IN4085: Pattern Recognition Project

Valentine Mairet
4141784

Tycho Marinus
4512014

Rhythima Shinde
4516389

April 20, 2017

Contents

1 Introduction

As part of the Pattern Recognition course, our group worked on a handwritten digit recognition project. We were instructed to preferably make use of MATLAB [1], so we opted for a MATLAB solution. We were given the NIST digit dataset [2] on which to perform the training and testing of our recognition system. Figure 1 shows a sample of this dataset.

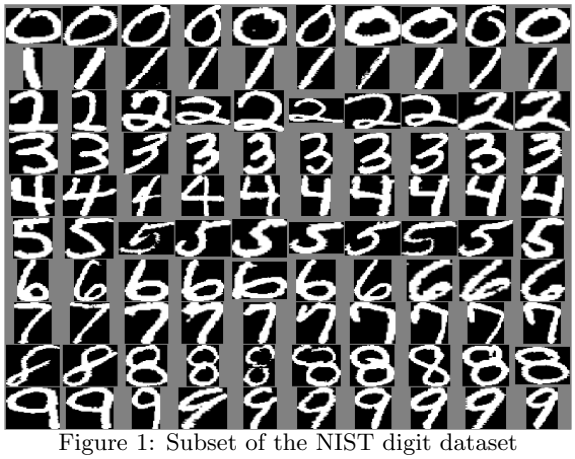

Figure 1: Subset of the NIST digit dataset

For this project, we took the role of a pattern recognition consultancy company working on a bank cheque processing application. For this application, two cases had to be considered.

1. Case 1: The application is trained once, and then applied in the field. The target accuracy for this case is 95%

2. Case 2: The application is trained for each batch of cheques to be processed.The target accuracy for this case is 75%

The first scenario suggests that the amount of training data available is sufficiently large in pattern recognition terms (200-1000 images per class). The second scenario indicate that this amount is much smaller, at most 10 objects per class as a cheque is composed of 10 digits.
We experimented with multiple feature extraction methods and three different classifiers, which we describe in the sections below. We benchmarked our results against the provided script and noted the differences we noticed between scenario 1 and 2. Additionally, we conducted a live testing session with self-handwritten digits. We each wrote down digits on a piece of paper, in the format resembling the NIST dataset. Then, we took a look at the classification results for our live testing. To top this, we wrote a few recommendations for banks, to minimise classification errors as much as possible.

2 Preprocessing

Before we could start developing our handwritten digit recognition system, some preprocessing on each single digit needed to occur. As can be seen in Figure 1,

not all the digits are straight, *clean*, and some may even be missing parts, e.g. because of the pen or photograph quality. There were hence a few things we needed to do before we could begin training our application. The preprocessing then goes as follows:

1. Restrict each digit to a bounding box such that each digit ends up centred in one box.
2. Remove possible noise to *clean* up each digit.
3. Straighten each digit.
4. Resize each digit to a proper, equal size for processing.

In the following subsections, we describe in detail how each of these steps is performed.

2.1 Boxing and Noise Removal

The NIST dataset provided resembles a matrix of images of digits sorted on the actual number they represent. It is possible to manually input the amount of digits desired. However, not all images are of the same size and not all digits are cut the same way. Figure 2 illustrates an example of what a "1" or a "5" would look like. It can be seen on these samples that the cropping of these digits is different, resulting in a significant difference in terms of the black and white pixel ratio.

Figure 2: Samples of the digit "1" and "5"

If we want to perform any operation involving a pixel colour comparison, we need to make sure we can work with this black and white ratio. The chosen solution is boxing the digit image using the PRTools [3] `im_box` command. This command allows us to add a defined number of empty columns on each side of the digit image.
Once we have boxed a digit, we can proceed to noise removal. The sample code below shows how we perform noise removal on each digit.

```
function [ out ] = remove_noise( in )

SE = strel('disk',1);

temp = imclose(in, SE);
temp = imerode(temp, SE);
temp = imdilate(temp, SE);

out = temp;
end
```

The noise removal function takes an input image of a digit and outputs a *clean* image. The primordial component is a structuring element that is used to interact with each digit to establish whether or not a certain set of pixels belongs to the digit at hand. We first use this structuring element to close holes with the `imclose` command, then we refine the digit by using `imerode` to scrape off potentially exaggerated contours. Finally, we use `imdilate` with the structuring element to rectify possibly eroded necessary pieces of the digit.

Figure 3: Digit "0" going through boxing and noise removal

The figures above show an image of digit "0" undergoing the steps described in this subsection. Figure 3 shows the; original digit image, a bounding box placed around the digit, and the digit image after going through noise removal.

2.2 Slant Correction

As can be seen in figure 1, some digits are tilted. To correct this, we seek inspiration in the literature [4] [5] [6] and come up with the idea to use image moments to detect the slant of a digit.

```
function [ out ] = straighten( in )

moments = im_moments(in, 'central');

variance_x = moments(1);
variance_y = moments(2);
covariance_xy = moments(3);

theta = atan( 2*covariance_xy / (variance_x - variance_y));

tform = affine2d([1 0 0; sin(0.5*pi-theta) cos(0.5*pi-theta) 0; 0 0 1]);

temp = imwrap(in, tform);

out = temp;
end
```

The sample code above shows how we use image moments and rotation to respectively detect and correct tilted digits. We use the `im_moments` function from PRTools directly on an input digit image. The specific function used returns a dataset containing the central moments of this image with respect to the image mean. From these moments, we can extract the variance in the x-axis, the variance in the y-axis, and the covariance between x and y. With these three values, we can approximate the orientation of the foreground object. This formula calculates the angle of the eigenvector associated with the largest eigenvalue towards the axis closest to this eigenvector. Simply put, we assume

that a digit tilted closer to the x-axis should be corrected towards the x-axis, while a digit closer to the y-axis should veer towards this axis. The computed angle is hence composed of the closest axis as reference.

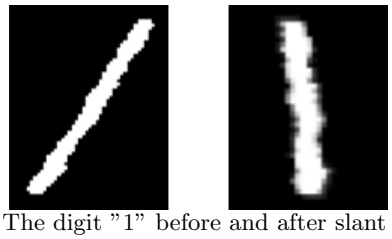

Figure 4: The digit "1" before and after slant correction

Figure 4 shows an example of a digit image before and after slant correction is applied. The image on the right is a bit blur because some resizing occurred before capture.

2.3 Finishing Touch

To finish the preprocessing, we resize each digit such that they obtain the same size. Each digit now has a decent black and white pixel ration, has sufficient noise removed, and the eventual slant is corrected to our best attempt. All digits now have the same size and are ready to be piped through the actual recognition system. We save each digit in a matrix with rows corresponding to the number each digit actually represents and columns being examples of each digit.

3 Classification

3.1 Feature extractions

Once the preprocessing is complete, we can begin feature extraction for classification. Feature extraction helps us derive features specific to each digit, which can be used for training a classifier based on the detection of these features. The following subsections describe which specific aspects of each digit we considered, which we could use for feature extraction. In the end, we opted for extracting histogram of oriented gradients (HOG) features to perform the classification [7]. This was the method which yielded the best results with regards to true classification error. We did combine HOG features with the others we mention, but the results were either equivalent or detrimental.

3.1.1 White pixel average

The first method we considered was an obvious one and we knew it could probably lead to inaccurate and inconclusive results. It basically consisted of averaging the number of white pixels across digits of the same class and use this calculated number as a feature for classification. The results, as expected, were not very helpful, as multiple digits can have the same average number of white pixels, and this average is not deterministic. It can, for example, depend on the handwriting.

3.1.2 White pixel x and y ratios

Because we still wanted to look into somehow involving a count of white pixels in this matter, we wanted to look at the white pixel ratios in the x and y directions. For this, we made two histograms for the number of white pixels on the x-axis and on the y-axis, and we created a histogram of the ratio of x versus y, which we could use as features. We looked at how the results compared among digits, but nothing was very conclusive. As with our first approach of simply averaging white pixels, multiple digits turned out to have very similar histograms, which made it hard to perform any kind of classification purely based on these features.

3.1.3 Hole and circle detection

Moving away from the idea of counting white pixels, we wanted to take a look at hole and circle detection. Hole detection works by finding connected components inside the inverted image of a digit. The number and placement of holes can be used as features for classification. The results actually looked promising, as they vary enough per digit to seemingly make a difference.
We considered looking at circle detection as well but the matlab function `imfindcircles` could not really find good partial circles without too many false positives. However, this could maybe be used for confirmation that a hole is actually circular.

3.1.4 Digit symmetry

Next, we looked into extracting digit symmetry as a feature. To compute a level of symmetry, we first flipped the digit horizontally then vertically, and looked at the number of overlapping pixels in each direction. One problem that struck immediately is that even after preprocessing, some digits were not straight enough to perform reliable symmetry measurements.

3.1.5 Histogram of oriented gradients

When looking into the literature for digit classification, we came across an article written by MathWorks that suggested HOG feature extraction as a method for digit classification [8]. The idea of HOG is that an object's shape and appearance can be described as a distribution of intensity gradients or edge directions. This distribution returns a set of *HOG features*. Based on the article mentioned above, we wrote a script to extract HOG features in a vector of digits, so we could directly extract all HOG features of the dataset used for training of our classifier.

```
function [ out ] = hog( in )

datasize = size(in, 3);

first_im = in(:, :,1);
[first_hog4x4, first_vis4x4] = extractHOGFeatures(first_im, 'CellSize',
    [4 4]);

temp = zeros(datasize, length(first_hog4x4));

temp(1, :) = first_hog4x4;
```

```
for i = 2:datasize
    im = in(:, :, i);

    [hog4x4, vis4x4] = extractHOGFeatures(im, 'CellSize', [4 4]);

    temp(i, :) = hog4x4;
end

out = temp;
```

This piece of code extracts the HOG features from the first digit image in the given set with the `extractHOGFeatures` function. This is done to determine the actual extracted feature vector size, as this size depends on the image itself. Once the size is determined, we loop through the dataset and create an array of extracted HOG features per digit image.
The `extractHOGFeatures` function returns a set of extracted HOG features and a visualisation of the intensity gradients. The features and visualisation depend on the side of the HOG cell, which is the size of the kernel used for the HOG computation. The greater the size, the higher loss of small-scale details. In our project, we chose a size of 4, for the simple reason that it looked good enough. Figure 5 shows a visualisation of the intensity gradients of one of the NIST digits "9" with different cell sizes.

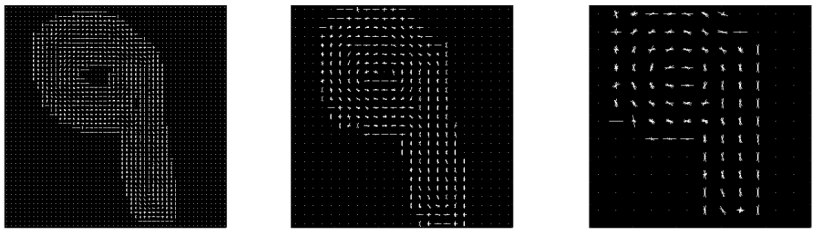

Figure 5: HOG visualisation with, from left to right, a cell of 2x2, 4x4, and 8x8

When looking at these different results, we hypothesised that a cell of 2x2 would yield too many details specific to this particular "9", and would perhaps cause the system to be overtrained on the tiny details. On the other hand, a cell of 8x8 does not have enough detail. This is why we settled for a 4x4 HOG cell.

3.2 Classifiers

After the feature extraction of the images, the next step is to start the recognition of the digits from the given images. This is achieved by using different MATLAB classifiers. A total of three classifiers will be used: one linear classifier and two density-based non-linear classifiers.
The chosen classifiers are thus:

- Support Vector Machine (SVM)
- k-Nearest Neighbour (kNN)

- (Verzakov) Decision Tree (DT)

Recall that the NIST dataset consists of handwritten digits. These digits have been separated as individual information with their corresponding number as its label from 0 to 9. Based on these data and labels, the pipeline for classification used here consists of the following steps:

1. The dataset is split into test and train set, sequentially and randomly. By sequentially, it is meant that the the first n values are chosen for testing and first m values for training per class, let n and m be respectively test and training size per class. While for random selection, the n and m respective selections for testing and training are done randomly from the whole set of the class.

2. For scenario 1 with large enough training dataset size, 600 and 800 images per class are used (the remainder is used for testing). For scenario 2 with training each batch of check, 10 images per class are used.

3. Repeat the steps for all, as there are random selections, where there are higher number of repetitions for case 2, because of more variance in the small dataset, and take the average accuracy (scenario 1: 3 repetitions, scenario 2: 6 repetitions).

The results of these classification are discussed in Section 4. To understand the performance of a classifier, the accuracy of this classifier is noted. More variations (sequential and random) of testing and training sizes are done with record of time and accuracy. Results for these are listed in detail in Appendix A.

4 Results

Based on the classifiers on the extracted features in the given two scenarios, different results are obtained, which helps decide which combination of features and classifiers is suitable for the given case. Since not all the features prove useful, we discuss in the first section below how features should be reduced. This is followed by discussions on the exact classification results and the comparative study of the classifier results. Finally, based on the results, optimal classifier and feature combinations are discussed per scenario.

4.1 Classifiers

The classifier results are gathered in various forms for comparison of different classifiers. First, results are in the form of a confusion matrix (example shown in table 1), which shows the falsely and correctly classified digits, giving the accuracy of the classifier. The table 2 shows the result for scenario 1 and scenario 2 on three different classifiers for HOG features. Throughout the experiments, the size of the image is kept constant at 50 by 50 pixels.

Table 1: Confusion matrix for SVM on scenario 2

digit	0	1	2	3	4	5	6	7	8	9
0	0.96	0.03	0	0	0	0	0	0	0	0
1	0.00	0.89	0	0	0	0	0	0.01	0.09	0
2	0.01	0	0.9	0.01	0	0	0.01	0.03	0.03	0
3	0.01	0	0.01	0.92	0	0.01	0.01	0.01	0.03	0.01
4	0.00	0.05	0.02	0	0.82	0	0	0	0.01	0.09
5	0.00	0.01	0	0.02	0	0.94	0.02	0	0.02	0
6	0.01	0.02	0	0	0	0	0.96	0	0.01	0
7	0.00	0.01	0	0	0.01	0	0	0.86	0.02	0.1
8	0.00	0.02	0.01	0.01	0	0	0	0	0.94	0.01
9	0.00	0.01	0	0	0	0	0	0.05	0.12	0.82
		correct:	8915	error:	985					

4.2 Feature reduction

The different features, which are discussed in Section 3.1, do not result in the same level of accuracy for classification and thus, based on the accuracy level, multiple features can be discarded. The results of other features, except for HOG, (white pixel average, hole detection and digit symmetry) are shown in the table 3 below. For other features, the required accuracy of 95% for scenario 1 and 75% for scenario 2 is not achieved, showing that the other features are not adding anything to the digit identification as required. kNN shows better results than other classifiers, but it still does not meet the requirements of accuracy. This table clearly shows that when compared with the results from Table 2, HOG feature extraction performs much better than the combination of other features for all the classifiers.

Table 2: Accuracy for different classifiers for HOG features

	Size per class		SVM	kNN	DT
Selection	Training	Testing	Accuracy(%)		
Sequential	600	400	97.98	97.4	85.4
	800	200	98.05	97.60	85.4
	10	990	83.28	81.04	57.48
Random	600	400	99.45	99.3	94.5
	800	200	99.4	99.65	91.95
	10	990	90.10	85.95	53.63

Table 3: Accuracy for different classifiers for a combination of hole+pixels+symmetry

	Size per class		SVM	kNN	DT
Selection	Training	Testing	Accuracy(%)		
Sequential	600	400	36.10	26.50	39.13
	800	200	36.45	25.25	38.60
	10	990	31.92	17.03	37.72
Random	600	400	34.03	68.53	57.75
	800	200	34.25	84.25	64.05
	10	990	30.24	19.17	32.85

***P*-value comparison:** For p-value comparison of classifiers, we should have at least 30 image results for each test. This can be done by performing three repetitions to ten fold cross validation. A separate script `k-fold` is made to carry out this computation. Upon performing such an analysis, we get the following average accuracies, as shown in Table 4 for different classifiers. Full results of this 10-fold validation are given in Appendix A, in Table 8. A statistical Wilcoxon signed rank test (a t-test is not suitable here [9]) on this results in a null hypothesis that states that SVM is similarly performing as KNN. The alternative hypothesis is that SVM is better than KNN. As the result shows that SVM is much better than DT, the test is not performed on the latter classifier. For this test, the command is shown below:

```
[p,h] = signrank(accuracy_SVM, accuracy_KNN)
```

Here, SVM and kNN accuracies are stored in the respective matrices obtained from *true* matrices (which give the correctly classified digit percentages). The test results show that for HOG feature: Test fails to reject null hypothesis that SVM is similar performing to KNN (p=0.002, h=1), confirming that SVM is the best classifier for this problem. Note that these results are only meant for p-value comparison and not belonging to any specific scenario. In fact, the training size here for each class is 100.

Table 4: 10 fold cross validation accuracy for classifiers

	SVM	KNN	DT
Average white pixels + Symmetry + Holes	31.17	20.27	32.61
HOG	96.56	95.54	69.81

4.3 Optimum combinations

For scenario 2, it can clearly be seen that with the HOG features, SVM performs better than kNN, and much better than DT. For case 1, with HOG features, the SVM and kNN results are comparatively closer. For other features than HOG features, the results are not satisfactory with regards to the accuracy requirements, though kNN performs better than other classifiers. HOG features score very well in comparison to the other feature extraction methods we came up with, and adding an other feature on top of HOG features only resulted in equal or worse outcome. This is why we decide to stick with HOG features only, and an SVM as classifier.

5 Benchmarking

Now that we have obtained the results, it is important to validate our best classifier to promote it correctly to the bank. Prior to that point, we had only been working with classifier functions provided by MATLAB only. Because the benchmarking helper script provided `nist_eval` could only work with a PRTools classifier, we converted part of our code to be able to train a PRTools SVM. We believe it to be safe to assume that a MATLAB classifier would perform similarly to a PRTools classifier, since they essentially base themselves on the same mathematics.

Thus, the following benchmarking steps are done:

1. We create a `my_rep` script that essentially is a function which takes a NIST `prdatafile` and outputs a `prdataset`. This dataset contains the HOG representation of all preprocessed NIST digit images. Our `my_rep` script can be found in Appendix B.

2. We train a classifier on a training set size of 200 images per digit and a test size of 800 for scenario 1, and training and test sizes of respectively 10 and 990 for scenario 2. We took the NIST data sequentially for both cases. Since we established that the best MATLAB classifier was the SVM for this problem, we choose to work with the `SVC` classifier from PRTools [10].

3. We use the provided `nist_eval` script with default `n` value to compute the classification error `e`. Table 5 illustrates the results we obtain.

The second code sample in Apprendix B shows the script we use to run the whole system. We first fetch the NIST data and pipe it through the `my_rep` script. Then, we train our SVM classifier with a predefined training set and test it on the predefined test set. We use the `labeld` function to extract the resulting labels and show the confusion matrix with the help of `confmat`. This is just to give us a deeper understanding of how our classifier performed. Additionally, we feed our `my_rep` script and generated classifier to the provided `nist_eval` script.

Table 5: Result of the nist_eval script on both scenarios with an SVM classifier

Scenario 1	Scenario 2
0.031	0.125

As we can see, we obtain an error of 0.031 for scenario 1, and 0.125 for scenario 2. The difference in error can simply be explained by the training size. A training size of 10 images per digit is not enough to train a well-performing classifier. If we plot the true error as a function of the training size, we can see that at low training size, the error is very high. To find a low true error, we need to have just enough data to have enough examples for our classifier, but not too much in order not to ovetrain our system.

6 Live Testing

Besides the provided test set, we also adjusted our digit recognition to be able to handle digits from paper. We have created two different functions for digit extraction from paper and classify them. The classification is done in exactly the same method as the previously described classification methods. For the extraction of the digits from the paper, we decided to approach two use-cases.

6.1 Input

Method 1 does not have many restrictions for the input image. It is important that the image does not contain too much shadow, and it works best when the digits are written on a black piece of paper. However lined paper still works

in most of the cases, when the contrast between pen and lines is clear enough. Besides the input image, the function requires the number of digits on the image and what the digit size should be for the individual extracted images. After extraction of the separate digits, the function shows the extracted digits and asks the user what the correct label is. This allows the input image to have any order of digits.
The second method allows for quick testing of larger sets of digits. However, the order of the digits has to be provided in a specific way. The input has to be 10 separate files, one for each digit, all containing an equal amount of digits. Since it is known at the start which file contains which digit, the creation of the labels is done automatically. Since labelling is done without user interaction, the number of digits can easily be larger than with the first method.
Our testing and reporting was done with this second method. We used 3 different sets of images, each having handwriting from a different team member. Two sets contained a total of 100 digits and one contained 50 digits for a total of 250 digits, distributed equally between 0 and 9. All the images have been added to the appendix C.

6.2 Extraction of digits

While there are some differences for the input of both functions, the digit extraction technique applied is the same. First, the images are converted into black and white binary images. This greatly increases the easiness for component extraction. Using the Matlab provided functions, listed in order of usage: `bwareafilt`, `padarray`, `bwconncomp`, `labelmatrix` and `regionprops`, we first select the largest objects, expecting those to be our digits, and the smaller components of the image are removed (filled in with black). After padding the image, we start looking at the connected components of our image. Since we have extracted only our top largest components, we expect to get exactly all the digits as connected components. Finally, we get the indexes of our separate digits, which we convert into regions.
Now, for each digit, we can just select the region that we found in our original black and white image, to get a single digit. After this, we apply the same preprocessing, as described in Section 2.

6.3 Classification Results

After the extraction, we tested our handwritten digits in three ways. First, we trained the classifier on the provided original dataset and classified the handwritten digits. Second, we trained the classifier on the handwritten digits and classified the original dataset. Lastly, we trained and tested based on a random subset of our handwritten digits.
In our first test, we trained our classifier as described before with either 50, 250, 500 or 1000 digits from the original NIST dataset. We started testing the individual handwriting images. We did see some differences between them, see table 6. For example, the images from *hand2* clearly show a better performance than *hand1*, *hand3*, and the *Combined* sets. Looking at the training size, it can be seen that a small training set of 50 images is on the small side for training a good classifier. Increasing this amount to 250 images greatly improves the performance, while increasing it above 250 might even damage the performance,

Training size	50	250	500	1000
hand1	46%	60%	50%	56%
hand2	57%	67%	68%	66%
hand3	46%	52%	51%	55%
Combined	50%	59%	57%	59%

Table 6: Table showing the percentage of correctly classified digits.

probably because of over-training. When combining all the self made digits, we logically just found the average of the three separate tests.
In our second test, we trained our classifier using all 250 handwritten digits to classify subsets of the original NIST dataset. Doing so, we found a 64% correct classification rate. This seems to be slightly higher than the other way around.
Lastly, we ran multiple tests training and testing on subsets of our self written digits. We either randomly selected a subset of 200 or 50 digits to use for training and used the leftover for testing. When using 200 training elements, we got an average at around 86% correct classification rate. Using only 50 digits for training and 200 for testing, we got a correctness of 67%, significantly smaller but still higher than the results when we mixed NIST with our own images. A possible explanation might be the small sample size of our own images.

7 Recommendation for the Bank

There are a few easy improvements that can be done for the bank to improve the digit recognition system. First, for the digit extraction, it is useful if there are clear boxes in which the digits have to be written. This allows to do digit extraction based on position on the cheque, instead of trying to detect the exact location of the digit. Making sure the contrast between the boxes and the ink of the digits is maximal, e.g. by having white boxes and asking clients to use black pens will increase the accuracy of the extracted digits. In a similar fashion, increasing the size of the boxes, encouraging larger digits, might improve the neatness of the writer, in combination with making the classification easier.
Besides adjusting the digit boxes, the bank can also request clients to write in block letters. Even though there is still variation in how people write digits in block letter style, the variation will probably be less. To further improve this, the cheques could contain examples of the digits. This would probably make most people copy the style and thus making it easier for the handwritten digit classification system.

8 Conclusion

The assignment aimed at correctly recognise handwritten digits. We were presented with two scenarios, which boiled down to one being a situation with a large training set and the other having a significantly smaller training set size.
We started with using the NIST dataset and preprocessing it with different steps, from creating equally sized bounding boxes, removing noise, to correcting slant. After preprocessing, different features were extracted from every image which would be later used for classification of the digits. These features were

histogram of oriented gradients (HOG) features, average of white pixels, white pixel ratios in the x and y axes, image symmetry, and hole and circle detection. The classifiers were tested according to the two scenarios, with three test cases each having a training set size of respectively 600, 800, and 10 images per digit. We also tested against two sets of features: one with HOG features only and the other with hole detection, white pixel average, and digit symmetry. We had two different type of selection of training data: sequential and random selection.
We considered three different classifiers, namely, a support vector machine, k-nearest neighbour classifier, and a Verzakov tree. To check which classifier performed the best, a 10 fold cross validation is performed, which clearly shows that the SVM performs much better the other classifiers we explored. kNN performs slightly better but still with a minimum error of 0.16 with 800 training set size. It is important to note that both classifiers meet the requirement of the assignment, with a minimum of 95% accuracy for case 1 and 75% accuracy for case 2. It should however be noted that scenario 2 is a risky method to directly be implemented for the banks, as even our best classifier results with 10 out of 100 wrongly detected cheques.
Because we discovered the HOG feature extraction method rather quickly in our project, we only considered but a few other possibilities. HOG performed so well from the start that we decided to stick to it. We tested combining our other features with HOG, but the results were either equivalent or detrimental. This could be due to our classifier hitting the curse of dimensionality. Since we performed our measurements on MATLAB classifiers, the results we have correspond to these classifiers. We did also implement the PRTools classifiers specifically for the benchmarking, but we deem it safe to assume that both types of classifiers would give similar results.
The benchmarking results are actually quite promising, with a true error of 0.0310 on a classifier trained on 200 training samples per digit, and an error of 0.1250 when trained on 10 samples. This difference can be explained by the risky nature of scenario 2: not having enough training data to be able to reliably classify enough to satisfy.
To close our project, we performed a live testing phase with our own handwritten digits. The most important outcome is, despite our outstanding results with the NIST dataset, the system does not handle our own digits quite well. This can be because our own digits were quite small, and usually *thinner* than the NIST digits. NIST digits, despite being quite variate, did have similar specific dimensions.
To conclude, we are happy to provide our client with our classifier, but we do recommend banks to apply standards on a certain cheque writing style. If people were restricted to digit boxes and were faced with an optimal example, we suspect they would do much better in writing similar, clear digits, which would make it easier for our handwritten digit recognition system to do its job (close to) flawlessly.

References

[1] MathWorks, "The Language of Technical Computing." https://www.mathworks.com/products/matlab.html. Online; accessed 20 April 2017.

[2] NIST, "Nist Special Database 19." https://www.nist.gov/srd/nist-special-database-19. Online; accessed 20 April 2017.

[3] F. van der Heijden, R. Duin, D. de Ridder, and D. Tax, *Classification, Parameter Estimation and State Estimation – An Engineering Approach Using MATLAB.* Wiley, 2004.

[4] C. Sun and D. Si, "Skew and Slant Correction for Document Images Using Gradient Direction," *4th International Conference on Document Analysis and Recognition*, 1997.

[5] C. Manisha, Y. Sundara Krishna, and E. Sreenivasa Reddy, "Slant Correction for Offline Handwritten Telugu Isolated Characters and Cursive Words," *International Journal of Applied Engineering Research*, 2016.

[6] S. Fitrianingsih, S. Madenda, and R. Widodo, "Slant Correction and Detection for Offline Cursive Handwriting Using 2D Affine Transform," *International Journal of Engineering Research & Technology*, 2016.

[7] MathWorks, "Extract histogram of oriented gradients (HOG) features." https://nl.mathworks.com/help/vision/ref/extracthogfeatures.html. Online; accessed 20 April 2017.

[8] MathWorks, "Digit Classification Using HOG Features." https://nl.mathworks.com/help/vision/examples/digit-classification-using-hog-features.html. Online; accessed 20 April 2017.

[9] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine learning research*, vol. 7, no. Jan, pp. 1–30, 2006.

[10] PRTools, "Svc." http://www.37steps.com/prhtml/prtools/svc.html. Online; accessed 20 April 2017.

Appendices

A Detailed Classification Results

Table 7: Accuracy and time for classifiers

	Size per class		SVM	kNN	DT	SVM	kNN	DT
Selection	Training	Testing	Time(s)			Accuracy		
Sequential	50	100	53.84	51.95	60.54	94.7	92.1	71.5
	100	100	65.82	66.48	74.06	95.1	94	73.1
	500	100	167.04	152.81	183.04	98.6	97.6	86.4
	300	500	282.92	290.05	319.05	97.68	97.1	84.96
	5	10	8.04	7.4	7.52	78	82	49
	10	10	9.31	10.33	8.97	82	83	57
	50	10	19.9	25.02	20.9	90	90	70
	30	50	32.99	35.62	39.2	88.6	88.6	69
	60	40	38	34.79	38.036	92.25	89	71.25
	600	400	319.73	332.13	346.6	97.98	97.4	85.4
	800	200	NA	NA	NA	98.05	97.60	85.4
	10	990	NA	NA	NA	83.28	81.04	57.48
Random	600	400	NA	NA	NA	99.45	99.3	94.5
	800	200	NA	NA	NA	99.4	99.65	91.95
	10	990	NA	NA	NA	90.10	85.95	53.63

Table 8: Detailed accuracy results of 10 fold cross validation

KNN	95.14	95.68	95.46	94.84	95.74	95.87	95.82	95.82	95.38	95.60
SVM	96.28	96.28	96.61	96.46	96.70	96.59	96.61	96.67	96.59	96.78

B MATLAB Scripts

my_rep.m

```
function [ out ] = my_rep( in )
%my_rep Convert digits images into a dataset.
%in NIST prdatafile
%out prdataset containing HOG features of each preprocessed digit image

dataset_size = size(in, 1) / 10;

labels = {{}};
hogs = [];

for i = 0:9
    for j = 1:dataset_size
        index = dataset_size*i + j;
        pr_digit = in(index);

        % convert to image
```

```
        digit = data2im(pr_digit);

        % restrict to bounding box
        digit = im_box(digit, [5, 5, 5, 5]);

        % remove noise
        digit = remove_noise(digit);

        % straighten the digit
        digit = straighten(digit);

        % resize image so they all have the same size
        digit = imresize(digit, [50 50]);

        % extract HOG features
        digit_hog = extractHOGFeatures(digit, 'CellSize', [4 4]);

        % save the HOG measurements into an array
        hogs(index, :) = digit_hog(:);

        % generate a label
        labels{index} = strcat('digit_', num2str(i));
    end
end

% return a dataset
out = prdataset(hogs, labels);

end
```

test_run.m

```
%% Logistics
% change this to change training size
training_size = 200;

%% NIST

% toggle if the data for training is chosen sequential
% training_data = prnist(0:9, 1:training_size);
% test_data = prnist(0:9, training_size+1:1000);

% toggle if the data for training is chosen at random
% training_data = prnist(0:9,randperm(1000,training_size));
% test_data = prnist(0:9,setdiff(1:1000,randperm(1000,training_size)));

%% my_rep
pr_training = my_rep(training_data);
pr_test = my_rep(test_data);

%% Training

w = svc(pr_training);
```

```
%% Testing

results_svm = labeld(pr_test, w);

%% Confusion Matrices

confmat(pr_test.labels, results_svm);

%% nist_eval

e = nist_eval('my_rep', w);
```

C Images for Live Testing

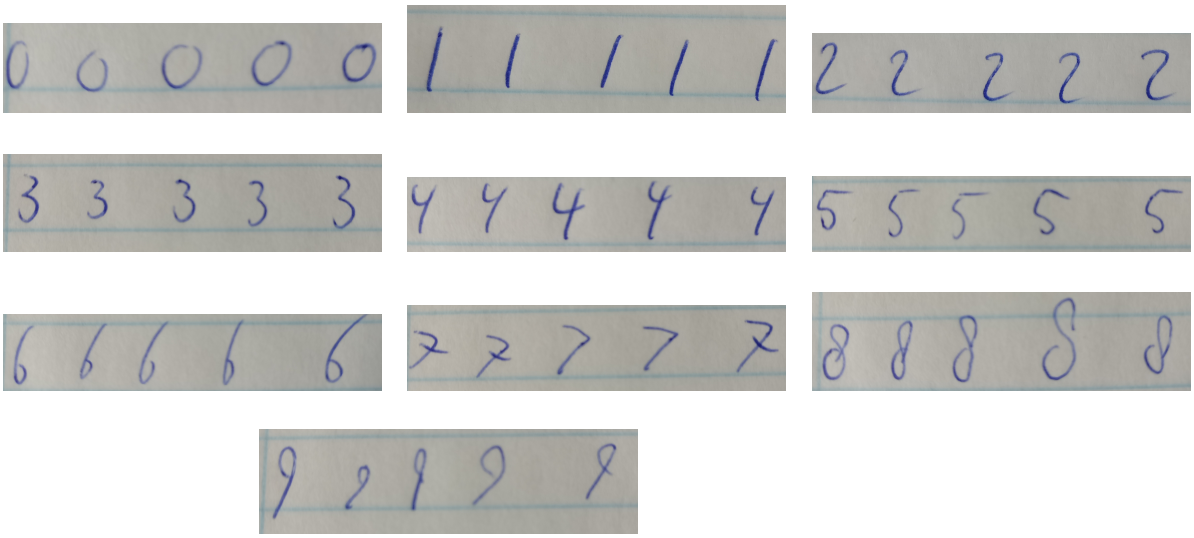

Figure 6: Images for hand1

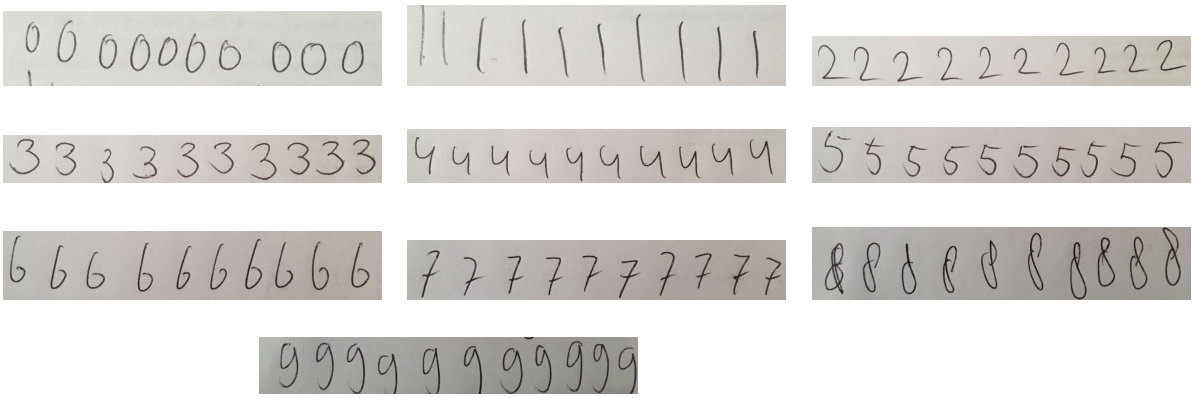

Figure 7: Images for hand2

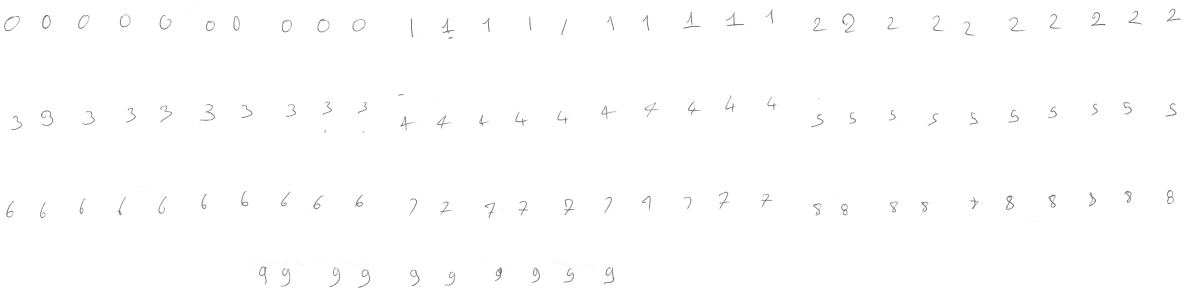

Figure 8: Images for hand3