specifying the amount of features. The smaller, the lower computational time for classification. However, we want to aim also for a dimension that describes the data well.

3.3 Dissimilarities

Another optional preprocessing step to apply to the data is to turn the pixel vectors into (dis)similarity vectors. This is done by comparing both the training and test set elements to the elements in the training set. Let $v^{(i)}$ be a vector in the training set and u^j a vector in the test set, then create the new vectors u' and v' as follows:

$$v_k'^{(i)} = sim(v^{(i)}, v^{(k)}) \tag{1}$$

$$u_k'^{(j)} = sim(u^{(j)}, v^{(k)}) \tag{2}$$

where the subscript indicates the element of the vector and *sim* is the similarity function.

There are four similarity measures available, cosine similarity, 1-norm, 2-norm, and the edit distance. The latter three are normalized to ensure that dimensions maintain the same level of importance to the algorithms.

4 Classifiers

This section lists the classifiers that were used.

4.1 Parametric

Parametric models use the training set to learn a set of parameters, during the testing phase these parameters are used to determine the category of the new data. These models have the advantage that their training phase reduces the amount of computation needed during production use.

- Linear Discriminant Analysis - model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix. No parameters need to be manually optimised.

- Quadratic Discriminant Analysis - similar as to LDA, no manual optimisation needed (except for data preprocessing).
- Logistic Regression - the training algorithm uses the one-vs-rest (OvR) scheme with reguarization parameter C to be set. The classifier is considered as a well-behaved classification algorithm that can be trained as long as one's expect features to be roughly linear.

4.2 Non-parametric

Non-parametric models do not have a training stage since there are no parameters to learn, these models are therefore usually slower during production than their parametric counterparts especially when facing larger datasets.

- K-Nearest Neighbours - classifier with k-nearest neighbors vote takes into account k - amount of neighbours and *weight* parameter as the distance measure. Possible choices of *distance measures* tested are:
  - uniform - all points in each neighborhood are weighted equally
  - distance - weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

The above classification algorithms are easy to explain, fast, and despite their simplicity they are capable to achieve very good results.

4.3 Advanced

The advanced models have promising capability to learn non-linear models. They are known to provide great results if their parameters are appropriately set.

- Multilayer Perceptron - using MLP even very complex models can be trained. However, many aspects needs to be taken into consideration in order to achieve good classifier, like:
  - the network architecture, that is the number of hidden layers and their size
  - regularization parameter (*alpha*)
  - activation function
- Support Vector Classifier - thanks to it's design, the classifier deals well with big amounts of data described by lots of features. On the other hand training can be time consuming. In order to optimise it's performance the following parameters have to be set:
  - kernel function
  - regularization parameter C

– kernel coefficient *gamma* (only for rbf, *poly* and *sigmoid* kernels) - used to configure the sensitivity to differences in feature vectors, which in turn depends on various things such as input space dimensionality and feature normalization.

5 Results

In this section the results of applying the previously discussed techniques and classifiers to the datasets associated with both scenarios described in the assignment as well as how this data was gathered. We included most[4] of our full log files in our public git repository, which can be found at `https://github.com/i3anaan/DrawALine/tree/master/results_example`

5.1 Data collection

The results obtained below were all gathered the same structure way using our implementation: The dataset is generated in accordance with the settings, fed to the specified classifier and then scored on accuracy and timing. The scoring and timing is done the same for every configuration (and classifier) and a detailed record of each 'run' is stored as a row in a .csv result file, containing the full configuration. This .csv file is maintained as a record of all past runs of each classifier so that results can be easily compared and the best classifiers and settings easily selected. Random states were specified to reduce the impact randomness had on our results. In order to be able to correctly compare timings, all results regarding time were obtained from the same machine.

The test accuracy reported in this section is based on test data that is split off from the 10000 total examples provided. As a consequence scenario 2 actually has significantly more test data than scenario 1 (9900 as opposed to 1000), we felt that doing this was justified because the classifiers weren't trained on this data.

The data provided in NISTEval was not used to arrive at the results shown in this section, the reason for this is that by running multiple classifiers multiple times on the same set of test data and then selecting the best classifier is actually a form of training in itself which also carries the risk of overfitting. The NISTEval data has therefore only been used to evaluate a single classifier in a single configuration for each scenario, these results are reported in section 7

5.2 Scenario 1

In scenario 1 the classifiers have access to up to 1000 training examples per class. However, because of the need for cross-validation these classifiers were trained on 900 examples per class and then tested on the remaining 100, this yields a total training set size of 9000 and a total test set size of 1000. By applying the distortions the size of the training set can be expanded to up to ten times this size.

[4]We ran our tests on multiple machines, the logs of one of those machines is added (the one with the most tests)

In table 1 the results of this scenario are displayed together with the configuration parameters needed to achieve them.

Classifier	test (%)	train time (s)	test time (s)	Preprocessing	Parameters
MLP	98.6	194.908	0.041	distort=all	activation=relu; alpha=0.01; layers=(800,200,30)
SVM	98.5	433.130	13.882	distort=shift	gamma=0.01; C=3
KNN	98.0	0.015	0.361	PCA=38	n_neighbors=4; weights=distance
QDA	96.5	0.060	0.003	distort=shift; PCA=35	-
LOG	90.5	37.743	0.004	distort=shift	C=0.03
LDA	89.3	0.075	0.001	PCA=100	-

Table 1: Best results for each classifier in scenario 1.

5.3 Scenario 2

In scenario 2 the classifiers have access to 10 training examples per class. Similar to the first scenario, the size of the training set can be expanded to ten times its size by applying distortions to the images.

In table 2 the results of this scenario are displayed together with the configuration parameters needed to achieve them.

Classifier	test (%)	train time (s)	test time (s)	Preprocessing	Parameters
SVM	82.9	0.027	0.277	distortion=shift; PCA=45	C=3; Gamma=0.01
LDA	81.8	0.003	0.002	PCA=37; similarity=sim_cos	-
MLP	78.6	2.052	0.395	distortion=shift	alpha=0.01; layers=(800,100)
LOG	77.9	0.293	0.016	distortion=shift; similarity=sim_cos	C=9
KNN	77.8	0.001	0.416	distortion=all; PCA=32	neighbours=2
QDA	75.6	0.003	0.015	distortion=all; similarity=sim_cos	-

Table 2: Best results for each classifier in scenario 2.

6 Discussion

6.1 Scenario 1 - Large

Observations we made for the large training dataset (scenario 1) specifically include the fact that some classifiers completely escalate regarding time required to train and/or test, such as the SVM classifier taking over an hour for some configuration of parameters. This shows that these types of classifiers rely on applying extensive mathematical operations that are more impacted by an increase in scale than others such as for example the LDA classifier. Aside from this we noted that applying distortion, to increase the dataset size, had less of an effect (in raw percentages) when compared to the small dataset, but nevertheless helped give a boost to the classifier performances (at a small cost of increased preprocessing time). Finally, while obvious, the overall accuracy of the classifiers for this scenario was higher than that for the smaller scenario.

Although KNN classifier can be considered quite straight forward, it proves to give good results even on the big dataset. With usage of PCA feature extraction, dimensionality is reduced leading to small computational time. Dimesionality reduction is beneficial for QDA and LDA as well.

6.2 Scenario 2 - Small

The fundamental observation in the training process on smaller datasets is that the there is not enough data available for training when compared to the features used. Only by applying preprocessing on the dataset, satisfactory results were obtained. Specifically, distortions were required for almost all classifiers, as these significantly enlarged the training data with new artificially modified data points and thereby increased the accuracy of the results.

Aside from distortions, we ended up often utilizing PCA feature extraction and/or changing the representation of data by switching to a dissimilarity representation. Both methods fight against the curse of dimensionality, albeit in different way. PCA does this simply by reducing the feature space to those features that vary the most throughout the data. Dissimilarities are found by comparing every data point to every point in the training set using some given similarity funciton, therefore the feature space will be of size of the training set size.

6.3 General

It can be observed that the QDA classifier requires more data as the dimensionality increases compared to the LDA classifier, or to put it differently: QDA requires a more significant and severe reduction in features. It can be reasoned that since LDA has substantially lower variance, it tends to be a better than QDA if there are relatively few training observations. In contrast, QDA is recommended if the training set is very large, so that the variance of the classifier is not a major concern. That's why in order to keep a large ratio between the amount of training data points and amount of features in QDA, we reduce the dimensionality more substantially compared to the LDA.

7 The benchmark

Having had the opportunity to improve our results by mean of trial and error on the 'default' dataset provided, this section provides the results from applying the learned optimal settings in (simulated) real world usage, by means of testing the classifier performance on the provided evaluation dataset. This is done by adding the `--test-set=eval` option while training a classifier. The results of this benchmark can be seen in table 3.

Scenario	Classifier	test (%)	difference (%)	train time (s)	test time (s)	Preprocessing	Parameters
Large	MLP	97.5	-1.1	221.947	0.036	distortion=all	alpha=0.01; activation=relu;
Small	SVM	83.0	-0.1	0.029	0.026	distortion=shift; PCA=45	C=3; Gamma=0.01

Table 3: Results of applying the previously best performing classifiers on the live test data

While the difference with the default provided training set is not immense, the difference on the large dataset did surprise us. Our best guess to explain why this gap in performance exists is to attribute it to a certain degree of overfitting as a result of running multiple classifiers many times and then selecting the best one.

8 Live test

As an extra challenge we performed a 'live test', testing the scores of the classifiers trained on the NIST dataset on a (small) dataset of our own handwriting. The original scan of the handwriting from which this dataset was constructed can be seen in figure 1 in Appendix B.

This image was loaded into GIMP[5], an open source image editor, where several actions were performed to convert this image into a usable dataset. Firstly a threshold was applied to the image to convert it to solely black or white pixels. After this a rough grid was laid over the individual images (figure 2), with some small corrections to ensure ever number was in its entirety inside 1 cell. Next a guillotine operation was used to seperate each cell into its own image file. After this the BIMP plugin[6] was used to perform actions as a batch job. The first of these batch jobs was to convert the images into the .bmp format[7] to allow quick human inspection (figure 3). Empty cells as a result of the guillotine operations were removed and the remaining digit cells were briefly inspected. Next the final image manipulation was performed, a batch job over the .bmp files that first trims (auto-crops) each image, and then resizes it to 28 by 28 pixels, centering the image and padding with white where necessary. At this point we had had for each of the 10 digits 11 image files, 28 by 28 pixels in black and white (Figure 4, 5).

Next up reading these images into python was made easy with the opencv python module. Besides the file reading some greyscale levels needed to be transformed and the data needed to be stored in the proper dataset format used elsewhere in the code. For more details on this precise step we refer to `import_images.py`.

[5]`https://www.gimp.org/`
[6]`https://alessandrofrancesconi.it/projects/bimp/`
[7].bmp was chosen as it is loss-less

In the end we managed to automate most of the process independent of amount of digits, yet still requiring some human intervention. During this entire process several choices had to be made, such as the threshold for a pixel to be black, as well as what exact procedure to use to trim, resize and possibly more. No doubt the decisions made in this impact the eventual readability and in such the accuracy of classifying the numbers. Correctly capturing, processing and importing images with the goal of classification has its very own challenges.

Scenario	Classifier	test (%)	train time (s)	test time (s)	Preprocessing	Parameters
Large	MLP	80.0	292.597	0.007	distortion=shift	alpha=0.01; activation=relu;
Small	SVM	60.0	0.064	0.004	distortion=shift; PCA=45	C=3; Gamma=0.01

Table 4: Results of applying the previously best performing classifiers on the live test data

9 Recommendations

Distortions were able to boost our performance a great deal, we suspect that there might be more or better distortions that could be investigated and applied to improve our results. Furthermore, near the end of our research we learned that there is a bug in our shift distortion algorithm that causes it to perform sub-optimally (creating some duplicates). Fixing this mistake might be a good start for this.

In order to combat some of the timing issues that occurred for the large amount of (training) data, investigating options to run algorithms on a GPU instead of a CPU might open up opportunities for some of the classifiers to handle an even larger amount of data (or reduce the current time).

Workflow related we would advice anyone employing a similar method to consider storing of preprocessed datasets and trained classifiers, in order to cut down the time it takes to experiment with multiple configurations.

Our current approach on finding optimal parameters was linear, in the sense that we provide a range of parameters to test on and that is all that is tested. It might be worth it to adapt this to make use of some kind of binary search (or gradient-ascent method) in order to more efficiently home in on the optimal value (and give it the possibility to look around specified variables instead of being limited to those exact variables)

While we track nearly every step of our method, and have documented it in a reliable way in this report, one thing that is causing problems in this regard is the fact that our data splitting method `cherry_pick_data_set` does so in a random method. This was a minor issue, but should be resolved if any improvements were to made to our program by providing a seed to this method, similar as to what was done with some of the classifiers. Adding this would make the results fully deterministic and reproducable.

Other questions and associated advice that could be applied:

- **Will more data help?** Having more training data is always advantageous for classification and might improve performance, especially in case of complex classifiers like MLP.

- **Does the application require reject options?** We believe that rejecting points would not be crucial for this application, since classifiers provide good classification rate.

- **Should other classifiers still be considered?** Convolutional Neural Networks are known to give very good results on images. For this reason this classifier should be considered.

- **Semi-supervised learning**, should a large amount of unlabelled data be available in addition to a small amount of labelled data then it is possible to run PCA on the combined dataset and then train a simple classifier on the labelled data. This allows the PCA to extract features that are of higher quality than if only the labelled data was used, which will improve the quality of the results of the classifier.

References

[1] Koutroumbas K. Theodoridis, S. Pattern recognition. 2009.

Appendices

A Implementation Guide

A.1 Installation Usage

For instalation, usage guidelines and some examples we refer to the readme included in our code, also found on `https://github.com/i3anaan/DrawALine`.

A.2 Implementation Overview

This section serves to briefly outline the structure of the code as found in `./src/`. It does not have as a goal to fully explain the workings of each method, for this we refer to the actual code, but it serves to help in knowing where to look for what.

The starting point of the program, and also the main flow, happens in `main.py`. `main.py` Makes use of the argument parser module for python which made it easy to implement a proper command line interface, from the values of the parser variables decide the settings of the run. The big `main` method implements the flow as described in section 2.1, yet outsources most of the actual algorithms to other methods and or files. For example the small data set is created using the `cherry_pick_data_set` which splits a provided dataset into 2 by randomly 'plucking' a specified amount of each class and returning this along with the remaining complimenting dataset. Similarily, the method `load_data` is implemented in such a way that it would be easy to extend to new datasets in the future

Classifier files (`cls_*`) are used to specify settings to use or experiment with, the actual logic for training, scoring and some tricks regarding overriding settings is largely hidden away in the `cls.py` file. This allowed easy experimenting (iterating over parameters) within the file, while still being able to run any specific configuration using arguments of the command. Likewise distortions, similarity and pca extraction have their own file for their logic.

Finally, abstracting as much as possible allowed to reuse methods for the actual scoring (including time) and outputting it consistently both in terminal and, using a csv library, to a .csv log file (found in `main.py`). All of this together allowed flexible mix and matching between different configurations for different classifiers, preprocessing and even datasets. In order to maintain a proper workflow and ensure working code automated testing was implemented using Travis-CI.

A.3 Dependencies

The dependencies for the code to run can be found in the `requirements.txt` file.

B Live test images

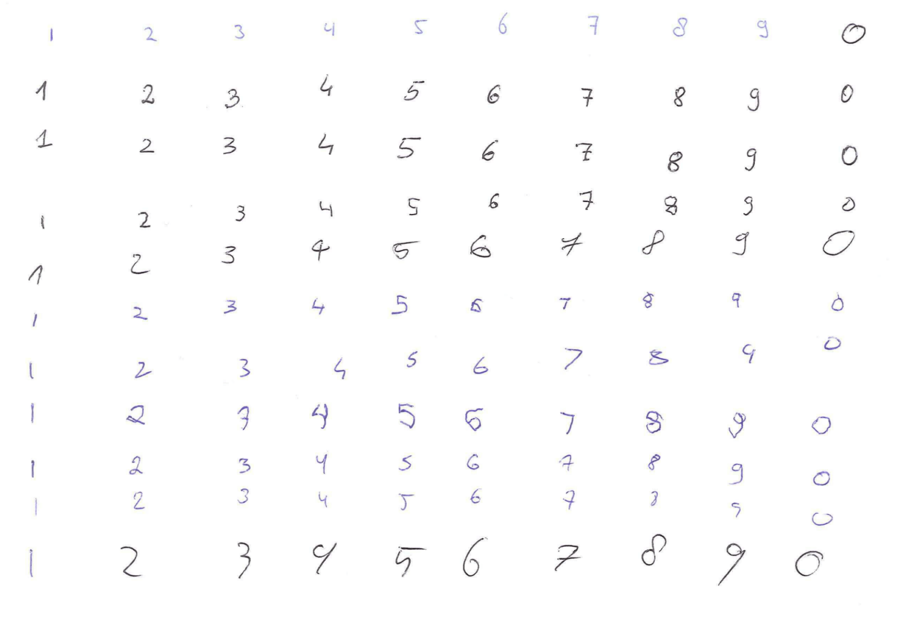

Figure 1: Hand-written digits used for live test

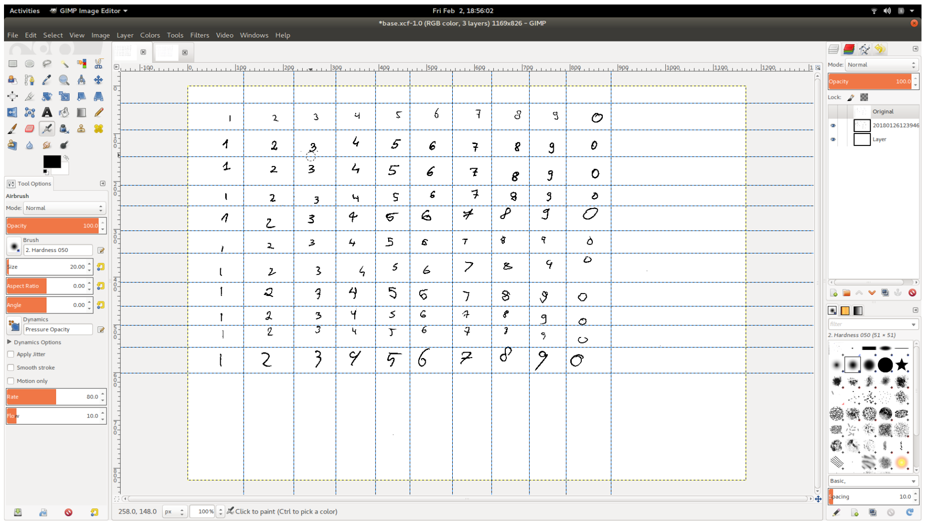
*base.xcf-1.0 (RGB color, 3 layers) 1169x826 – GIMP
File Edit Select View Image Layer Colors Tools Filters Video Windows Help
Airbrush
Opacity
Brush
Size
Aspect Ratio
Angle
Dynamics
Pressure Opacity
Apply Jitter
Smooth stroke
Motion only
Rate
Flow
Click to paint (Ctrl to pick a color)

Figure 2: The digits file loaded in GIMP and prepared for slicing into separate files

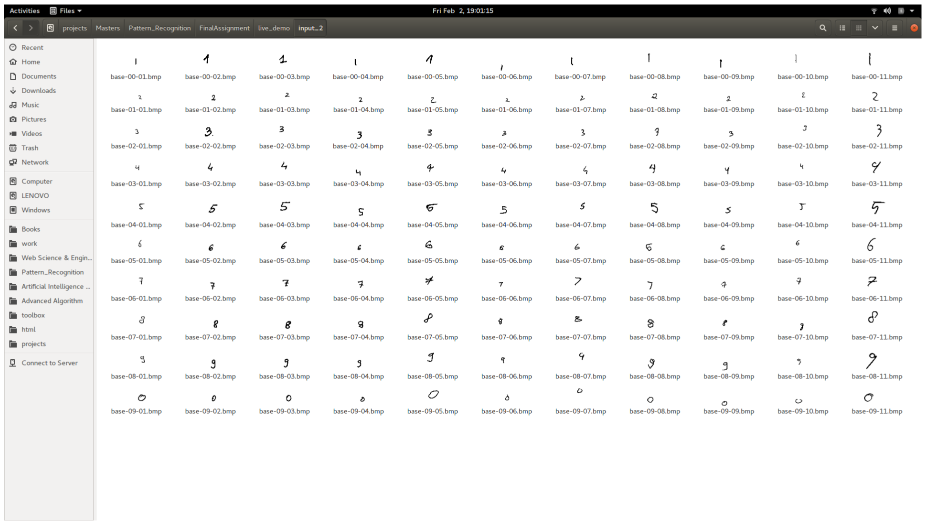

Figure 3: Separate image files shown in a file browser

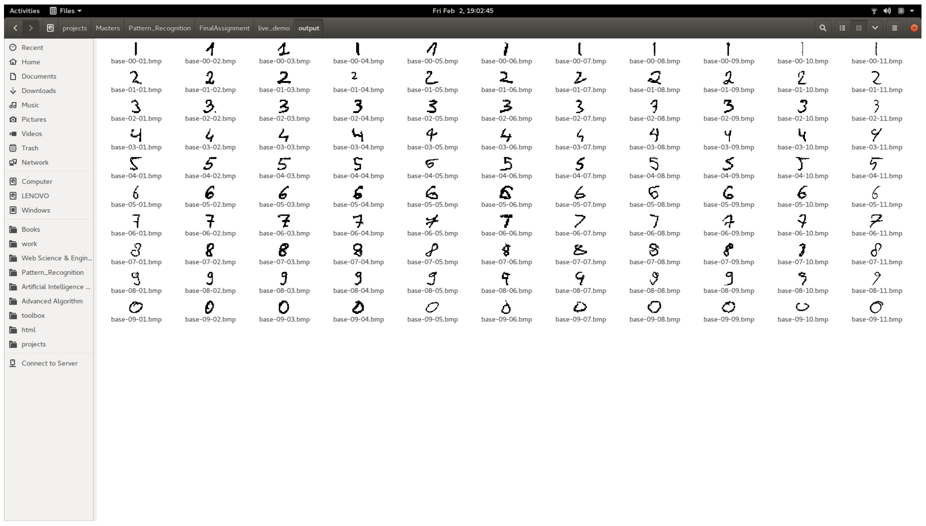
Activities
Files
Fri Feb 2, 19:02:45
projects
Masters
Pattern_Recognition
FinalAssignment
live_demo
output
Recent
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Network
Computer
LENOVO
Windows
Books
work
Web Science & Engin...
Pattern_Recognition
Artificial Intelligence ...
Advanced Algorithm
toolbox
html
projects
Connect to Server
base-00-01.bmp base-00-02.bmp base-00-03.bmp base-00-04.bmp base-00-05.bmp base-00-06.bmp base-00-07.bmp base-00-08.bmp base-00-09.bmp base-00-10.bmp base-00-11.bmp
base-01-01.bmp base-01-02.bmp base-01-03.bmp base-01-04.bmp base-01-05.bmp base-01-06.bmp base-01-07.bmp base-01-08.bmp base-01-09.bmp base-01-10.bmp base-01-11.bmp
base-02-01.bmp base-02-02.bmp base-02-03.bmp base-02-04.bmp base-02-05.bmp base-02-06.bmp base-02-07.bmp base-02-08.bmp base-02-09.bmp base-02-10.bmp base-02-11.bmp
base-03-01.bmp base-03-02.bmp base-03-03.bmp base-03-04.bmp base-03-05.bmp base-03-06.bmp base-03-07.bmp base-03-08.bmp base-03-09.bmp base-03-10.bmp base-03-11.bmp
base-04-01.bmp base-04-02.bmp base-04-03.bmp base-04-04.bmp base-04-05.bmp base-04-06.bmp base-04-07.bmp base-04-08.bmp base-04-09.bmp base-04-10.bmp base-04-11.bmp
base-05-01.bmp base-05-02.bmp base-05-03.bmp base-05-04.bmp base-05-05.bmp base-05-06.bmp base-05-07.bmp base-05-08.bmp base-05-09.bmp base-05-10.bmp base-05-11.bmp
base-06-01.bmp base-06-02.bmp base-06-03.bmp base-06-04.bmp base-06-05.bmp base-06-06.bmp base-06-07.bmp base-06-08.bmp base-06-09.bmp base-06-10.bmp base-06-11.bmp
base-07-01.bmp base-07-02.bmp base-07-03.bmp base-07-04.bmp base-07-05.bmp base-07-06.bmp base-07-07.bmp base-07-08.bmp base-07-09.bmp base-07-10.bmp base-07-11.bmp
base-08-01.bmp base-08-02.bmp base-08-03.bmp base-08-04.bmp base-08-05.bmp base-08-06.bmp base-08-07.bmp base-08-08.bmp base-08-09.bmp base-08-10.bmp base-08-11.bmp
base-09-01.bmp base-09-02.bmp base-09-03.bmp base-09-04.bmp base-09-05.bmp base-09-06.bmp base-09-07.bmp base-09-08.bmp base-09-09.bmp base-09-10.bmp base-09-11.bmp

Figure 4: The separate image files after trimming and resizing

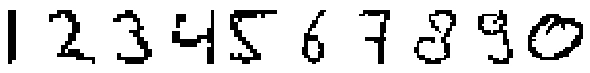

Figure 5: Some examples of digits after processing and ready to be read and classified by our program