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.

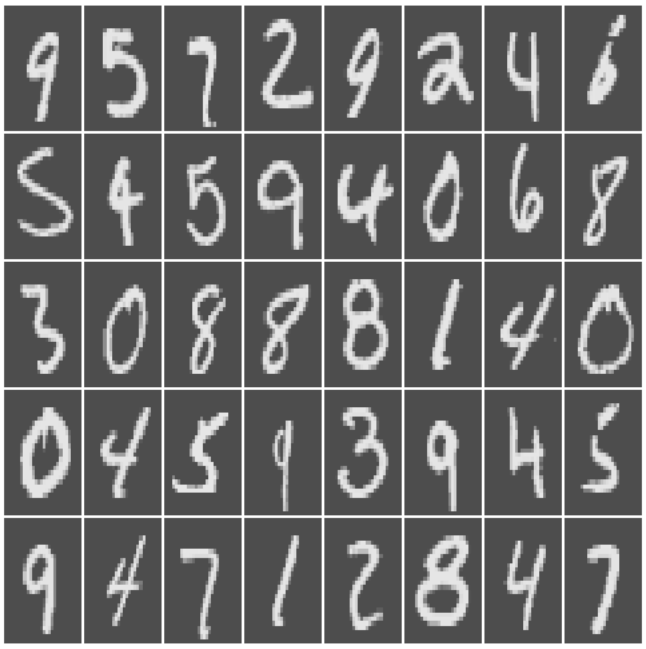


Figure 1: 40 digits from the data set

Contents

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 $m << M$ 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 λ 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

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

, where l indicate the layer, i the input, j the output and θ 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 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:

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

, 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:

`ntree`	`mtry`
500	$\frac{1}{3} \times 784 \approx 261$

Table 1

, where `ntree` is the amount of trees and `mtry` 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 `mtry`. Here the subset of variables is the whole set of variables. This is implemented with the parameters:

`ntree`	`mtry`
500	784

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 λ (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= 5×5, num_filter $= 30$
activation_1	"tanh"
pooling_1	pool_type = "max", kernel $= 2 \times 2$
convolution_2	kernel= 5×5, 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	$p \in [1, 2, ..., 30]$
cost	$cost \in [0.5, 1, 1.5, ..., 10]$

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.

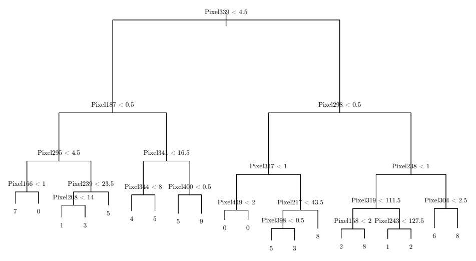
4 Results

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.

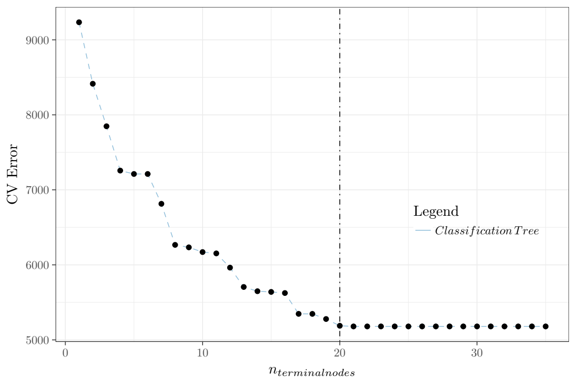
4.1 Tree based methods

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 in 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.

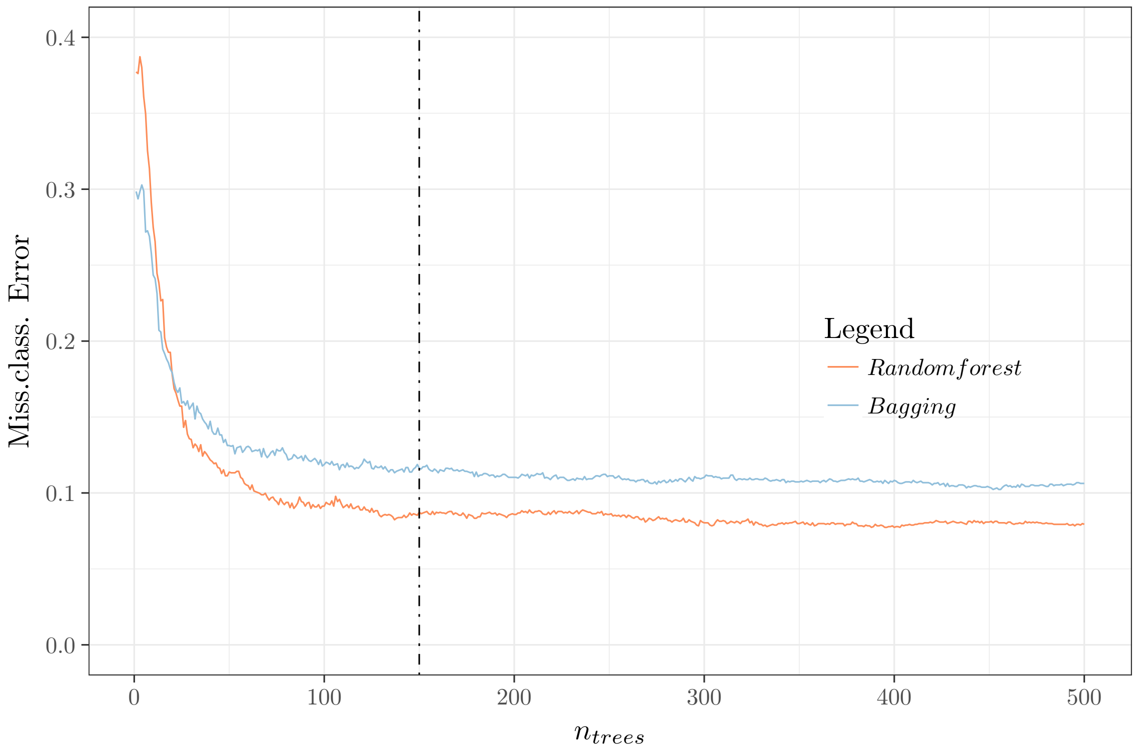


Figure 3: Bagging and Random Forest misclassification over 500 trees.

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 ≈ 2900 trees.

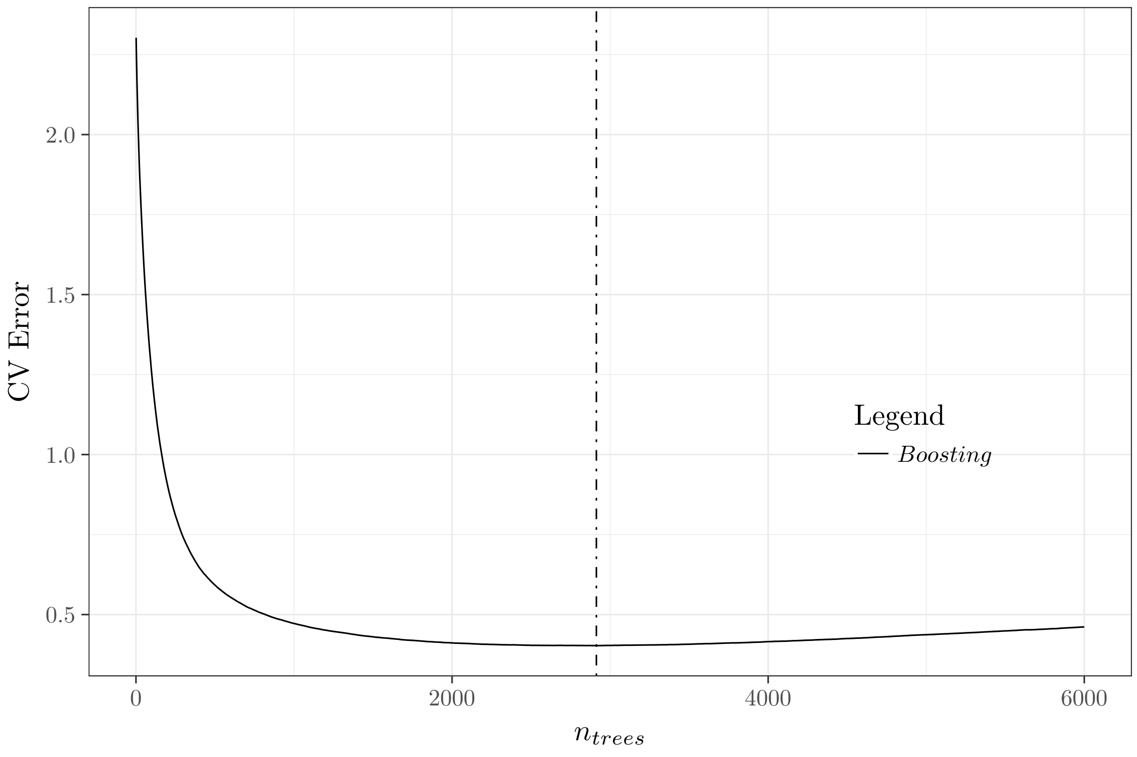


Figure 4: Boosting

Using the best number of trees to predict on the test set giving an error of 11.9%. This means it performes 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`	`l1`
$(540, 320)$	0.2	20	`"Rectifier"`	$1.4e-5$

Table 11

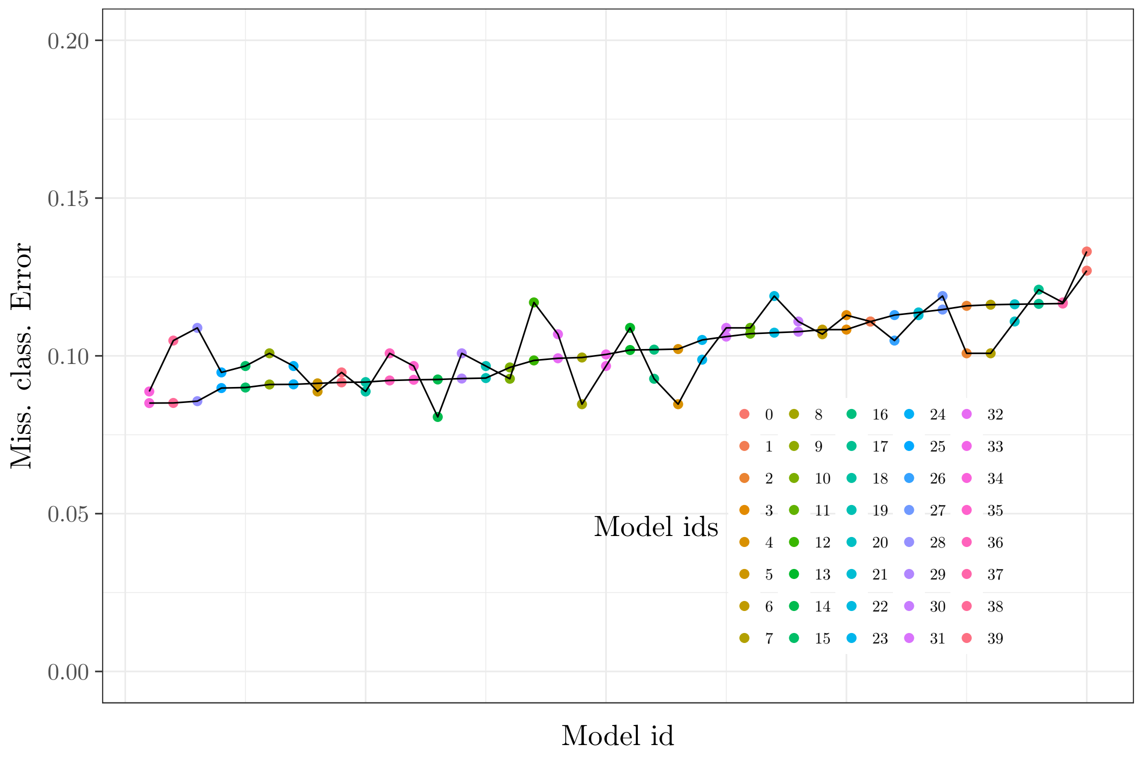


Figure 5: Misclassification error for each model in the grid sorted by the cv-error.

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.

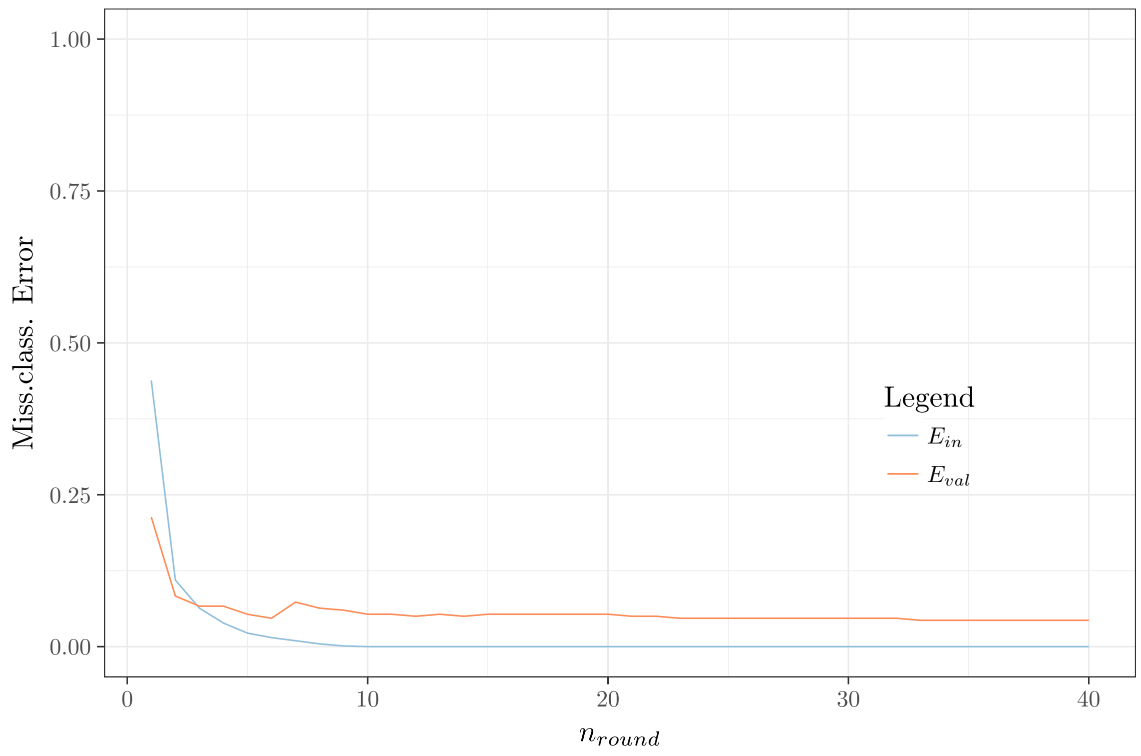


Figure 6: E_{in} and E_{val} for training the network over 40 rounds.

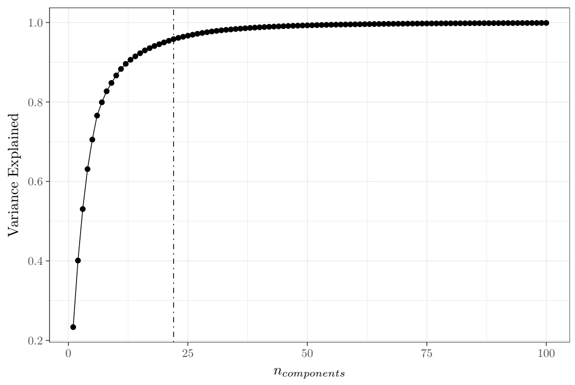
Below we see the predictions made on the test set(20% of the training data) after using taking back the validation set into the training set. The predictions have a misclassification error of 3.6% clearly out concurring all the tree models and the ANN. This demonstrates how powerful CNN is when used 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

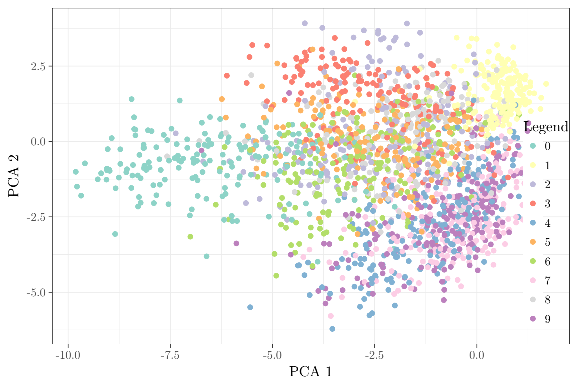
Table 13: Confusion Matrix: Convolutional Neural Network

4.3 Support vector machines w/ PCA

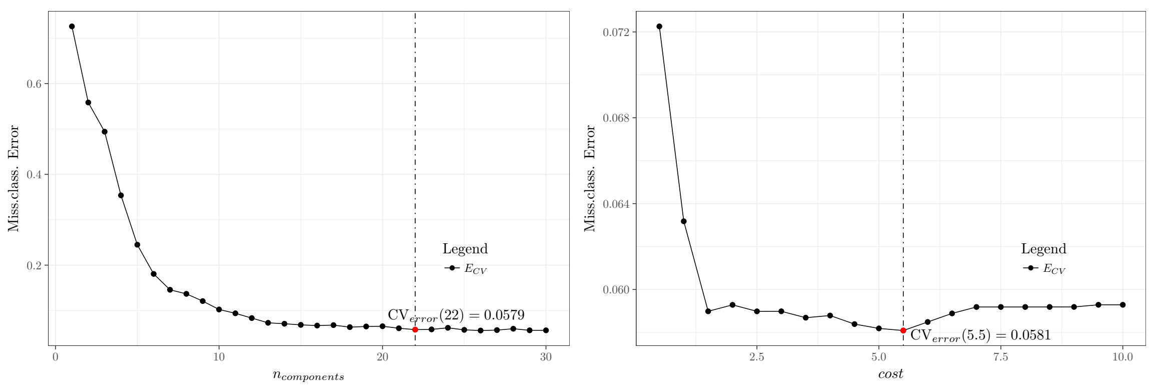
Principal Components Analysis(PCA) showed quite powerful in combination with Support Vector Machines(SVM). First the principal components(PC) was calculated and a as low as possible while still describing the data well number of components were selected as seen in the first plot below. When plotting 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



(a) CV-error for SVM over number of PC's (b) CV-error for 22 PC's over cost parameter

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 is in this task 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 $k = 4, 6, \ldots$.

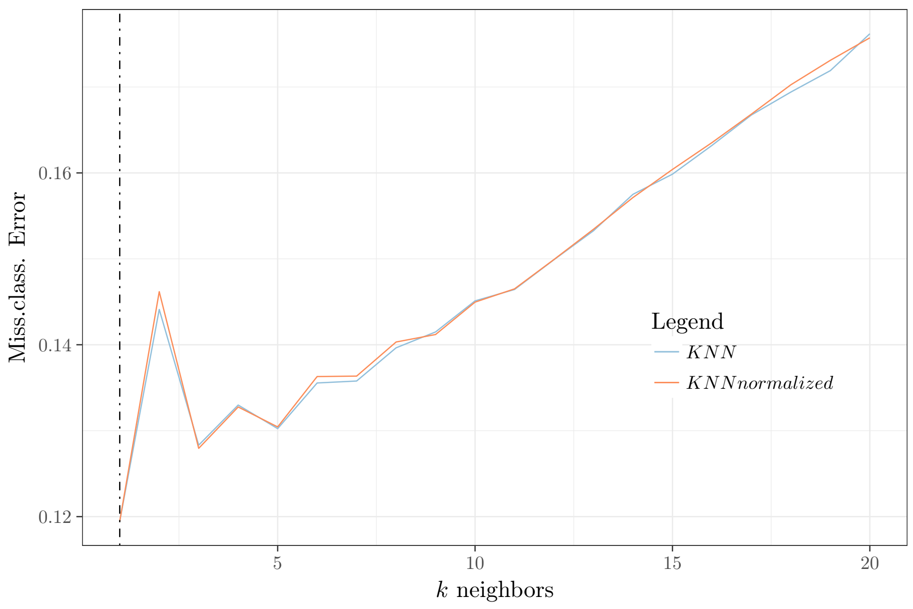


Figure 9

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

```r
require(caret)           # useful library to split up data set
library(tikzDevice)      # library to export plots to .tex files
library(xtable)          # library to export data frames to tables in .tex files

options(tikzMetricPackages = c("\\usepackage[utf8]{inputenc}", "\\usepackage[T1]{fontenc}",
                               "\\usetikzlibrary{calc}", "\\usepackage{amssymb}"))

ggplot_to_latex <- function(
    ggplot,
    destination_path,
    width,
    height
){
    tikz(file = paste0(destination_path, ".tex"), width = 6, height = 4)
    print(ggplot)
    dev.off()
}

create_confusion_matrix <- function(
    predicted_value,
    true_value,
    destination_path
){
    conf <- confusionMatrix(predicted_value, true_value)
    conf_df <- as.data.frame.matrix(conf$table) # extract confusion matrix
    # add row for total error
    conf_df <- rbind(conf_df, Total = rep(0, ncol(conf_df)))

    rows_in_df <- nrow(conf_df)

    classification_frac <- rep("", rows_in_df)
    classification_float <- rep(0, rows_in_df)

    total_wrong <- 0
    total_classified <- 0

    # make columns that shows accuracy
    for(i in 1:(rows_in_df - 1)){
        correct_classified <- conf_df[i, i]
        amount_classified <- sum(conf_df[i, ])
        missclassified <- amount_classified - correct_classified

        classification_frac[i] <- paste0(missclassified, "/", amount_classified)
        classification_float[i] <- missclassified / amount_classified

        total_wrong <- total_wrong + missclassified
        total_classified <- total_classified + amount_classified
    }

    classification_frac[rows_in_df] <- paste0(total_wrong, "/", total_classified)
    classification_float[rows_in_df] <- total_wrong / total_classified

    conf_df <- cbind(temp = row.names(conf_df),           # added extra
        column
                     conf_df,                             # to get predicted
                          classes
                     Error = classification_float,
                     Rate = classification_frac)
    names(conf_df) <- c("", names(conf_df)[-1]) # remove name of predicted
        classes

    write.csv(x = conf_df, file = paste0(destination_path, "_Confusion_
        Matrix.csv"))
    print(xtable(conf_df, display = c("s", rep("d", 11), "f", "s"),
                 digits = c(rep(0, 12), 3, 0)),
          #table.placement = "H",
          only.contents = TRUE,
          file = paste0(destination_path ,"_Confusion_Matrix.tex"),
          include.rownames = FALSE)

}

create_cv_indexes <- function(N, n_folds){
    indexes_per_fold <- floor(N/n_folds)
    index_matrix <- matrix(0L, nrow = n_folds, ncol = indexes_per_fold)
    index_available <- 1:N
    for(i in 1:n_folds){
        selected_indexes <- sample(index_available, indexes_per_fold)
        index_available <- index_available[! index_available %in% selected_
            indexes]

        index_matrix[i, ] <- selected_indexes
    }
    return(index_matrix)
}
```

../R_scripts/Help_Scripts/to_latex_functions.R

6.2 R-Code: Regression Tree

```r
## Libraries and seed
rm(list = ls())
library(caret)
library(readr)
library(tree)            # package used for regression tree
library(tikzDevice)      # library to export plots to .tex files

options(tikzMetricPackages = c("\\usepackage[utf8]{inputenc}", "\\usepackage
    [T1]{fontenc}",
                               "\\usetikzlibrary{calc}", "\\usepackage{
                                   amssymb}"))

set.seed(420)

if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#-------------------#

```

```r
## Data
path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path
    to project
# which is one folder over

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
    header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
    , header = TRUE)

# Remove unnessesary varibles which have a low variance
split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

# Remove variable with low variance which are near zero. Doing it after
# splitting in train/test set to avoid contaminating the data.
near_zero_variables <- nearZeroVar(train_data[,-1], saveMetrics = T, freqCut
     = 10000/1,uniqueCut = 1/7)
cut_variables <- rownames(near_zero_variables[near_zero_variables$nzv ==
    TRUE,])
variables <- setdiff(names(train_data), cut_variables)

train_data <- train_data[, variables]
test_data <- test_data[, variables]

# make the label Digit factor
train_data[, 1] <- as.factor(train_data[, 1])
test_data[, 1] <- as.factor(test_data[, 1])

unclassified_data[,1] <- as.factor(unclassified_data[,1])

#-------------------#

## REGRESSION - tree

regression <- function(
    minimum_development,
    train_data,
    test_data
    ){
    # Chanage name of pixel columns to work with tikz library

    colnames(train_data)[ 2:length(train_data[1,])] <- c(paste0("Pixel", 1:(
        length(train_data[1,]) - 1)))
    colnames(test_data)[ 2:length(train_data[1, ])] <- c(paste0("Pixel", 1:(
        length(train_data[1,]) - 1)))

    # Set minimum development required for making a new split
    minimum_development <- 0.005
    tree_model <- tree(Digit ~ ., data = train_data, mindev = minimum_
        development)
    plot(tree_model)
    text(tree_model, cex = .5)
    print(summary(tree_model))

    cross_validation <- cv.tree(tree_model, K = 10)
    cross_validation$k[1] <- 0
    alpha <- round <- round(cross_validation$k)

    plot(cross_validation$size, cross_validation$dev, type = "b",
         xlab = "Number of terminal nodes", ylab = "CV error")

    ggplot_df <- data.frame(size = cross_validation$size, dev = cross_
       validation$dev)

    destination_path <- paste0(path_to_here, "/Results_TBM/Regression_Tree")

    ggplot1 <- ggplot(data = ggplot_df, aes(x = size, y = dev)) +
               geom_line(aes(colour = "$RegressionTree$"), linetype = "
                  dashed") +
               geom_point() +
               geom_vline(xintercept = 20, color = "black", linetype = "
                  dotdash") +
               xlab("n\\_{terminal nodes}") +
               ylab("CV Error") +
               scale_colour_manual("Legend",
                                   breaks = c("$RegressionTree$"),
                                   values = c("#91bfdb"),
                                   guide = guide_legend(override.aes = list(
                                       linetype = c("solid"),
                                       shape = c(16)
                                   ))) +
               theme_bw() +
               theme(legend.position = c(0.8, 0.355),
                  legend.background = element_rect(fill=alpha('white', 0)))
    ggsave(paste0(destination_path, ".png"))

    ggplot_to_latex(ggplot1, destination_path, width = 5, height = 5)
    tree_prune <- prune.tree(tree_model, best = 20)
    summary(tree_prune)

    tikz(file = paste0(destination_path, "_Tree.tex"), width = 6, height =
       4)
    plot(tree_prune)
    text(tree_prune, cex = .5)
    dev.off()

    predicted <- predict(tree_prune, test_data, type = "class")
    create_confusion_matrix(predicted, test_data[,1], destination_path)
    }

regression(0.05, train_data, test_data)
#-------------------#
```

../R_scripts/Tree_Based_Methods/Regression_Tree.R

6.3 R-Code: Random Forest

```r
## Libraries and seed
rm(list = ls())
library(randomForest)   # library giving a easy-to-use random forest method
library(caret)          # useful library to split up data set
library(tikzDevice)     # library to export plots to .tex files
library(xtable)         # library to export data frames to tables in .tex files
set.seed(420)           # seed to replicate results and get consistent test and training set

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#--------------------#

## Data
path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path to project
                                                        # which is one folder over

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"), header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv"), header = TRUE)

train_data[,1] <- as.factor(train_data[, 1])

# split training set into training and test set

split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list = FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

#--------------------#

## Random forest
# Train forest
train_random_forest <- function(
    data,
    n_trees,
    minimum_development = 0.01
    ){
        random_forest <- randomForest(Digit ~ .,
                                      data = data,
                                      ntree = n_trees,
                                      #mindev = minimum_development,
                                      importance = TRUE,
                                      na.action = na.exclude)
        return(random_forest)
    }

```

```r
# Plot error as the number of trees increase

plot_error_development <- function(
    random_forest_data,
    destination_path
    ){
        error_data <- data.frame(n_trees = 1:nrow(random_forest_data$err.rate),
                                 error <- random_forest_data$err.rate[,"OOB"])

        write.csv(error_data, file = paste0(destination_path, ".csv"))

        ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
            geom_line(aes(y = error, colour = "$Random forest")) +
            xlab("$n\\_{trees}$") +
            ylab("Miss. class. Error") +
            scale_colour_manual("Legend",
                                breaks = c("$Random forest$"),
                                values = c("black"),
                                guide = guide_legend(override.aes = list(
                                    linetype = c("solid"),
                                    shape = c( 16)
                                ))) +
            theme(legend.position = c(0.9, 0.2))
        ggsave(paste0(destination_path, ".png"))

        ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
}

main <- function(){
    n_trees = 50
    random_forest <- train_random_forest(train_data, n_trees)
    plot_error_development(random_forest, paste0(path_to_here, "/Results_TBM/Random_Forest_",
                                                 n_trees, "trees_Error_plot"))

    prediction <- predict(random_forest, newdata = test_data)
    create_confusion_matrix(predicted_value = prediction, true_value = test_data$Digit,
                            paste0(path_to_here, "/Results_TBM/Random_Forest_",
                                                                n_trees, "trees"))
}

main()
```

../R_scripts/Tree_Based_Methods/Random_Forest.R

6.4 R-Code: Bagging

```r
## Libraries and seed
rm(list = ls())
library(randomForest)   # library giving a easy-to-use random forest method
library(caret)          # useful library to split up data set
library(tikzDevice)     # library to export plots to .tex files
library(xtable)         # library to export data frames to tables in .tex
    files
set.seed(420)           # seed to replicate results and get consistent test
    and training set

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#-------------------#

## Data

path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path
    to project
# which is one folder over

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
    header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
    , header = TRUE)

train_data[,1] <- as.factor(train_data[, 1])

# split training set into training and test set

split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

#-------------------#

## Random forest
# Train forest
train_bagging <- function(
    data,
    n_trees
){
    n_features <- ncol(data) - 1
    bagging <- randomForest(Digit ~ .,
                                  data = data,
                                  ntree = n_trees,
                                  mtry = n_features,
                                  importance = TRUE,
                                  na.action = na.exclude)
    return(bagging)
}

```

```r
# Plot error as the number of trees increase

plot_error_development <- function(
    random_forest_data,
    destination_path
){
    error_data <- data.frame(n_trees = 1:nrow(random_forest_data$err.rate),
                             error <- random_forest_data$err.rate[,"OOB"])

    write.csv(error_data, file = paste0(destination_path ,".csv"))
    ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
        geom_line(aes(y = error, colour = "$Bagging$")) +
        xlab("n\\_{trees}") +
        ylab("Miss.class. Error") +
        scale_colour_manual("Legend",
                            breaks = c("$Bagging$"),
                            values = c("black"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid"),
                                shape = c( 16)
                            ))) +
        theme(legend.position = c(0.9, 0.2))
    ggsave(paste0(destination_path, ".png"))

    ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
}

main <- function(){
    n_trees <- 50
    bagging <- train_bagging(train_data, n_trees)
    plot_error_development(bagging, paste0(path_to_here, "/Results_TBM/
        Bagging_",
                                            n_trees, "trees_Error_plot"))

    prediction <- predict(bagging, newdata = test_data)

    create_confusion_matrix(predicted_value = prediction, true_value = test_
        data$Digit,
                            paste0(path_to_here, "/Results_TBM/Bagging_",
                                   n_trees, "trees"))
}

main()
```
../R_scripts/Tree_Based_Methods/Bagging.R

6.5 R-Code: Boosting

```r
## Libraries and seed
rm(list = ls())
library(caret)          # useful library to split up data set
library(gbm)            # library with powerful boosting method
set.seed(420)           # seed to replicate results and get consistent test and training set

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#-------------------#

## Data

path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path to project

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"), header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv"), header = TRUE)

train_data[,1] <- as.factor(train_data[, 1])

# split training set into training and test set

split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list = FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

# Remove variable with low variance which are near zero. Doing it after
# splitting in train/test set to avoid contaminating the data.
near_zero_variables <- nearZeroVar(train_data[,-1], saveMetrics = T, freqCut = 10000/1,uniqueCut = 1/7)
cut_variables <- rownames(near_zero_variables[near_zero_variables$nzv == TRUE,])
variables <- setdiff(names(train_data), cut_variables)

train_data <- train_data[, variables]
test_data <- test_data[, variables]

#-------------------#

## Boosting
# Train booster
boosting <- function(
    data,
    n_trees,
    interaction_depth = 2,
    shrinkage = 0.001
){
    boosting <- gbm(Digit ~ .,
                    data = data,
                    distribution = "multinomial",
                    n.trees = n_trees,
                    interaction.depth = interaction_depth,
                    shrinkage = shrinkage,
                    cv.folds = 10,
                    n.cores = 4)
    return(boosting)
}


# Plot error as the number of trees increase

plot_error_development <- function(
    boosting_data,
    best_n_trees,
    destination_path
){
    error_data <- data.frame(n_trees = 1:length(boosting_data$cv.error),
                             error <- boosting_data$cv.error)
    write.csv(error_data, file = paste0(destination_path ,".csv"))

    ggplot1 <- ggplot(data = error_data, aes(x = n_trees)) +
        geom_line(aes(y = error, colour = "$Boosting$")) +
        xlab("$n_{trees}$") +
        ylab("Miss.class. Error") +
        geom_vline(xintercept = best_n_trees, color = "black", linetype = "
            dotdash") +
        scale_colour_manual("Legend",
                            breaks = c("$Boosting$"),
                            values = c("black"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid"),
                                shape = c( 16)
                            ))) +
        theme(legend.position = c(0.9, 0.2)) +
        theme_bw() +
        theme(legend.position = c(0.8, 0.355),
              legend.background = element_rect(fill=alpha('white', 0)))
    ggsave(paste0(destination_path, ".png"))

    ggplot_to_latex(ggplot1, destination_path, width = 6, height = 4)
}

predict_data <- function(
    boosting_train,
    best_n_trees,
    test_data
){
    predicted <- predict(boosting_train, newdata = test_data, n.trees = best
        _n_trees, type = "response")

    predicted <- apply(predicted, 1, function(x) which.max(x) - 1)
    return(predicted)
}

main <- function(){
    n_trees = 10000
    boosting_train <- boosting(train_data,n_trees, shrinkage = 0.01)

    # Get amount of trees with best performance based on 10 fold cv
    best_n_trees <- gbm.perf(boosting_train, method = "cv", plot.it = FALSE)

    # Plot error
    plot_error_development(boosting_train, best_n_trees, paste0(path_to_here,
                                                 "/Results_TBM/Boosting_",
                                                 n_trees,
                                                 "trees_Error_plot"))

    # Perdict test set labels
    predicted <- predict_data(boosting_train, best_n_trees, test_data)

    # Create confusion matrix for the result
    create_confusion_matrix(predicted, test_data$Digit, paste0(path_to_here,
                                                              "/Results_TBM/Boosting_",
                                                              n_trees))
}

main()
```

../R_scripts/Tree_Based_Methods/Boosting.R

6.6 R-Code: Plot Random Forest w/ Bagging

```r
rm(list = ls())
library(ggplot2)
library(tikzDevice)     # library to export plots to .tex files

path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path
    to project
                                                        # which is one
                                                           folder over

plot_random_forest_bagging <- function(
    n_trees,
    path,
    destination_path
    ){
        rf_path <- paste0(path,
                          "Random_Forest_",
                          n_trees,
                          "trees_Error_plot_",
                          n_trees,
                          "trees.csv")
        bagging_path <- paste0(path,
                          "Bagging_",
                          n_trees,
                          "trees_Error_plot_",
                          n_trees,
                          "trees.csv")

        random_forest_error <- read.csv(rf_path)
        bagging_error <- read.csv(bagging_path)

        ggplot_df <- data.frame(n_trees = 1:nrow(bagging_error),
                                rf = random_forest_error[3],
                                bag = bagging_error[3])
        names(ggplot_df) <- c("n_trees", "rf", "bag")
        print(str(ggplot_df))

        tikz(file = paste0(destination_path, ".tex"), width = 6, height = 4)
        ggplot1 <- ggplot(data = ggplot_df, aes(x = n_trees)) +
            geom_line(aes(y = rf, colour = "$Random forest$")) +
            geom_line(aes(y = bag, colour = "$Bagging$")) +
            geom_vline(xintercept = 150, color = "black", linetype = "
                dotdash") +
            xlab("$n_{trees}$") +
            ylab("Miss.class. Error") +
            scale_colour_manual("Legend",
                                breaks = c("$Random forest$", "$Bagging$"),
                                values = c("#91bfdb", "#fc8d59"),
                                guide = guide_legend(override.aes = list(
                                    linetype = c("solid", "solid"),
                                    shape = c( 16, 16)
                                ))) +
            scale_y_continuous(limits = c(0, 0.4)) +
            theme_bw() +
            theme(legend.position = c(0.8, 0.455),
                  legend.background = element_rect(fill=alpha('white', 0)))
        ggsave(paste0(destination_path, ".png"))
        print(ggplot1)
        dev.off()
    }

main <- function(

    ){
        n_trees = 500
        path = paste0(path_to_here, "/Results_TBM/")
        destination_path = paste0(path, "/Random_Forest_Bagging_",
                                  n_trees, "trees")

        plot_random_forest_bagging(n_trees, path, destination_path)

    }

main()
```
../R_scripts/Tree_Based_Methods/Plot_Random_Forest_Bagging.R

6.7 R-Code: Neural Network

```r
## Libraries and seed
rm(list = ls())
library(h2o)
library(caret)
library(reshape2)

set.seed(420)

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}
#-------------------#

## Data

path_data <- getwd()
path_to_here <- paste0(getwd(), "/Neural_Networks")

train_data <- read.csv(paste0(path_data, "/data/Train_Digits_20171108.csv"))
unclassified_data <- read.csv(paste0(path_data, "/data/Test_Digits_20171108.
    csv"))

local.h2o <- h2o.init(ip = "localhost", port = 54321, startH2O = TRUE,max_
    mem_size = "7G", nthreads = -1)

train_data[,1] <- as.factor(train_data[, 1])
split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

train_data <- as.h2o(train_data)
unclassified_data <- as.h2o(unclassified_data)
test_data <- as.h2o(test_data)

#-------------------#

## Getting useful data from grid run of neural networkss

get_data_in_df <- function(
    data
)
{
    n <- length(data@model_ids)
    mse_errors <- rep(0,n)
    mean_per_class_errors <- rep(0,n)
    hidden <- rep("", n)
    str(hidden)
    rate <- rep(0,n)
    l1 <- rep(0,n)
    epochs <- rep(0,n)
    model_numbers <- rep(0,n)
    train_error <- rep(0,n)
    train_mse <- rep(0,n)
    test_error <- rep(0,n)
    test_mse <- rep(0,n)
    activation <- rep("",n)
    input_dropout_ratio <- rep(0,n)
    nesterov_accelerated_gradient <- rep("", n)

    model_df <- data.frame(model_numbers = mse_errors, hidden, rate, l1,
        epochs,
                            train_error, test_error, train_mse, test_mse,
                                activation, input_dropout_ratio,
                                stringsAsFactors = FALSE)
    str(model_df)

    for(i in 1:n){
        model <- h2o.getModel(data@model_ids[[i]])
        model_df$mse_errors[i] <- h2o.mse(model)
        #model_df$mean_per_class_error[i] <- model@model$cross_validation_
            metrics@metrics$mean_per_class_error
        model_df$mean_per_class_error[i] <- h2o.performance(model, xval = T)
            @metrics$mean_per_class_error

        model_paramaters <- model@allparameters
        model_name <- model@model_id
        model_number <- sub(".*model_(.*)$", "\\1", model_name)
        model_df$model_numbers[i] <- as.integer(model_number)
        model_df$hidden[i] <- paste(as.character(model_paramaters$hidden),
            sep = " ", collapse = ", ")
        model_df$rate[i] <- model_paramaters$rate
        model_df$l1[i] <- model_paramaters$l1
        model_df$epochs[i] <- model_paramaters$epochs
        model_df$activation[i] <- model_paramaters$activation
        model_df$input_dropout_ratio[i] <- model_paramaters$input_dropout_
            ratio
        model_df$nesterov_accelerated_gradient[i] <- model_paramaters$
            nesterov_accelerated_gradient

        train_performance <- h2o.performance(model, train_data)@metrics
        train_performance_error <- train_performance$mean_per_class_error
        train_performance_mse <- train_performance$MSE

        model_df$train_error[i] <- train_performance_error
        model_df$train_mse[i] <- train_performance_mse

        test_performance <- h2o.performance(model, test_data)@metrics
        test_predictions <- h2o.predict(model, test_data)
        test_accuracy <- test_predictions$predict == test_data$Digit
        test_performance_error <- 1 - mean(test_accuracy)
        test_performance_mse <- test_performance$MSE

        model_df$test_error[i] <- test_performance_error
        model_df$test_mse[i] <- test_performance_mse

    }
    model_df <- model_df[with(model_df, order(model_numbers)),]
    model_df
}

# Run and plot grid of n neural nets
run_neural_network_grid <- function(){
    activation <- list("Rectifier", "RectifierWithDropOut")# "Tanh")
```

```r
    hidden <- list(c(100,100), c(150, 150),  c(540, 320), c(100, 100, 100),
        c(540, 320, 100))
    input_dropout_ratio <- list(0, 0.2)
    nesterov_accelerated_gradient <- list( TRUE)
    epochs <- list(20)
    l1 = list(0,1.4e-5)
    hyper_params <- list(activation = activation, hidden = hidden, input_
        dropout_ratio = input_dropout_ratio, nesterov_accelerated_gradient =
         nesterov_accelerated_gradient, epochs = epochs, l1 = l1)

    grid_deep_learning <- h2o.grid(algorithm = "deeplearning",
                             x = 2:785,
                             y = 1,
                             training_frame = train_data,
                             nfolds = 10,
                             stopping_metric = "MSE",
                             stopping_tolerance = 0.0025,
                             hyper_params = hyper_params)
    save_results <- function(results){
    write.csv(results, file = paste0(path_to_here, "/Neural_Networks/results
        _NN/grid_run_20.csv"))
    }

    df <- get_data_in_df(grid_deep_learning)
    save_results(df)

    #results_df <- df

    results_df <- read.csv(paste0(path_to_here, "/Neural_Networks/results_NN
        /grid_run_40.csv"))

    results_df <- results_df[with(results_df, order(mean_per_class_error)),]
    results_df$row_names <- 1:length(results_df[,1])

    melt_datas <- melt(results_df[c("test_error","mean_per_class_error", "
        row_names",
                              "model_numbers")], id = c("row_names", "model_
                                  numbers"))
    plot_list  <- list()
    # Plot classification error
    plot_list[[1]] <- ggplot(data=melt_datas,
                       aes(x=row_names, y=value)) +
    geom_point(aes(colour = as.factor(model_numbers), group = as.factor(
        model_numbers)), size = 1.25) +
    geom_line(aes(group = variable)) +
    xlab("Model id") +
    ylab("Miss. class. Error") +
    scale_y_continuous(limits = c(0, 0.2)) +
    theme_bw() +
    theme(legend.position = c(0.675, 0.255),
      legend.background = element_rect(fill=alpha('white', 0)),
      legend.direction = "horizontal",
      legend.text = element_text(size=6),
      legend.key = element_rect(size = 3),
      legend.key.size = unit(1.0, 'lines'),
      axis.text.x=element_blank(),
      axis.ticks.x=element_blank()) +
    scale_colour_discrete(name = "Model ids") +
    guides(fill = guide_legend(title = "Legend"))

    ggplot_to_latex(plot_list[[1]],
            paste0(path_to_here, "/results_NN/per_class_error"), width = 6,
                height = 4)
    ggsave(paste0(path_to_here,"/results_NN/per_class_error3.png"))
}

# Run final model based on results from grid run and make confusion matrix
    for prediction on test set
run_final_model <- function(){
    deep_learning_results3<- h2o.deeplearning(x = 2:785,
                                              y = 1,
                                              training_frame = train_data,
                                              activation = "Rectifier",
                                              input_dropout_ratio = 0.2,
                                              nfolds = 10,
                                              balance_classes = TRUE,
                                              hidden = c(540, 320),
                                              l1 = 1.4e-5,
                                              stopping_metric = "MSE",
                                              stopping_tolerance = 0.0025,
                                              nesterov_accelerated_gradient =
                                                  TRUE,
                                              epochs = 20)

    h2o.performance(deep_learning_results3, test_data)


    predicted <- predict(deep_learning_results3, test_data, type = "response
        ")

    predicted_confusion_matrix <- as.factor(as.vector(predicted$predict))
    test_data_confusion_matrix <- as.data.frame(test_data)
    create_confusion_matrix(predicted_confusion_matrix,
                            test_data_confusion_matrix[, "Digit"],
                            paste0(path_to_here, "/results_NN/540_320_neural
                                _net"))
}

run_final_model

h2o.shutdown()
```

../R_scripts/Neural_Networks/neural_network.R

6.8 R-Code: Convolutional Neural Network

```r
## Libraries and seed
rm(list = ls())
library(mxnet)      #library for running convolutional neural network
library(caret)

set.seed(420)

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}
#--------------------#

## Data

path_data <- getwd()
path_to_here <- paste0(getwd(), "/Neural_Networks")

train_data <- read.csv(paste0(path_data, "/data/Train_Digits_20171108.csv"))
unclassified_data <- read.csv(paste0(path_data, "/data/Test_Digits_20171108.
    csv"))

train_data[, 1] <- as.factor(train_data[, 1])
train_data_full <- train_data

# Split the data into training, test and validation set
split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)

test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

split_train_validation <- createDataPartition(train_data$Digit, p = 0.85,
    list = FALSE)

validation_data <- train_data[-split_train_validation, ]
train_data_val <- train_data[split_train_validation, ]

# Setting up datasets as matrices in order to get correct input for mxnet

# Training after tuning
train <- train_data
train_x <- t(train[, -1])
train_y <- train[, 1]
train_array <- train_x
dim(train_array) <- c(28, 28, 1, ncol(train_x))

# Training under tuning
train_val <- train_data_val
train_x_val <- t(train_val[, -1])
train_y_val <- train_val[, 1]
train_array_val <- train_x_val
dim(train_array_val) <- c(28, 28, 1, ncol(train_x_val))

# Test set to compare with other methods
test_x <- t(test_data[, -1])
```

```r
test_y <- test_data[, 1]
test_array <- test_x
dim(test_array) <- c(28, 28, 1, ncol(test_x))

# Validation set used to validate under tuning
validation_x <- t(validation_data[, -1])
validation_y <- validation_data[, 1]
validation_array <- validation_x
dim(validation_array) <- c(28, 28, 1, ncol(validation_x))

# Get right format on the unclassified set
unclassified_x <- t(unclassified_data[, -1])
unclassified_array <- unclassified_x
dim(unclassified_array) <- c(28, 28, 1, ncol(unclassified_x))

# Full training data when making prediction for unclassified data
train_full <- train_data_full
train_x_full <- t(train_full[, -1])
train_y_full <- train_full[, 1]
train_array_full <- train_x_full
dim(train_array_full) <- c(28, 28, 1, ncol(train_x_full))

#-------------------#

data <- mx.symbol.Variable("data")

two_layer_concolutional_network <- function(){
    # Setting up first convolutional layer

    convolution_1 <- mx.symbol.Convolution(data = data, kernel = c(5,5), num
        _filter = 30)
    activation_1 <- mx.symbol.Activation(data = convolution_1, act_type = "
        tanh")
    pooling_1 <- mx.symbol.Pooling(data = activation_1, pool_type = "max",
        kernel = c(2, 2), stride = c(2,2))

    # Setting up second convolutional layer

    convolution_2 <- mx.symbol.Convolution(data = pooling_1, kernel = c(5,5)
        , num_filter = 50)
    activation_2 <- mx.symbol.Activation(data = convolution_2, act_type = "
        tanh")
    pooling_2 <- mx.symbol.Pooling(data = activation_2, pool_type = "max",
        kernel = c(2, 2), stride = c(2,2))

    # Setting up first fully connected layer

    flatten <- mx.symbol.Flatten(data = pooling_2)
    fully_1 <- mx.symbol.FullyConnected(data = flatten, num_hidden = 500)
    activation_3 <- mx.symbol.Activation(data = fully_1, act_type = "relu")

    # Setting up second fully connected layer

    fully_2 <- mx.symbol.FullyConnected(data = activation_3, num_hidden =
        40)

    return(fully_2)

}
```

```r

# three layer convolutional neural network to experiment, turned out to perform worse
three_layer_concolutional_network <- function(){
    # Setting up first convolutional layers

    convolution_1 <- mx.symbol.Convolution(data = data, kernel = c(5,5), num_filter = 30)
    activation_1 <- mx.symbol.Activation(data = convolution_1, act_type = "tanh")
    pooling_1 <- mx.symbol.Pooling(data = activation_1, pool_type = "max", kernel = c(2, 2), stride = c(2,2))

    # Setting up second convolutional layers

    convolution_2 <- mx.symbol.Convolution(data = pooling_1, kernel = c(5,5), num_filter = 50)
    activation_2 <- mx.symbol.Activation(data = convolution_2, act_type = "tanh")
    pooling_2 <- mx.symbol.Pooling(data = activation_2, pool_type = "max", kernel = c(2, 2), stride = c(2,2))

    # Setting up thrid convolutional layers
    convolution_3 <- mx.symbol.Convolution(data = pooling_2, kernel = c(3,3), num_filter = 50)
    activation_3 <- mx.symbol.Activation(data = convolution_3, act_type = "tanh")
    pooling_3 <- mx.symbol.Pooling(data = activation_3, pool_type = "max", kernel = c(2, 2), stride = c(2,2))

    # Setting up first fully connected layer

    flatten <- mx.symbol.Flatten(data = pooling_3)
    fully_1 <- mx.symbol.FullyConnected(data = flatten, num_hidden = 500)
    activation_3 <- mx.symbol.Activation(data = fully_1, act_type = "tanh")

    # Setting up second fully connected layer

    fully_3 <- mx.symbol.FullyConnected(data = activation_3, num_hidden = 40)

    return(fully_3)

}

# Output layer, softmax gives probabilies for the output

run_convolutional_neural_network <- function(
    neural_net_model,
    train_array,
    train_y,
    test_array
){
    mx.set.seed(100)

    cpu_used <- mx.cpu()


    logger_mx <- mx.metric.logger$new()
    train_model <- mx.model.FeedForward.create(neural_net_model,
                                               X = train_array,
                                               y = train_y,
                                               #eval.data = list(data = validation_array, label = validation_y), #must be set when validating
                                               ctx = cpu_used,
                                               num.round = 40,
                                               array.batch.size = 50,
                                               learning.rate = 0.01,
                                               momentum = 0.9,
                                               initializer = mx.init.Xavier(),
                                               eval.metric = mx.metric.accuracy,
                                               epoch.end.callback = mx.callback.log.train.metric(100, logger_mx)
                                               )

    predicted <- predict(train_model, test_array)
    #mxnet gives wrong labels by 1, that is 1-10 instead of 0-9
    #fix by subtracting 2
    predicted_labels <- max.col(t(predicted)) - 2

    return(list(logger_mx, predicted_labels))
}

plot_error_development <- function(
    neural_net_model,
    train_array_val,
    train_y_val,
    test_array,
    test_y
){
    # NB! remember to remove "#" from "eval.data" in "run_convolutional_neural_network"
    # run cnn to find error in and val
    error <- run_convolutional_neural_network(neural_net_model,
                                             train_array_val,
                                             train_y_val,
                                             test_array)

    # extract error from results
    error_t <- error[[1]]

    error_in_sample <- (1 - error_t$train)
    error_val <- (1 - error_t$eval)

    ggplot_df <- data.frame(round = 1:length(error_in_sample),
                            error_in_sample = error_in_sample,
                            error_val = error_val)

    # plot error in and validation
    ggplot1 <- ggplot(data = ggplot_df, aes(x = round)) +
        geom_line(aes(y = error_in_sample, colour = "$E_{in}$")) +
        geom_line(aes(y = error_val, colour = "$E_{val}$")) +
        xlab("$n_{round}$") +
        ylab("Miss.class. Error") +
        scale_y_continuous(limits = c(0, 1.0)) +
        scale_colour_manual("Legend",
                            breaks = c("$E_{in}$", "$E_{val}$"),
                            values = c("#91bfdb", "#fc8d59"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid", "solid"),
                                shape = c( 16, 16)
                            ))) +
        theme_bw() +
        theme(legend.position = c(0.8, 0.355),
              legend.background = element_rect(fill=alpha('white', 0)))
    ggplot_to_latex(ggplot1, paste0(path_to_here, "/results_NN/convolutional
        _neural_network_40_rounds"),
                    width = 6, height = 4)
}

# Predict on test set splitted from the training data to compare with other
    methods
predict_on_test_set <- function(
    neural_net_model,
    train_array,
    train_y,
    test_array,
    test_y
){
    predicted <- run_convolutional_neural_network(neural_net_model,
                                                   train_array,
                                                   train_y,
                                                   test_array)
    predicted <- predicted[[2]]

    create_confusion_matrix(as.factor(predicted), test_y,
                            paste0(path_to_here, "/results_NN/convolutional_
                                neural_network_40_rounds"))
}

# Give predictions on unclassified data
predict_on_unclassified_data <- function(
    neural_net_model,
    train_array_full,
    train_y_full,
    unclassified_array
){
    predicted <- run_convolutional_neural_network(neural_net_model,
                                                   train_array_full,
                                                   train_y_full,
                                                   unclassified_array)
    predicted <- predicted[[2]]

    predicted_digit <- predicted
    predicted_odd_even <- as.factor(as.integer(predicted) %% 2)
    prediction_csv <- data.frame(number = 1:length(predicted_digit),
                                digits = predicted_digit,
                                odd_even = predicted_odd_even)
    write.csv(prediction_csv, file = paste0(path_data, "/data/predictions_
        STNKAR012.csv"))

}
```

```r

main <- function()
    {
    # Validation used under tuning, set to false so it does run after tuning
    validation_boolean <- FALSE
    # Test used under testing of the final tuned model to compare with other
        methods
    train_boolean <- FALSE

    fully_2 <- two_layer_concolutional_network()
    neural_net_model <- mx.symbol.SoftmaxOutput(data = fully_2)

    if(validation_boolean){
        plot_error_development(neural_net_model,
                               train_array_val,
                               train_y_val,
                               test_array,
                               test_y)
    }

    if(train_boolean){
        predict_on_test_set(neural_net_model,
                            train_array,
                            train_y,
                            test_array,
                            test_y)
    }

    predict_on_unclassified_data(neural_net_model,
                                 train_array_full,
                                 train_y_full,
                                 unclassified_array)

}
main()
```

../R_scripts/Neural_Networks/convolutional_neural_network.R

6.9 R-Code: K-nearest Neighbours

```r
## Libraries and seed
rm(list = ls())

library(foreach)        # library for running loop in parallel
library(doParallel)     # library for running loop in parallel
library(caret)          # useful library to split up data set
library(tikzDevice)     # library to export plots to .tex files
library(xtable)         # library to export data frames to tables in .tex files
library(class)          # library for knn method
set.seed(420)           # seed to replicate results and get consistent test and training set

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#-------------------#

## Data
path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Tree_Based_Methods")   # getwd give path to project
# which is one folder over


train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"), header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv"), header = TRUE)

train_data[,1] <- as.factor(train_data[, 1] )

# split training set into training and test set

split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list = FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]


k_nearest_neighbors <- function(
    train_data,
    train_data_norm,
    k_folds,
    k_neighbors = 20
){
    # First find number of k(neighbors) using crossvalidation
    cv_indexes <- create_cv_indexes(nrow(train_data), k_folds)

    cores=detectCores()
    cl <- makeCluster(cores[1]-1) #not to overload your computer
    registerDoParallel(cl)

    cv_error <- c()
    indexes <- 1:nrow(train_data)

    final_df <- foreach(i = 1:k_neighbors,
                        .combine = "rbind",
                        .packages = "class") %dopar%{
            error_kfolds <- 0
            error_kfolds_norm <- 0
            for(k in 1:k_folds){
                train_val <- train_data[!indexes %in% cv_indexes[k, ], ]
                validation_val <- train_data[cv_indexes[k, ],]

                train_val_norm <- train_data_norm[!indexes %in% cv_indexes[k
                    , ], ]
                validation_val_norm <- train_data_norm[cv_indexes[k, ],]

                # without normalization
                knn_pred_class <- knn(train_val[-1],
                                      validation_val[-1],
                                      train_val[, 1],
                                      k = i)
                # with normalization
                knn_pred_class_norm <- knn(train_val_norm[-1],
                                           validation_val_norm[-1],
                                           train_val_norm[, 1],
                                           k = i)

                error <- 1 - mean(validation_val[, 1] == knn_pred_class)
                error_norm <- 1 - mean(validation_val_norm[, 1] == knn_pred_
                    class_norm)
                error_kfolds <- error_kfolds + error
                error_kfolds_norm <- error_kfolds_norm + error_norm
            }
            temp_df <- data.frame(k_neighbors = i,
                                  error_kfolds = error_kfolds,
                                  error_kfolds_norm = error_kfolds_norm)
            temp_df
    }
    final_df[, 2:3] <- final_df[, 2:3] / k_folds
    stopCluster(cl)
    return(final_df)
}

average_cv_error <- function(
    train_data,
    test_data,
    train_data_norm,
    test_data_norm,
    n_avg
){
    k_neighbors <- 20

    # set up error data frame
    error_df <- data.frame(k_neighbors = 1:k_neighbors,
                           error_kfolds = rep(0, k_neighbors),
                           error_kfolds_norm = rep(0, k_neighbors))

    # progress bar
    pb <- txtProgressBar(min = 0, max = n_avg, style = 3)

    # average cv-error over several runs to get more precise measure
    for(i in 1:n_avg){
        setTxtProgressBar(pb, i)
        error_df[-1] <- error_df[-1] + k_nearest_neighbors(train_data,
                                train_data_norm,
                                2,
                                k_neighbors)[-1]
    }
    close(pb)
    error_df[-1] <- error_df[-1]/n_avg
    ggplot1 <- ggplot(data = error_df, aes(x = k_neighbors)) +
        geom_line(aes(y = error_kfolds, colour = "$KNN$")) +
        geom_line(aes(y = error_kfolds_norm, colour = "$KNN normalized$")) +
        geom_vline(xintercept = 1, color = "black", linetype = "dotdash") +
        xlab("$k$ neighbors") +
        ylab("Miss.class. Error") +
        scale_colour_manual("Legend",
                            breaks = c("$KNN$", "$KNN normalized$"),
                            values = c("#91bfdb", "#fc8d59"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid", "solid"),
                                shape = c( 16, 16)
                            ))) +
        theme_bw() +
        theme(legend.position = c(0.8, 0.355),
              legend.background = element_rect(fill=alpha('white', 0)))
    ggplot_to_latex(ggplot1,
                    paste0("K_Nearest_Neighbors/Results_KNN/knn_error_
                        compare",n_avg),
                    width = 5, height = 5)
    return(error_df)
}

predict_on_test <- function(
    best_k,
    train_data,
    test_data
){
    knn_pred_class <- knn(train_data[-1],
                          test_data[-1],
                          train_data[, 1],
                          k = best_k)
    create_confusion_matrix(predicted_value = knn_pred_class,
                            true_value = test_data[, 1],
                            destination_path = "K_Nearest_Neighbors/Results_
                                KNN/")
}
#cv_final <- k_nearest_neighbors(train_data, 10)
train_data_norm <- train_data
train_data_norm[-1] <- train_data_norm[-1]/255
test_data_norm <- test_data
test_data_norm[-1] <- test_data_norm[-1]/255
#cv_norm <- k_nearest_neighbors(train_data, train_data_norm, 10)

average_error <- average_cv_error(train_data = train_data,
                  test_data = test_data,
                  train_data_norm = train_data_norm,
                  test_data_norm = test_data_norm,
                  n_avg = 30)
```

```

predict_on_test(1, train_data_norm, test_data_norm)
```

../R_scripts/K_Nearest_Neighbors/kknn.R

6.10 R-Code: Support Vector Machines

```r
## Libraries and seed
rm(list = ls())
library(e1071)          # library to make margins for svm
library(caret)          # useful library to split up data set
library(readr)

set.seed(420)           # seed to replicate results and get consistent test
    and training set

# Load help script with functions to export the results to latex
# These functions gathered to avoid duplicate code
if(!exists("create_confusion_matrix", mode = "function")){
    source("Help_Scripts/to_latex_functions.R")
}

#-------------------#

## Data
path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Support_Vector_Machines")   # getwd give
    path to project
# which is one folder over

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
    header = TRUE)
unclassified_data <- read.csv(paste0(path_data, "/Test_Digits_20171108.csv")
    , header = TRUE)

train_data[,1] <- as.factor(train_data[, 1])

nzr <- nearZeroVar(train_data[,-1], saveMetrics = TRUE, freqCut = 10000/1,
    uniqueCut = 1/7)
sum(nzr$zeroVar)

sum(nzr$nzv)

cut_variables <- rownames(nzr[nzr$nzv == TRUE, ])
variable <- setdiff(names(train_data), cut_variables)
train_data <- train_data[, variable]

# split data into test and training set
split_train_test <- createDataPartition(train_data$Digit, p = 0.8, list =
    FALSE)
test_data <- train_data[-split_train_test, ]
train_data <- train_data[split_train_test, ]

# remove label temporarely by storeing as it's own vector
digit <- train_data[1]
train_data$Digit <- NULL
train_data <- train_data/255
cov_train <- cov(train_data)

digit_test <- test_data[1]
test_data$Digit <- NULL
test_data <- test_data/255

# use prcomp to do PCA on the covariance matrix
```

```r
train_pc <- prcomp(cov_train)

# get information of about the variance
var_explained <- train_pc$sdev^2/sum(train_pc$sdev^2)
var_explained_cumsum <- cumsum(var_explained)
var_explained_df <- data.frame(number = 1:length(train_pc$sdev),
                               variance_explained = var_explained,
                               cumsum = var_explained_cumsum)
# see plot in end of script


# find optimal number of components
find_number_of_components <- function(
    components_range,
    increase_by
){
    components <- seq(from = components_range[1],
                      to = components_range[2],
                      by = increase_by)
    missclassification_error <- rep(0, length(components))
    cv_error <- rep(0, length(components))

    components_error_df <- data.frame(components,
                                      missclassification_error,
                                      cv_error)

    for(i in 1:length(components)){

        # traing the svm for the components given
        train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:
            components[i]]
        train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))

        tune_svm <- tune.svm(Digit ~ ., data = train_data_temp, cost = 1:10,
             kernal = "radial")
        fit_svm <- tune_svm$best.model

        # review the performance

        predicted <- predict(fit_svm, train_score, type = "response")
        missclassification_error <- 1 - mean(train_data_temp[, "Digit"] ==
            predicted)

        components_error_df[i, "cv_error"] <- tune_svm$best.performance
        components_error_df[i, "missclassification_error"] <-
            missclassification_error
    }
    return(components_error_df)
}

# make ggplot of cv error for different amount of components
# also marks selected number of components manually (22 in this task)
plot_components_error <- function(
    components_error_df
){
    components_chosen <- components_error_df[22, ]

    ggplot1 <- ggplot(data = components_error_df, aes(x = components)) +
        geom_line(aes(y = cv_error, colour = "$E_{CV}$")) +
        geom_point(aes(y = cv_error, colour = "$E_{CV}$")) +
        geom_vline(xintercept = 22, color = "black", linetype = "dotdash") +
        xlab("$n_{components}$") +
        ylab("Miss.class. Error") +
        geom_point(data = components_chosen,
                   aes(x = components, y = cv_error),
                   color = "red") +
        geom_text(data = components_chosen,
                  aes(y = cv_error,
                      label = paste0("$\\mathrm{CV}_{error}(22) = ",
                                     round(cv_error,4), "$")),
                  hjust = 0.2, vjust = -1.0) +
        scale_colour_manual("Legend",
                            breaks = c("$E_{CV}$"),
                            values = c("black"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid"),
                                shape = c( 16)
                            ))) +
        theme_bw() +
        theme(legend.position = c(0.8, 0.255),
              legend.background = element_rect(fill=alpha('white', 0)))
    ggplot_to_latex(ggplot1,
                    paste0(path_to_here,
                           "/Results_SVM/number_of_components_cv_error"),
                    width = 5, height = 5)
}

# Calculate best cost parameter for svm
find_optimal_cost_for_components <- function(
    selected_components,
    cost_range,
    cost_increase,
    n_avg
){
    train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:selected_
        components]
    train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))

    cost <- seq(from = cost_range[1], to = cost_range[2], by = cost_increase
        )

    cost_error <- data.frame(cost = cost, cv_error = rep(0, length(cost)))
    for(i in 1:n_avg){
        tune_svm <- tune.svm(Digit ~ ., data = train_data_temp, cost = cost,
             kernal = "radial")

        cv_error <- tune_svm$performances$error
        cost_error[, "cv_error"] <- cost_error[, "cv_error"] + cv_error

    }
    cost_error[, "cv_error"] <- cost_error[, "cv_error"]/n_avg

    best_cost <- cost_error[which(cost_error[, "cv_error"] == min(cost_error
        [, "cv_error"])),]

    ggplot1 <- ggplot(data = cost_error, aes(x = cost)) +
        geom_line(aes(y = cv_error, colour = "$E_{CV}$")) +
        geom_point(aes(y = cv_error, colour = "$E_{CV}$")) +
        geom_vline(xintercept = best_cost[, "cost"],
                   color = "black", linetype = "dotdash") +
        xlab("$n_{components}$") +
        ylab("Miss.class. Error") +
        geom_point(data = best_cost,
                   aes(x = cost, y = cv_error),
                   color = "red") +
        geom_text(data = best_cost,
                  aes(y = cv_error,
                      label = paste0("$\\mathrm{CV}_{error}(",
                                     cost,
                                     ") = ",
                                     round(cv_error,4), "$")),
                  hjust = -0.05, vjust = 0.9) +
        scale_colour_manual("Legend",
                            breaks = c("$E_{CV}$"),
                            values = c("black"),
                            guide = guide_legend(override.aes = list(
                                linetype = c("solid"),
                                shape = c( 16)
                            ))) +
        theme_bw() +
        theme(legend.position = c(0.8, 0.255),
              legend.background = element_rect(fill=alpha('white', 0)))
    ggplot_to_latex(ggplot1,
                    paste0(path_to_here,
                           "/Results_SVM/cost_cv_error"),
                    width = 5, height = 5)
    return(best_cost)
}

# Make prediction on the test set and make confusion matrix
predict_on_test <- function(
    selected_components,
    cost
){
    train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:selected_
        components]
    train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))

    test_score <- as.matrix(test_data) %*% train_pc$rotation[,1:selected_
        components]
    test_data_temp <- cbind(Digit = digit_test, as.data.frame(test_score))

    svm_predict <- svm(Digit ~ ., data = train_data_temp, cost = cost,
        kernal = "radial")

    predicted <- predict(svm_predict, test_data_temp)

    create_confusion_matrix(predicted_value = predicted,
                            true_value = test_data_temp[, "Digit"],
                            destination_path = paste0(path_to_here,
                                                      "/Results_SVM/
                                                          confusion_matrix")
                                                          )

}

# some help-plots for pca
```

```r
plot_variance_explained <- function(
    var_explained_df,
    chosen_number_components
){
    # plot amount of varibles and how much of the variance this describe
    ggplot1 <- ggplot(data = var_explained_df[1:100,], aes(x = number, y =
        cumsum)) +
        geom_line() +
        geom_point() +
        geom_vline(xintercept = chosen_number_components,
                   color = "black", linetype = "dotdash") +
        xlab("$n_{components}$") +
        ylab("Variance Explained") +
        theme_bw() +
        theme(legend.position = c(0.8, 0.255),
              legend.background = element_rect(fill=alpha('white', 0)))

    # plot the data as described by the two first principle components

    train_score <- as.matrix(train_data) %*% train_pc$rotation[,1:chosen_
        number_components]
    train_data_temp <- cbind(Digit = digit, as.data.frame(train_score))

    ggplot2 <- ggplot(data = train_data_temp, aes(x = PC1, y = PC2)) +
        geom_point(aes(colour = Digit)) +
        xlab("PCA 1") +
        ylab("PCA 2") +
        scale_colour_manual("Legend",
                            values = c('#8dd3c7','#ffffb3','#bebada','#
                                fb8072',
                                       '#80b1d3','#fdb462','#b3de69','#
                                           fccde5',
                                       '#d9d9d9','#bc80bd')
                            ) +
        theme_bw() +
        theme(legend.position = c(0.9545, 0.355),
              legend.background = element_rect(fill=alpha('white', 0)))

    ggplot_to_latex(ggplot1,
                    paste0(path_to_here,
                           "/Results_SVM/variance_explained_pca"),
                    width = 5, height = 5)

    ggplot_to_latex(ggplot2,
                    paste0(path_to_here,
                           "/Results_SVM/map_pca1_pca2"),
                    width = 5, height = 5)
}

main <- function(){
    # Find best number of components to use with svm
    components_range <- c(1, 30)
    increase_by <- 1
    components_error_df <- find_number_of_components(components_range =
        components_range,
                                                     increase_by = increase_
                                                         by)
    # Plot best cv error over number of components
    plot_components_error(components_error_df)


    selected_components <- 22

    # Find optimal cost parameter for the svm function
    best_cost <- find_optimal_cost_for_components(cost_range = c(0.5, 10),
                                                  cost_increase = 0.5,
                                                  selected_components =
                                                      selected_compnents,
                                                  n_avgs = 5)
    # Plot pca variance and to principle components
    plot_variance_explained(var_explained_df, selected_components)
}

main()
```

../R_scripts/Support_Vector_Machines/support_with_pca.R

6.11 R-Code: Visualize data examples

```r

## Data
path_data <- paste0(getwd(), "/data")
path_to_here <- paste0(getwd(), "/Support_Vector_Machines")   # getwd give
    path to project

train_data <- read.csv(paste0(path_data, "/Train_Digits_20171108.csv"),
    header = TRUE)

train_images <- train_data[, -1]

rotate <- function(x) t(apply(x, 2, rev))

# plot 5 x 8 images from the data set
png(paste0(path_data,"/images_digits.png"))
opar <- par(no.readonly = T)

par(mfrow = c(5, 8), mar = c(.1,.1,.1,.1))
for(i in 1:40){
    im_matrix <- matrix(train_images[i, ], nrow = 28, ncol = 28)
    im_matrix <- apply(im_matrix, c(1, 2), function(x) as.numeric(x))
    im_matrix <- rotate(im_matrix)
    image(im_matrix, col = grey.colors(255), axes = F)
}
par(opar)
dev.off()
```

../R_scripts/Help_Scripts/print_images.R