

# Machine Learning: Project

Karsten Standnes - STNKAR012

November 2017

## 1 Introduction

In the world today there are many methods that fall under the category of "Machine Learning", some being very similar and some very different. All of them share in common that they use data to produce a model that can be used to tell us something on previously unseen data. When successful this is very appealing in today's society where we got lots of data on situations where there is likely to be a underlying pattern, another requirement for learning. There is broad agreement that Machine Learning is a good way to make prediction and classification models, and often the only computational feasible way to do so. This makes it a task to decide which method in machine learning to choose for a given problem. The answer is not always the same and several methods can be good, but for different reasons. In this report I will utilize several of the most used machine learning algorithms and show how they perform on classifying images of handwritten digits.

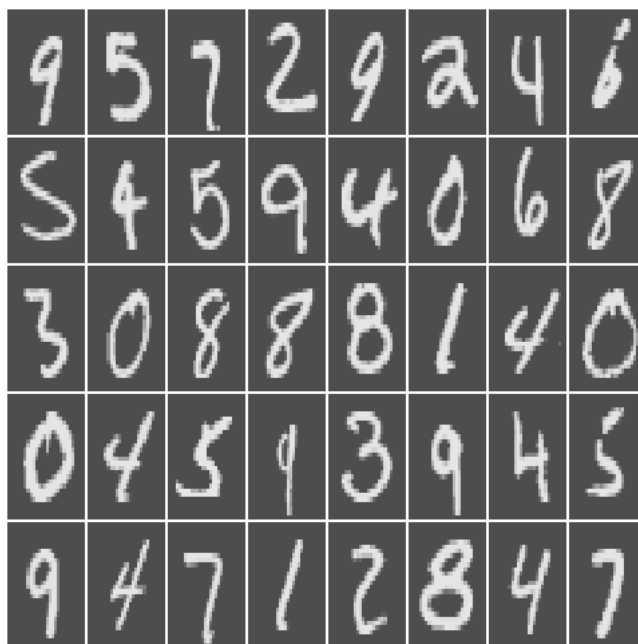


Figure 1: 40 digits from the data set

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theory</b>	<b>3</b>
2.1	Tree based methods . . . . .	3
2.1.1	Classification tree . . . . .	3
2.1.2	Random Forest . . . . .	3
2.1.3	Bagging . . . . .	3
2.1.4	Boosting . . . . .	3
2.2	Deep Learning . . . . .	3
2.2.1	Artificial Neural Network . . . . .	3
2.2.2	Convolutional Neural Network . . . . .	4
2.3	Support Vector Machines . . . . .	4
2.4	K-Nearest Neighbours . . . . .	4
<b>3</b>	<b>Methods</b>	<b>4</b>
3.1	Tree based methods . . . . .	5
3.1.1	Classification tree . . . . .	5
3.1.2	Random Forest . . . . .	5
3.1.3	Bagging . . . . .	5
3.1.4	Boosting . . . . .	6
3.2	Deep Learning . . . . .	6
3.2.1	Artificial Neural Network . . . . .	6
3.2.2	Convolutional Neural Network . . . . .	6
3.3	Support vector machines/PCA . . . . .	7
3.4	K-Nearest Neighbours . . . . .	7
<b>4</b>	<b>Results</b>	<b>7</b>
4.1	Tree based methods . . . . .	8
4.1.1	Classification tree . . . . .	8
4.1.2	Random Forest and Bagging . . . . .	9
4.1.3	Boosting . . . . .	11
4.2	Deep Learning . . . . .	12
4.2.1	Artificial Neural Network . . . . .	12
4.2.2	Convolutional Neural Networks . . . . .	13
4.3	Support vector machines w/ PCA . . . . .	14
4.4	K-Nearest Neighbours . . . . .	16
4.5	Summary . . . . .	17
4.6	Conclusion . . . . .	17
<b>5</b>	<b>Discussion</b>	<b>17</b>
<b>6</b>	<b>Appendices</b>	<b>18</b>
6.1	R-Code: Help Functions . . . . .	18
6.2	R-Code: Regression Tree . . . . .	19
6.3	R-Code: Random Forest . . . . .	22
6.4	R-Code: Bagging . . . . .	24
6.5	R-Code: Boosting . . . . .	26
6.6	R-Code: Plot Random Forest w/ Bagging . . . . .	29
6.7	R-Code: Neural Network . . . . .	31
6.8	R-Code: Convolutional Neural Network . . . . .	35
6.9	R-Code: K-nearest Neighbours . . . . .	41
6.10	R-Code: Support Vector Machines . . . . .	45
6.11	R-Code: Visualize data examples . . . . .	51

## 2 Theory

### 2.1 Tree based methods

#### 2.1.1 Classification tree

Classification tree is maybe the method in Machine Learning which is easiest to interpret due to it's intuitive construction and clear visualization. The method uses a greedy approach using recursive binary splitting to structure a tree that can classify input based on it's variables. The goal of the classification tree is to classify a set of data as best as possible while have a low complexity to avoid overfitting. Below we see that one classification tree is not enough to make a great model for the problem, but combining many of them gives us a "Random Forest" which is discussed in subsection 2.1.2. Classification trees also gives a nice visual image of the classification.

#### 2.1.2 Random Forest

Random Forest is as mentioned (subsection 2.1.1) constructed of many classification tree. After such a construction by using a set of training data, new data can be ran through the "forest". The new data is classified with the label the majority of trees labelled it. The trees in other words "vote" for the best classification for the data. As in real life, voting makes little difference if all the votes are the same. To avoid making a forest out of  $n$  ( $n \in \mathbb{Z}_{>0}$ ) identical trees two aspects are changed when the constructing the classification trees. The first is that the data of size  $N$  used in training a tree is sampled from the whole set of  $N$  data **with** replacement. This will on average lead to about  $\frac{2}{3}$  data to be used in creating each tree leaving on average  $\frac{1}{3}$  Out-Of-Bag(OOB) which is used for validation. The OOB-data is critical in making insuring the Random Forest doesn't overfit. The other is that  $m$  randomly selected variables are used for each tree, where  $m \ll M$  and  $M$  is the whole set of variables in the problem. This gives a rich variety of trees where with enough trees a predictive model can be created.

#### 2.1.3 Bagging

Bagging is basically a Random Forest (see 2.1.2) where the number of variables considered for each split is the whole variable set for the problem. It shares the same behaviour as a Random Forest when sampling out the data and in classifying new data. The big difference is that the variables are not sampled in Bagging, this does that Bagging creates trees that are more correlated than a Random Forest making it more susceptible to dominating features. This will not directly lead to a bad classification model, but it does have the unwanted consequence of making semi-important features of the data to be underestimated in the contribution towards new predictions.

#### 2.1.4 Boosting

Boosting takes a different direction to find make a good predictive tree model than the two previous tree based methods. Instead of growing  $n$  trees independently of each other, boosting "grow" trees sequentially moving towards a good model. Boosting is a slowly improving method and often use a small amount of splits for each tree. It also have a shrinkage parameter  $\lambda$  that controls the rate of change in the model. Boosting learns slower, which means more trees are required compared to the random forest or bagging method in order to achieve the same predictive power.

## 2.2 Deep Learning

### 2.2.1 Artificial Neural Network

Artificial Neural Networks got the name because it to some degree mimic the behaviour of neurons in the brain. It does so by connecting nodes (neurons) together with weights (synapses) connecting them. In the nodes that are not input nodes there is an activation function altering the values that are brought to the nodes through weights from other nodes. The activation is one of several factors

that can be tuned when creating a neural network. The relationship between the weights can be seen the equation for calculating the nodes in the hidden layers

$$x_j = \theta(s_j^{(l)}) = \theta(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}) \quad (1)$$

, where  $l$  indicate the layer,  $i$  the input,  $j$  the output and  $\theta$  the activation function. Other factors that can be altered is the size of the hidden layers and the amount of nodes in each layer of the network. This method learn through updating the weights of using stochastic gradient descent(SGD). This is applied through the backpropagation algorithm. This method takes run data forward through the network and the backpropagates back updating the weights based on the error in the output using SGD. The backpropagation will penalize weights that contributed a lot to the error more than those less relevant. Neural networks are exposed to overfitting, but there are several techniques that can be used to decrease the chance of overfitting. Some of these are validation, using dropout in input and/or hidden layers and regularization.

### 2.2.2 Convolutional Neural Network

Convolutional Neural Network(CNN) is in theory a great method to classify digits because of it's great reputation in performance for problems that can be represented as an "image". This because it CNN's works by recognizing patterns in problems that can be represented as by matrix or tensor(multidimensional matrix) in the computer. It does so by using convolutional, pooling and voting layers. Multiple of these can be stacked to create a very a precise model for recognising images.

## 2.3 Support Vector Machines

Support vector machines(SVM) is a powerful methods used either to classify data or in regression analysis. SVM tries as big a margin as possible between different categories. This is strived after to decrease the number of possible dichotomies. SVM's are powerful because of the ability to classify non-linearly separable data by transforming/mapping the data to higher dimensional space, using the inner product to classify the data. SVM's works well when using feature extraction like Principal Component Analysis(PCA) to reduce the amount of features to evaluate.

## 2.4 K-Nearest Neighbours

K-Nearest Neighbours(KNN) is a easy-to interpret and maybe one of the simplest methods in Machine Learning. The method does exactly as the name suggest; find the  $k$  nearest neighbours, where  $k$  is a number and labels each data point based on those neighbours. KNN doesn't learn in the fashion random forest or neural net does, it rather just classifies data using the most similar known data. The method is attractive because of it's easy interpretability, fast execution time and often precise classification and regression power. There are many variants of the method where the distance measure and way to handle draws differ. KNN's with  $k = 1$  have a special property that the hyperplane of the data points will be partitioned into an equal amount of partitions as there are data points. Each partition consist of all point in the plane closer a specific data point than any other. The amount of neighbours  $k$  can have a great impact on the accuracy of the classification. In some cases a low  $k$  will be preferable with very separated data, in other case it can lead to overfitting due to higher influence from noise and outliers. Because a even number of categories is classified in this problem the concerns of draws is present when using a even  $k$ . This makes a odd  $k$  more attractive in this problem.

## 3 Methods

As mentioned in the introduction different methods in Machine Learning will be used to try to build a model that manage to correctly classify if pictures of handwritten numbers are odd or even. This

would be simple if the method should only classify previously seen data, the challenge arise when asked to make a model that is highly accurately in classifying new data. The goal is to make a model that trains without overfitting on the in-sample data in order to well estimate the out-of-sample data. In the methods below this is done by using different techniques like cross-validation, validation-set and regularization. In all of the methods the data is classified into the numbers 0 – 9. Another approach would be to classify the number's directly into either a odd or even category. The reason for using all ten numbers is mainly because this makes the extension of the use purpose a lot smoother, f.ex. to also classify if the number is a prime number or not.

### 3.1 Tree based methods

#### 3.1.1 Classification tree

The regression tree is implemented with the R-package "tree" which has a function with the same name. In this method the feature being classified is set and which variables it should consider in the splits. When classifying digits many of the pixels will have a low variance nearing the borders, these will most likely not be used in any of the splits. To reduce unnecessary computation, columns with variance close to zero are removed. Other parameters that can be altered is to control when the tree should stop splitting. Here the minimum development is set as stopping metric. This is measured by calculating the reduction in Residual Sum of Squared - RSS given by:

$$RSS = \sum_{i=1}^n (y_i - f(x_i))^2 \quad (2)$$

, where  $i$  is a data point in the set of  $n$  data points. After a tree has been fitted it might be that the number of terminal nodes are too high and a lower amount of nodes can classify the data as well meanwhile reducing overfitting.

#### 3.1.2 Random Forest

To create a random forest to classify digits the package "randomForest" is used. The function with the same name as the library is used both for creating the random forest and the bagging. These parameters was used when running the random forest:

$$\begin{array}{cc} \text{n tree} & \text{m try} \\ \hline 500 & \frac{1}{3} \times 784 \approx 261 \end{array}$$

Table 1

, where **n tree** is the amount of trees and **m try** the number of variables used in each tree. The fraction  $\frac{1}{3}$  is comes from the default size of the subset of variables in the "randomForest" package. The total number of variables is 784, equal to the amount of pixels in each image.

#### 3.1.3 Bagging

The implementation of bagging or bootstrap aggregation is identical to random forest as described above except for **m try**. Here the subset of variables is the whole set of variables. This is implemented with the parameters:

$$\begin{array}{cc} \text{n tree} & \text{m try} \\ \hline 500 & 784 \end{array}$$

Table 2

. For both bagging and random forest there is no need to specify a form of validation due to the OOB-estimate for the ut-of-sample error as explain in the theory (2.1.2).

### 3.1.4 Boosting

For the implementation of boosting, the library "gbm" is used. The function "gbm" is very similar in use compared to other tree-functions, but "gbm" has an advantage in being parallelized speeding up big runs. Boosting has the possibility to overfit and therefore the built-in validation metric is set to 10-fold cross-validation. When running boosting these parameter were set to:

ntree	distribution	interaction.depth	shrinkage	cv.folds	n.cores
10000	"multinomial"	2	0.01	10	4

Table 3

. Here the distribution is set to be multinomial in order to make the boosting method handle the 10 categories for the digits. Here it is possible to note if one directly classify digits as odd or even a binomial distribution would possible to use. The interaction depth is set to two in order to keep the trees small with a  $\lambda$  (shrinkage) of 0.01 slowly improving the performance. The last two parameters gives the number of cross-validation folds and cores to use.

## 3.2 Deep Learning

### 3.2.1 Artificial Neural Network

For the neural network a "h2o"-grid is run in order to find a good set up for the neural network. "h2o" is a 'R' package originally written in 'Java'. This is a powerful tool which has a deep learning function that suits very well for setting up a neural network. The function runs in parallel and can be used with the grid-function in "h2o". This makes it easier to build and compare several neural nets to find the best parameters for the network of a specific problem. To avoid overfitting 10-fold cross-validation is activated for all the models. The different parameters below were combined producing 40 neural networks:

Hyper parameter	value(s)
hidden	(100, 100), (150, 150), (540, 320), (100, 100, 100), (540, 320, 100)
input_dropout_ratio	0, 0.2
epochs	20
activation	"Rectifier", "RectifierWithDropout"
l1	0, $1.4e - 5$

Table 4

. The hidden layers are controlled by the "hidden"-parameter with the number of layers being the length of the vector and the size of each given by the number. "l1" regularization and drop-out in the input layers are given tried to deal with overfitting. To keep the neural network within feasible range 20 epochs were used for each model. The two rectifier activation functions for deep learning in "h2o" were in training the models.

### 3.2.2 Convolutional Neural Network

To implement a convolutional neural network the package "mxnet" was used. This is originally written in 'C++' and is available in several programming languages. The network is built by piecing together different layers and activation functions. A validation set were used when tweaking to find the best parameters. 15% of the training data was sampled out and used to estimate the out-of-sample error. Parameters for convolutional neural network with two convolutional layers:

Layer	parameters
convolution_1	kernel= $5 \times 5$ , num_filter = 30
activation_1	"tanh"
pooling_1	pool_type = "max", kernel = $2 \times 2$
convolution_2	kernel= $5 \times 5$ , num_filter = 50
activation_2	"tanh"
pooling_2	pool_type = "max", kernel = $2 \times 2$
flatten	
fully_1	num_hidden = 500
activation	"Rectifier Linear Unit - relu"
fully_2	num_hidden = 40

Table 5: Parameters Covolutional Neural Network

. The full network is constructed of two convolution layers, two pooling layer, three activation function, one flatten layer and two fully connected layers. The network can be trained using either the CPU or a GPU, here the CPU is used due to lack of GPU.

### 3.3 Support vector machines/PCA

Before using support vector machines with the package "e1071" feature selection was performed on the data to reduce the number of features and thereby the complexity of the classification problem. To extract the features, principal component analysis was applied. PCA takes the original features and combines them making new features that tries to describe as much of the variance in the data as possible. After running PCA on the features the  $p$  most important PCA features is selected. PCA returns the most describing features in decreasing importance. Running the SVM with different amounts of PCA features yield different cross-validation error which is used to find the best number of principal components to use with SVM. The parameters used for support vector machines:

Parameter	value(s)
amount PCA	$p \in [1, 2, \dots, 30]$
cost	$cost \in [0.5, 1, 1.5, \dots, 10]$

Table 6

### 3.4 K-Nearest Neighbours

The library used for the knn implementation is "Class". The only parameters one can really change in a k-nearest-neighbour model is the  $k$  number of neighbour and the distance metric used to obtain them. The standard euclidean distance is used, making it only a challenge to find the best  $k$ . This is done by performing cross-validation on the dataset for each  $k$ . This is implemented using a parallelized cross-validation function to speed it up. To experiment a bit with the effect of normalizing the data both a normalized and untouched dataset is run through KNN.

## 4 Results

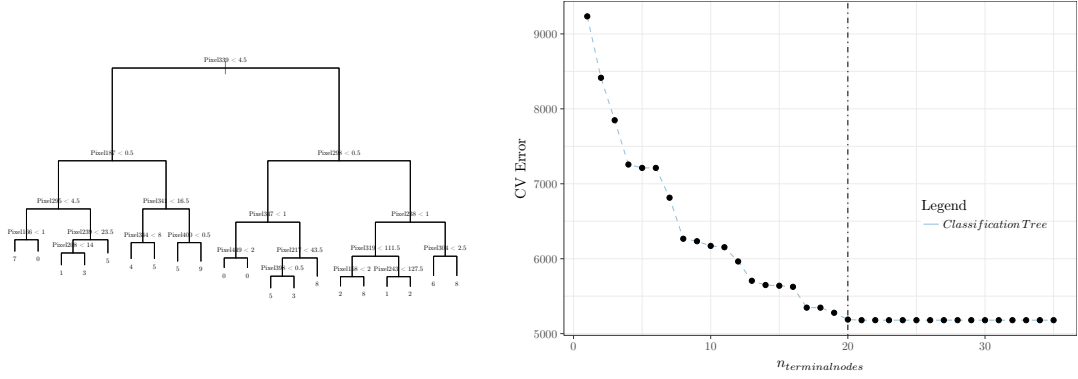
The result for each model are presented with a confusion matrix showing how well each model classified the different digits. With the figures are some plot justifying or illustrating the choices made when making the models. To decide which of the models, 20% of the training data is separated out and used to estimate the out-of-sample error when comparing the models. I will refer to this as the "test set" and the given test set of unlabelled data as the "unclassified set". The model with the lowest error is in used to train on the whole training data and used to predict on the unclassified data. This prediction is stored as .csv-file with the predicted digit for each image and if it is a even

or odd number. It is tempting to use the separated test set when training the individual model, but this would be snooping in the data, leading to contamination of the results.

## 4.1 Tree based methods

### 4.1.1 Classification tree

The is very fast to make given that it is only one tree. As seen in the plot over the CV Error below there is little to gain in reduction of the error after 20 terminal nodes(NB! not the y-scale starts at 6000). The trees is pruned down to 20 terminal nodes before predicting on the created test set.



(a)  $E_{CV}$  for different amounts of leaf nodes.

(b) See that 20 terminal nodes is enough.

Figure 2: Classification tree

After the tree is pruned, the test set is run through the model and each data point is giving a predicted label. This is compared to the real value which gives an error of 37.1%. This is clearly the worst of all the models, but still not that bad thinking it is made using only one tree. The classification tree give a visual of how the tree models classify the data, but is to simple to make a good prediction.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	31	0	5	1	1	3	4	1	0	1	0.340	16/47
1	0	44	1	2	0	2	3	0	1	0	0.170	9/53
2	0	3	32	2	1	0	0	0	2	0	0.200	8/40
3	0	1	1	26	1	0	1	1	0	3	0.235	8/34
4	0	0	0	0	26	0	10	0	1	1	0.316	12/38
5	4	2	2	12	9	31	6	10	3	7	0.640	55/86
6	0	1	3	1	2	1	23	0	0	2	0.303	10/33
7	1	0	1	2	0	0	2	34	0	5	0.244	11/45
8	0	2	5	3	6	5	6	0	39	3	0.435	30/69
9	3	0	2	2	7	3	2	1	6	27	0.491	26/53
Total	8	9	20	25	27	14	34	13	13	22	0.371	185/498

Table 7: Confusion Matrix: Classification tree



#### 4.1.2 Random Forest and Bagging

Since the random forest and bagging are very similar in how they are build up the results from these two models are presented in the same plot and confusion matrices together. As mentioned in the theory 2.1.2 we expect random forest to perform better than bagging due to it having less correlated trees. This is clearly the case when plotting the misclassification error function for each method. We see a very similar behaviour in each curve. Bagging starts with a lower error and keeps this up to around 25 – 30 trees when random forest passes it. This further indicates that the randomness when creating the subset of variables for the random forest in the start will produce worse results in the beginning, but quickly will be better when enough trees are created. This is because after some time there will be on average trees splitting on thee most important variables which creates more uncorrelated trees and a better performance than bagging. Another interesting feature of the models, is that after around 150 the improvement stalls and the solutions stabilizes. This is the limit of the models and increasing the number of trees only contributes to a longer wait when producing the models. This also show how the OOB-data for each tree keeps the models from overfitting.

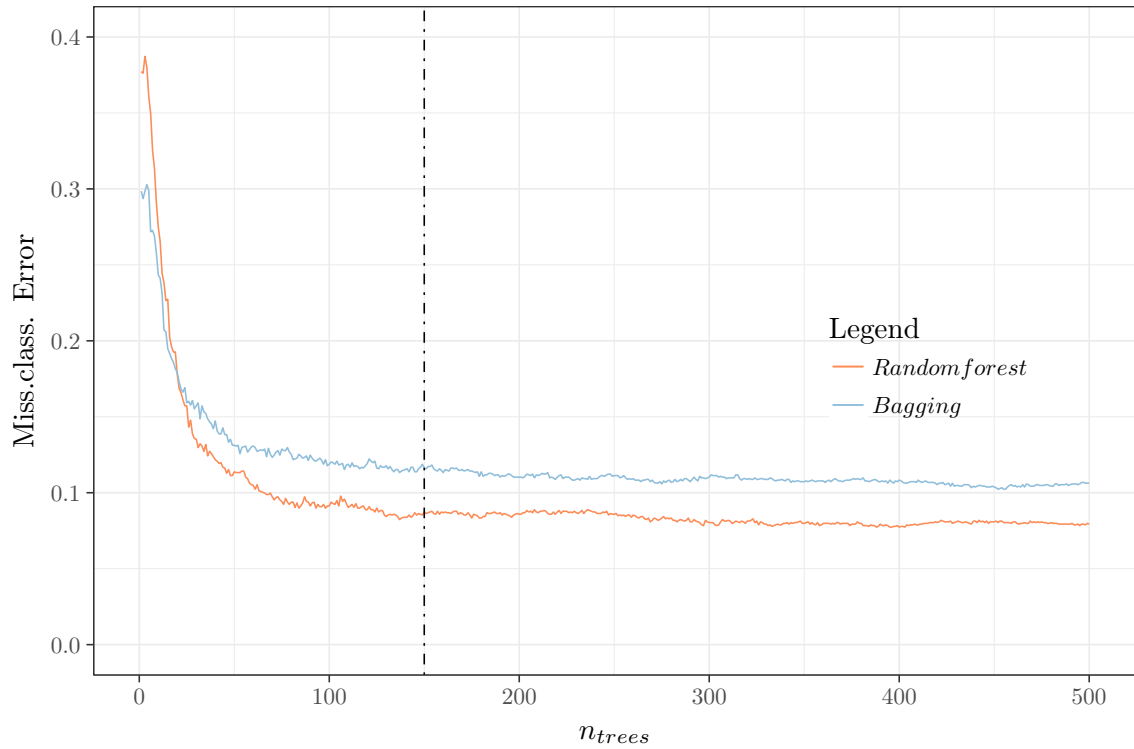


Figure 3: Bagging and Random Forest misclassification over 500 trees.

Below is the confusion matrices for bagging and random forest. As seen in the "Error" tab random forest yield an error of 8.1% while bagging give an error of 10.3%. This makes both of the models a clear improvement of the single classification tree with random forest being the superior over bagging.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	46	0	0	1	0	0	0	0	0	0	0.021	1/47
1	0	42	0	0	0	1	0	1	0	0	0.045	2/44
2	0	2	48	0	0	0	0	1	0	0	0.059	3/51
3	0	0	0	41	0	1	0	0	3	0	0.089	4/45
4	0	1	0	0	52	1	1	4	0	1	0.133	8/60
5	0	0	0	4	0	42	1	0	1	0	0.125	6/48
6	0	0	0	0	1	0	47	0	2	0	0.060	3/50
7	0	1	0	0	0	0	0	48	0	2	0.059	3/51
8	1	2	0	1	0	0	0	0	42	1	0.106	5/47
9	0	0	0	1	2	1	0	0	1	48	0.094	5/53
Total	1	6	0	7	3	4	2	6	7	4	0.081	40/496

Table 8: Confusion matrix: Random Forest

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	47	0	0	2	0	3	0	0	0	0	0.096	5/52
1	0	42	0	0	0	0	0	0	0	0	0.000	0/42
2	0	2	46	0	0	0	0	1	0	0	0.061	3/49
3	0	1	0	41	0	2	0	1	2	0	0.128	6/47
4	0	1	0	0	49	1	1	2	0	2	0.125	7/56
5	0	0	0	3	0	38	1	0	1	0	0.116	5/43
6	0	0	1	0	1	1	47	0	2	1	0.113	6/53
7	0	0	0	0	0	0	0	47	0	3	0.060	3/50
8	0	2	1	1	1	0	0	2	43	1	0.157	8/51
9	0	0	0	1	4	1	0	1	1	45	0.151	8/53
Total	0	6	2	7	6	8	2	7	6	7	0.103	51/496

Table 9: Confusion matrix: Bagging

### 4.1.3 Boosting

From the plot below showing the cv-error over the number of trees made. We see that it decrease quickly in the start before flattening and after around 3000 trees start to increase. When the error starts increasing the boosting model begins to overfit. The best number of trees meaning the amount that gives the lowest cv-error is used to predict on the test set. This number is as pointed out in the plot  $\approx 2900$  trees.

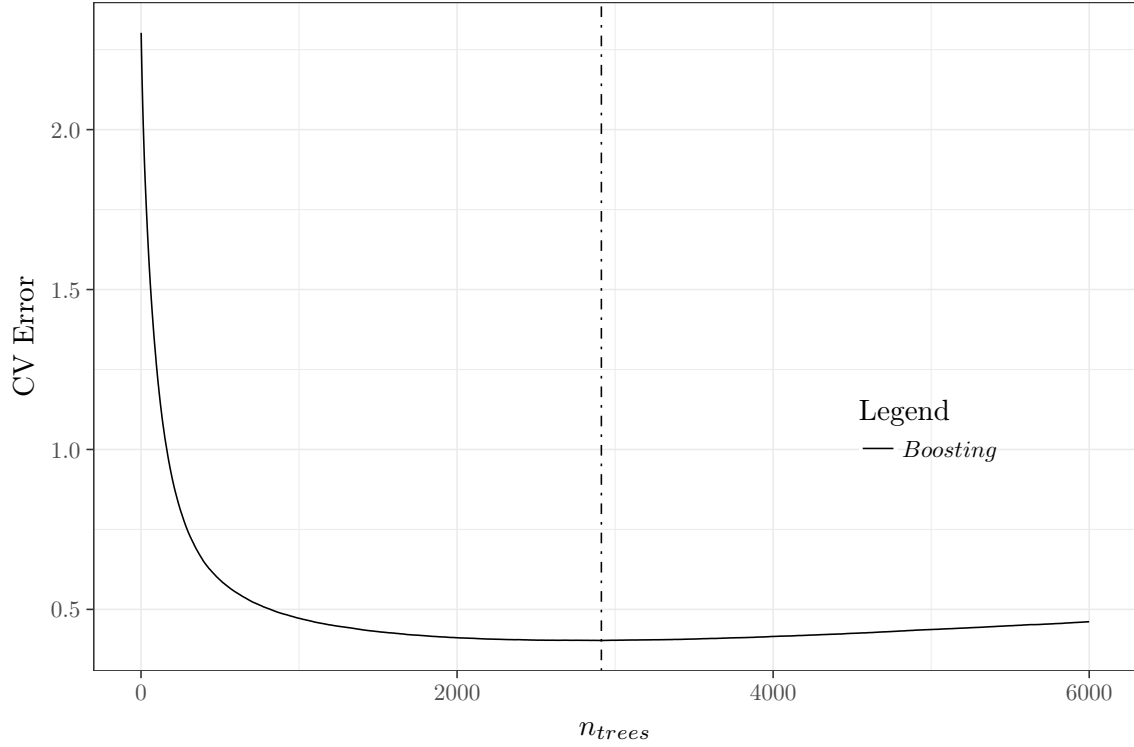


Figure 4: Boosting

Using the best number of trees to predict on the test set giving an error of 11.9%. This means it performs worse than both bagging and random forest, beating single classification tree.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	45	0	0	1	0	1	2	0	2	0	0.118	6/51
1	0	41	0	0	0	1	0	1	0	0	0.047	2/43
2	1	1	43	1	0	1	0	2	1	0	0.140	7/50
3	0	1	2	41	0	0	0	1	1	0	0.109	5/46
4	0	1	0	0	46	1	0	4	0	2	0.148	8/54
5	0	0	0	3	0	40	1	0	1	0	0.111	5/45
6	0	0	1	1	2	1	45	0	0	0	0.100	5/50
7	1	0	0	0	0	0	0	43	0	1	0.044	2/45
8	0	3	2	1	2	0	1	2	44	0	0.200	11/55
9	0	1	0	0	5	1	0	1	0	49	0.140	8/57
Total	2	7	5	7	9	6	4	11	5	3	0.119	59/496

Table 10: Confusion matrix: Boosting

## 4.2 Deep Learning

Below the results for Artificial and Convolutional Neural Network showcasing the difference between them by pure performance. This clearly indicate that they are two distinct methods that share some features.

### 4.2.1 Artificial Neural Network

The artificial neural network(ANN) was set up by running a grid with different networks as given by the parameter overview in 3.1.2. This is a way to find a good set-up for the neural net by basically just trial and error. Below is a plot where the misclassification error of the models is displayed. There are two lines where the slowly increasing one is the 10-fold cv-error and the oscillating one is the error on the test set. Here we come into the temptation of snooping as mentioned earlier. If I would use the results obtained using the test set I would have been snooping and increase the chance of underestimating the complexity of the problem. So the models are sorted by the cv-error and this is purely used to chose the best model. This even though we can see that there are models that gives lower error on the test set. Set-up for the best neural network based used to predict on the test set:

Hidden layers	input_dropout_ratio	epochs	activation	11
(540, 320)	0.2	20	"Rectifier"	$1.4e - 5$

Table 11

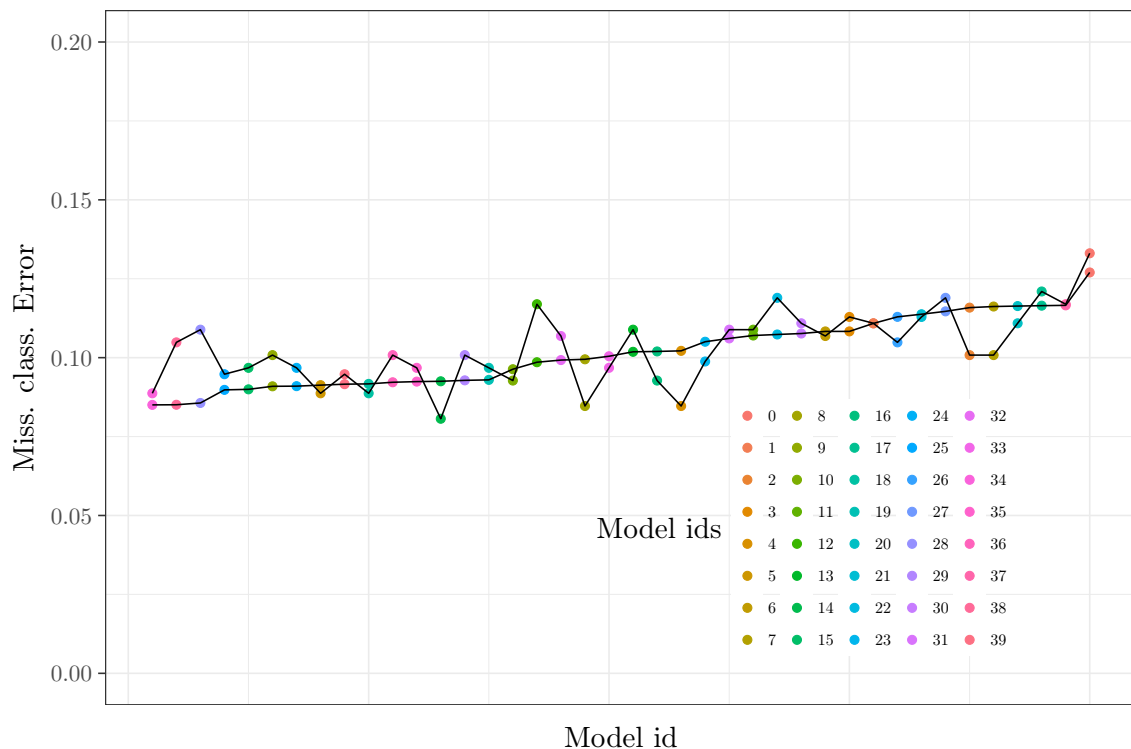


Figure 5: Misclassification error for each model in the grid sorted by the cv-error.

In the confusion matrix below we can see that the final ANN misclassified 9.1% of the digits in the test set. This places it between the bagging and random forest methods performance wise.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	42	0	0	0	0	2	0	0	0	0	0.045	2/44
1	0	47	0	0	1	0	0	0	4	0	0.096	5/52
2	2	1	46	2	0	0	1	0	0	0	0.115	6/52
3	0	0	0	42	1	1	0	1	0	0	0.067	3/45
4	0	0	0	0	50	0	0	2	0	2	0.074	4/54
5	1	0	0	2	1	43	0	1	6	1	0.218	12/55
6	1	0	0	1	1	0	48	0	0	0	0.059	3/51
7	1	0	2	0	1	0	0	49	1	3	0.140	8/57
8	0	0	0	1	0	0	0	0	38	0	0.026	1/39
9	0	0	0	0	0	0	0	1	0	46	0.021	1/47
Total	5	1	2	4	5	3	1	5	11	6	0.091	45/496

Table 12: Confusion matrix: Artificial Neural Network

#### 4.2.2 Convolutional Neural Networks

Using a convolutional neural network on this problem showed to be very successful. With a bit of tweaking in the layers show in 3.2.2, the error when validating(validation set 15% of training data) quickly went below 5%. In the plot below both the in-sample and validation error is plotted for the final set-up for the network as described 5. It shows the in-sample error quickly going to zero(overfitting) and the validation error converging to an accuracy around 95% after 10 rounds of training. The convergence of the error sped up when heavily when using "Xavier" distribution to initialize the "mxnet" network.

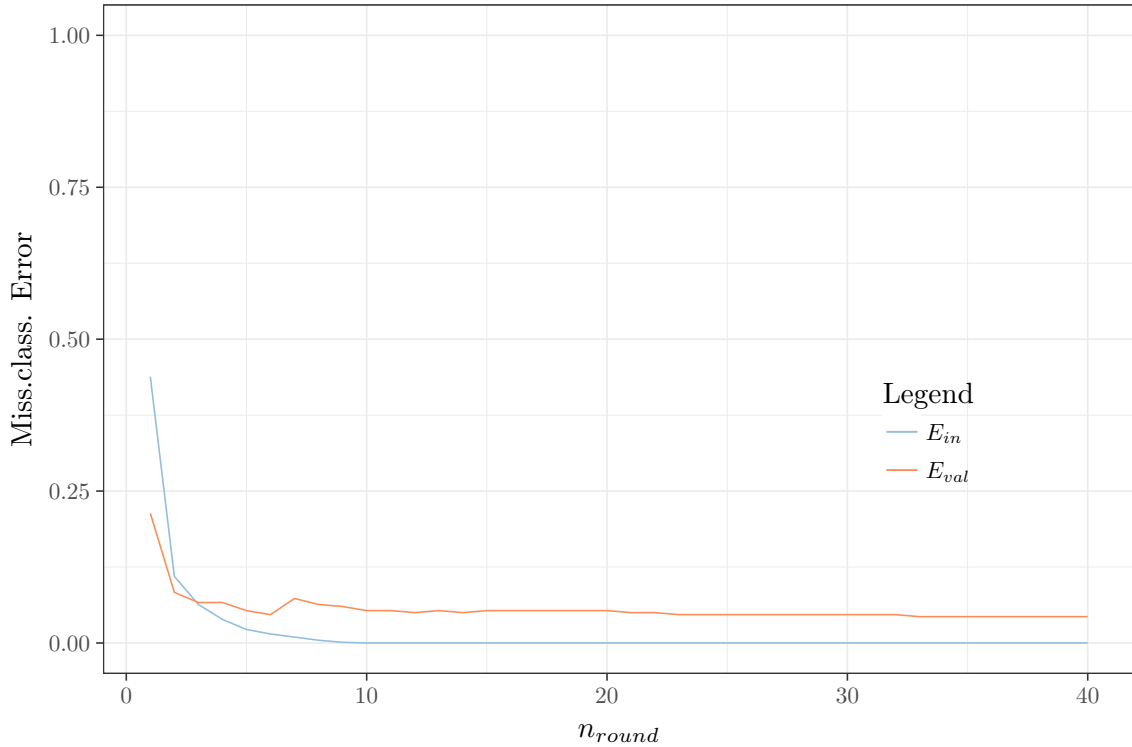


Figure 6:  $E_{in}$  and  $E_{val}$  for training the network over 40 rounds.

Below we see the predictions made on the test set(20% of the training data) after using taking back the validation set into the training set. The predictions have a misclassification error of 3.6% clearly

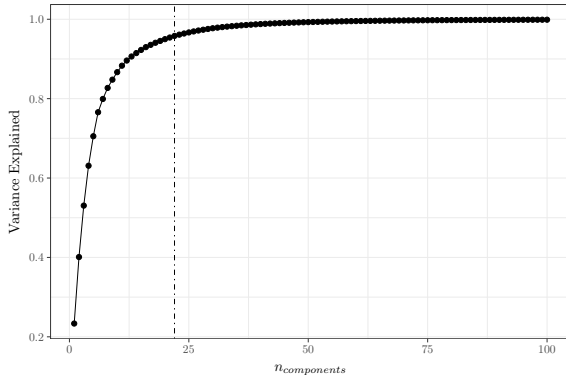
out concurring all the tree models and the ANN. This demonstrates how powerful CNN is when used on image classification.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	47	0	0	0	0	0	1	0	0	0	0.021	1/48
1	0	43	0	0	0	0	0	0	0	0	0.000	0/43
2	0	1	47	0	1	0	0	1	1	0	0.078	4/51
3	0	1	0	48	0	0	0	0	0	0	0.020	1/49
4	0	1	0	0	51	1	0	1	0	0	0.056	3/54
5	0	0	0	0	0	44	1	0	0	0	0.022	1/45
6	0	1	0	0	1	0	47	0	0	0	0.041	2/49
7	0	0	1	0	0	0	0	52	0	1	0.037	2/54
8	0	1	0	0	0	1	0	0	48	0	0.040	2/50
9	0	0	0	0	2	0	0	0	0	51	0.038	2/53
Total	0	5	1	0	4	2	2	2	1	1	0.036	18/496

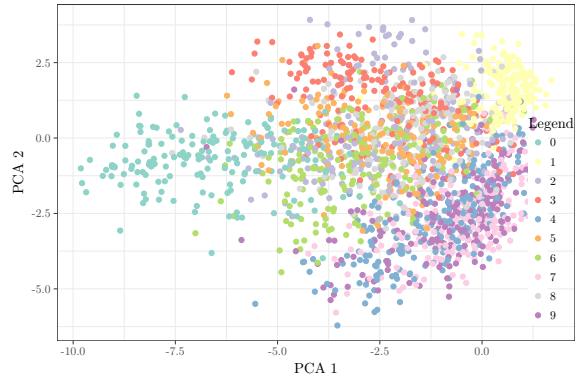
Table 13: Confusion Matrix: Convolutional Neural Network

### 4.3 Support vector machines w/ PCA

Principal Components Analysis(PCA) showed quite powerful in combination with Support Vector Machines(SVM). First the principal components(PC) was calculated and as low as possible while still describing the data well number of components were selected as seen in the first plot below. When plotting the first 100 PC 22 is a good number to satisfy these two objectives. On the right of this plot the training data is plotted defined by the two first principal components where the each digit is represented by a unique colour. In the third plot the error over each number of PC's used when using SVM. Here we see that 22 components give a small cv-error while not necessarily been the lowest. Since we want to keep the number of components as low as possible, picking more will lead to increased complexity without a justifiable increase in accuracy. A grid of cost parameters always returning the best were used to find optimal number of components using SVM. After having the PC's, the cost parameter is calculated the same way. This is shown in the last plot giving the best cost parameter to be 5.5. NB! Note that the last two plots have different scales in order to show the alteration in cv-error.



(a) Variance described by 100 first PC's



(b) Data by two first components from PCA

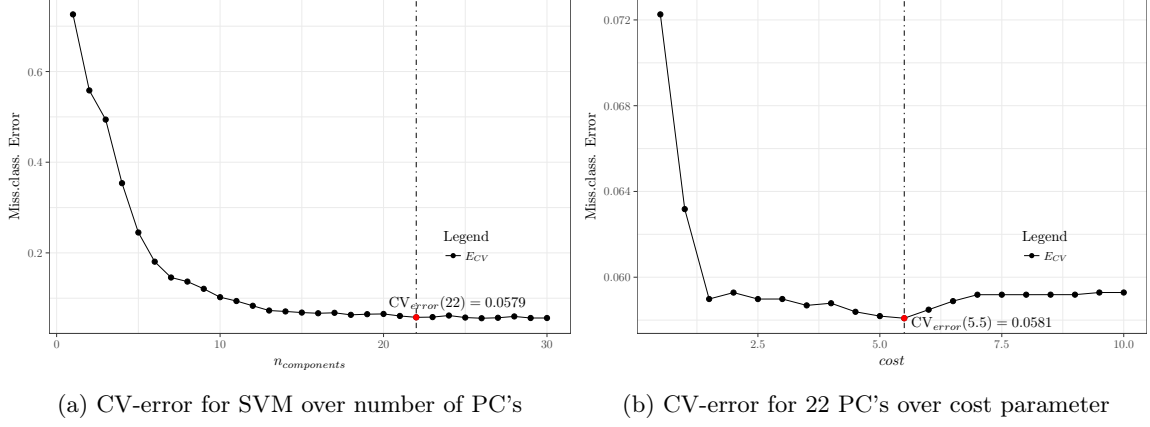


Figure 8: Finding the best set up for PCA/SVM

With the set-up described above the PCA/SVM-model give a misclassification error of 5.6% meaning it predicts better than the other models with exception to the CNN still being the best model.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	47	0	0	0	0	0	0	0	0	0	0.000	0/47
1	0	48	0	0	0	0	1	0	0	0	0.020	1/49
2	0	0	46	0	0	0	0	0	1	1	0.042	2/48
3	0	0	1	41	0	0	0	1	2	0	0.089	4/45
4	0	0	0	0	53	0	0	2	0	1	0.054	3/56
5	0	0	0	4	0	46	0	0	0	0	0.080	4/50
6	0	0	0	0	1	0	48	0	1	0	0.040	2/50
7	0	0	0	0	0	0	0	49	0	4	0.075	4/53
8	0	0	1	1	0	0	0	1	45	1	0.082	4/49
9	0	0	0	2	1	0	0	1	0	45	0.082	4/49
Total	0	0	2	7	2	0	1	5	4	7	0.056	28/496

Table 14: Confusion Matrix: Support Vector Machine w/ PCA

#### 4.4 K-Nearest Neighbours

The K-Nearest Neighbours algorithm in this task is used with the standard euclidean distance when calculating finding the nearest neighbours. This means the task is to find the optimal number of neighbours  $k$  to classify the data. In order to find a good measure for the out-of-sample using CV. This is sped up by parallelization of the CV. As seen below normalizing the data only slightly altered the cv-error and not in a clearly positive or negative way which makes sense since it here meant dividing all the pixels by maximum value for the greyscale colour. As seen in the plot below  $k = 1$  give the lowest cv-error meaning each new data point is classified by the single most similar neighbour. It is also worth noting the peak for  $k = 2$  which is the consequence of using a even number of neighbours for a problem with a even number of categories. This comes into greatest affect here when  $k = 2$ , but also decreasingly visible for  $k = 4, 6, \dots$

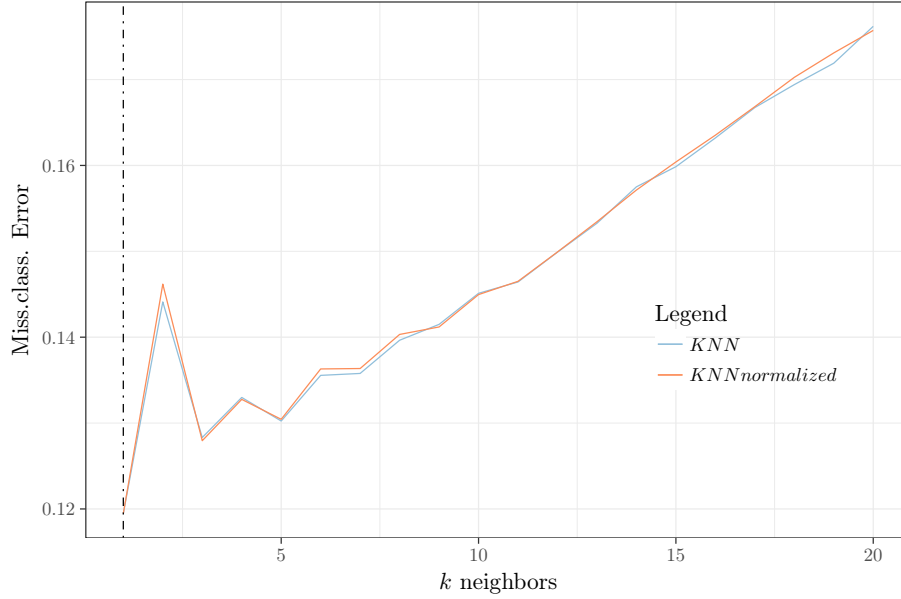


Figure 9

The test predicted using  $k = 1$  on the training set yield a misclassification error equal to 10.1% making it slightly better than bagging by 0.2%.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	45	0	0	1	0	1	0	0	1	0	0.062	3/48
1	0	46	3	0	0	2	0	2	1	0	0.148	8/54
2	0	0	45	0	0	0	0	2	1	0	0.062	3/48
3	0	0	0	40	0	1	0	0	6	0	0.149	7/47
4	0	1	0	0	52	0	0	3	1	4	0.148	9/61
5	1	0	0	4	0	41	1	0	1	0	0.146	7/48
6	1	0	0	0	1	0	48	0	0	0	0.040	2/50
7	0	1	0	0	0	0	0	45	0	1	0.043	2/47
8	0	0	0	3	0	0	0	1	37	0	0.098	4/41
9	0	0	0	0	2	1	0	1	1	47	0.096	5/52
Total	2	2	3	8	3	5	1	9	12	5	0.101	50/496

Table 15: Confusion Matrix: K-Nearest Neighbours



## 4.5 Summary

Below is a table containing all the methods together with the digit classification error rate obtained on the test set sampled from the training set.

	Method	Digit Classification Error
1	Convolutional NN	0.036
2	Support Vector Machine	0.056
3	Random Forest	0.081
4	Artificial NN	0.091
5	K-Nearest Neighbours	0.101
6	Bagging	0.103
7	Boosting	0.119
8	Classification Tree	0.371

## 4.6 Conclusion

By the result obtained by testing each methods on the test set I conclude that Convolutional Neural Network produce the best method for classifying handwritten digits. Further I say that this means it also performs best on classifying if a digit is even or odd. The CNN as defined in 5 is used to train on the whole data set and make predictions for the unlabelled data. A .csv-file containing predicted digit and if it is odd or even in the columns `Digit` and `isOdd(1 = odd)`.

## 5 Discussion

As viewed in the results 4 the different methods performed fairly well on classifying handwritten digits, with exception to the single classification tree where a higher error is expected. Four out of the eight models trained obtained an error below 10%. All these was obtained using around 2000 data points and with what we know from the theory the precision would most likely increase if given more data. We can clearly see that there are two models that outperform the other, these being convolutional neural network and support vector machines. This confirms the notion that these are very powerful methods within machine learning and why they are popular in use. Especially the convolutional neural network which was the clearly best model used in this report. Though some criteria are required to use this type of network, it is even more powerful where it shines, namely "image recognition". This being a field that is very popular today, makes machine learning a very interesting and exciting branch of applications to practise and study.

## 6 Appendices

### 6.1 R-Code: Help Functions

```
1 require(caret)           # useful library to split up data set
2 library(tikzDevice)      # library to export plots to .tex files
3 library(xtable)          # library to export data frames to tables in .tex
4                             files
5 options(tikzMetricPackages = c("\\usepackage[utf8]{inputenc}", "\\usepackage
6                             [T1]{fontenc}",
7                             "\\usetikzlibrary{calc}", "\\usepackage{
8                             amssymb}"))
9
10 ggplot_to_latex <- function(
11     ggplot,
12     destination_path,
13     width,
14     height
15 ){
16     tikz(file = paste0(destination_path, ".tex"), width = 6, height = 4)
17     print(ggplot)
18     dev.off()
19 }
20
21 create_confusion_matrix <- function(
22     predicted_value,
23     true_value,
24     destination_path
25 ){
26     conf <- confusionMatrix(predicted_value, true_value)
27     conf_df <- as.data.frame.matrix(conf$table) # extract confusion matrix
28     # add row for total error
29     conf_df <- rbind(conf_df, Total = rep(0, ncol(conf_df)))
30
31     rows_in_df <- nrow(conf_df)
32
33     classification_frac <- rep("", rows_in_df)
34     classification_float <- rep(0, rows_in_df)
35
36     total_wrong <- 0
37     total_classified <- 0
38
39     # make columns that shows accuracy
40     for(i in 1:(rows_in_df - 1)){
41         correct_classified <- conf_df[i, i]
42         amount_classified <- sum(conf_df[i, ])
43         missclassified <- amount_classified - correct_classified
44
45         classification_frac[i] <- paste0(missclassified, "/", amount_
46         classified)
47         classification_float[i] <- missclassified / amount_classified
48
49         total_wrong <- total_wrong + missclassified
50         total_classified <- total_classified + amount_classified
51     }
52
53     classification_frac[rows_in_df] <- paste0(total_wrong, "/", total_
54     classified)
```

```

51 classification_float[rows_in_df] <- total_wrong / total_classified
52
53 conf_df <- cbind(temp = row.names(conf_df),           # added extra
54                 column
55                 conf_df,                             # to get predicted
56                 classes
57                 Error = classification_float,
58                 Rate = classification_frac)
59 names(conf_df) <- c("", names(conf_df)[-1]) # remove name of predicted
60                                     classes
61
62 write.csv(x = conf_df, file = paste0(destination_path, "_Confusion_
63 Matrix.csv"))
64 print(xtable(conf_df, display = c("s", rep("d", 11), "f", "s"),
65         digits = c(rep(0, 12), 3, 0)),
66       #table.placement = "H",
67       only.contents = TRUE,
68       file = paste0(destination_path, "_Confusion_Matrix.tex"),
69       include.rownames = FALSE)
70 }
71
72 create_cv_indexes <- function(N, n_folds){
73   indexes_per_fold <- floor(N/n_folds)
74   index_matrix <- matrix(0L, nrow = n_folds, ncol = indexes_per_fold)
75   index_available <- 1:N
76   for(i in 1:n_folds){
77     selected_indexes <- sample(index_available, indexes_per_fold)
78     index_available <- index_available[! index_available %in% selected_
79       indexes]
80
81     index_matrix[i, ] <- selected_indexes
82   }
83   return(index_matrix)
84 }

```

../R\_scripts/Help\_Scripts/to\_latex\_functions.R

## 6.2 R-Code: Regression Tree

```

1 ## Libraries and seed
2 rm(list = ls())
3 library(caret)
4 library(readr)
5 library(tree)           # package used for regression tree
6 library(tikzDevice)     # library to export plots to .tex files
7
8 options(tikzMetricPackages = c("\\usepackage[utf8]{inputenc}", "\\usepackage
9   [T1]{fontenc}",
10                                "\\usetikzlibrary{calc}", "\\usepackage{
11                                amssymb}"))
12
13 set.seed(420)
14
15 if(!exists("create_confusion_matrix", mode = "function")){
16   source("Help_Scripts/to_latex_functions.R")
17 }
18
19 #-----#

```

```

19 ## Data
20 path_data <- paste0(getwd(), "/data")
21 path_to_here <- paste0(getwd(), "/Tree-Based-Methods") # getwd give path
    to project
22 # which is one folder over
23
24 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
    header = TRUE)
25 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
    , header = TRUE)
26
27 # Remove unnessesary variables which have a low variance
28 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
29 test_data <- train_data[-split_train_test, ]
30 train_data <- train_data[split_train_test, ]
31
32 # Remove variable with low variance which are near zero. Doing it after
33 # splitting in train/test set to avoid contaminating the data.
34 near_zero_variables <- nearZeroVar(train_data[, -1], saveMetrics = T, freqCut
    = 10000/1, uniqueCut = 1/7)
35 cut_variables <- rownames(near_zero_variables[near_zero_variables$nzv ==
    TRUE, ])
36 variables <- setdiff(names(train_data), cut_variables)
37
38 train_data <- train_data[, variables]
39 test_data <- test_data[, variables]
40
41 # make the label Digit factor
42 train_data[, 1] <- as.factor(train_data[, 1])
43 test_data[, 1] <- as.factor(test_data[, 1])
44
45 unclassified_data[, 1] <- as.factor(unclassified_data[, 1])
46
47 #-----#
48
49 ## REGRESSION - tree
50
51 regression <- function(
52   minimum_development,
53   train_data,
54   test_data
55 ){
56   # Chanage name of pixel columns to work with tikz library
57
58   colnames(train_data)[ 2:length(train_data[1,])] <- c(paste0("Pixel", 1:(
    length(train_data[1,]) - 1)))
59   colnames(test_data)[ 2:length(train_data[1,])] <- c(paste0("Pixel", 1:(
    length(train_data[1,]) - 1)))
60
61   # Set minimum development required for making a new split
62   minimum_development <- 0.005
63   tree_model <- tree(Digit ~ ., data = train_data, mindev = minimum_
    development)
64   plot(tree_model)
65   text(tree_model, cex = .5)
66   print(summary(tree_model))
67
68   cross_validation <- cv.tree(tree_model, K = 10)

```

```

69 cross_validation$k[1] <- 0
70 alpha <- round <- round(cross_validation$k)
71
72 plot(cross_validation$size, cross_validation$dev, type = "b",
73       xlab = "Number of terminal nodes", ylab = "CV error")
74
75 ggplot_df <- data.frame(size = cross_validation$size, dev = cross_
76                           validation$dev)
77
78 destination_path <- paste0(path_to_here, "/Results_TBM/Regression_Tree")
79
80 ggplot1 <- ggplot(data = ggplot_df, aes(x = size, y = dev)) +
81   geom_line(aes(colour = "$RegressionTree$"), linetype = "
82     dashed") +
83   geom_point() +
84   geom_vline(xintercept = 20, color = "black", linetype = "
85     dotdash") +
86   xlab("n\\_{terminal nodes}") +
87   ylab("CV Error") +
88   scale_colour_manual("Legend",
89                       breaks = c("$RegressionTree$"),
90                       values = c("#91bfd9"),
91                       guide = guide_legend(override.aes = list(
92                         linetype = c("solid"),
93                         shape = c(16)
94                       ))) +
95   theme_bw() +
96   theme(legend.position = c(0.8, 0.355),
97         legend.background = element_rect(fill=alpha('white', 0)))
98 ggsave(paste0(destination_path, ".png"))
99
100 ggplot_to_latex(ggplot1, destination_path, width = 5, height = 5)
101 tree_prune <- prune.tree(tree_model, best = 20)
102 summary(tree_prune)
103
104 tikz(file = paste0(destination_path, "_Tree.tex"), width = 6, height =
105       4)
106 plot(tree_prune)
107 text(tree_prune, cex = .5)
108 dev.off()
109
110 predicted <- predict(tree_prune, test_data, type = "class")
111 create_confusion_matrix(predicted, test_data[,1], destination_path)
112 }
113 regression(0.05, train_data, test_data)
114 #-----#
115
116 ../R_scripts/Tree_Based_Methods/Regression_Tree.R

```

## 6.3 R-Code: Random Forest

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(randomForest) # library giving a easy-to-use random forest method
4 library(caret)        # useful library to split up data set
5 library(tikzDevice)   # library to export plots to .tex files
6 library(xtable)       # library to export data frames to tables in .tex
7                         # files
8 set.seed(420)          # seed to replicate results and get consistent test
9                         # and training set
10
11 # Load help script with functions to export the results to latex
12 # These functions gathered to avoid duplicate code
13 if(!exists("create_confusion_matrix", mode = "function")){
14   source("Help_Scripts/to_latex_functions.R")
15 }
16
17 #-----#
18 ## Data
19 path_data <- paste0(getwd(), "/data")
20 path_to_here <- paste0(getwd(), "/Tree_Based_Methods") # getwd give path
21                                                         # which is one
22                                                         # folder over
23
24 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
25   header = TRUE)
26 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
27   , header = TRUE)
28
29 train_data[,1] <- as.factor(train_data[, 1])
30
31 # split training set into training and test set
32
33 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
34   FALSE)
35 test_data <- train_data[-split_train_test, ]
36 train_data <- train_data[split_train_test, ]
37
38 #-----#
39 ## Random forest
40 # Train forest
41 train_random_forest <- function(
42   data,
43   n_trees,
44   minimum_development = 0.01
45 ){
46   random_forest <- randomForest(Digit ~ .,
47     data = data,
48     ntree = n_trees,
49     #mindev = minimum_development,
50     importance = TRUE,
51     na.action = na.exclude)
52   return(random_forest)
53 }
```

```

51 # Plot error as the number of trees increase
52
53 plot_error_development <- function(
54   random_forest_data,
55   destination_path
56 ){
57   error_data <- data.frame(n_trees = 1:nrow(random_forest_data$err.
58     rate),
59     error <- random_forest_data$err.rate[, "OOB"
60     ])
61
62   write.csv(error_data, file = paste0(destination_path, ".csv"))
63
64   ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
65     geom_line(aes(y = error, colour = "$Random forest")) +
66     xlab("$n\\_{trees}$") +
67     ylab("Miss. class. Error") +
68     scale_colour_manual("Legend",
69       breaks = c("$Random forest$"),
70       values = c("black"),
71       guide = guide_legend(override.aes = list(
72         linetype = c("solid"),
73         shape = c( 16)
74       ))) +
75     theme(legend.position = c(0.9, 0.2))
76   ggsave(paste0(destination_path, ".png"))
77
78   ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
79 }
80
81 main <- function(){
82   n_trees = 50
83   random_forest <- train_random_forest(train_data, n_trees)
84   plot_error_development(random_forest, paste0(path_to_here, "/Results_TBM
85     /Random_Forest-",
86     n_trees, "trees_Error_plot"
87     ))
88
89   prediction <- predict(random_forest, newdata = test_data)
90   create_confusion_matrix(predicted_value = prediction, true_value = test_
91     data$Digit,
92     paste0(path_to_here, "/Results_TBM/Random_Forest
93     -",
94     n_trees, "
95     trees"))
96 }
97
98 main()

```

../R\_scripts/Tree\_Based\_Methods/Random\_Forest.R

## 6.4 R-Code: Bagging

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(randomForest) # library giving a easy-to-use random forest method
4 library(caret)        # useful library to split up data set
5 library(tikzDevice)    # library to export plots to .tex files
6 library(xtable)        # library to export data frames to tables in .tex
7                         # files
8 set.seed(420)          # seed to replicate results and get consistent test
9                         # and training set
10
11 # Load help script with functions to export the results to latex
12 # These functions gathered to avoid duplicate code
13 if(!exists("create_confusion_matrix", mode = "function")){
14   source("Help_Scripts/to_latex_functions.R")
15 }
16
17 #-----#
18 ## Data
19
20 path_data <- paste0(getwd(), "/data")
21 path_to_here <- paste0(getwd(), "/Tree_Based_Methods") # getwd give path
22 # to project
23 # which is one folder over
24
25 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
26   header = TRUE)
27 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
28   , header = TRUE)
29
30 train_data[,1] <- as.factor(train_data[, 1])
31
32 # split training set into training and test set
33
34 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
35   FALSE)
36 test_data <- train_data[-split_train_test, ]
37 train_data <- train_data[split_train_test, ]
38
39 #-----#
40 ## Random forest
41 # Train forest
42 train_bagging <- function(
43   data,
44   n_trees
45 ){
46   n_features <- ncol(data) - 1
47   bagging <- randomForest(Digit ~ .,
48     data = data,
49     ntree = n_trees,
50     mtry = n_features,
51     importance = TRUE,
52     na.action = na.exclude)
53   return(bagging)
54 }
```



```

52 # Plot error as the number of trees increase
53
54 plot_error_development <- function(
55   random_forest_data,
56   destination_path
57 ){
58   error_data <- data.frame(n_trees = 1:nrow(random_forest_data$err.rate),
59                             error <- random_forest_data$err.rate[, "OOB"])
60
61   write.csv(error_data, file = paste0(destination_path, ".csv"))
62   ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
63     geom_line(aes(y = error, colour = "$Bagging$")) +
64     xlab("n\\_{trees}") +
65     ylab("Miss.class. Error") +
66     scale_colour_manual("Legend",
67                         breaks = c("$Bagging$"),
68                         values = c("black"),
69                         guide = guide_legend(override.aes = list(
70                           linetype = c("solid"),
71                           shape = c( 16)
72                         ))) +
73     theme(legend.position = c(0.9, 0.2))
74   ggsave(paste0(destination_path, ".png"))
75
76   ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
77 }
78
79 main <- function(){
80   n_trees <- 50
81   bagging <- train_bagging(train_data, n_trees)
82   plot_error_development(bagging, paste0(path_to_here, "/Results_TBM/
83     Bagging-",
84                                     n_trees, "trees_Error_plot"))
85
86   prediction <- predict(bagging, newdata = test_data)
87
88   create_confusion_matrix(predicted_value = prediction, true_value = test_
89     data$Digit,
90     paste0(path_to_here, "/Results_TBM/Bagging-",
91           n_trees, "trees"))
92 }
93
94 main()

```

../R\_scripts/Tree\_Based\_Methods/Bagging.R

## 6.5 R-Code: Boosting

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(caret)           # useful library to split up data set
4 library(gbm)             # library with powerful boosting method
5 set.seed(420)            # seed to replicate results and get consistent test
6                           # and training set
7
8 # Load help script with functions to export the results to latex
9 # These functions gathered to avoid duplicate code
10 if(!exists("create_confusion_matrix", mode = "function")){
11   source("Help_Scripts/to_latex_functions.R")
12 }
13 #-----#
14
15 ## Data
16
17 path_data <- paste0(getwd(), "/data")
18 path_to_here <- paste0(getwd(), "/Tree_Based_Methods") # getwd give path
19               # to project
20
21 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
22   header = TRUE)
23 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
24   , header = TRUE)
25
26
27 train_data[,1] <- as.factor(train_data[, 1])
28
29 # split training set into training and test set
30
31 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
32   FALSE)
33 test_data <- train_data[-split_train_test, ]
34 train_data <- train_data[split_train_test, ]
35
36
37 # Remove variable with low variance which are near zero. Doing it after
38 # splitting in train/test set to avoid contaminating the data.
39 near_zero_variables <- nearZeroVar(train_data[, -1], saveMetrics = T, freqCut
40   = 10000/1, uniqueCut = 1/7)
41 cut_variables <- rownames(near_zero_variables[near_zero_variables$nzv ==
42   TRUE,])
43 variables <- setdiff(names(train_data), cut_variables)
44
45
46 train_data <- train_data[, variables]
47 test_data <- test_data[, variables]
48
49 #-----#
50
51 ## Boosting
52 # Train booster
53 boosting <- function(
54   data,
55   n_trees,
56   interaction_depth = 2,
57   shrinkage = 0.001
58 ){
59   boosting <- gbm(Digit ~ .,
```

```

51         data = data,
52         distribution = "multinomial",
53         n.trees = n_trees,
54         interaction.depth = interaction_depth,
55         shrinkage = shrinkage,
56         cv.folds = 10,
57         n.cores = 4)
58     return(boosting)
59 }
60
61
62 # Plot error as the number of trees increase
63
64 plot_error_development <- function(
65     boosting_data,
66     best_n_trees,
67     destination_path
68 ){
69     error_data <- data.frame(n_trees = 1:length(boosting_data$cv.error),
70                             error <- boosting_data$cv.error)
71     write.csv(error_data, file = paste0(destination_path, ".csv"))
72
73     ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
74         geom_line(aes(y = error, colour = "$Boosting$")) +
75         xlab("$n_{trees}$") +
76         ylab("Miss.class. Error") +
77         geom_vline(xintercept = best_n_trees, color = "black", linetype = "
78             dotdash") +
79         scale_colour_manual("Legend",
80                             breaks = c("$Boosting$"),
81                             values = c("black"),
82                             guide = guide_legend(override.aes = list(
83                                 linetype = c("solid"),
84                                 shape = c( 16)
85                             ))) +
86         theme(legend.position = c(0.9, 0.2)) +
87         theme_bw() +
88         theme(legend.position = c(0.8, 0.355),
89               legend.background = element_rect(fill=alpha('white', 0)))
90     ggsave(paste0(destination_path, ".png"))
91     ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
92 }
93
94 predict_data <- function(
95     boosting_train,
96     best_n_trees,
97     test_data
98 ){
99     predicted <- predict(boosting_train, newdata = test_data, n.trees = best
100                          _n_trees, type = "response")
101
102     predicted <- apply(predicted, 1, function(x) which.max(x) - 1)
103     return(predicted)
104 }
105
106 main <- function(){
107     n_trees = 10000
108     boosting_train <- boosting(train_data,n_trees, shrinkage = 0.01)

```

```

108
109 # Get amount of trees with best performance based on 10 fold cv
110 best_n_trees <- gbm.perf(boosting_train, method = "cv", plot.it = FALSE)
111
112 # Plot error
113 plot_error_development(boosting_train, best_n_trees, paste0(path_to_here
114     ,
115                                     "/Results_TBM/Boosting_",
116                                     n_trees,
117                                     "trees_Error_plot"))
118
119 # Predict test set labels
120 predicted <- predict_data(boosting_train, best_n_trees, test_data)
121
122 # Create confusion matrix for the result
123 create_confusion_matrix(predicted, test_data$Digit, paste0(path_to_here,
124     "/Results_TBM/
125     Boosting_"
126     ,
127     n_trees))
128 }
129
130 main()

```

../R\_scripts/Tree\_Based\_Methods/Boosting.R

## 6.6 R-Code: Plot Random Forest w/ Bagging

```
1 rm(list = ls())
2 library(ggplot2)
3 library(tikzDevice)      # library to export plots to .tex files
4
5 path_data <- paste0(getwd(), "/data")
6 path_to_here <- paste0(getwd(), "/Tree_Based_Methods") # getwd give path
   to project
7
   # which is one
   folder over
8
9 plot_random_forest_bagging <- function(
10   n_trees,
11   path,
12   destination_path
13 ){
14   rf_path <- paste0(path,
15                     "Random_Forest_",
16                     n_trees,
17                     "trees_Error_plot_",
18                     n_trees,
19                     "trees.csv")
20   bagging_path <- paste0(path,
21                          "Bagging_",
22                          n_trees,
23                          "trees_Error_plot_",
24                          n_trees,
25                          "trees.csv")
26
27   random_forest_error <- read.csv(rf_path)
28   bagging_error <- read.csv(bagging_path)
29
30   ggplot_df <- data.frame(n_trees = 1:nrow(bagging_error),
31                          rf = random_forest_error[3],
32                          bag = bagging_error[3])
33   names(ggplot_df) <- c("n_trees", "rf", "bag")
34   print(str(ggplot_df))
35
36   tikz(file = paste0(destination_path, ".tex"), width = 6, height = 4)
37   ggplot1 <- ggplot(data = ggplot_df, aes(x = n_trees)) +
38     geom_line(aes(y = rf, colour = "$Random forest$")) +
39     geom_line(aes(y = bag, colour = "$Bagging$")) +
40     geom_vline(xintercept = 150, color = "black", linetype = "
       dotdash") +
41     xlab("$n_{trees}$") +
42     ylab("Miss.class. Error") +
43     scale_colour_manual("Legend",
44                         breaks = c("$Random forest$", "$Bagging$"),
45                         values = c("#91b9db", "#fc8d59"),
46                         guide = guide_legend(override.aes = list(
47                           linetype = c("solid", "solid"),
48                           shape = c(16, 16)
49                         ))) +
50     scale_y_continuous(limits = c(0, 0.4)) +
51     theme_bw() +
52     theme(legend.position = c(0.8, 0.455),
53           legend.background = element_rect(fill=alpha('white', 0)))
54   ggsave(paste0(destination_path, ".png"))
```

```

55     print(ggplot1)
56     dev.off()
57 }
58
59 main <- function(
60
61 ){
62     n_trees = 500
63     path = paste0(path_to_here, "/Results_TBM/")
64     destination_path = paste0(path, "/Random_Forest_Bagging-",
65                               n_trees, "trees")
66
67     plot_random_forest_bagging(n_trees, path, destination_path)
68
69 }
70
71 main()

```

../R\_scripts/Tree\_Based\_Methods/Plot\_Random\_Forest\_Bagging.R

## 6.7 R-Code: Neural Network

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(h2o)
4 library(caret)
5 library(reshape2)
6
7 set.seed(420)
8
9 # Load help script with functions to export the results to latex
10 # These functions gathered to avoid duplicate code
11 if(!exists("create_confusion_matrix", mode = "function")){
12   source("Help_Scripts/to_latex_functions.R")
13 }
14 #-----#
15
16 ## Data
17
18 path_data <- getwd()
19 path_to_here <- paste0(getwd(), "/Neural_Networks")
20
21 train_data <- read.csv(paste0(path_data, "/data/Train_Digits_20171108.csv"))
22 unclassified_data <- read.csv(paste0(path_data, "/data/Test_Digits_20171108.
   csv"))
23
24 local.h2o <- h2o.init(ip = "localhost", port = 54321, startH2O = TRUE, max_
   mem_size = "7G", nthreads = -1)
25
26 train_data[,1] <- as.factor(train_data[, 1])
27 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
   FALSE)
28 test_data <- train_data[-split_train_test, ]
29 train_data <- train_data[split_train_test, ]
30
31 train_data <- as.h2o(train_data)
32 unclassified_data <- as.h2o(unclassified_data)
33 test_data <- as.h2o(test_data)
34
35 #-----#
36
37 ## Getting useful data from grid run of neural networkss
38
39 get_data_in_df <- function(
40   data
41 )
42 {
43   n <- length(data@model_ids)
44   mse_errors <- rep(0,n)
45   mean_per_class_errors <- rep(0,n)
46   hidden <- rep("", n)
47   str(hidden)
48   rate <- rep(0,n)
49   l1 <- rep(0,n)
50   epochs <- rep(0,n)
51   model_numbers <- rep(0,n)
52   train_error <- rep(0,n)
53   train_mse <- rep(0,n)
54   test_error <- rep(0,n)
```

```

55 test_mse <- rep(0,n)
56 activation <- rep("",n)
57 input_dropout_ratio <- rep(0,n)
58 nesterov_accelerated_gradient <- rep("", n)
59
60 model_df <- data.frame(model_numbers = mse_errors, hidden, rate, l1,
61 epochs,
62 train_error, test_error, train_mse, test_mse,
63 activation, input_dropout_ratio,
64 stringsAsFactors = FALSE)
65
66 str(model_df)
67
68 for(i in 1:n){
69   model <- h2o.getModel(data@model_ids[[i]])
70   model_df$mse_errors[i] <- h2o.mse(model)
71   #model_df$mean_per_class_error[i] <- model@model$cross_validation_
72   metrics@metrics$mean_per_class_error
73   model_df$mean_per_class_error[i] <- h2o.performance(model, xval = T)
74   @metrics$mean_per_class_error
75
76   model_paramaters <- model@allparameters
77   model_name <- model@model_id
78   model_number <- sub(".*model_(.*)$", "\\1", model_name)
79   model_df$model_numbers[i] <- as.integer(model_number)
80   model_df$hidden[i] <- paste(as.character(model_paramaters$hidden),
81 sep = " ", collapse = " ")
82   model_df$rate[i] <- model_paramaters$rate
83   model_df$l1[i] <- model_paramaters$l1
84   model_df$epochs[i] <- model_paramaters$epochs
85   model_df$activation[i] <- model_paramaters$activation
86   model_df$input_dropout_ratio[i] <- model_paramaters$input_dropout_
87   ratio
88   model_df$nesterov_accelerated_gradient[i] <- model_paramaters$
89   nesterov_accelerated_gradient
90
91   train_performance <- h2o.performance(model, train_data)@metrics
92   train_performance_error <- train_performance$mean_per_class_error
93   train_performance_mse <- train_performance$MSE
94
95   model_df$train_error[i] <- train_performance_error
96   model_df$train_mse[i] <- train_performance_mse
97
98   test_performance <- h2o.performance(model, test_data)@metrics
99   test_predictions <- h2o.predict(model, test_data)
100   test_accuracy <- test_predictions$predict == test_data$Digit
101   test_performance_error <- 1 - mean(test_accuracy)
102   test_performance_mse <- test_performance$MSE
103
104   model_df$test_error[i] <- test_performance_error
105   model_df$test_mse[i] <- test_performance_mse
106
107 }
108
109 model_df <- model_df[with(model_df, order(model_numbers)),]
110 model_df
111 }
112
113 # Run and plot grid of n neural nets
114 run_neural_network_grid <- function(){
115   activation <- list("Rectifier", "RectifierWithDropOut")# "Tanh")

```



```

106 hidden <- list(c(100,100), c(150, 150), c(540, 320), c(100, 100, 100),
107               c(540, 320, 100))
108 input_dropout_ratio <- list(0, 0.2)
109 nesterov_accelerated_gradient <- list( TRUE)
110 epochs <- list(20)
111 l1 = list(0,1.4e-5)
112 hyper_params <- list(activation = activation, hidden = hidden, input_
113                       dropout_ratio = input_dropout_ratio, nesterov_accelerated_gradient =
114                         nesterov_accelerated_gradient, epochs = epochs, l1 = l1)
115
116 grid_deep_learning <- h2o.grid(algorithm = "deeplearning",
117                               x = 2:785,
118                               y = 1,
119                               training_frame = train_data,
120                               nfolds = 10,
121                               stopping_metric = "MSE",
122                               stopping_tolerance = 0.0025,
123                               hyper_params = hyper_params)
124
125 save_results <- function(results){
126   write.csv(results, file = paste0(path_to_here, "/Neural_Networks/results
127     _NN/grid_run_20.csv"))
128 }
129
130 df <- get_data_in_df(grid_deep_learning)
131 save_results(df)
132
133 #results_df <- df
134
135 results_df <- read.csv(paste0(path_to_here, "/Neural_Networks/results_NN
136   /grid_run_40.csv"))
137
138 results_df <- results_df[with(results_df, order(mean_per_class_error)),]
139 results_df$row_names <- 1:length(results_df[,1])
140
141 melt_datas <- melt(results_df[c("test_error","mean_per_class_error", "
142   row_names",
143                               "model_numbers")], id = c("row_names", "model_
144     numbers"))
145
146 plot_list <- list()
147 # Plot classification error
148 plot_list[[1]] <- ggplot(data=melt_datas,
149                          aes(x=row_names, y=value)) +
150   aes(x=row_names, y=value)) +
151   geom_point(aes(colour = as.factor(model_numbers), group = as.factor(
152     model_numbers)), size = 1.25) +
153   geom_line(aes(group = variable)) +
154   xlab("Model id") +
155   ylab("Miss. class. Error") +
156   scale_y_continuous(limits = c(0, 0.2)) +
157   theme_bw() +
158   theme(legend.position = c(0.675, 0.255),
159         legend.background = element_rect(fill=alpha('white', 0)),
160         legend.direction = "horizontal",
161         legend.text = element_text(size=6),
162         legend.key = element_rect(size = 3),
163         legend.key.size = unit(1.0, 'lines'),
164         axis.text.x=element_blank(),
165         axis.ticks.x=element_blank()) +
166   scale_colour_discrete(name = "Model ids") +
167   guides(fill = guide_legend(title = "Legend"))

```

```

157 |
158 |     ggplot_to_latex(plot_list[[1]],
159 |         paste0(path_to_here, "/results_NN/per_class_error"), width = 6,
160 |         height = 4)
161 |     ggsave(paste0(path_to_here, "/results_NN/per_class_error3.png"))
162 | }
163 | # Run final model based on results from grid run and make confusion matrix
164 |   for prediction on test set
165 | run_final_model <- function(){
166 |     deep_learning_results3<- h2o.deeplearning(x = 2:785,
167 |         y = 1,
168 |         training_frame = train_data,
169 |         activation = "Rectifier",
170 |         input_dropout_ratio = 0.2,
171 |         nfolds = 10,
172 |         balance_classes = TRUE,
173 |         hidden = c(540, 320),
174 |         l1 = 1.4e-5,
175 |         stopping_metric = "MSE",
176 |         stopping_tolerance = 0.0025,
177 |         nesterov_accelerated_gradient =
178 |             TRUE,
179 |         epochs = 20)
180 |
181 |     h2o.performance(deep_learning_results3, test_data)
182 |
183 |     predicted <- predict(deep_learning_results3, test_data, type = "response
184 |         ")
185 |
186 |     predicted_confusion_matrix <- as.factor(as.vector(predicted$predict))
187 |     test_data_confusion_matrix <- as.data.frame(test_data)
188 |     create_confusion_matrix(predicted_confusion_matrix,
189 |         test_data_confusion_matrix[, "Digit"],
190 |         paste0(path_to_here, "/results_NN/540_320_neural
191 |             _net"))
192 | }
193 | run_final_model
194 | h2o.shutdown()

```

../R\_scripts/Neural\_Networks/neural\_network.R

## 6.8 R-Code: Convolutional Neural Network

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(mxnet)          #library for running convolutional neural network
4 library(caret)
5
6 set.seed(420)
7
8 # Load help script with functions to export the results to latex
9 # These functions gathered to avoid duplicate code
10 if(!exists("create_confusion_matrix", mode = "function")){
11     source("Help_Scripts/to_latex_functions.R")
12 }
13 #-----#
14
15 ## Data
16
17 path_data <- getwd()
18 path_to_here <- paste0(getwd(), "/Neural_Networks")
19
20 train_data <- read.csv(paste0(path_data, "/data/Train_Digits_20171108.csv"))
21 unclassified_data <- read.csv(paste0(path_data, "/data/Test_Digits_20171108.
    csv"))
22
23 train_data[, 1] <- as.factor(train_data[, 1])
24 train_data_full <- train_data
25
26 # Split the data into training, test and validation set
27 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
28
29 test_data <- train_data[-split_train_test, ]
30 train_data <- train_data[split_train_test, ]
31
32 split_train_validation <- createDataPartition(train_data$Digit, p = 0.85,
    list = FALSE)
33
34 validation_data <- train_data[-split_train_validation, ]
35 train_data_val <- train_data[split_train_validation, ]
36
37 # Setting up datasets as matrices in order to get correct input for mxnet
38
39 # Training after tuning
40 train <- train_data
41 train_x <- t(train[, -1])
42 train_y <- train[, 1]
43 train_array <- train_x
44 dim(train_array) <- c(28, 28, 1, ncol(train_x))
45
46 # Training under tuning
47 train_val <- train_data_val
48 train_x_val <- t(train_val[, -1])
49 train_y_val <- train_val[, 1]
50 train_array_val <- train_x_val
51 dim(train_array_val) <- c(28, 28, 1, ncol(train_x_val))
52
53 # Test set to compare with other methods
54 test_x <- t(test_data[, -1])
```

```

55 test_y <- test_data[, 1]
56 test_array <- test_x
57 dim(test_array) <- c(28, 28, 1, ncol(test_x))
58
59 # Validation set used to validate under tuning
60 validation_x <- t(validation_data[, -1])
61 validation_y <- validation_data[, 1]
62 validation_array <- validation_x
63 dim(validation_array) <- c(28, 28, 1, ncol(validation_x))
64
65 # Get right format on the unclassified set
66 unclassified_x <- t(unclassified_data[, -1])
67 unclassified_array <- unclassified_x
68 dim(unclassified_array) <- c(28, 28, 1, ncol(unclassified_x))
69
70 # Full training data when making prediction for unclassified data
71 train_full <- train_data_full
72 train_x_full <- t(train_full[, -1])
73 train_y_full <- train_full[, 1]
74 train_array_full <- train_x_full
75 dim(train_array_full) <- c(28, 28, 1, ncol(train_x_full))
76
77 #-----#
78
79 data <- mx.symbol.Variable("data")
80
81 two_layer_concolutional_network <- function(){
82   # Setting up first convolutional layer
83
84   convolution_1 <- mx.symbol.Convolution(data = data, kernel = c(5,5), num
     _filter = 30)
85   activation_1 <- mx.symbol.Activation(data = convolution_1, act_type = "
     tanh")
86   pooling_1 <- mx.symbol.Pooling(data = activation_1, pool_type = "max",
     kernel = c(2, 2), stride = c(2,2))
87
88   # Setting up second convolutional layer
89
90   convolution_2 <- mx.symbol.Convolution(data = pooling_1, kernel = c(5,5)
     , num_filter = 50)
91   activation_2 <- mx.symbol.Activation(data = convolution_2, act_type = "
     tanh")
92   pooling_2 <- mx.symbol.Pooling(data = activation_2, pool_type = "max",
     kernel = c(2, 2), stride = c(2,2))
93
94   # Setting up first fully connected layer
95
96   flatten <- mx.symbol.Flatten(data = pooling_2)
97   fully_1 <- mx.symbol.FullyConnected(data = flatten, num_hidden = 500)
98   activation_3 <- mx.symbol.Activation(data = fully_1, act_type = "tanh")
99
100  # Setting up second fully connected layer
101
102  fully_2 <- mx.symbol.FullyConnected(data = activation_3, num_hidden =
    10)
103
104  return(fully_2)
105
106 }

```

```

107
108 # three layer convolutional neural network to experiment, turned out to
    perform worse
109 three_layer_convolutional_network <- function(){
110     # Setting up first convolutional layers
111
112     convolution_1 <- mx.symbol.Convolution(data = data, kernel = c(5,5), num
        _filter = 30)
113     activation_1 <- mx.symbol.Activation(data = convolution_1, act_type = "
        tanh")
114     pooling_1 <- mx.symbol.Pooling(data = activation_1, pool_type = "max",
        kernel = c(2, 2), stride = c(2,2))
115
116     # Setting up second convolutional layers
117
118     convolution_2 <- mx.symbol.Convolution(data = pooling_1, kernel = c(5,5)
        , num_filter = 50)
119     activation_2 <- mx.symbol.Activation(data = convolution_2, act_type = "
        tanh")
120     pooling_2 <- mx.symbol.Pooling(data = activation_2, pool_type = "max",
        kernel = c(2, 2), stride = c(2,2))
121
122     # Setting up third convolutional layers
123     convolution_3 <- mx.symbol.Convolution(data = pooling_2, kernel = c(3,3)
        , num_filter = 50)
124     activation_3 <- mx.symbol.Activation(data = convolution_3, act_type = "
        tanh")
125     pooling_3 <- mx.symbol.Pooling(data = activation_3, pool_type = "max",
        kernel = c(2, 2), stride = c(2,2))
126
127     # Setting up first fully connected layer
128
129     flatten <- mx.symbol.Flatten(data = pooling_3)
130     fully_1 <- mx.symbol.FullyConnected(data = flatten, num_hidden = 500)
131     activation_3 <- mx.symbol.Activation(data = fully_1, act_type = "tanh")
132
133     # Setting up second fully connected layer
134
135     fully_3 <- mx.symbol.FullyConnected(data = activation_3, num_hidden =
        40)
136
137     return(fully_3)
138
139 }
140
141 # Output layer, softmax gives probabilities for the output
142
143 run_convolutional_neural_network <- function(
144     neural_net_model,
145     train_array,
146     train_y,
147     test_array,
148     validation_array = NULL,
149     validation_y = NULL
150 ){
151     mx.set.seed(100)
152
153     cpu_used <- mx.cpu()
154

```

```

155 train_y <- as.factor(as.integer(train_y) %% 2)
156 validation_y <- as.factor(as.integer(validation_y) %% 2)
157
158 logger_mx <- mx.metric.logger$new()
159 train_model <- mx.model.FeedForward.create(neural_net_model,
160                                             X = train_array,
161                                             y = train_y,
162                                             eval.data = list(data =
163                                                                     validation_array, label =
164                                                                     validation_y), #must be
165                                                                     set when validating
166                                             ctx = cpu_used,
167                                             num.round = 40,
168                                             array.batch.size = 50,
169                                             learning.rate = 0.01,
170                                             momentum = 0.9,
171                                             initializer = mx.init.Xavier
172                                             ()),
173                                             eval.metric = mx.metric.
174                                             accuracy,
175                                             epoch.end.callback = mx.
176                                             callback.log.train.metric
177                                             (100, logger_mx)
178                                             )
179
180 predicted <- predict(train_model, test_array)
181 #mxnet gives wrong labels by 1, that is 1-10 instead of 0-9
182 #fix by subtracting 2
183 predicted_labels <- max.col(t(predicted)) - 2
184
185 return(list(logger_mx, predicted_labels))
186 }
187
188 plot_error_development <- function(
189     neural_net_model,
190     train_array_val,
191     train_y_val,
192     test_array,
193     test_y,
194     validation_array,
195     validation_y
196 ){
197     # NB! remember to remove "#" from "eval.data" in "run_convolutional_
198     #       neural_network"
199     # run cnn to find error in and val
200     error <- run_convolutional_neural_network(neural_net_model,
201                                               train_array_val,
202                                               train_y_val,
203                                               test_array,
204                                               validation_array = validation_
205                                               array,
206                                               validation_y = validation_y)
207
208     # extract error from results
209     error_t <- error[[1]]
210
211     error_in_sample <- (1 - error_t$train)
212     error_val <- (1 - error_t$eval)
213
214

```

```

205     ggplot_df <- data.frame(round = 1:length(error_in_sample),
206                             error_in_sample = error_in_sample,
207                             error_val = error_val)
208
209     # plot error in and validation
210     ggplot1 <- ggplot(data = ggplot_df, aes(x = round)) +
211       geom_line(aes(y = error_in_sample, colour = "$E_{in}$")) +
212       geom_line(aes(y = error_val, colour = "$E_{val}$")) +
213       xlab("$n_{round}$") +
214       ylab("Miss.class. Error") +
215       scale_y_continuous(limits = c(0, 1.0)) +
216       scale_colour_manual("Legend",
217                           breaks = c("$E_{in}$", "$E_{val}$"),
218                           values = c("#91bfdb", "#fc8d59"),
219                           guide = guide_legend(override.aes = list(
220                             linetype = c("solid", "solid"),
221                             shape = c( 16, 16)
222                           ))) +
223       theme_bw() +
224       theme(legend.position = c(0.8, 0.355),
225             legend.background = element_rect(fill=alpha('white', 0)))
226     ggplot_to_latex(ggplot1, paste0(path_to_here, "/results_NN/98_attempt_
227                                   convolutional_neural_network_40_rounds"),
228                                   width = 6, height = 4)
229 }
230 # Predict on test set splitted from the training data to compare with other
231   methods
232 predict_on_test_set <- function(
233   neural_net_model,
234   train_array,
235   train_y,
236   test_array,
237   test_y
238 ){
239   predicted <- run_convolutional_neural_network(neural_net_model,
240                                                 train_array,
241                                                 train_y,
242                                                 test_array)
243   predicted <- predicted[[2]]
244   create_confusion_matrix(as.factor(predicted), test_y,
245                           paste0(path_to_here, "/results_NN/convolutional_
246                                   neural_network_40_rounds_test"))
247 }
248 # Give predictions on unclassified data
249 predict_on_unclassified_data <- function(
250   neural_net_model,
251   train_array_full,
252   train_y_full,
253   unclassified_array
254 ){
255   predicted <- run_convolutional_neural_network(neural_net_model,
256                                                 train_array_full,
257                                                 train_y_full,
258                                                 unclassified_array)
259   predicted <- predicted[[2]]
260 }

```

```

261 predicted_digit <- predicted
262 predicted_odd_even <- as.factor(as.integer(predicted) %% 2)
263 prediction_csv <- data.frame(number = 1:length(predicted_digit),
264                               digits = predicted_digit,
265                               odd_even = predicted_odd_even)
266 write.csv(prediction_csv, file = paste0(path_data, "/data/predictions_
    STNKAR012.csv"))
267
268 }
269
270 main <- function()
271 {
272   # Validation used under tuning, set to false so it does run after tuning
273   validation_boolean <- TRUE
274   # Test used under testing of the final tuned model to compare with other
    methods
275   train_boolean <- FALSE
276
277   fully_2 <- two_layer_concolutional_network()
278   neural_net_model <- mx.symbol.SoftmaxOutput(data = fully_2)
279
280   if(validation_boolean){
281     plot_error_development(neural_net_model,
282                             train_array_val,
283                             train_y_val,
284                             test_array,
285                             test_y,
286                             validation_array,
287                             validation_y)
288   }
289
290   if(train_boolean){
291     predict_on_test_set(neural_net_model,
292                         train_array,
293                         train_y,
294                         test_array,
295                         test_y)
296   }
297
298   predict_on_unclassified_data(neural_net_model,
299                               train_array_full,
300                               train_y_full,
301                               unclassified_array)
302 }
303 }
304 main()

```

../R\_scripts/Neural\_Networks/convolutional\_neural\_network.R



## 6.9 R-Code: K-nearest Neighbours

```
1 ## Libraries and seed
2 rm(list = ls())
3
4 library(foreach)          # library for running loop in parallel
5 library(doParallel)       # library for running loop in parallel
6 library(caret)            # useful library to split up data set
7 library(tikzDevice)       # library to export plots to .tex files
8 library(xtable)           # library to export data frames to tables in .tex
9                             files
10 library(class)            # library for knn method
11 set.seed(420)             # seed to replicate results and get consistent test
12                             and training set
13
14 # Load help script with functions to export the results to latex
15 # These functions gathered to avoid duplicate code
16 if(!exists("create_confusion_matrix", mode = "function")){
17   source("Help_Scripts/to_latex_functions.R")
18 }
19
20 #-----#
21 ## Data
22 path_data <- paste0(getwd(), "/data")
23 path_to_here <- paste0(getwd(), "/Tree-Based-Methods") # getwd give path
24                             to project
25 # which is one folder over
26
27 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
28   header = TRUE)
29 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
30   , header = TRUE)
31
32 train_data[,1] <- as.factor(train_data[, 1] )
33
34 # split training set into training and test set
35
36 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
37   FALSE)
38 test_data <- train_data[-split_train_test, ]
39 train_data <- train_data[split_train_test, ]
40
41
42 k_nearest_neighbors <- function(
43   train_data,
44   train_data_norm,
45   k_folds,
46   k_neighbors = 20
47 ){
48   # First find number of k(neighbors) using crossvalidation
49   cv_indexes <- create_cv_indexes(nrow(train_data), k_folds)
50
51   cores=detectCores()
52   cl <- makeCluster(cores[1]-1) #not to overload your computer
53   registerDoParallel(cl)
54
55   cv_error <- c()
```

```

52     indexes <- 1:nrow(train_data)
53
54     final_df <- foreach(i = 1:k_neighbors,
55                         .combine = "rbind",
56                         .packages = "class") %dopar%{
57         error_kfolds <- 0
58         error_kfolds_norm <- 0
59         for(k in 1:k_folds){
60             train_val <- train_data[!indexes %in% cv_indexes[k, ], ]
61             validation_val <- train_data[cv_indexes[k, ], ]
62
63             train_val_norm <- train_data_norm[!indexes %in% cv_indexes[k
64                                     , ], ]
65             validation_val_norm <- train_data_norm[cv_indexes[k, ], ]
66
67             # without normalization
68             knn_pred_class <- knn(train_val[-1],
69                                   validation_val[-1],
70                                   train_val[, 1],
71                                   k = i)
72
73             # with normalization
74             knn_pred_class_norm <- knn(train_val_norm[-1],
75                                       validation_val_norm[-1],
76                                       train_val_norm[, 1],
77                                       k = i)
78
79             error <- 1 - mean(validation_val[, 1] == knn_pred_class)
80             error_norm <- 1 - mean(validation_val_norm[, 1] == knn_pred_
81                                   class_norm)
82             error_kfolds <- error_kfolds + error
83             error_kfolds_norm <- error_kfolds_norm + error_norm
84         }
85         temp_df <- data.frame(k_neighbors = i,
86                               error_kfolds = error_kfolds,
87                               error_kfolds_norm = error_kfolds_norm)
88     }
89     final_df[, 2:3] <- final_df[, 2:3] / k_folds
90     stopCluster(cl)
91     return(final_df)
92 }
93
94 average_cv_error <- function(
95     train_data,
96     test_data,
97     train_data_norm,
98     test_data_norm,
99     n_avg
100 ){
101     k_neighbors <- 20
102
103     # set up error data frame
104     error_df <- data.frame(k_neighbors = 1:k_neighbors,
105                           error_kfolds = rep(0, k_neighbors),
106                           error_kfolds_norm = rep(0, k_neighbors))
107
108     # progress bar
109     pb <- txtProgressBar(min = 0, max = n_avg, style = 3)

```

```

109 # average cv-error over several runs to get more precise measure
110 for(i in 1:n_avg){
111     setTxtProgressBar(pb, i)
112     error_df[-1] <- error_df[-1] + k_nearest_neighbors(train_data,
113                                                         train_data_norm,
114                                                         2,
115                                                         k_neighbors)[-1]
116 }
117 close(pb)
118 error_df[-1] <- error_df[-1]/n_avg
119 ggplot1 <- ggplot(data = error_df, aes(x = k_neighbors)) +
120   geom_line(aes(y = error_kfolds, colour = "$KNN$")) +
121   geom_line(aes(y = error_kfolds_norm, colour = "$KNN normalized$")) +
122   geom_vline(xintercept = 1, color = "black", linetype = "dotted") +
123   xlab("$k$ neighbors") +
124   ylab("Miss.class. Error") +
125   scale_colour_manual("Legend",
126                       breaks = c("$KNN$", "$KNN normalized$"),
127                       values = c("#91b9db", "#fc8d59"),
128                       guide = guide_legend(override.aes = list(
129                         linetype = c("solid", "solid"),
130                         shape = c( 16, 16)
131                       ))) +
132   theme_bw() +
133   theme(legend.position = c(0.8, 0.355),
134         legend.background = element_rect(fill=alpha('white', 0)))
135 ggplot_to_latex(ggplot1,
136                 paste0("K_Nearest_Neighbors/Results_KNN/knn_error_
137                       compare",n_avg),
138                 width = 5, height = 5)
139 return(error_df)
140 }
141
142 predict_on_test <- function(
143   best_k,
144   train_data,
145   test_data
146 ){
147   knn_pred_class <- knn(train_data[-1],
148                         test_data[-1],
149                         train_data[, 1],
150                         k = best_k)
151   create_confusion_matrix(predicted_value = knn_pred_class,
152                           true_value = test_data[, 1],
153                           destination_path = "K_Nearest_Neighbors/Results_
154                           KNN/")
155 }
156
157 #cv_final <- k_nearest_neighbors(train_data, 10)
158 train_data_norm <- train_data
159 train_data_norm[-1] <- train_data_norm[-1]/255
160 test_data_norm <- test_data
161 test_data_norm[-1] <- test_data_norm[-1]/255
162 #cv_norm <- k_nearest_neighbors(train_data, train_data_norm, 10)
163
164 average_error <- average_cv_error(train_data = train_data,
165                                   test_data = test_data,
166                                   train_data_norm = train_data_norm,
167                                   test_data_norm = test_data_norm,
168                                   n_avg = 30)

```

```
166 |  
167 |predict_on_test(1, train_data_norm, test_data_norm)  
    |../R_scripts/K_Nearest_Neighbors/kknn.R
```

## 6.10 R-Code: Support Vector Machines

```
1 ## Libraries and seed
2 rm(list = ls())
3 library(e1071)           # library to make margins for svm
4 library(caret)           # useful library to split up data set
5 library(readr)
6
7 set.seed(420)            # seed to replicate results and get consistent test
                        and training set
8
9 # Load help script with functions to export the results to latex
10 # These functions gathered to avoid duplicate code
11 if(!exists("create_confusion_matrix", mode = "function")){
12   source("Help_Scripts/to_latex_functions.R")
13 }
14
15 #-----#
16
17 ## Data
18 path_data <- paste0(getwd(), "/data")
19 path_to_here <- paste0(getwd(), "/Support_Vector_Machines") # getwd give
                        path to project
20 # which is one folder over
21
22 train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
                        header = TRUE)
23 unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
                        , header = TRUE)
24
25 train_data[,1] <- as.factor(train_data[, 1])
26
27 nzs <- nearZeroVar(train_data[, -1], saveMetrics = TRUE, freqCut = 10000/1,
                        uniqueCut = 1/7)
28 sum(nzs$zeroVar)
29
30 sum(nzs$nzv)
31
32 cut_variables <- rownames(nzs[nzs$nzv == TRUE, ])
33 variable <- setdiff(names(train_data), cut_variables)
34 train_data <- train_data[, variable]
35
36 # split data into test and training set
37 split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
                        FALSE)
38 test_data <- train_data[-split_train_test, ]
39 train_data <- train_data[split_train_test, ]
40
41 # remove label temporarily by storeing as it's own vector
42 digit <- train_data[1]
43 train_data$Digit <- NULL
44 train_data <- train_data/255
45 cov_train <- cov(train_data)
46
47 digit_test <- test_data[1]
48 test_data$Digit <- NULL
49 test_data <- test_data/255
50
51 # use prcomp to do PCA on the covariance matrix
```

```

52 train_pc <- prcomp(cov_train)
53
54 # get information of about the variance
55 var_explained <- train_pc$sdev^2/sum(train_pc$sdev^2)
56 var_explained_cumsum <- cumsum(var_explained)
57 var_explained_df <- data.frame(number = 1:length(train_pc$sdev),
58                               variance_explained = var_explained,
59                               cumsum = var_explained_cumsum)
60 # see plot in end of script
61
62
63 # find optimal number of components
64 find_number_of_components <- function(
65   components_range,
66   increase_by
67 ){
68   components <- seq(from = components_range[1],
69                     to = components_range[2],
70                     by = increase_by)
71   missclassification_error <- rep(0, length(components))
72   cv_error <- rep(0, length(components))
73
74   components_error_df <- data.frame(components,
75                                     missclassification_error,
76                                     cv_error)
77
78   for(i in 1:length(components)){
79
80     # traing the svm for the components given
81     train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:
82       components[i]]
83     train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))
84
85     tune_svm <- tune.svm(Digit ~ ., data = train_data_temp, cost = 1:10,
86       kernal = "radial")
87     fit_svm <- tune_svm$best.model
88
89     # review the performance
90
91     predicted <- predict(fit_svm, train_score, type = "response")
92     missclassification_error <- 1 - mean(train_data_temp[, "Digit"] ==
93       predicted)
94
95     components_error_df[i, "cv_error"] <- tune_svm$best.performance
96     components_error_df[i, "missclassification_error"] <-
97       missclassification_error
98   }
99   return(components_error_df)
100 }
101
102 # make ggplot of cv error for different amount of components
103 # also marks selected number of components manually (22 in this task)
104 plot_components_error <- function(
105   components_error_df
106 ){
107   components_chosen <- components_error_df[22, ]
108
109   ggplot1 <- ggplot(data = components_error_df, aes(x = components)) +
110     geom_line(aes(y = cv_error, colour = "$E_{CV}$")) +

```

```

107     geom_point(aes(y = cv_error, colour = "$E_{CV}$")) +
108     geom_vline(xintercept = 22, color = "black", linetype = "dotdash") +
109     xlab("$n_{components}$") +
110     ylab("Miss.class. Error") +
111     geom_point(data = components_chosen,
112               aes(x = components, y = cv_error),
113               color = "red") +
114     geom_text(data = components_chosen,
115              aes(y = cv_error,
116                  label = paste0("$\\mathrm{CV}_{error}(22) = ",
117                                  round(cv_error,4), "$")),
118              hjust = 0.2, vjust = -1.0) +
119     scale_colour_manual("Legend",
120                         breaks = c("$E_{CV}$"),
121                         values = c("black"),
122                         guide = guide_legend(override.aes = list(
123                           linetype = c("solid"),
124                           shape = c( 16)
125                         ))) +
126     theme_bw() +
127     theme(legend.position = c(0.8, 0.255),
128           legend.background = element_rect(fill=alpha('white', 0)))
129 ggplot_to_latex(ggplot1,
130                 paste0(path_to_here,
131                         "/Results_SVM/number_of_components_cv_error"),
132                 width = 5, height = 5)
133 }
134
135 # Calculate best cost parameter for svm
136 find_optimal_cost_for_components <- function(
137   selected_components,
138   cost_range,
139   cost_increase,
140   n_avg
141 ){
142   train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:selected_
143     components]
144   train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))
145   cost <- seq(from = cost_range[1], to = cost_range[2], by = cost_increase
146     )
147   cost_error <- data.frame(cost = cost, cv_error = rep(0, length(cost)))
148   for(i in 1:n_avg){
149     tune_svm <- tune.svm(Digit ~ ., data = train_data_temp, cost = cost,
150                          kernal = "radial")
151     cv_error <- tune_svm$performances$error
152     cost_error[, "cv_error"] <- cost_error[, "cv_error"] + cv_error
153   }
154   cost_error[, "cv_error"] <- cost_error[, "cv_error"]/n_avg
155   best_cost <- cost_error[which(cost_error[, "cv_error"] == min(cost_error
156     [, "cv_error"]))],]
157
158   ggplot1 <- ggplot(data = cost_error, aes(x = cost)) +
159     geom_line(aes(y = cv_error, colour = "$E_{CV}$")) +
160     geom_point(aes(y = cv_error, colour = "$E_{CV}$")) +

```

```

162     geom_vline(xintercept = best_cost[, "cost"],
163               color = "black", linetype = "dotted") +
164     xlab("$n_{components}$") +
165     ylab("Miss.class. Error") +
166     geom_point(data = best_cost,
167               aes(x = cost, y = cv_error),
168               color = "red") +
169     geom_text(data = best_cost,
170              aes(y = cv_error,
171                 label = paste0("$\\mathrm{CV}_{\\mathrm{error}}(",
172                                cost,
173                                ") = ",
174                                round(cv_error,4), "$")),
175              hjust = -0.05, vjust = 0.9) +
176     scale_colour_manual("Legend",
177                        breaks = c("$E_{CV}$"),
178                        values = c("black"),
179                        guide = guide_legend(override.aes = list(
180                          linetype = c("solid"),
181                          shape = c( 16)
182                        ))) +
183     theme_bw() +
184     theme(legend.position = c(0.8, 0.255),
185           legend.background = element_rect(fill=alpha('white', 0)))
186 ggplot_to_latex(ggplot1,
187                paste0(path_to_here,
188                       "/Results_SVM/cost_cv_error"),
189                width = 5, height = 5)
190 return(best_cost)
191 }
192
193 # Make prediction on the test set and make confusion matrix
194 predict_on_test <- function(
195   selected_components,
196   cost
197 ){
198   train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:selected_
199     components]
200   train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))
201   test_score <- as.matrix(test_data) %*% train_pc$rotation[,1:selected_
202     components]
203   test_data_temp <- cbind(Digit = digit_test, as.data.frame(test_score))
204   svm_predict <- svm(Digit ~ ., data = train_data_temp, cost = cost,
205                     kernal = "radial")
206   predicted <- predict(svm_predict, test_data_temp)
207
208   create_confusion_matrix(predicted_value = predicted,
209                           true_value = test_data_temp[, "Digit"],
210                           destination_path = paste0(path_to_here,
211                                                       "/Results_SVM/
212                                                       confusion_matrix")
213 )
214 }
215 # some help-plots for pca

```



```

216 plot_variance_explained <- function(
217   var_explained_df,
218   chosen_number_components
219 ){
220   # plot amount of variables and how much of the variance this describe
221   ggplot1 <- ggplot(data = var_explained_df[1:100,], aes(x = number, y =
      cumsum)) +
222     geom_line() +
223     geom_point() +
224     geom_vline(xintercept = chosen_number_components,
225               color = "black", linetype = "dotted") +
226     xlab("$n_{components}$") +
227     ylab("Variance Explained") +
228     theme_bw() +
229     theme(legend.position = c(0.8, 0.255),
230           legend.background = element_rect(fill=alpha('white', 0)))
231
232   # plot the data as described by the two first principle components
233
234   train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:chosen_
      number_components]
235   train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))
236
237   ggplot2 <- ggplot(data = train_data_temp, aes(x = PC1, y = PC2)) +
238     geom_point(aes(colour = Digit)) +
239     xlab("PCA 1") +
240     ylab("PCA 2") +
241     scale_colour_manual("Legend",
242                         values = c('#8dd3c7', '#ffffb3', '#bebada', '#
      fb8072',
243                                   '#80b1d3', '#fdb462', '#b3de69', '#
      fccde5',
244                                   '#d9d9d9', '#bc80bd')
245     ) +
246     theme_bw() +
247     theme(legend.position = c(0.9545, 0.355),
248           legend.background = element_rect(fill=alpha('white', 0)))
249
250   ggplot_to_latex(ggplot1,
251                   paste0(path_to_here,
252                           "/Results_SVM/variance_explained_pca"),
253                   width = 5, height = 5)
254
255   ggplot_to_latex(ggplot2,
256                   paste0(path_to_here,
257                           "/Results_SVM/map_pca1_pca2"),
258                   width = 5, height = 5)
259 }
260
261 main <- function(){
262   # Find best number of components to use with svm
263   components_range <- c(1, 30)
264   increase_by <- 1
265   components_error_df <- find_number_of_components(components_range =
      components_range,
266                                                     increase_by = increase_
      by)
267
268   # Plot best cv error over number of components
269   plot_components_error(components_error_df)

```

```

269
270
271     selected_components <- 22
272
273     # Find optimal cost parameter for the svm function
274     best_cost <- find_optimal_cost_for_components(cost_range = c(0.5, 10),
275                                                    cost_increase = 0.5,
276                                                    selected_components =
277                                                         selected_components,
278                                                         n_avgs = 5)
279
280     # Plot pca variance and to principle components
281     plot_variance_explained(var_explained_df, selected_components)
282 }
283 main()

```

../R\_scripts/Support\_Vector\_Machines/support\_with\_pca.R

## 6.11 R-Code: Visualize data examples

```
1 |
2 | ## Data
3 | path_data <- paste0(getwd(), "/data")
4 | path_to_here <- paste0(getwd(), "/Support_Vector_Machines") # getwd give
   | path to project
5 |
6 | train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
   | header = TRUE)
7 |
8 | train_images <- train_data[, -1]
9 |
10 | rotate <- function(x) t(apply(x, 2, rev))
11 |
12 | # plot 5 x 8 images from the data set
13 | png(paste0(path_data, "/images_digits.png"))
14 | opar <- par(no.readonly = T)
15 |
16 | par(mfrow = c(5, 8), mar = c(.1,.1,.1,.1))
17 | for(i in 1:40){
18 |   im_matrix <- matrix(train_images[i, ], nrow = 28, ncol = 28)
19 |   im_matrix <- apply(im_matrix, c(1, 2), function(x) as.numeric(x))
20 |   im_matrix <- rotate(im_matrix)
21 |   image(im_matrix, col = grey.colors(255), axes = F)
22 | }
23 | par(opar)
24 | dev.off()
```

../R\_scripts/Help\_Scripts/print\_images.R