university of groningen / faculty of science and engineering

Handwritten Digit Recognition

Marloes van Asselt s3744531
Dewi Batista s3313093
Giacomo Bagnato s4918428
Andrej Kovács s4410742

February 4, 2024

Abstract

We applied four classification methods to a dataset of images of handwritten digits. The methods used were k-Nearest Neighbors (kNN) and Random Forests as a baseline, contrasted with two more elaborate methods: a Convolutional Neural Network (CNN) and a Support Vector Machine (SVM) paired with an autoencoder. The baseline methods are equipped with a series of handcrafted features in order to enhance their performance relative to only using the raw pixel data. The results consist of 3.0% misclassification for our kNN on the handcrafted feature data, 1.5% for our random forest on the handcrafted feature data, 1.8% for our CNN on the raw pixel data and 1.1% for our autoencoding-paired SVM on the raw pixel data.

1 Introduction

Digit recognition is a typical benchmark to study new techniques in machine learning, especially computer vision. The typical dataset used for this benchmark is the MNIST dataset (consisting of 70,000 digits in total), published originally in 1998. Models with an error rate of less than 1% have been published for this dataset many times, so it is no longer considered a challenging dataset [1].

Since digit recognition can be considered the *hello world* of machine learning [1], it is great for educational purposes, too. The objective of this project is to train and compare four different models (a k-Nearest Neighbors model, a random forest, a convolutional neural network and an autoencoding-equipped SVM) for the classification of handwritten digits. Rather than MNIST, a much smaller dataset is used, originally published by Duin and Tax [7] in 2000. This report considers the entire pipeline used for the aforementioned methods. To begin, preprocessing and data augmentation is discussed, the latter being used to improve the performance of our CNN. To follow, handcrafted features are engineered, which are used for our kNN and random forest models. Finally, all four methods are explained and discussed.

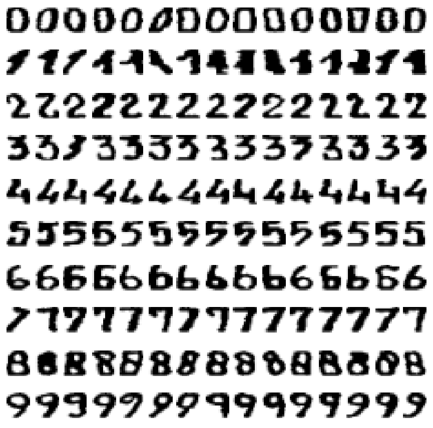
Figure 1: Sample of the images of the handwritten digits contained in the dataset.

2 Data Preprocessing and Augmentation

The original dataset, or the ‘raw data’ as it is referred to in this report, contains 2000 images of handwritten digits, consisting of 200 images of each digit 0 through

9. Each sample is given in the form of a 240−element array whose pixels (or elements) represent an image of a handwritten digit that is 16 pixels in width and 15 in height. Each pixel takes on a value in $\{0, 1, 2, 3, 4, 5, 6\}$ representing its gray scale. Higher pixel values indicate a brighter pixel.

The data is complete in that there are no missing values. As such no imputation process is needed. That said, if we split the raw data into 1000 training and 1000 testing samples, it is expected that our CNN will not have enough data to perform particularly well and so we consider data augmentation methods.

2.1 Data Augmentation

Data augmentation is the practice of performing certain transformations on samples within a dataset with the intention of enlarging it in a way that does not just repeat the effects of the samples that were augmented. This can be particularly useful when dealing with a small training sample. Considering the classification methods that are used in this report, the only one that will be tested with data augmentation is the CNN. This is because the CNN uses the dataset in their raw image format and does not make use of any of the handcrafted features that the baseline methods use. Thus, data augmentation is a different method of providing the model with more training data.

In our case of digit recognition, there are a few transformations that could prove to be useful but also a few that should be avoided which we now discuss.

2.1.1 Mirroring

One common augmentation method is to mirror the images either horizontally or vertically (or both). However, for digits this can completely alter the 'correct' classification of the image. The most clear example being a 6 turning into a 9 by mirroring in both directions.

2.1.2 Rotations

Another augmentation method is to rotate images. Here the images get rotated by some degree θ either clockwise or counter-clockwise. For small rotation angles this can be a useful augmentation method but for larger angles this could once again transform the digits to the point where they could no longer conform to the human classification of that digit. It could also, as with mirroring, change them in a way that would resemble other digits. The most straightforward example again being a 6 and 9 being 180 degree rotations of each other. However, this is likely the most useful augmentation method to obtain a varied extension of the existing dataset when only using small angles.

3 Handcrafted Features

In order to improve the performance of our baseline methods of kNNs and random forests compared to their performance on the raw data, we implemented a set of seven handcrafted features. Five of the handcrafted features are relatively simple in their use and implementation while the remaining two, namely Gabor filters and the histogram of oriented gradients, are relatively elaborate. It is worth noting that while some handcrafted features, such as hole area and vertical ratios, distinguish digits very well individually (acting as semi-classifiers in themselves), this is not the case for all handcrafted features and will be considered in their description.

Now, onto describing each in detail including their dependencies where relevant.

3.1 Row and column non-zero counts

The first of the relatively simple handcrafted features is that of row and column non-zero counts. Given a sample in 'matrix form', i.e. as a 16 by 15 image, this feature simply counts the number of non-zero pixel values in a given row or column. In constructing our handcrafted features dataset we include this feature for all possible rows and columns. That is, we compute and store both

$$\mathcal{F}^{(i)}_{\mathbf{rc}}(X) = \sum_{j=1}^{15} \mathbb{1}(X_{i,j} > 0)$$

$$\mathcal{F}^{(j)}_{\mathbf{cc}}(X) = \sum_{i=1}^{16} \mathbb{1}(X_{i,j} > 0)$$

for $i = 1, \ldots, 16$ and $j = 1, \ldots, 15$ where $\mathbb{1}(\cdot)$ denotes the indicator function and $X_{i,j}$ denotes the element belonging to X in the ith row and jth column.

This feature is an example where digits are not separated by the handcrafted feature particularly well. That said, when included in the handcrafted features dataset, model performance improves noticeably and so it is included.

3.2 Row and column sums

Similarly to the row and column non-zero counts, in this feature we compute the sum of the pixel values in a given row or column. That is, we compute and store both

$$\mathcal{F}^{(i)}_{\mathbf{rs}}(X) = \sum_{j=1}^{15} X_{i,j}$$

$$\mathcal{F}^{(j)}_{\mathbf{cs}}(X) = \sum_{i=1}^{16} X_{i,j}$$

for $i = 1, \ldots, 16$ and $j = 1, \ldots, 15$. As with the first handcrafted feature, the inclusion of this feature improves model performance but does not individually distinguish digits particularly well.

3.3 Vertical ratios

A relatively intuitive handcrafted feature that relates to the vertical symmetry of an image is that of vertical ratios. The vertical ratio of an image is the ratio of the sum of all pixel values in the first k rows to the sum of *all* pixel values. The subtlety of summing over the first k rows instead of just the first eight (which pertains to the vertical symmetry of the image) is that different values of k help distinguish different digits. To compute the vertical ratio of an image while summing over the first k rows we compute

$$\mathcal{F}_{\mathbf{vr}}^{(k)}(X) = \frac{\sum_{i=1}^{k}\sum_{j=1}^{15} X_{i,j}}{\sum_{i=1}^{16}\sum_{j=1}^{15} X_{i,j}}.$$

It should be noted that this idea of taking the first k rows in the sum of the numerator is not our own. The idea was taken from [13] in which they show that $\mathcal{F}_{\mathbf{vr}}^{(3)}$ helps to distinguish 7 and $\mathcal{F}_{\mathbf{vr}}^{(8)}$ helps to distinguish 8. The latter can be seen to an extent in Figure 2 in which we see that 8 and 9 have minimum vertical ratio values of around 0.55 and maximum values of around 0.75 while all other digits lie at lower values.

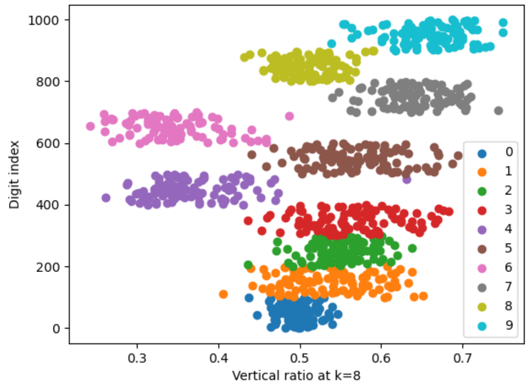

Figure 2: The vertical ratios of all training samples for each digit where $k = 8$.

3.4 Hole area

When considering the pixel intensity distribution of a given image, a natural observation is that digits such as 0, 6, 8 and 9 have hole-like areas, or closed contours. This distinction in hole area between digits makes for a helpful handcrafted feature.

In implementing a program that computes the total area of holes (closed contours) in an image, we made use of the `cv2` library [2] which offers a plethora of image processing functionalities. For the hole area feature, we make use of its `findContours` and `contourArea` functions.

In their application, we first 'binarise' the image. That is, we apply

$$\tau(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases}$$

to each pixel in the image. We then use `findContours` to traverse the binarised image beginning from the top left pixel. From the point that a non-background pixel is found (a pixel whose value is 1 in our case) it continues to traverse the image taking into account what sort of border these series of non-background pixels enclose, resulting in a contour. After all closed contours are found in the image, we use `contourArea` to compute the total area of the holes belonging to the image. This function uses a numerical implementation of the application of Green's theorem to the area of closed contours [11]. We expectedly found that greater hole areas are found in 0, 8, 6 and 9 in descending order (Figure 3).

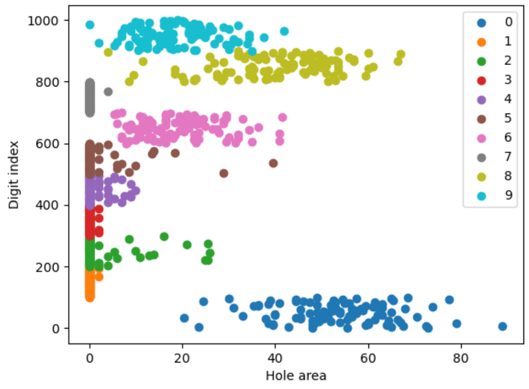

Figure 3: The distribution of hole area in the training data.

3.5 Prototype matching

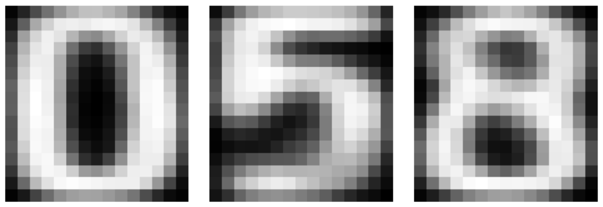

Figure 4: Prototype constructions of 0, 5 and 8.

By constructing an 'average' (or prototype) representation of a given digit, one can compare the similarity of a given sample by taking its dot product with said prototype.

In computing the prototype representation of a given digit we simply compute the average of each of the 240

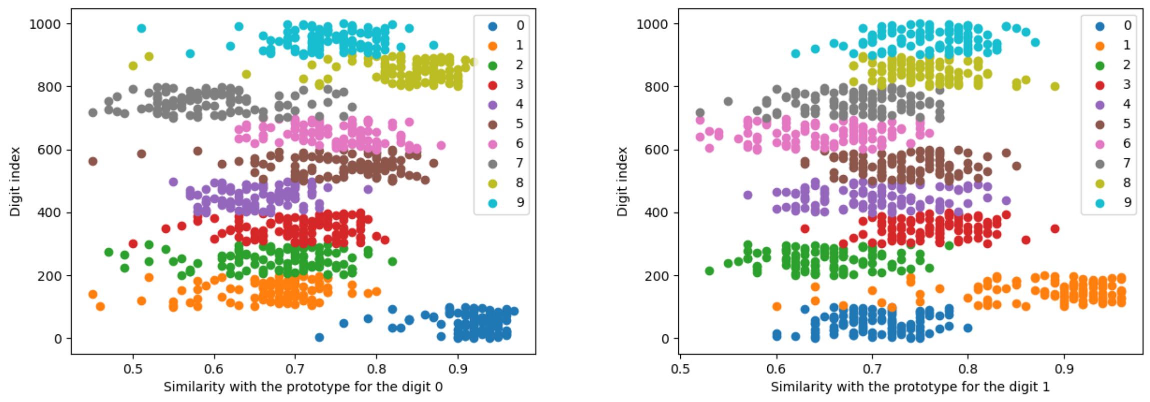

Figure 5: The prototype matching values of all training samples for 0 (left) and 1 (right).

pixels over the relevant samples. That is, we compute

$$\mathcal{P}_d = \frac{1}{N}\sum_{i=1}^{N} d_i$$

for each digit $d \in \{0, \dots, 9\}$ where d_i is the ith sample corresponding to d and N is the number of such samples. From here, the prototype feature $\mathcal{F}_{\mathbf{pr}}^{(d)}$ corresponding to digit d is the similarity of a given image I and this prototype $\mathcal{P}_d$ given by their dot product $\mathcal{F}_{\mathbf{pr}}^{(d)} = I \cdot \mathcal{P}_d$.

As can be seen in Figure 5, individual digits are distinguished *very* well when compared to their constructed prototype. This is of course expected: a given digit's sample is of course most similar to others similar to it.

3.6 Histogram of Oriented Gradients (HoGs)

For the first elaborate handcrafted feature we make use of the histogram of oriented gradients (HoGs) whose purpose is to detect objects within an image via the intensity of the pixel values along detected edges.

We implement this using `cv2`'s `HOGDescriptor` function which first computes the gradients of each pixel in the image X in matrix form via

$$\nabla^x_{i,j} = X_{i,j+1} - X_{i,j-1}$$

and

$$\nabla^y_{i,j} = X_{i-1,j} - X_{i+1,j}$$

for $i = 1, \dots, 16$ and $j = 1, \dots, 15$ where negative indices of X are given by 0. From here, the gradient's magnitude and angle is given by

$$\text{mag}(\nabla)_{i,j} = \sqrt{(\nabla^x_{i,j})^2 + (\nabla^y_{i,j})^2}$$

and

$$\theta^{\nabla}_{i,j} = \left|\arctan\left(\frac{\nabla^y_{i,j}}{\nabla^x_{i,j}}\right)\right|.$$

In this fashion, the gradient magnitude and angle matrices $\text{mag}(\nabla)$ and θ^{∇} are constructed. Then, the image is split into (typically) square cells of pixels and those cells are split into blocks. Using the sections of $\text{mag}(\nabla)$ and θ^{∇} corresponding to these blocks the values of said matrices are used to allocate values to n bins systematically. This n-bin histogram is the output of `HOGDescriptor` and is what is used to extract the relevant features.

In our case, the image is split into cells of 6 by 6 pixels, the blocks are made up of four cells in a 2 by 2 formation and me choose $n = 10$ bins. Our choice of cell and block sizes were mostly the result of model performance. Given that our image size is 16 by 15, there are not many choices of cell or block sizes that produce an appropriate number of output features and so brute force assessing model performance on each choice was possible. For the remaining details on the full HoGs process consider [4].

3.7 Gabor Filters

The second elaborate handcrafted feature is that of Gabor filters [9]. Our implementation makes use of `cv2`'s `getGaborKernel` which consists of computing the convolution filter $g(x, y; \lambda, \theta, \psi, \sigma, \gamma)$ and 'masking' a given image using this filter. Briefly put, these parameters dictate the filter's sensitivity to a variety of characteristics of the image it is used to filter, such as $\psi \in [-\pi, \pi]$ corresponding to the filter's sensitivity to lines or well-defined bars within an image. As a whole, these values determine the convolution filter as follows: the row and column dimensions of the square filter must be odd as to ensure the existence of a centre pixel whose coordinates we denote by $(0, 0)$. From here, the element of $g = g(x, y; \lambda, \theta, \psi, \sigma, \gamma)$ that is x elements horizontally and y elements vertically from this centre is given by

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\Big(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\Big)\cos\Big(2\pi\frac{x'}{\lambda} + \psi\Big)$$

where $x' = x \cdot \cos(\theta) + y \cdot \sin(\theta)$ and $y' = -x \cdot \sin(\theta) + y \cdot \cos(\theta)$ account for the orientation of the filter. In our case, we construct an 11 by 11 convolution filter using $\lambda = 10$, $\psi = 0$, $\sigma = 3$ and $\gamma = 0.8$ and we vary the orientation $\theta \in \{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\}$ making for four separate applications of our Gabor filter. From here, an image X is filtered using g to obtain the filtered image $X^{\mathbf{filt}}$ via

$$X^{\mathbf{filt}}_{i,j} = \sum_{m=-5}^{5} \sum_{n=-5}^{5} X_{(i+m,j+n)} g_{m,n}.$$

Note that the indices for g are oriented around its centre pixel, for example $g_{-1,2}$ corresponds to the element of g that is one element left and two above its centre pixel. In the context of negative indices of the image X, this is accounted for by padding the image with a sufficient number of 0s.

Once computed for $\theta \in \{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\}$ we extract the mean and standard deviation of each filtered matrix which form our features. That is,

$$\mathcal{F}^{\theta}_{\mathbf{gf\_avg}} = \text{mean}(X^{\mathbf{filt}})$$

$$\mathcal{F}^{\theta}_{\mathbf{gf\_std}} = \text{standard\_deviation}(X^{\mathbf{filt}}).$$

Admittedly, the parameters chosen for our Gabor filter were not fine tuned as the improvement of each model after their inclusion using the default parameters was already noticeable. On top of this, the difference in model performance did not seem to change noticeably after altering parameters.

4 Methods

We discuss our two baseline methods consisting of a kNN and a random forest followed by our two more elaborate methods: a CNN and an autoencoding-equipped SVM. The hyperparameter tuning of each method is considered including other relevant criterion for an optimal model. This section is followed by a discussion of the results of each method, including possible explanations for unexpected model behaviour and unusual samples.

4.1 *k*-Nearest Neighbors

The k-nearest neighbor algorithm is a simple but robust classifier, requiring no training and works well with smaller datasets. It classifies test data by finding the k-closest (by a chosen distance function) data points in the training data, and classifying it by a chosen voting method. More information about kNNs can be found in the 2023 review by Syriopoulos et al. [12].

In this project, the `KNeighborsClassifier` class of the Python package `scikit-learn 1.3.0` was used [10]. The brute force algorithm and Euclidean distance function were used. The Euclidean distance function is the most often used in kNNs [12]. Also, for the purposes of this project, only the simplest variant of kNN is considered, though more elaborate variants exist.

Since the range of values possible in different features is different, and kNNs rely on distances, the variance in every dimension has to be normalised. Therefore, some preprocessing of the data is needed. The same distance along every dimension should be equivalent to the same degree of *closeness*. The simplest solution for this was the inclusion of `StandardScaler` in the pipeline. This scaler moves the data points, so along every dimension a zero mean and unit standard deviation is achieved. This scaler might be preferable to one which relies on the minimum and maximum data points along the dimensions, as it is less sensitive to outliers.

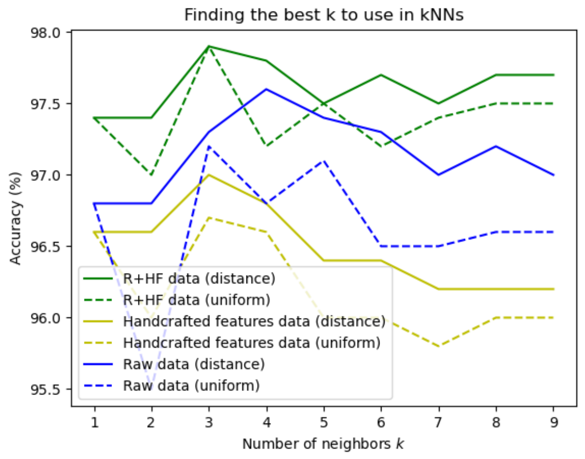

Figure 6: Finding k and comparing weighting methods for neighbors

There are two remaining important choices, the discussion of which is left for the remainder of this subsection. The two relevant parameters were k and the choice of voting algorithm. These were subject to hyperparameter optimization through cross-validation. The dataset contained 2000 digits, 1000 of which were chosen as training data, the other 1000 of which as test data. Values of 1 to 9 were tried for k. The choices for the voting algorithm possible in the package were *uniform* and *distance*. In the latter, training data points that are closer to the test data point have a larger effect on the outcome of the voting, unlike the former. All combinations of these hyperparameters were tried for three different datasets: the raw (pixel) data, the handcrafted feature data, and a combination of both. The results of the classification can be seen on Figure 6. The distance voting function always performed better, and the ideal k for this problem seems to be $k = 3$.

Finally, it should be noted that a few interesting things can be seen in Figure 6. As one would expect, the two voting functions are identical for $k = 1$, and the distance voting function gives identical results for $k = 1$ and $k = 2$. Furthermore, the uniform voting function seems to be doing substantially worse when k is an even number at lower k values, probably due to tiebreaker situations, where distance weighing gives better results.

4.2 Random Forest

A Random Forest (RF) model is an ensemble learning method that leverages the power of multiple decision trees to enhance classification accuracy and robustness. Each tree independently analyzes the features extracted from the digit images, ultimately culminating in a collective prediction through majority vote.

The core aspects of a RF method are:

Bagging: training data is randomly sampled with replacement (bootstrapping), creating new datasets (bootstrap samples) of the same size. Each tree is trained on a unique bootstrap sample, leading to diversified decision trees, a technique known as bagging. This mitigates overfitting by preventing any single tree from overly specializing on specific data points.

Feature Randomness: at each node in a decision tree, instead of considering all available features for splitting, only a random subset (determined by the parameter `max_features`) is evaluated by the information gain criterion. This injects further diversity into the forest, reducing reliance on any single feature and improving robustness to irrelevant or noisy features.

Prediction: each tree in the forest makes its own prediction for the class of a new digit image based on its learned decision rules. The final prediction of the RF is obtained by aggregating the individual prediction through majority vote. This ensemble approach capitalizes on the strengths of individual trees while reducing their weaknesses, leading to improved generalization performance.

We used the `RandomForestClassifier` from `scikit-learn` [10] and performed a search for the best values of the parameters. The results of the search are in Table 1. Here is an explanation for the parameters' function and choice:

- `n_estimators` is the number of trees in the forest. Increasing the number of trees generally improves the model's performance. However, there are diminishing returns after a certain number of trees (Figure 7). We obtained the best results with 147 trees on raw data, 167 trees on handcrafted features, and 81 trees on the combination of both.

- `min_sample_split` is the minimum number of samples required to split an internal node. Small values lead to deeper trees that might capture more details from the training data, with the risk of increasing overfitting. The best results were obtained with a minimum number of samples of 3 on raw data and handcrafted features, while the best value for the combination of both is 5.

- `max_features` is the number of features to consider when looking for the best split, influencing the diversity of individual trees. The value `"log2"`, meaning it considers the base 2 logarithm of the total number of features, was the one that delivered the best results for raw data and handcrafted features. Instead, on the combination of both the best value was `"sqrt"`, i.e. considering the square root of the total number of features.

- `criterion` is the function used to measure the quality of a split. The difference between Gini and Entropy measure were not significant, showing that the dataset and the features do not have a strong preference for one over the other. We chose Gini as the default one.

It is important to notice that there exist many combinations of parameters that would lead to results comparable with the best ones reported here, and even the best values were not unique. Given the best results in accuracy, we chose the values with the least number of trees, because the increased number of trees implies an increased computational cost.

	Best values for dataset type		
Parameter	**Raw Pixel**	**HF**	**Both**
`n_estimators`	147	167	81
`min_sample_split`	3	3	5
`max_features`	`"log2"`	`"log2"`	`"sqrt"`
`criterion`	`"gini"`	`"gini"`	`"gini"`

Table 1: Tuned RF classifier parameters for datasets composed of raw pixel data, handcrafted features data, and the combination of both.

4.3 CNN

Convolutional Neural Networks (CNNs) are a method of deep learning often used for image classification. It gets its name from the use of convolutions layers.

Convolutional layers are composed of several filter. These filters perform the convolutional operation over the input of the layer, and are able to detect certain patters (depending on the filter), that is why convolutional layer are said to be feature extractors. A convolutional layer consists of n filters. The filters in a convolutional layer will all have the same kernel size but can differ per layer. Filters can be 3-dimensional but in this report we will only be making use of 2-dimensional filters. Filters will scan over the input and map this to a new feature map. Depending on the step-size (known as 'stride') this

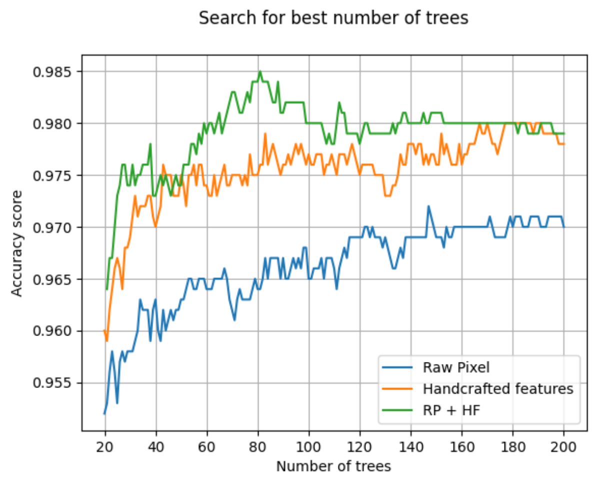

Figure 7: Behaviour of the accuracy as a function of the number of trees in the Random Forest classifier working on raw pixel data, handcrafted features data, and the combination of both. We notice a fast increase of accuracy up to ~ 60 trees. After that, the increase becomes slower and dominated by the randomness of the classification process. The data is calculated with the best parameters values shown in Table 1.

can reduce the dimensions of your feature map. However, it is possible to add padding around the input to avoid this.

It is also possible to use pooling layers. These layers are used to down sample the features from the convolutional layers. Typically pooling layers have a dimension of 2 by 2 and a stride of 2. This means that they will reduce the dimensions of the feature map by half.

After the convolutional and pooling layers, CNNs also make use of flattening and dense layers. Flattening simply flattens the feature map into a 1-dimensional vector. Dense layers change the dimensions of the array. This is done through matrix multiplication, if we multiply an array with dimensions m by n and a dense layer with dimensions n by k, the resulting array will have the dimensions m by k. Dense layers are the classification part of the CNN. They are thus used as final layers to obtain a prediction.

The final architecture that was used in this report is shown in figure 8. It uses four convolutional layers in total. The first using a 5 by 5 filter and the remaining 3 using a 3 by 3 filter. The final dense layer compresses it to a 10 dimensional vector since the classification deals with 10 classes. The model was build and trained using TensorFlow[5].

In order to compile the CNN an optimizer, a loss and a metric are chosen. These are used during the training of the model. The optimizer is the method that gets used to chose the weights associated with each filter.

The loss function compares the predicted outcome with the truth value of the image. For classification problems typically an entropy loss function is used. Since this problem has more than two classes and integer class inputs are used (instead of hot-one encoding), the sparse categorical crossentropy class is used.

The metric, like the loss function, is a way to evaluate the performance of the model. Since we are not dealing with a binary classification issue the most logical metric to use is the sparse categorical accuracy.

4.4 Autoencoding-equipped SVM

For our final method, we combined an autoencoder with a support vector machine (SVM). We first discuss autoencoding, an unsupervised neural network used to reduce the dimensionality of a dataset. We then consider the performance of such an autoencoder with a support vector machine as the core model.

To implement our autoencoder we used TensorFlow's Keras library [3] and for our SVM, scikit-learn's [10] implementation is used. We will now discuss some of the details of how the relevant parts of said libraries are implemented as well as our hyperparameter tuning efforts.

4.4.1 Autoencoding

In the context of our report, an autoencoder can informally be thought of as a more elaborate version of principal component analysis (PCA) with some key additional features. The purpose of an autoencoder, like PCA, is to learn to represent a dataset in a way that preserves the underlying structure to a sufficient extent. Once an autoencoder is trained, it consists of an encoder and a decoder. The encoder is used to reduce the dimensionality of a given image and the decoder is used to reconstruct an encoded image sufficiently well. Autoencoders offer significant advantages for our task over PCA such as the lack of a need for inherent linearity in the data it learns to compress. This is helpful for us as our data is in no way linear. Secondly, for reconstructing an encoded image, autoencoders offer far less loss for non-linear data such as ours. To add to this, autoencoders unsurprisingly offer far better flexibility in the form of many tuneable hyperparameters such as learning rate, the number of hidden layers and the activation function of the input and output layers.

Our simple autoencoder makes use of an input layer consisting of 240 neurons (matching the number of pixels in a given sample), one hidden layer consisting of 64 neurons and one output layer of 240 neurons. The intention of choosing far fewer neurons in our hidden layer is for it to act as a 'bottleneck' for the learning process. As a result, the encoder is made to learn to represent the input data using far fewer dimensions than 240. The exact choice of 64 neurons is the result of performing 10-fold cross validation on the range of values of 48 to 128 neurons.

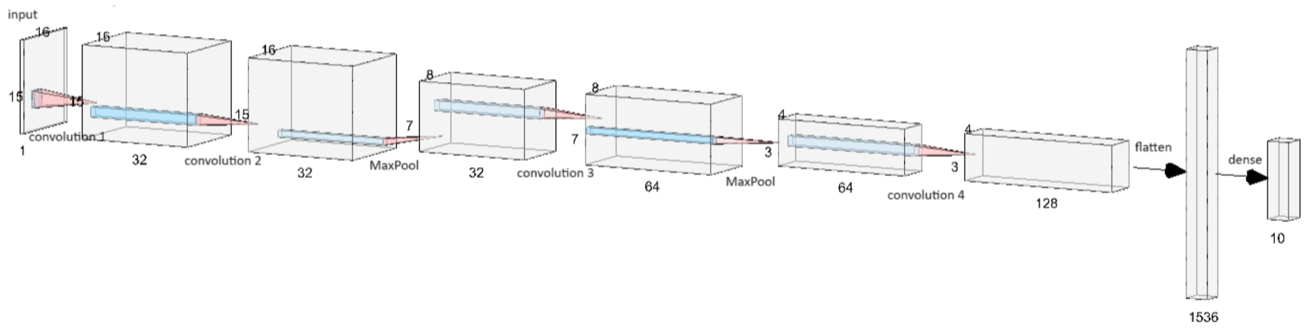

Figure 8: Architecture of the CNN model

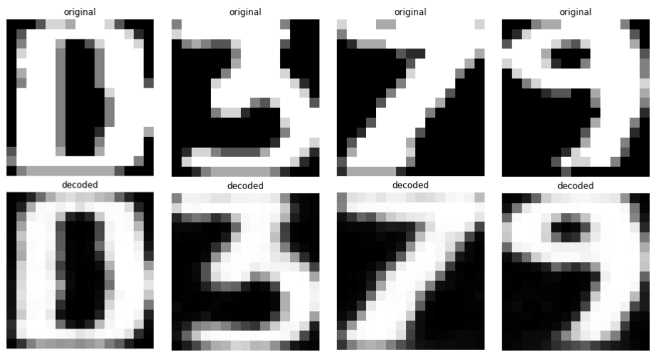

Figure 9: Original and decoded compositions of some 0, 3, 7 and 9.

During learning, the quality of the output of our autoencoder is assessed using a loss function for which we chose the mean squared error. That is, the quality of some output $\hat{y}$ as compared to its 'true' value y is given by

$$\mathrm{MSE}(y, \hat{y}) = \frac{1}{240} \sum_{i=1}^{240} (y_i - \hat{y_i})^2$$

where both are in vector form and y_i and $\hat{y}_i$ denote the ith pixel of each image. During tuning, the choice of our loss function did not seem to influence the accuracy of our SVM heavily hence our decision of going with this relatively simple loss function.

In choosing our encoding and decoding activation functions we simply assessed the accuracy of our model via misclassification rate using the Sigmoid, ReLU, Softmax and tanh functions. Since we first normalised the pixel values to within $[0, 1]$ by dividing each by 6, we first considered the Sigmoid function, given by

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

for our encoding activation function. This function can be seen as one that returns probability-like values given some real number x. Surprisingly, this performed noticeably worse than the rectified linear unit activation function (ReLU) given by

$$f(x) = \max(0, x).$$

It is not immediately clear why but the performance is noticeably better. For the decoding activation function, the accuracy using the tanh and Softmax function was noticeably worse than ReLU and the Sigmoid function. We ultimately chose the Sigmoid function as it seemed appropriate in that it output pixel values in $[0, 1]$, matching the pixel values in the normalised input data, and because the performance of the model was seemingly the same.

Onto the most interesting component of an autoencoder: the choice of optimisation algorithm. The purpose of this optimisation algorithm is to minimise the loss function, in our case to minimise the mean square error of the model's output, during training. This is done by updating each parameter of the model by edging them in the direction in which the gradient of the loss function with respect to each parameter is minimised. The parameter space in a model like this can be immensely complex in its composition and so finding a point at which the parameters minimise the loss function is not straightforward.

After considering Adagrad, RMSprop, Adadelta, Adam and Nadam, we chose Nadam whose full title is Nesterov-accelerated Adaptive Moment Estimation. Nadam is an adaptation of Adam [6] which incorporates Neseterov momentum which uses a parameter 'look ahead' mechanism to compute the next iteration of parameters. That is, given suitable initial parameters θ_0 and gradient $g_{t+1} = \nabla_{\theta_t} L(\theta_t)$, Nadam iteratively updates these parameters via

$$\theta_{t+1} = \theta_t - \eta \frac{\bar{m}_t}{\sqrt{\hat{n}_t} + \epsilon}$$

for $t \in \mathbb{N}$. Here, η is the learning rate, typically a small value of say 10^{-3} which dictates how quickly the model converges. A smaller value pertains to a slower converging but more stable and less overfitting model. A value of around 10^{-8} is given to the stability parameter ϵ whose purpose is to ensure that there is no division by 0 in iterating over new parameter values. Additionally, $\bar{m}_t$ is the Nesterov-accelerated gradient making for the predominant adaptation from the Adam optimisation algorithm. This is given by

$$\bar{m}_{t+1} = (1-\mu_t)\hat{g}_t + \mu_{t+1}\left(\frac{\mu \cdot m_t + (1-\mu)g_t}{1-\prod_{i=1}^{t+2}\mu_t}\right)$$

where

$$\hat{g}_t = \frac{g_t}{1-\prod_{i=1}^{t+2}\mu_t} = \frac{\nabla_{\theta_{t-1}}}{1-\prod_{i=1}^{t+2}\mu_t}$$

is the normalised gradient and $\mu_t = \mu\big(1-\frac{1}{2}\cdot 0.96^{\frac{t}{250}}\big)$ is the momentum schedule, in line with [6]. The final component is that of the second moment estimate given by

$$\hat{n}_{t+1} = \frac{\nu n_t + (1-\nu)g_t^2}{1-\nu^t}.$$

Though Adam itself performs relatively well in our use case, Nadam offers faster convergence and slight but noticeable improvements in the number of misclassifications made by the model.

As the final component of fine-tuning the SVM, we consider the number of epochs and the batch size pertaining to the fitting of the autoencoder. For this we simply iterated over a series of possible values using 10-fold cross-validation. For the number of epochs we iterated over $\{32,\ldots,128\}$ and found that 97 epochs gave the smallest validation error during cross-validation. Similarly, for the batch size number we iterated over $\{64,\ldots,192\}$ and performed 10-fold cross-validation ultimately choosing a batch size of 108.

4.4.2 Support Vector Machine (SVM)

The core model paired with our autoencoder is an SVM. While simpler in concept than autoencoding, their application to even small datasets like ours yields impressive results. At its core, an SVM is a linear model that fits a hyperplane in the space of features such that it separates classes while also maximising the distance of the nearest points of each class to the hyperplane. Note that in our case, the SVM is not applied to the raw data but instead to the encoded raw data.

The application of an SVM to binary classification can be formulated as follows. In our n-dimensional feature space, a hyperplane is produced by the set of points $\mathbf{x}$ that satisfy the affine linear map

$$\mathbf{w}\cdot\mathbf{x} - b = 0$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of weights (or coefficients), $b \in \mathbb{R}$ is a bias scalar that dictates the elevation of the hyperplane with respect to the origin and $\cdot$ denotes the dot product. The points of each class separated by this hyperplane that are closest to $\mathbf{w}$ are the support vectors. In training an SVM, the intention is to ensure that these support vectors are as far as possible from the hyperplane. Since these points are (ideally) completely separated by the hyperplane, and our feature space is non-linearly separable, this is done using the soft-margin method which utilises the hinge loss function

$$\max(0, 1-y_i(\mathbf{w}\cdot\mathbf{x}_i - b))$$

by minimising

$$\frac{1}{N}\sum_{i=1}^{N}\Big[\max(0,1-y_i(\mathbf{w}\cdot\mathbf{x}_i - b))\Big] + \lambda||\mathbf{w}||^2$$

where $(\mathbf{x}_i, y_i) \in \{(\mathbf{x}_1,y_1),\ldots,(\mathbf{x}_N,y_N)\}$ is the ith sample of the dataset, both $\mathbf{w}$ and b are as before and λ is a positive real number that helps to ensure that each feature vector $\mathbf{x}_i$ remains on the correct side of the margin of the hyperplane. This reduces to minimising

$$C\sum_{i=1}^{N}\xi_i + ||\mathbf{w}||_2^2 \qquad (1)$$

such that

$$y_i(\mathbf{w}\cdot\mathbf{x}_i - b) \geq 1-\xi_i$$

for all $i = \{1,\ldots,N\}$ with $C \geq 1$ and $\xi_i \geq 0$. The appropriate naming of the soft margin method is clear as this allows for support vectors to be within a non-hardline distance to the hyperplane. We also observe that given a large value of $C \geq 1$, each time a misclassification is observed, a large penalty is applied to the model, and so we would expect to have smaller values of ξ_i yielding something similar to the hardline distance of ensuring that

$$y_i(\mathbf{w}\cdot\mathbf{x}_i - b) \geq 1.$$

As such, a higher value of C can be thought of as ensuring a smaller error within the model. That said, the choice of C should also take into account the possibility of overfitting.

In our implementation, we use `scikit-learn`'s [10] implementation of SVMs which offers us the flexibility of which kernel to use along with any corresponding parameters. We chose the Radial Basis Function (RBF) kernel whose use we now discuss. The RBF kernel is given by

$$K(\mathbf{x}_1,\mathbf{x}_2) = \exp\Big(-\frac{1}{2\sigma^2}||\mathbf{x}_1-\mathbf{x}_2||^2\Big)$$

where σ is used as a normalisation of the distance between $\mathbf{x}_1$ and $\mathbf{x}_2$. In tuning our model, we took C, as in 1, in a way that minimised the risk of overfitting. We ultimately chose a small penalty of $C = 1.2$. That is, our model aims to minimise

$$1.2 \sum_{i=1}^{N} \xi_i + ||\mathbf{w}||_2^2$$

such that

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 - \xi_i$$

for all $i = 1, \ldots, N$.

Finally, we discuss how our model decides which label to classify a given sample. Intuitively, the classification of a given sample can be thought of as simply the side of the hyperplane in which it lies. Ultimately, this decision is made by choosing a number of sufficiently close support vectors $\{s_1, \ldots, s_k\} \in SV$ and computing

$$D(x) = \sum_{i \in SV} y_i \alpha_i K(x, s_i) + b$$

where α_i is a lagrange multiplier associated with the support vector s_i and y_i is its label. Finally, simply taking the sign of $D(x)$ returns the model's decision. Admittedly, finding an explicit form like this was difficult and so we relied on an answer on the machine learning stack exchange [14].

The reader will likely notice that this section, up until now, has focused only on the use of SVMs for binary classification. While initially developed for binary classification tasks, SVMs can be extended to multiclass classification tasks via the *one-vs-one strat*. The one-vs-one strat involves learning a binary SVM between all class pairs (so each pair of digits in our case). For us, there are 10 classes, corresponding to the number of digits, and so this involves learning $\binom{10}{2} = 45$ SVMs. Once learned, given an image, the overarching classifier uses a voting strategy, reminiscent of Adaptive Boosting [8], weighting the outputs of all 45 binary classifiers in a way that results in the most reliable final decision.

5 Results and Discussion

Class. method	**Data form**	**Error**
kNN ($k = 3$)	Raw pixel values	2.7%
kNN ($k = 3$)	Handcrafted features	3.0%
kNN ($k = 3$)	Raw + Handcrafted	2.1%
Random Forest	Raw pixel values	2.8%
Random Forest	Handcrafted features	2.0%
Random Forest	Raw + Handcrafted	1.5%
CNN	Raw pixel values	1.8%
A.E. + SVM	Raw pixel values	1.1%

Table 2: The rate of misclassification on testing data of each model by data form.

To conclude, we consider the misclassification rate of each method, including interesting cases of misclassified images. Potential improvements to each model are then discussed with these edge case misclassification in mind.

5.1 kNNs

The k-nearest neighbor (kNN) classifier was tried on the raw data, feature data and both datasets combined. The smallest test error was reached when k was set to 3, and the distance weight function was used. An accuracy of 97.0 % was reached when it was used to classify the test data based on their features. The best accuracy reached was 97.9% (21 misclassifications in the test data of 1000) for the combined raw and feature data. These results can be seen in Table 2.

The main reason for the inclusion of kNNs in this project was for comparison with random forests (which they share handcrafted features with) and other more sophisticated methods.

In this project, it is notable, that including the raw pixel data in the training data used for kNNs gave better results. In fact, unlike the random forests, only using the raw pixel data performed better than only using the features. Unfortunately, when using larger training datasets, especially with higher resolution pictures, the computational cost for doing this might be too high. Ideally, the performance on the lower dimension data (handcrafted features) is a better indicator of how usable and scalable the approach might perform in real life.

How could the rate of misclassifications be reduced when only using the feature data as training data? One of the main sources of error in kNNs is class overlap [12]. Therefore, the best features for kNNs are those, which help distinguish classes with the least amount of overlap. New features should be chosen specially to reduce class overlap in the dimension reduced space. This can be achieved by crafting more handcrafted features, or by using other dimension reduction approaches, such as PCA or LCA. However, kNNs are also vulnerable to irrelevant or redundant features, which should be minimized, a certain sense of balance among the features should be achieved. The standard scaler used to equalize the distance along dimensions is also simplistic. Distances along different dimensions might not translate to the same distance in conceptual space using only a simple scaler. Perhaps an ideal scaling factor could be determined through cross-validation for every dimension individually.

5.2 Random Forests

The results obtained using the Random Forest Classifier show varying accuracies for different types of input data: combination of data, handcrafted features, and raw pixel data. Combining both raw pixel data and handcrafted features provides a more comprehensive representation of the input. While handcrafted features are designed to capture specific patterns and characteristics, the combination of data allows the model to benefit from both the inherent patterns in pixel values and the engineered features. In fact, the best result are achieved with the classifier working on the set of data containing both kinds of data, reaching an accuracy of 98.5%. On the feature data the best results reached an accuracy of 98%, showing that the features are well designed and informative. Finally, the accuracy reached on raw pixel data is 97.2%, lower then the others given that the dataset contains a large amount of redundant or irrelevant information, suggesting that a more complex model is needed to further optimize the result on this kind of input data.

5.3 CNNs

Multiple models were tested on raw image data and on augmented dataset that included the raw images in addition to rotated images. The images were rotated by varying angles θ in both the clockwise and counter-clockwise direction. The augmentation was only done on the training set and not on the test set.

The first thing that was done, was to compare the performance of a fairly standard model to augmented datasets. The results of this can be seen in table 3. They were compared over two training sessions with different epochs. This was done because a larger dataset might need more epochs to reach a stable result. From the table it is clear that the data augmentation does not improve the final results much. For most angles the model performs consistently worse. For datasets such as the $\theta = 3$ set the model performs significantly worse, 2 - 3 %. For 10 epochs the $\theta = 9$ set performs slightly better than the non-augmented set and for 50 epochs the $\theta = 10$ set performs slightly better. However, this is a very minor difference and is more likely due to a "lucky roll" on the weights of the CNN. This analysis was also done for rotations with up to a 45 degree angle but all of these showed significantly worse results than the raw dataset.

	10 epochs	**50 epochs**
No augmentation	0.951	0.963
θ=1	0.941	0.942
θ=2	0.940	0.935
θ=3	0.926	0.938
θ=4	0.937	0.938
θ=5	0.948	0.955
θ=6	0.927	0.959
θ=7	0.942	0.958
θ=8	0.938	0.962
θ=9	0.955	0.962
θ=10	0.947	0.967

Table 3: Comparison of CNN model's accuracies for different augmented datasets over two different numbers of epochs.

After running multiple training sessions over a number of different CNNs, with both the raw and the augmented data, the best performing model was obtained with the raw training data. CNN consisted of 4 convolution layers, the full architecture is shown in figure 8. The model reached a 98.2% accuracy, or a 1.8% misclassification rate after running for 15 epochs.

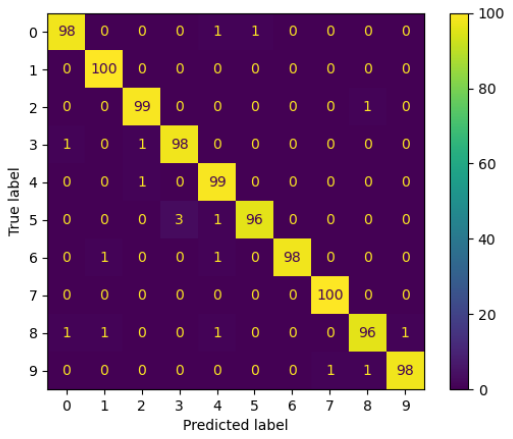

Figure 10: Confusion matrix for the CNN.

The confusion matrix is shown in figure 10. The model did not have issues with classifying the 1s and 7s correctly. The most commonly misclassified digits where the 8s and the 5s. Some of the digits that were misclassified seemed to not have been very clean data as shown in Figure 11.

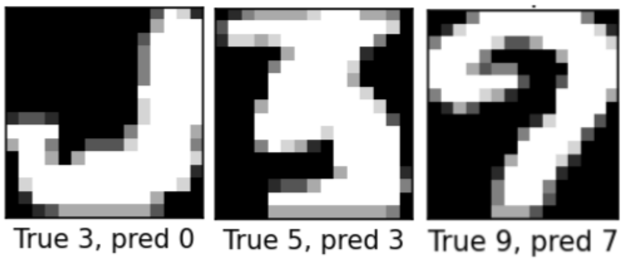

Figure 11: Misclassifications by CNN of digits that are difficult to classify by eye.

However, figure 12 shows a sample of the other misclassifications and these could very be due to the rotation of the digits. Due to misclassifications like this, one would expect that a dataset with the adding of augmented, rotated images would improve the performance of the model but this was not the case as discussed before. It was unclear what the reason for this was until plots were made of some of the rotated images where it became evident that rotation augmentation did not work as intended. Unfortunately (and stupidly) this was only noticed shortly before the deadline and there was not enough time to change the code and re-train the model.

We suspect that if we were to do this again with properly working augmentation, the model would see an improvement in accuracy.

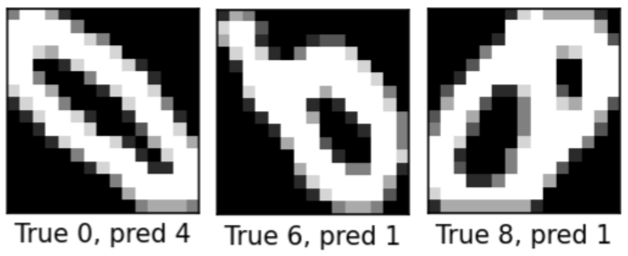

Figure 12: Misclassifications by the CNN that seem to be due to rotations.

The CNN performs only moderately better than the KNN and worse than the Random Forest with raw + handcrafted features. Since CNNs are a deep-learning method one would generally expect it to perform better than the baseline models. The main reason why this is likely not the case here is due to the small dataset. Since the CNN does not work with the handcrafted features, the only parameters it has are the pixel values of each image. This problem deals with images of 15x16 pixels and a trainingset of 1000 images. This gives the model very little data to train on, especially without expanding the dataset using data-augmentation.

Additionally, figure 11 shows that the test data did contain some atypical digits. This makes it not unreasonable to assume that there could be similar anomalies in the training data which could lead to inaccuracies.

A straightforward way to improve the performance of the CNN would be to check and clean up the training data for any digits that might cause issues and to perform better data-augmentation to increase the training set.

5.4 Autoencoding-equipped SVM

As the best performing model, our autoencoding-equipped SVM yielded an average misclassification rate of just 1.1% on the raw pixel values over 100 runs. Note that the variability of the misclassification of this model is attributed to the randomisation of the weights used at the beginning of autencoding as well as the partially stochastic nature of the Nadam optimisation algorithm. The confusion matrix for the output of the model is given in Figure 14.

The model performs effectively perfectly with regards to 1s, 2s, 4s, 7s and 9s while struggling minimally, sometimes completely understandably, with the remaining digits. To demonstrate, consider Figure 13 in which we illustrate four plots of misclassified digits. The third and fourth image in the figure can be described as a 6 and an 8 rotated anticlockwise by some noticeable amount.

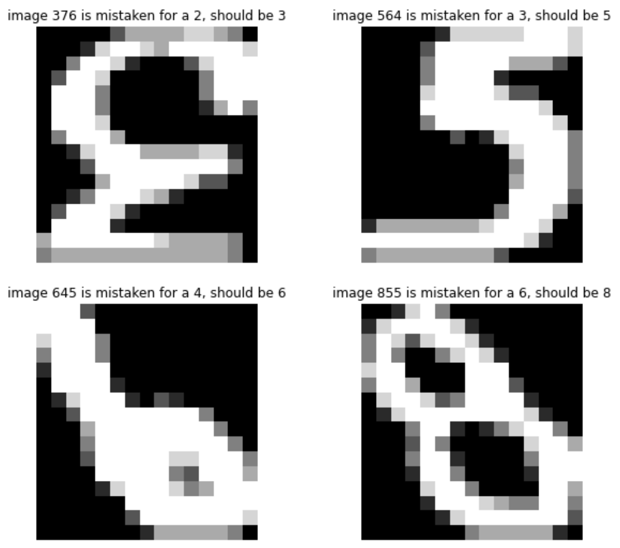
image 376 is mistaken for a 2, should be 3
image 564 is mistaken for a 3, should be 5
image 645 is mistaken for a 4, should be 6
image 855 is mistaken for a 6, should be 8

Figure 13: Collection of images misclassified by the autoencoding-paired SVM.

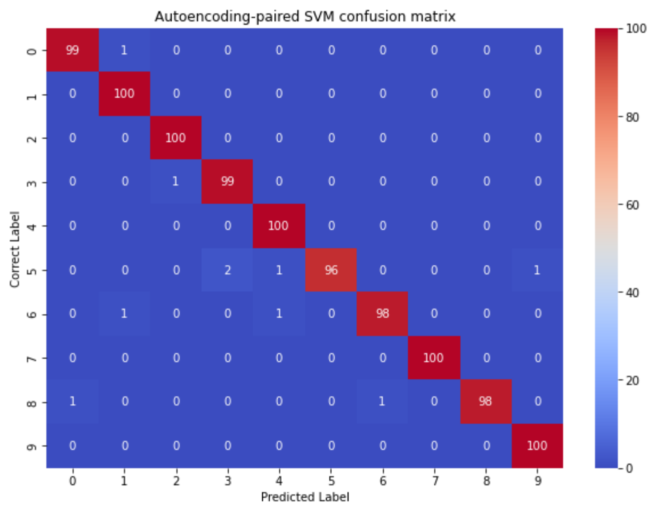

Figure 14: Confusion matrix for the autoencoder-equipped SVM.

The model could be improved to account for cases like these by simply augmenting the data to include small but noticeable rotations of the training data. For the second image, which is unambiguously a 5, the model unexpectedly mistakes it for a 3. This could be due to the long tail belonging to the bottom of the image but it is not immediately clear to us how the model could be improved to account for this misclassification. As for the first, it is rather clearly a 3 rotated 180 degrees. Following this is a rather odd case which would hopefully not reflect a handwritten 3 in practice. That said, such unexpected cases can occur and so to account for this, we could perhaps rotate a number of out training samples by 180 degrees before training. It is not immediately clear what portion of each digit this should be applied to: if we were to augment the data by augmenting *every* digit by 180 degrees then this may pose an overfitting-related problem. Perhaps just 10% of the samples pertaining to each digit.

6 Conclusion

The baseline methods: kNN and Random Forest were implemented as a baseline. Their expected risk was 2.7% and 2.8% respectively. Seven handcrafted features were also implemented as a method of dimension reduction, which improved the performance of both methods (to 2.1 % and 1.5 % respectively, when considering combined datasets). These methods were contrasted with two more advanced methods: CNNs and Autoencoder-equipped SVMs. These methods achieved relatively low error rates (1.8 % and 1.1 % respectively) without the need for engineering handcrafted features.

References

[1] Alejandro Baldominos, Yago Sáez, and Pedro Isasi. A Survey of Handwritten Character Recognition with MNIST and EMNIST. *Applied Sciences*, 9(15):3169, August 2019.

[2] Gary Bradski. Opencv. `https://opencv.org/`, 2020. Version 4.5.

[3] François Chollet et al. Keras. `https://keras.io`, 2015.

[4] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[5] TensorFlow Developers. Tensorflow, November 2023. `https://doi.org/10.5281/zenodo.10126399`.

[6] Timothy Dozat. Incorporating Nesterov Momentum into Adam. `https://cs229.stanford.edu/proj2015/054_report.pdf`. CS229: Machine Learning.

[7] Robert P. W. Duin and David M. J. Tax. Experiments with classifier combining rules. In *Multiple Classifier Systems*, pages 16–29, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[8] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[9] Dennis Gabor. Theory of communication. *Journal of the Institution of Electrical Engineers*, 93(26):429–457, 1946.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[11] James Stewart. *Calculus: Early transcendentals*. Cengage Learning, 2015.

[12] Panos K. Syriopoulos, Nektarios G. Kalampalikis, Sotiris Kotsiantis, and Michael N. Vrahatis. kNN Classification: a review. *Annals of Mathematics and Artificial Intelligence*, September 2023.

[13] Unknown. Handwritten Digit Recognition (Example paper 2 given in the course materials), 2022.

[14] Sanjar Adilov (`https://stats.stackexchange.com/users/191906/sanjar-adilov`). Why is Scikit's Support Vector Classifier returning support vectors with decision scores outside [-1,1]? Is this a mistake? `https://stats.stackexchange.com/q/564134` (version: 2022-02-12).