Digit Classification

November 27, 2017

STA5068Z Machine Learning
Comparing the abilities of different learning algorithms to classify handwritten digits
Halvor Reiten - RTNHAL001

Contents

1 Introduction

The aim of this project is to compare the performance of several different learning algorithms tasked with classifying a set of handwritten digits and determine if they are even or odd. The end goal is the deliver a set of predictions on a test set with unknown labels using the most promising machine learning techinque found in this problem.

2 Data

The data used to train and test the learning algorithms is downloaded from the online learning platform 'Vula'. The training and testings sets consist of 2500 unique, greyscale images of handwritten digits of size 28×28 px. Each row in the data sets represents an image of a handwritten digit. The first column is titled 'Digit' and represents the true label of the digit depicted. In the testing set, this label is masked with NA values. The remaining 784 columns contains a pixel value between 0 and 255, indicating its degree of lightness or darkness.

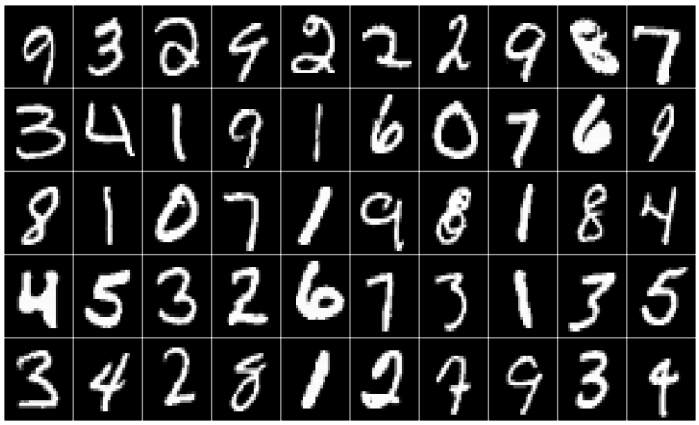


Figure 1: The first 50 handwritten digits plotted from the input data.

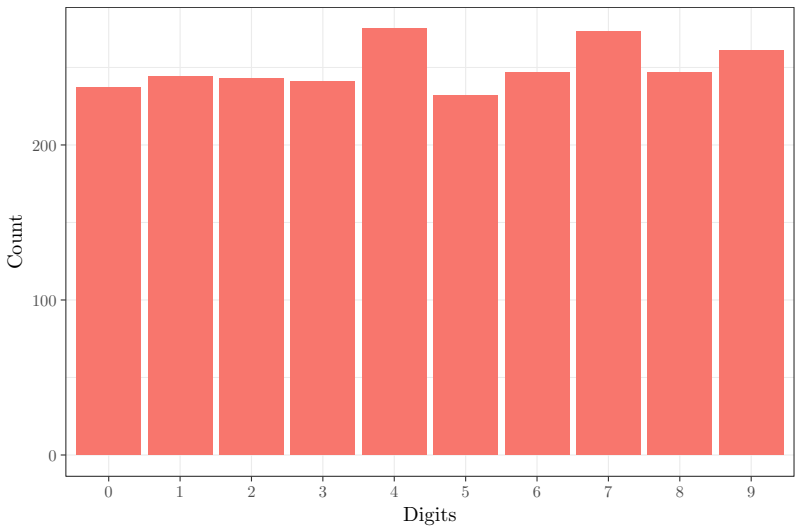


Figure 2: The distribution of the digits in the training data.

3 Method

The task is strictly to classify whether a digit is a even or odd, *not* to categorize seperate digits from one another. One approach is to treat this as a binary classification problem by first rewriting the digit labels into binary labels (even $\sim$ 0, odd $\sim$ 1), and then using learning techniques to learn whether the input is `TRUE` or `FALSE`. Another approach is to treat this as a multiclass classification problem, where each digit represents a class. One will then try and classify the digit as digits (from 0 to 9) and then afterwards determine whether the number is even or odd. Personally I find the latter approach more interesting. A model that precisely manages to classify individual digits, will also manage to precisely classify if the digit is even or odd. This is not the case the other way around. For this problem there is no guarantee that one approach is more accurate or better than the other, but I am more motivated and see more advantages in classifying individual digits. This project will therefore treat this problem as a multiclass problem. Each unique digit is treated as a class and sought classified.

The following machine learning methods will be implemented and compared in this project.

- **Tree-based Methods**
  - Classification trees
  - Random Forest
  - Bagging
  - Boosting
- **Nearest Neighbors**
  - K Nearest Neighbors
- **Support Vector Machines**
- **Neural Networks**
  - Artificial Neural Networks
  - Convolutional Neural Networks

Common for all of these methods is that they are all part of the learning paradigm *supervised learning*. No *unsupervised learning techniques* will be implemented in this project.

Each method will be evaluated in the following manner:

1. **Splitting data into training/testing sets**
   The training set will be divided into a training set (80%) and a test set (20%). The test set will *not* be used in the training process, and is only there to evaluate the finally chosen hypothesis. If required, 20% of the training data will be extracted to form a validation set.
2. **Finding the optimal parameters using validation**
   Some form of validation will be used to choose the optimal parameters for each model. Parameters can be regularization factors, learning rates, required complexities etc. The validation will most often be in the form of *10-fold Cross Validation*, *OOB error*, otherwise own validation sets. The model with the optimal parameters is chosen with respect to $E_{val}(h)$, and the model with the best parameters is denoted g^* for each method.
3. **Evaluating the best model on the test set**
   The best hypothesis, g^*, will then be evaluated on the test set and produce $E_{out}(g^*)$. The value of $E_{out}(g^*)$ is what will rank the different machine learning methods to each other. The lower $E_{out}(g^*)$, the better.

The best machine learning method will be chosen to predict the labels on the final testing set (with masked labels). Which one this is depends on not only $E_{out}(g^*)$, but also computational cost and memory requirements. The chosen method will then be trained on the *whole* training set (100% of the training data) with the optimal parameters found in the validation process) before predicting values on the testing set. The end predictions will be of binary form, i.e. even numbers ~ 0, odd numbers ~ 1.

The issue of *data snooping* has during the whole project been a consern. It must be stated that in all actions made that might affect the learned model, only the training and validation data is used. All data preprocessing, all feature selection, all validation, fitting etc. is exclusively based on the training and validation data. The testing set remains untouched and is only used in the final process when the best hypothesis is to be evaluated.

4 Implementing the learning algorithms

4.1 Classification Trees

4.1.1 Description

The most simple decision trees are single Classification and Regression Trees (CART). These are the building blocks of more complex decision trees (Bagged trees, Random Forests, Boosting etc.). They are popular for their ease of implementation and interpretation. The trees are grown using a top-down, greedy algorithm called *recursive binary splitting*. The algorithm splits a feature space into J distinct, non-overlapping regions. In classification problems the splits are made so that they maximize the reduction in impurity, i.e. the reduction in Gini Index.

4.1.2 Implementation

The dataset is split into a training set (80%) and a test set (20%). R's `tree`-package is utilized to grow the trees with the standard settings. The tree is first fully grown. The CV error for each tree is then plotted, and the tree is pruned down to where the error stops improving notably. The pruned tree will be evaluated on the test set.

4.1.3 Results

The error for an increasing number of terminal nodes is plotted in figure 3.

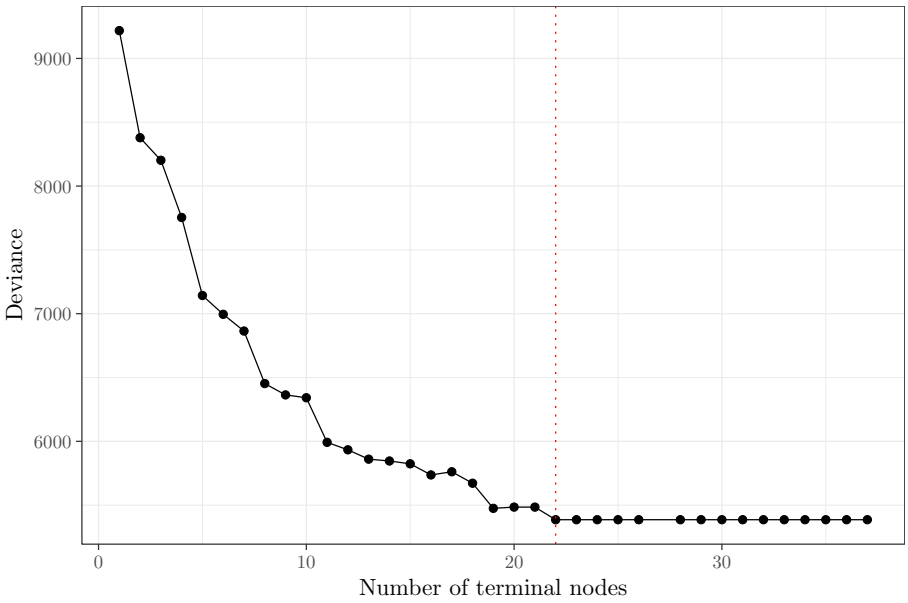
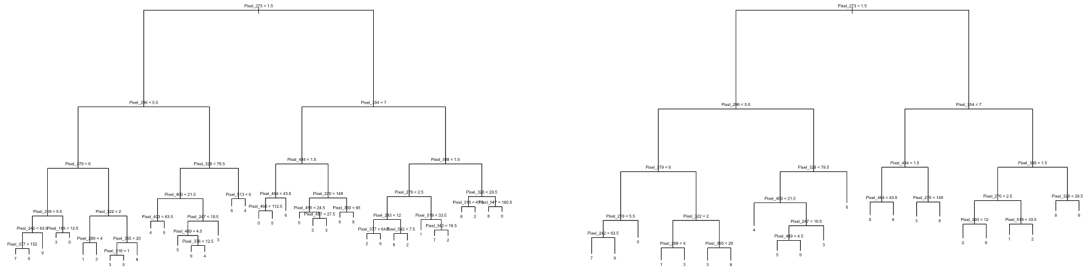


Figure 3: Deviance plotted for an increasing number of terminal nodes in the classifiaction tree.

We can see that the error stops improving from 22 terminal nodes and onwards. This point is marked with a red, dotted line. The increase in complexity is not worth the improvement in error. To avoid overfitting we prune the tree down to 22 terminal nodes from the originally 37 nodes. The pruned and unpruned tree can be viewed respectively in the figure 4.



(a) The unpruned classification tree

(b) The pruned classification tree

Figure 4: The unpruned classification tree in (a) with 37 terminal nodes, and the pruned classification tree in (b) with 22 terminal nodes.

The performance of the pruned classification tree is shown in the confusion matrix below.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	37	1	1	3	0	3	0	1	0	0	0.1956	9 / 46
1	0	38	2	0	1	1	2	0	1	0	0.1556	7 / 45
2	1	0	31	2	2	0	2	2	0	0	0.2250	9 / 40
3	1	2	6	33	1	27	1	3	5	2	0.5926	48 / 81
4	0	0	1	0	32	1	1	3	0	2	0.2000	8 / 40
5	0	0	1	5	3	7	0	1	0	1	0.6111	11 / 18
6	5	0	4	0	4	4	36	3	3	2	0.4098	25 / 61
7	1	2	4	1	1	1	0	42	0	3	0.2364	13 / 55
8	2	3	9	1	0	2	9	2	24	4	0.5714	32 / 56
9	0	0	0	4	14	0	0	4	1	35	0.3966	23 / 58
Totals	47	46	59	49	58	46	51	61	34	49	0.3700	185 / 500

Table 1: Confusion matrix of the classification tree.

Awful results from the classification tree, with a staggering 185 out of 500 misclassifications, giving a total accuracy of 63.0%.

4.2 Bagging and Random Forest

4.2.1 Description

One of the obvious disadvantages with classification trees is their high sampling variability and comparably poor predictive abilities. Bagging and Random Forest are two methods designed to improve accuracy and stability. By averaging a lot of trees grown on different samples of the training data, they can reduce the variance. Bagging, or *Bootstrap Aggregation*, will bootstrap B samples from the dataset, fit a tree for every sample (as described in section 4.1) and average the outcome (majority vote for classification trees).

By averaging the outcomes of many trees, Bagging will reduce the sampling variablity. However, if the bagged trees are highly correlated and produce similar outcomes, the improvement won't be substantial. This will be the case if the dataset has one or more dominant predictors that most trees consistently will use as a split-predictor. Random Forest decorrelates the trees by randomly sampling m predictors to be considered every time a tree is to be grown. As with Bagging, Random Forests are built by bootstrapping B samples from the dataset. For every sample, it will fit a tree, but only m predictors is used as split candidates for the trees. Notice that Bagging is the same as Random Forests for $m = p$, where p is the total number of predictors in the data.

4.2.2 Implementation

The dataset is split into a training set (80%) and a test set (20%). R's `randomForest`-package is utilized to grow the bagged trees and the random forest. The settings used are shown in table 2.

Method	`ntree`	`importance`	`mtry`
Random Forest	1000	`TRUE`	$\sqrt{p}$
Bagging	1000	`TRUE`	p

Table 2: The settings used to grow the bagged trees and random forests.

The OOB error for each tree is stored and plotted for both models. The OOB error is used as reference for validation, and by comparing the OOB error for the two models, one can confidently choose which of the models that's wise to proceed with. One will choose the model with the least OOB error, and proceed with this model only. The least complex model possible is desirable, so the model will be 'cut off' after whatever amount of trees where the error stops improving notably, calling this break point N^* trees. A new model is grown for the selected method with `ntree` $= N^*$, and this final model is evaluated on the test set.

4.2.3 Results

Figure 5 shows the OOB error development for the Random Forest and Bagging models.

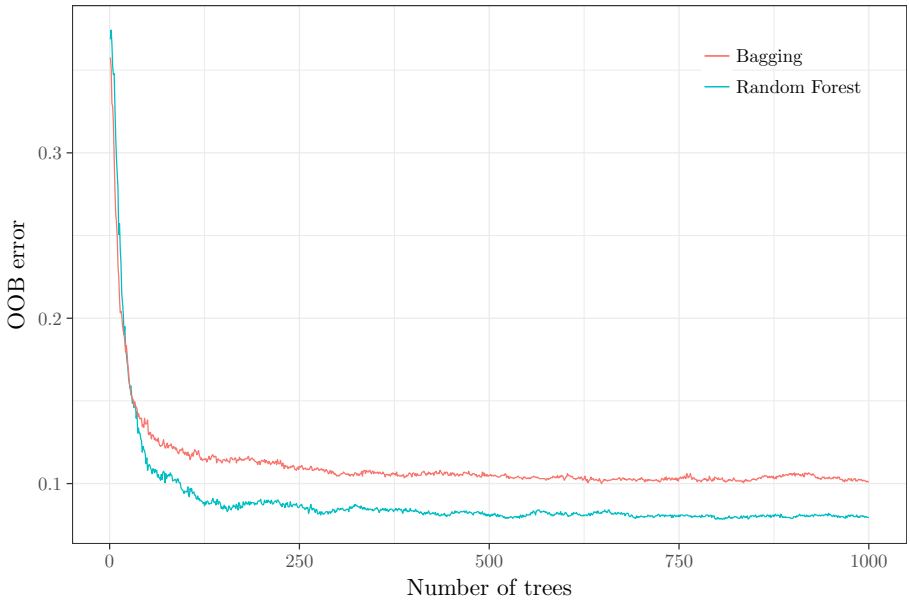


Figure 5: Random Forest compared to Bagging

The comparison of the two in figure 5 clearly shows that Random Forest is fitting a better model than the bagged model. We will therefore only use the Random Forest further in this analysis, stating that the bagged model produces a higher OOB error than the Random Forest.

We see that the OOB error stops improving after $N^* = 500$ trees, so we will grow a new Random Forest with `ntree` $= N^*$. This model is then evaluated on the test set. The confusion matrix for the Random Forest is shown in table 3.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	45	0	0	1	0	0	0	0	0	0	0.0217	1 / 46
1	0	44	1	0	1	0	0	0	1	0	0.0638	3 / 47
2	0	1	56	1	0	0	0	1	0	0	0.0508	3 / 59
3	0	0	0	43	0	0	0	0	0	0	0.0000	0 / 43
4	0	1	0	0	52	1	0	1	0	1	0.0714	4 / 56
5	0	0	0	1	0	45	0	0	0	0	0.0217	1 / 46
6	2	0	1	0	2	0	51	0	1	0	0.1053	6 / 57
7	0	0	0	1	1	0	0	57	0	2	0.0656	4 / 61
8	0	0	1	1	0	0	0	1	31	1	0.1143	4 / 35
9	0	0	0	1	2	0	0	1	1	45	0.1000	5 / 50
Totals	47	46	59	49	58	46	51	61	34	49	0.0620	31 / 500

Table 3: Confusion matrix of the Random Forest.

Quite good results obtained with the Random Forest, and a remarkable improvement from the single Classification Trees in section 4.1. The model managed to correctly classify 469 out of 500 trees in the test set, yielding a total accuracy of 93.8%.

4.3 Boosting

4.3.1 Description

Boosting is another ensemble method that aims to improve stability and predictive accuracy of a single tree by combining many of them. In Random Forests and Bagging the trees are grown independently of each other. This is not the case for Boosting, however, where the trees are grown using information from previously grown trees. More specifically, the residuals from the previously grown tree is used as the response variable when growing the next. The trees will only have a small number of splits each, denoted d. This yields a sequence of B trees, where each tree accounts for some variation i y that was not captured by the previous trees. The boosting method is a slow learner, and will in general require many trees to get good results. In the end, boosting will often beat methods like Random Forest, although it is comparably much more computationally expensive.

4.3.2 Implementation

The dataset is splitted into a training set (80%) and a testing set (20%). R's `gbm`-package will be used to grow the boosted trees with settings as shown in table 4.

Method	`n.trees`	`distribution`	`interaction.depth`	`cv.folds`	`shrinkage`
GBM	10000	Multinomial	2	5	0.01

Table 4: The settings used to grow the boosted trees.

There are a number of settings to tune using the gbm package. However, Boosting is a very computationally costly method, and it takes a long time to run the number of iterations required to achieve acceptable results with the boosting algorithm. This project will therefore keep to the settings as described in table 4 and only tune the number of trees used. The optimal number of trees, N^* will be found via cross validation, and the boosted model of N^* trees will be evaluated on the test set.

4.3.3 Results

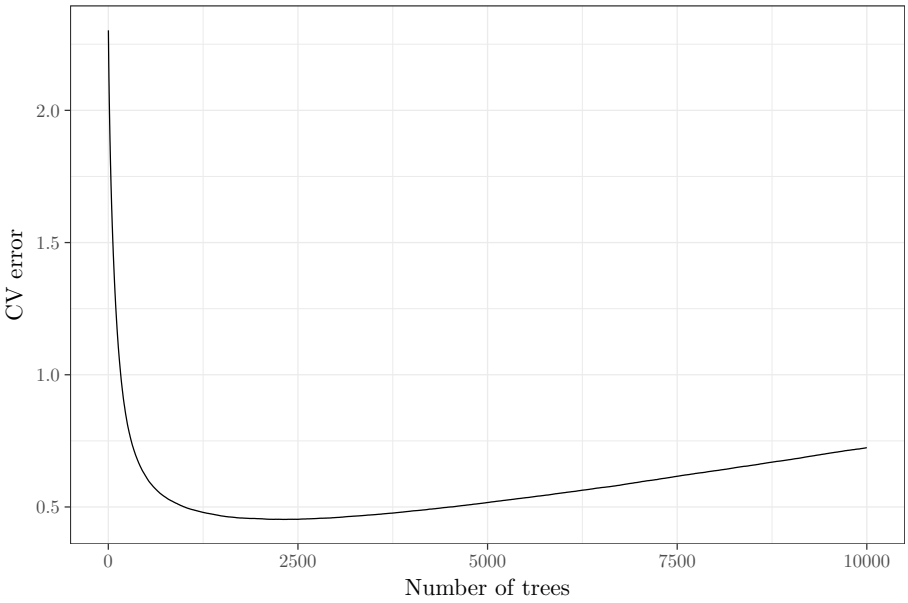


Figure 6: CV error per boosted tree.

We see that the CV error has a minimum at about $N^* = 2500$ trees.
We evaluate the boosted model after N^* trees on the test set. The results are shown in table 5.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	43	0	0	0	0	0	0	0	0	0	0.0000	0 / 43
1	0	45	0	0	1	0	0	0	1	0	0.0425	2 / 47
2	0	0	43	2	0	0	2	0	2	1	0.1400	7 / 50
3	0	0	0	52	0	2	0	0	1	0	0.0546	3 / 55
4	0	1	1	0	49	0	0	2	1	0	0.0926	5 / 54
5	0	0	0	6	0	39	0	0	2	0	0.1702	8 / 47
6	0	0	4	0	1	2	47	0	1	0	0.1454	8 / 55
7	0	0	1	0	0	1	0	50	0	1	0.0566	3 / 53
8	0	0	1	2	0	2	2	0	32	0	0.1795	7 / 39
9	0	0	0	1	4	1	0	2	2	47	0.1754	10 / 57
Totals	43	46	50	63	55	47	51	54	42	49	0.1060	53 / 500

Table 5: Confusion matrix of Boosting.

The boosted model managed to correctly classify 447 out of 500 digits, yielding an accuracy of 89.40%.

4.4 K Nearest Neighbors

4.4.1 Description

K Nearest Neighbors algorithms are very simple, yet effective and precice. In classification the KNN algorithm works by forming a majority vote between the K nearest neighbors. The to-be-classified observation will then be categorized to the same class as the other observations it is most *similar* to. Similarity is here defined as the difference in distance between two observations. *Distance* is a relative term and can be calcualted in numerous ways, but the standard is to use Euclidian distance, where $dist(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$.

To classify an unseen observation x, the KNN classifier will first calculate the distance d between x and every observation in the training data. It will extract the K training observations closest to x and calculate the conditional probability that the observation belongs to each class. The unseen observation x will then be assigned to whichever class having the largest conditional probability.

One can think of K as the complexity of the model in this case. A large value for K implies a very flexible fit with a possibly complex decision boundary. This will lead to low in-sample bias, but high variance. Choosing a higher K will average more observations in each prediction and therefore smoothen the decision boundary. The fit will be more resilient to outliers, leading to an increase in bias, but lowering the variance.

4.4.2 Implementation

The data is split into a training set (80%) and a test set (20%). Some data preprocessing is then done to try and improve the performance of the algorithm. The near zero variance predictors in the training set are found and removed from both the training set and the testing set. The data is then normalized, before PCA is applied to it. PCA will reduce the dimensionality and extract the most important features, making the algorithm faster and more precise. The `knn()`-method in R's `class`-package is used to execute the K nearest neighbors algorithm. The standard settings is used here, including the standard distance measure, Euclidian distance. The optimal value for K, K^*, is then found using 5-fold cross validation. Finally, the best model g^* is evaluated on the test set.

4.4.3 Results

PCA is applied to the data. The number of components to include is chosen on behalf of figure 7. By using the first 25 components, almost 97% of the variance in the data is explained.

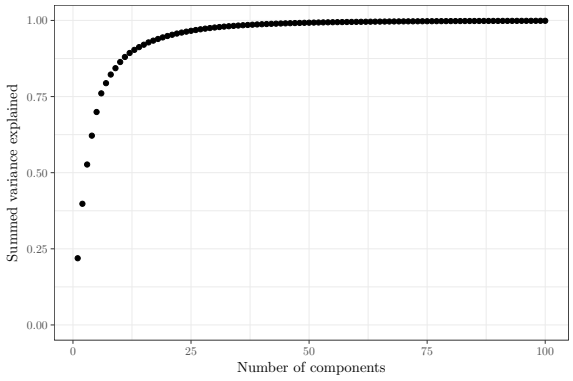


Figure 7: The summed variance explained per included principal component.

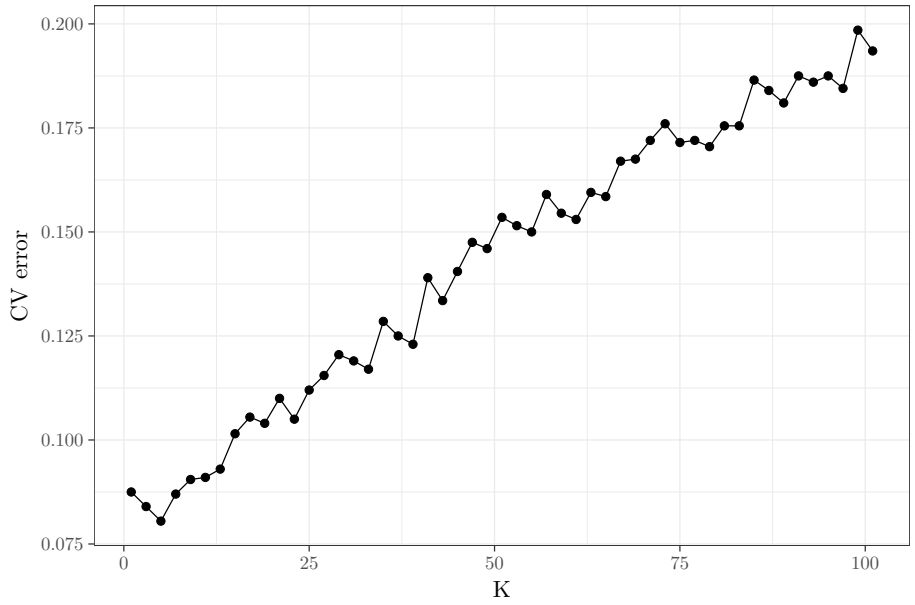


Figure 8: The expected 5-fold CV error for an increasing odd K.

Figure 8 shows the expected CV error as a function of K. Only odd values for K is considered. We see that the CV error in general increases as K increases, and that the K that returns the lowest CV error is $K^* = 5$.

A new model is now fitted with K^*, and is finally evaluated on the test set. The confusion matrix is shown in table 6.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	51	0	1	0	0	0	0	0	0	0	0.0192	1 / 52
1	0	67	1	0	0	0	0	2	0	0	0.0429	3 / 70
2	0	1	36	1	0	0	0	0	0	0	0.0526	2 / 38
3	0	1	0	38	0	1	0	0	1	0	0.0732	3 / 41
4	0	0	0	0	49	1	0	0	2	1	0.0755	4 / 53
5	0	0	0	1	1	36	0	1	2	0	0.1220	5 / 41
6	4	1	1	0	0	0	47	0	0	0	0.1132	6 / 53
7	0	0	0	0	0	0	0	59	0	2	0.0328	2 / 61
8	0	0	0	3	0	1	0	0	33	0	0.1081	4 / 37
9	0	0	0	1	2	0	0	1	4	46	0.1482	8 / 54
Totals	55	70	39	44	52	39	47	63	42	49	0.0760	38 / 500

Table 6: Confusion matrix of KNN.

Considering that this is an algorithm that requires essentially no training, these are very good results. The KNN algorithm with K = 5 manages to correctly classify 462 out of 500 digits, yielding a total accuracy of 92.40%.

4.5 Support Vector Machines

4.5.1 Description

Support Vector Machines is arguably the most successful classification method in machine learning. The algorithm is easy to grasp, it is highly accurate, and not to mension computationally efficient.

If the data is linearly seperable, there is more than one hyperplane that can seperate the data. The hyperplane with the biggest margin, i.e. the hyperplane with the greatest distance from the plane to the closest datapoints, has an advantage. SVM will fit the hyperplane that maximizes this margin. The datapoints that are 'supporting' the plane on the edges of the margin are called support vectors. By having this 'safety cushion' between the plane and the support vectors, SVM is more robust to noise, helping with the issue of overfitting and improving the out-of-sample performance.

In cases where the data is not linearly sepearble in some dimension, one can use non-linear transforms to transform the data to a higher dimension where the data is linearly sepearble. SVM work seamlessly with *kernels* that efficiently can perform these high dimensional non-linear transforms. The seperating hyperplane is then fitted in this high-dimensional space, before mapping it back to the original space.

Shortly summarized, SVM is combining the *maximized margin* and the *coordination with kernels*, resulting in a very powerful, nonlinear learning model with built-in regularization.

4.5.2 Implementation

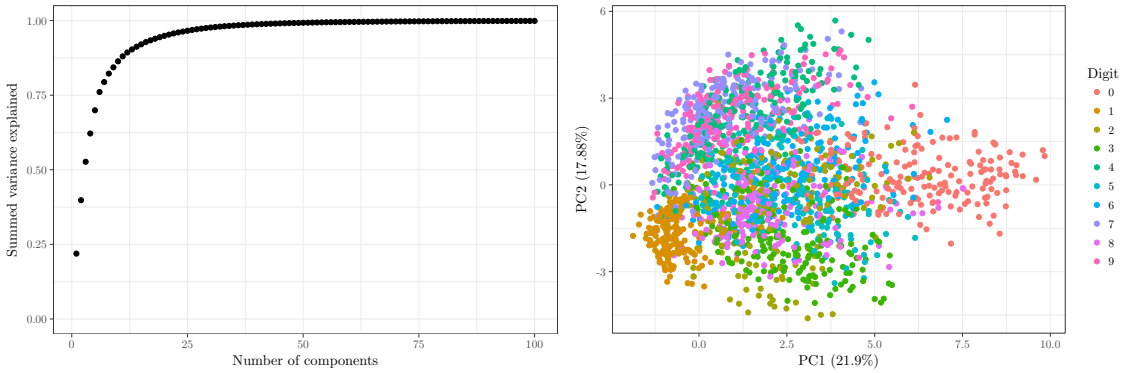
The dataset is first split into a training set (80%) and a testing set (20%). Some simple preprocessing of the data is then done. Many of the predictors have few non-zero values, and will therefore have a variance close to zero. The near-zero-variance predictors are found in the training set, and these predictors are removed from both the training set and the testing set. The data is then normalized, before PCA is applied to it. By applying PCA to the data one can extract the most important features and select the N most important to work with, significantly decreasing the dimensionality of the problem.

The SVM model is implemented using the `svm`-function of package `e1071` in R. The *Radial Basis Function kernel* is used, and all other settings used are the standard settings. The optimal number of principal components to include in the model is then found by cross validation, analyzing the error development for an increasing number of included components in the model. After choosing N, one can again use cross validation to find the optimal regularization parameter C.

The model with the optimal parameters, using N principal components and the optimal cost parameter C, is then evaluated on the testing set.

4.5.3 Results

Feature extraction is done by applying PCA. Figure 9 visualizes some data from the PCA.



(a) 100 principal components and variance explained

(b) The digits described by two PCs.

Figure 9: The top 100 PC and variance explained in (a) and the digits described by two PCs in (b).

By reviewing figure 9a one can clearly see that the first components in the PCA are the most important ones. The cumulative sum of the variance explained for each component converges to 1, but reaches 95% after only about 20-25 components. That means that we can compactly describe a great majority of the data using very few, important components (found by applying PCA). Figure 9b visualizes the digits described by the first two (and most important) principal components. One can clearly see some patterns in the classes (1's in the far left corner, 0's in the far right), but it is far from perfect. With more dimensions (including more PCA components) we will be able to better and better seperate the data, although harder (or impossible) to visualize.

Now let's find the optimal number of components to include in our model. Figure 10 shows the expected 10-fold CV error development for an increasing number of included components.

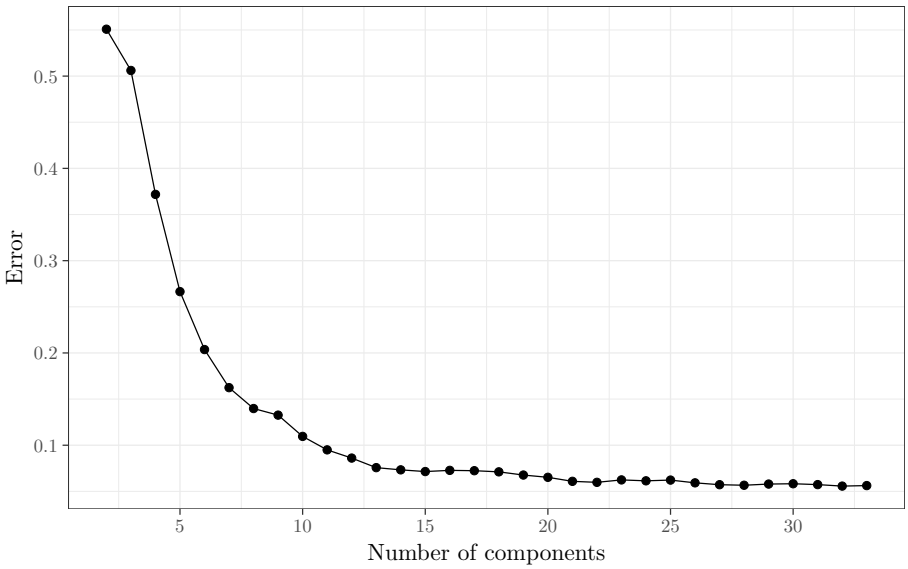


Figure 10: 10-fold CV error development for an increasing number of included components.

Figure 10 shows that the improvement in CV error is minimal to none from 25 components and up. Figure 9a shows us that by choosing to represent the data by 25 components, we can describe

almost 97% of the variance in our data - pretty good! The graph also show us that the increase in variance explained is minimal for components after the 25th. Therefore there is little to gain in both reduction of error and data loss by choosing more than 25 components. Based on this we will choose to work with $N = 25$ principal components in our model.

The optimal value of the regularization parameter C is then found using 10-fold cross validation.

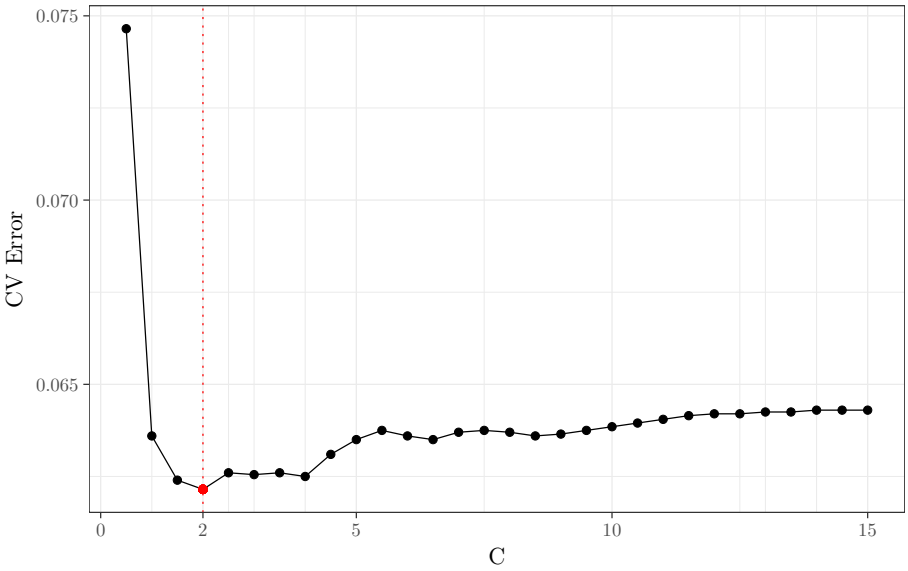


Figure 11: Error development for an increasing regularization parameter C.

Figure 11 shows that the optimal parameter of C is 2.0. Using the optimal number of components, $N = 25$, and the optimal regularization parameter, $C = 2.0$, we now fit a new model. This model is evaluated on the test set, and the confusion matrix is shown in table 7

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	46	0	0	0	0	1	0	0	0	0	0.0213	1 / 47
1	0	44	0	0	1	0	0	0	0	0	0.0222	1 / 45
2	0	1	57	0	0	0	0	1	0	0	0.0339	2 / 59
3	0	0	1	44	0	1	0	1	2	0	0.1020	5 / 49
4	0	1	0	0	54	0	0	0	0	1	0.0357	2 / 56
5	0	0	0	3	0	43	0	0	0	0	0.0652	3 / 46
6	1	0	0	0	2	0	51	0	0	0	0.0556	3 / 54
7	0	0	0	1	0	0	0	59	0	2	0.0484	3 / 62
8	0	0	1	1	0	1	0	0	31	1	0.1143	4 / 35
9	0	0	0	0	1	0	0	0	1	45	0.0425	2 / 47
Totals	47	46	59	49	58	46	51	61	34	49	0.0520	26 / 500

Table 7: Confusion matrix of the SVM.

Very good results from the SVM, correctly classifying 474 out of 500 digits, yielding an accuracy of 94.80%,

4.6 Artificial Neural Networks

4.6.1 Description

A neural network is a biologically inspired learning model, aiming to mimic (some) of the behaviour of the neurons in the brain. The neural network consist of neurons organized in layers, including an input layer, an output layer and with one or more hidden layers in between. Each of the neurons in the hidden layers are fully connected to the neurons in the previous and the next by synapses ('weights'). The neurons work by receiving an input signal s, non-linearly transforming s using an activation function, and finally returning an output x which it passes on to the next layer of neurons. As a system, these neurons can efficiently approximate very complex target functions.

All a neural network does is to transform the input data in a particular way and predict an output. The way the transformation is done is dependent on the neurons and the weights that connects them. The learning process is therefore about altering the weights so that they can minimize the output error. The optimized weights are found by minimizing a cost function, the error function, and Stochastic Gradient Descent (SGD) and the backpropogation are the most widely used algorithms for doing just this.

4.6.2 Implementation

R's `h2o`-package is utilized to make the neural network. 10-fold cross validation is used on the training set when fitting a model to evaluate its performance. The `grid()`-function is then used to run the neural network with different combination of parameters, making it easy to compare different models to each other, specifically comparing their performance by looking at the CV error that the respective model produce. The parameters that's used to make the grid is shown in table 8.

Parameter	Values
Number of Hidden Layers	1, 2
Number of Neurons in Hidden Layers	(100), (200), (400), (300, 300), (400, 200), (500,100)
Learning Rate	0.1, 0.01, 0.001
Number of Epochs	100
Number of Folds (CV)	10
Activation function	$Tanh()$
Stopping Metric	Misclassification
Stopping Tolerance	0.0001
Stopping Rounds	2
L1 Regularisation	10^{-5}
Input Dropout Ratio	0.20

Table 8: Parameters of different ANNs in the grid

Both input layer dropout and L1 regularization can add stability and improve generalization. They help with the issue of overfitting, and are therefore included in the parameters of the models.

Three different learning ratios are used in this grid, one high (0.1), one medium (0.01) and one small (0.001). Higher learning rate means less stable fitting, in contrast to the lower learning rate that makes the model converge slower.

Stopping metrics are used to ensure that the model stops running as soon as the improvement is negligble. The stopping metric is in this project is the misclassification rate (i.e. the number of falsely predicted observations), the stopping threshold is 0.0001 and the stopping rounds defined as

2. This means that the model will stop iterating if the misclassification rate hasn't improved by 0.01% over 2 iterations.

Many combinations of hidden layers and number of neurons are being tried out in the grid. It is interesting to see the impact these parameters have on the model's performance. The maximum number of layers have been set to 2 and the maximum number of neurons in a layer to 500. A rule of thumb is that the number of neurons in the hidden layers should be $\frac{2}{3}*$ input-layer-size + output-layer-size. As the size of the input layer is 784 neurons, this means that a good indication of the number of neurons in the hidden layers is approximately 500.

The number of epochs (or iterations) has been set to 100, but due to early stopping the models won't reach that many epochs due to the stopping metrics. The activation function used is $tanh()$, an activation function that's well suited for a classification problem like this one.

These parameters make up a total of 18 unique models. The model that in total generates the least CV-error (mse) is the model that will be chosen as the best model. The best model will further be evaluated by testing its performance on the test set.

4.6.3 Results

The result of the gridsearch is shown in table 9.

	epochs	hidden	rate	stopping_tolerance	mse
1	12.38	[400, 200]	0.01	1.0E-4	0.09072
2	11.78	[400, 200]	0.001	1.0E-4	0.09492
3	13.93	[300, 300]	0.01	1.0E-4	0.09556
4	10.06	[500, 100]	0.001	1.0E-4	0.09624
5	11.62	[400, 200]	0.1	1.0E-4	0.09751
6	9.29	[500, 100]	0.01	1.0E-4	0.09764
7	9.81	[500, 100]	0.1	1.0E-4	0.09765
8	14.27	[300, 300]	0.001	1.0E-4	0.09837
9	14.09	[300, 300]	0.1	1.0E-4	0.09972
10	24.55	[100]	0.01	1.0E-4	0.10479
11	21.47	[100]	0.001	1.0E-4	0.10593
12	14.25	[200]	0.1	1.0E-4	0.10598
13	17.63	[200]	0.01	1.0E-4	0.10709
14	20.91	[100]	0.1	1.0E-4	0.10860
15	13.29	[400]	0.1	1.0E-4	0.10869
16	13.32	[400]	0.01	1.0E-4	0.10953
17	15.23	[200]	0.001	1.0E-4	0.10985
18	13.30	[400]	0.001	1.0E-4	0.11015

Table 9: The results from the gridsearch run with parameters as given in table 8.

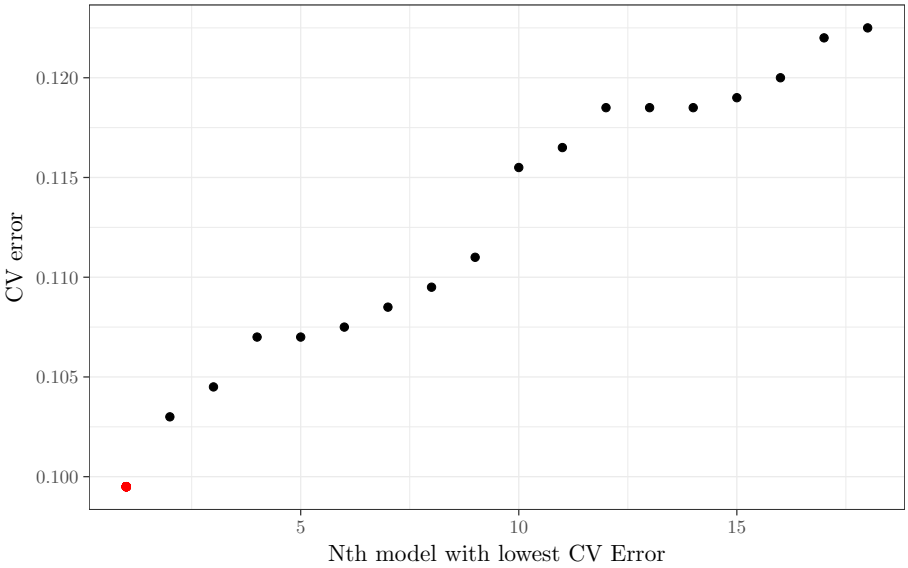


Figure 12: CV error for each of the models tried in the grid.

The parameters of the best model (the model generating the least MSE) are:

- Number of Hidden Layers: 2
- Number of Neurons in Hidden Layers: (400, 200)
- Number of Epochs: 12
- Learning Rate: 0.01

One can see from the grid results in table 9 that all of the models with 2 layers did better than the 1-layer networks. In addition, the 'medium' sized learning rate of 0.01 was beneficial in this case. This model is now evaluated on the testing set. The confusion matrix is shown in the table below.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	44	0	0	0	0	1	1	1	0	0	0.0638	3 / 47
1	0	44	1	0	0	0	1	0	0	0	0.0435	2 / 46
2	0	1	51	2	0	0	1	1	2	1	0.1356	8 / 59
3	0	0	1	45	0	0	1	2	0	0	0.0816	4 / 49
4	0	1	1	0	53	0	2	0	0	1	0.0862	5 / 58
5	0	0	0	5	1	40	0	0	0	0	0.1304	6 / 46
6	0	0	0	0	2	0	49	0	0	0	0.0392	2 / 51
7	0	0	1	0	2	0	0	55	0	3	0.0984	6 / 61
8	0	1	2	0	0	1	0	0	27	3	0.2059	7 / 34
9	0	0	0	0	0	0	0	3	1	45	0.0816	4 / 49
Totals	44	47	57	52	58	42	55	62	30	53	0.0940	47 / 500

Table 10: Confusion matrix of ANN.

The best neural network manages to correctly classify 453 out of 500 observations, yielding a total accuracy of 90.6%. This might indicate that we have too sparse amount of data for the ANN to perform well.

4.7 Convolutional Neural Networks

4.7.1 Description

Convolutional Neural Networks (CNN) is a subcategory of Neural Networks that has proven very effective in the area of image recognition and computer vision. A CNN will assume that the input is an image, and the task of the CNN is to classify what kind of image this is. It will do this by finding, matching and comparing certain features found in the image. One can imagine the case where the CNN is set to classify a set of handwritten digits. There are 10 seperate classes, one for every digit, and each class has a set of features that are typical for that class. The better the features of one class corresponds with the features of the to-be-classified image, the more likely it is that the image belongs to that class. Very simplified, this is what happens in a Convolutional Neural Network.

The network will compare the images to each other *piece by piece*. It doesn't automatically know where the features will be found in an image, so it tries to match the features in every position possible. This process, where the network is convoluting through all the "mini-images" and trying to match for features, is called 'convolution'. The CNN consists of three types of layers: Convolutional layers, Pooling layers and Fully Connected layers. The convolutional layers will convolute through the input, calculate the feature matchings and map a filtered version of the input. The pooling layers will mainly compress the input without little loss of important information. Fully Connected are of the more traditional sort, and is most closely described as a hidden layer as you find in ANNs. The fully connected layers will convert the transformed signal from the convolutional and pooling layers to a probability.

4.7.2 Implementation

The package `mxnet` is used to implement the CNN. The data is split into a training set (64%), validation set (16%) and a test set (20%). The network will be of a LeNet architecture, made up of two convolutional layers, two pooling layers, and lastly two fully connected layers. The first convolutional layer takes a 28×28 image as input, and will filter the input using 20 kernels of size 5×5. The first pooling layer will downsize the output of the first convolutional layer by applying 2×2 max pooling. The second convolutional layer will filter the output from the pooling layer with 50 kernels of size 5×5. Another pooling layer will again apply 2×2 max pooling to the output of the convolutional layer, downscaling the data. The fully connected layers will have 500 and 10 neurons, respectively. The hyperbolic tangent function is used at the outputs of convolutional and fully connected layers as the activation function, with exception of the last fully connected layer. The output is there passed through a 10-way softmax, which is convenient as the output then directly can be interpreted as a probability bounded between 0 and 1.

The function `mx.model.FeedForward.create()` is used to 'run' the CNN. The optimal hyperparameters of the CNN will be found by a gridsearch. The CNN will be run with the following parameters

Learning Rate	Array Batch Size	Epochs	Momentum	Optimizer
(0.01, 0.001, 0.005)	(50, 100)	40	0.90	'SGD'

Table 11: The parameters used in the CNN gridsearch.

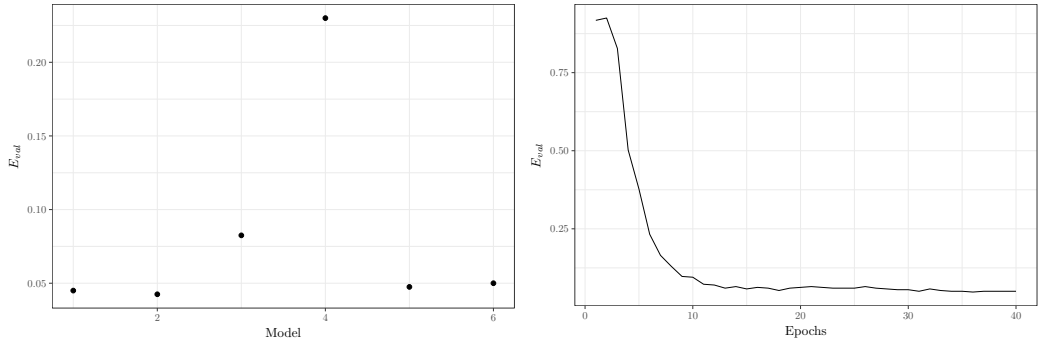
E_{val} is recorded for each model, and the model with the parameters generating the least validation error will be chosen as the best model. A new model with these parameters will now be trained on the whole set (training set + validation set) before it is evaluated on the testing set.

4.7.3 Results

The results from the hyperparameter grid search is shown below.

id	rate	array_size	e_val
1	0.010	50	0.0450
2	0.010	100	0.0425
3	0.001	50	0.0825
4	0.001	100	0.2300
5	0.005	50	0.0475
6	0.005	100	0.0500

Table 12: The results from the gridsearch with CNN.



(a) Validation error per model in the gridsearch.

(b) E_{val} per epoch for the best model.

Figure 13: Results from the gridsearch.

We can see that the model with parameters `learning rate` = 0.01 and `array batch size` = 100, generated the least E_{val}. The evolution of the validation error E_{val} is plotted in figure 13b for the model with these parameters.

The validation data and training data is now merged into one, and a new model with the best parameters is trained on the merged data. The newly trained model is now tested on the test set. The confusion matrix of the best CNN is shown in table 13.

	0	1	2	3	4	5	6	7	8	9	Error	Rate
0	47	0	1	0	0	0	0	0	0	0	0.0208	1 / 48
1	0	43	1	0	1	0	0	0	0	0	0.0444	2 / 45
2	0	1	55	1	0	0	0	1	1	0	0.0678	4 / 59
3	0	0	0	47	0	0	0	0	0	0	0.0000	0 / 47
4	0	1	0	0	54	0	0	0	0	2	0.0526	3 / 57
5	0	0	0	0	0	46	0	0	0	0	0.0000	0 / 46
6	0	0	1	0	3	0	51	0	0	0	0.0727	4 / 55
7	0	0	1	1	0	0	0	59	0	2	0.0635	4 / 63
8	0	1	0	0	0	0	0	0	33	1	0.0571	2 / 35
9	0	0	0	0	0	0	0	1	0	44	0.0222	1 / 45
Totals	47	46	59	49	58	46	51	61	34	49	0.0420	21 / 500

Table 13: Confusion matrix of CNN.

A precision of 95.80% makes this the most accurate algorithm of all attempted methods. Very impressive results.

5 Discussion

5.1 Comparison of the learning methods

Method	$E_{out}(g^*)$	Rate
Classification Trees	0.3700	185 / 500
Random Forest	0.0620	31 / 500
Boosting	0.1060	53 / 500
K Nearest Neighbors	0.0760	38 / 500
Support Vector Machines	0.0520	26 / 500
Artificial Neural Networks	0.0940	47 / 500
Convolutional Neural Networks	0.0420	21 / 500

Table 14: Summary of the methods and the produced E_{out} for each of them.

The best of the tree based methods was by far the Random Forest, with an accuracy of 93.8%. The single classification trees and the boosted trees didn't even get close the results achieved with the Random Forest. In addition, the boosting process was a lot more computationally costly than any of the other tree based methods. Based on performance as a whole, boosting and single classification trees are therefore not recommended algorithms for this problem.

The K Nearest Neighbors model did achieve impressive accuracy in spite of being perhaps the simplest of the supervised learning techniques implemented in this project. The required 'training' time is essentially zero, and the results are respectable. The Artificial Neural Network did not perform as well as initially expected. It is beat by most other methods precision wise, and is a computationally heavy system to train. For this problem, with the sparse amount of data available, it is therefore discouraged to use Artificial Neural Networks before most other supervised learning techniques. It is believed that the ANN is in need of more data to achieve any good results in this matter.

Although KNN and Random Forests granted good results, they couldn't quite reach up to the performance of the two superior methods in this classification problem - Support Vector Machines and Convolutional Neural Networks. An out-of-sample accuracy of 94.8% makes the Support Vector Machines the second best of all methods attempted. Combining computational efficiency and high accuracy makes SVM one of the most attractive methods for this problem. There was, however, some work required in data preprocessing and feature extraction to achieve the high results in this problem.

The best model of all was the Convolutional Neural Network. With little to no data preprocessing needed, the CNN is particularly easy to implement and use. It is very efficient to train and extremely accurate. The out-of-sample error of 4.20 % makes the CNN the most precice of all the methods attempted. This proves that the CNN is particularly well suited for image recognition problem like this.

5.2 Final predictions and expected performance

As the highest performing method, the Convolutional Neural Network is chosen as the model to produce the final predictions for the yet unseen testing set with masked labels. The network is trained with the same parameters as was found by validation in section 4.7. The *whole* training set is now used for training the model, i.e. the available data with known labels is not split into a training set and a test set as before. The predictions are saved to a seperate csv file with 3

columns: `row_id`, `pred_digit` and `pred_isUneven`. `row_id` is just the row index of the image being classified in the test set, `pred_digit` is the classified digit and `pred_isUneven` is a boolean stating whether the digit is an uneven (`isUneven` = 1) or an even number (`isUneven` = 0).

When it comes to commenting the expected performance of the model, it is important to remember that the already found $E_{out}(g^*)$ is a biased estimate of the true E_{out}, and that $\mathbb{E}[E_{out}(g^*)] \neq E_{out}(g^*)$. $E_{out}(g^*)$ is dependent on the data it was trained and tested on, and the results could have easily been shifted in one direction or the other based on how the training set and the test sets were sampled. Still, it is reasonable to believe that the true value of $E_{out}(g^*)$ is close to the already calculated $E_{out}(g^*)$. The assignemnt is to classify whether a digit is even or odd, *not* to pinpoint what digit it actually is. It is therefore also reasonable to presume that the the model will in some cases 'get lucky', i.e. it classifies the wrong digit, but it turns out to correctly classify whether it is uneven or even. For example the true label is '1' but the model is predicting is to be '7': The model has predicted the wrong digit, but will correctly classify it as 'uneven'. This will be an advantage for the model, and might actually make the model perform better than the estimated out-of-sample error already calculated.

6 Conclusion

Eight different machine learning methods were implemented when trying to find the best model for classifying handwritten digits. Their respective performance was assessed by the out-of-sample error that the best hypothesis generated, $E_{out}(g^*)$. The model with the best overall performance was the Convolutional Neural Network, being particularly precise and easy to implement. $E_{out}(g^*)$ for this model was calculated to 0.0420, yielding an accuracy of 95.80%. Second best was the Support Vector Machine, with an accuracy of 94.80%.

Appendices

A R-code

A.1 Help functions

```
require(caret)
require(ggplot2)
require(tikzDevice)
require(xtable)


# ====================== #
# GENERAL HELP FUNCTIONS #
# ====================== #

makeConfusionMatrix <- function(pred, true){
  mod <- confusionMatrix(pred,true)
  confM <- mod$table
  confM <- rbind(confM, Totals = colSums(confM))
  sums <- rowSums(confM)

  confM <- as.data.frame.matrix(confM)
  confM <- cbind(confM, Error = 0, Rate = NA)
  correct_vec = numeric(nrow(confM))
  wrong_vec = numeric(nrow(confM))

  for (row in 1:(nrow(confM)-1)){
    correct <- as.numeric(confM[row,row])
    wrong <- sums[row] - correct
    correct_vec[row] = correct
    wrong_vec[row] = wrong

    confM$Error[row] = round(wrong/(correct+wrong),5)
    confM$Rate[row] = paste0(wrong, " / ", (correct+wrong))
  }
  confM$Error[nrow(confM)] <- sum(wrong_vec) / length(pred)
  confM$Rate[nrow(confM)] <- paste0(sum(wrong_vec), " / ", length(pred))

  return(confM)
}


# will export a table or a data frame to tex format
# must be in the format of the confusion matrix above.
exportTableToLatex <- function(table, folder_path, name){
  complete_filepath = paste0(folder_path, name)
  print(xtable(table,
               display = c("d", "d", "d", "d", "d", "d", "d", "d", "d", "d", "
                   d", "f", "s"),
               digits = c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0)),
      file = complete_filepath,
      only.contents = T,
      include.rownames = T)
}
```

```
49
50
51 # will export a ggplot to tex format
52 exportPlotToLatex <- function(plot, folder_path, name){
53   complete_filepath = paste0(folder_path, name)
54
55   tikz(file = complete_filepath, width = 6, height = 4)
56     print(plot)
57   dev.off()
58 }
59
60
61 # ================== #
62 # PCA HELP FUNCTIONS #
63 # ================== #
64
65 # will return the near zero variance predictors of 'data'
66 getNZVPredictors <- function(data){
67   zeroVarPred <- nearZeroVar(data[,-1], freqCut=10000/1, saveMetrics = TRUE,
         unique = 1/7)
68   return(cutPredictors <- rownames(zeroVarPred[zeroVarPred$nzv == TRUE,]))
69 }
70
71 # will remove cutPredictors from 'data' and normalize it
72 getScaledAndTrimmedData <- function(data, cutPredictors){
73
74   data <- data[,-which(colnames(data) %in% cutPredictors)]
75   data[,-1] <- data[,-1]/255
76
77   return(data)
78 }
79
80 # returns PCA of the data input
81 applyPCA <- function(scaledTrimmedData){
82
83   # apply PCA
84   data.cov <- cov(scaledTrimmedData[,-1])
85   pca <- prcomp(data.cov)
86
87   return(pca)
88 }
89
90 # will return the data input represented with principal components
91 getDataAsPCA <- function(scaledData, pca, numComponents){
92   labels = scaledData$Digit
93   data <- scaledData[,-1] # remove response variables
94
95   score <- as.matrix(data) %*% pca$rotation[,1:numComponents]
96   dataAsPCA <- cbind(Digit = labels, as.data.frame(score))
97   return(dataAsPCA)
98 }
```

r_files/HelpFunctions.R

A.2 Classification Trees

```
rm(list=ls())
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(caret) # used to make a confusion matrix
library(tikzDevice) # used to compile ggplots in latex
library(xtable) # used to compile tables and dataframes in latex
library(tree)

source("HelpFunctions.R")
exportspath <- "../exports/tree_based_methods/classificationtrees/"

# read in data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

# divide into training and testing sets
sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]

# save the pruned tree to file
fit <- tree(Digit ~ ., data = train, mindev = 1/200)
png(paste0(exportspath, "classtree_unpruned.png"), width = 1000, height =
    600)
  plot(fit); text(fit, cex = 0.6)
dev.off()

cv.fit <- cv.tree(fit)
df <- data.frame(nodes = cv.fit$size, err = cv.fit$dev)

# assign the best node where the tree should be splitted from
best = 22

# plot the deviance per tree
pl <- ggplot(df) +
  geom_line(aes(x = nodes, y = err)) +
  geom_point(aes(x = nodes, y = err)) +
  geom_vline(xintercept = best, linetype = "dotted", col = "red") +
  theme_bw() +
  xlab("Number of terminal nodes") + ylab("Deviance")
exportPlotToLatex(pl, exportspath, "classtree_errorPerTree.tex")


# Will prune the tree down to 'best' trees
cv.fit$k[1] <- 0
alpha <- round(cv.fit$k)
fit.pruned <- prune.tree(fit, best = best)

# save the pruned tree to file
png(paste0(exportspath, "classtree_pruned.png"), width = 1000, height = 600)
  plot(fit.pruned); text(fit.pruned, cex = 0.6)
dev.off()
```

```
54
55
56 # evaluate values on test set
57 pred <- predict(fit.pruned, newdata = test, type="class")
58 pred_df <- data.frame(pred = pred, true = test$Digit)
59
60 confM <- makeConfusionMatrix(pred,test$Digit)
61 exportTableToLatex(confM, exportspath, "classtree_confusionmatrix.tex")
```

r_files/ClassificationTrees.R

A.3 Bagging

```r
set.seed(1000)

library(ggplot2)
library(randomForest)
library(tikzDevice)
library(caret)
library(xtable)

source("HelpFunctions.R")
exportspath <- "../exports/tree_based_methods/bagging/"

# read in data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)
sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]

ntree = 1000

## fit bagging model ##
bag <- randomForest(Digit ~ ., data = train,
                    ntree = ntree,
                    mtry = (ncol(train) - 1),
                    importance=TRUE,
                    na.action = na.exclude,
                    do.trace = floor(ntree/10)
                    )

# plot the OOB error evolution
err_df <- data.frame(trees = 1:length(bag$err.rate[,"OOB"]), err = bag$err.
    rate[,"OOB"])
pl <- ggplot(err_df) +
  geom_line(aes(x = trees, y = err)) +
  xlab("Number of trees") + ylab("OOB error") + ggtitle("") +
  theme_bw()
exportPlotToLatex(pl, exportspath, "bag_oob_err.tex")

# predict values on test set
pred <- predict(bag, newdata = test)
pred_df <- data.frame(pred = pred, true = test$Digit)

# write confusion matrix to tex file
confM <- makeConfusionMatrix(pred,test$Digit)
exportTableToLatex(confM, exportspath, "bag_confusionmatrix.tex")

# save data to .csv-files
write.table(err_df, file = paste(exportspath, "bag_oob_error.csv", sep=""),
    row.names=F, col.names=T, sep = ",", append=F)
write.table(pred_df, file = paste(exportspath, "bag_predictions.csv", sep=""
    ), row.names=F, col.names=T, sep = ",", append=F)
```

r_files/Bagging.R

A.4 Random Forest

```
rm(list=ls(all = T))
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(randomForest)
library(tikzDevice)
library(caret)
library(xtable)

source("HelpFunctions.R")
exportspath <- "../exports/tree_based_methods/randomforest/"

# read in data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]

ntree = 1000

# ## RANDOM FOREST IN PARALLEL ##
# # will not get OOB measures, so will not use it in this excercise
#
# library(doParallel)
# ncores <- detectCores()
# cl <- makeCluster(ncores)
# registerDoParallel(cl)
#
# rf <- foreach(ntree=rep(floor(ntree/ncores), ncores), .combine = combine,
    .packages = "randomForest") %dopar% {
#   tit_rf <- randomForest(Digit ~ ., data = train, ntree = ntree,
    importance=TRUE, na.action = na.exclude)
# }
# stopCluster(cl)


## RANDOM FOREST NORMAL ##
rf <- randomForest(Digit ~ ., data = train,
                   ntree = ntree,
                   importance=TRUE,
                   na.action = na.exclude,
                   do.trace = floor(ntree/10))

err_df <- data.frame(trees = 1:length(rf$err.rate[,"OOB"]), err = rf$err.
    rate[,"OOB"])
pl <- ggplot(err_df) +
  geom_line(aes(x = trees, y = err)) +
  xlab("Number of trees") + ylab("OOB error") + ggtitle("") +
  theme_bw()
exportPlotToLatex(pl, exportspath, "rf_oob_err.tex")

```

```
52
53 # Grow new Random Forest with optimal number of trees
54 ntree = 500
55 rf <- randomForest(Digit ~ ., data = train,
56                    ntree = ntree,
57                    importance=TRUE,
58                    na.action = na.exclude,
59                    do.trace = floor(ntree/10))
60
61
62 # predict values on test set
63 pred <- predict(rf, newdata = test)
64 pred_df <- data.frame(pred = pred, true = test$Digit)
65 confM <- makeConfusionMatrix(pred,test$Digit)
66 exportTableToLatex(confM, exportspath, "rf_confusionmatrix.tex")
67
68 # save data to .csv-files
69 write.table(err_df, file = paste(exportspath, "rf_oob_error.csv", sep=""),
       row.names=F, col.names=T, sep = ",", append=F)
70 write.table(pred_df, file = paste(exportspath, "rf_predictions.csv", sep="")
       , row.names=F, col.names=T, sep = ",", append=F)
```

r_files/RandomForest.R

A.5 Boosting

```
rm(list=ls(all = T))
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(tikzDevice)
library(xtable)
library(caret)
library(gbm)

source("HelpFunctions.R")
exportspath <- "../exports/tree_based_methods/boosting/"

# read in data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]

n.trees = 10000
cv.folds = 4
int.depth = 2
n.cores = 4
shrinkage = 0.01

# remove insignificant predictors to speed things up
cutPredictors <- getNZVPredictors(train)
train.scaled <- getScaledAndTrimmedData(train, cutPredictors)
test.scaled <- getScaledAndTrimmedData(test, cutPredictors)

boost <- gbm(Digit ~., data = train.scaled, distribution = "multinomial",
             n.trees = n.trees, interaction.depth = int.depth,
             cv.folds = cv.folds, shrinkage = shrinkage, n.cores = n.cores)

# plot the CV error as a function of the number of trees
best.iter <- gbm.perf(boost, method="cv", plot.it = F)
err_df <- data.frame(trees = 1:length(boost$cv.error), err = boost$cv.error)
pl <- ggplot(err_df) +
  geom_line(aes(x = trees, y = err)) +
  xlab("Number of trees") + ylab("CV error") +
  theme_bw()
exportPlotToLatex(pl, exportspath, "boost_cverr.tex")


# predict values for the test set
predict_values <- function(model, newdata, n.trees = best.iter){
  pred <- predict(model, newdata = newdata, n.trees = n.trees, type="
      response")
  return(apply(pred, 1, function(x) which.max(x) - 1))

  # factors = 0:9
  # return(sapply(pred, function(x) factors[which.min(abs(x - factors))]))
```

```
}


pred <- predict_values(boost, newdata = test.scaled, n.trees = best.iter)
pred_df <- data.frame(pred = pred, true = test.scaled$Digit)

confM <- makeConfusionMatrix(pred, test.scaled$Digit); print(confM)
exportTableToLatex(confM, exportspath, "boost_confusionmatrix.tex")

# save data to .csv-files
write.table(err_df, file = paste(exportspath, "boost_cv_error.csv", sep=""),
    row.names=F, col.names=T, sep = ",", append=F)
write.table(pred_df, file = paste(exportspath, "boost_predictions.csv", sep=
    ""), row.names=F, col.names=T, sep = ",", append=F)
```

r_files/Boosting.R

A.6 K Nearest Neighbors

```
rm(list=ls())
set.seed(1)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(caret)
library(tikzDevice)
library(xtable)
library(class)

source("HelpFunctions.R")
exportspath <- "../exports/KNN/"

# read data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]


# ============== #
# HELP FUNCTIONS #
# ============== #

# returns predictions found with knn()
executeKNN <- function(train, test, k, numComponents){
  # find near zero predictors and delete these
  cutPredictors <- getNZVPredictors(train)

  # normalize data and remove near zero predictors
  train.scale <- getScaledAndTrimmedData(train, cutPredictors)
  test.scale <- getScaledAndTrimmedData(test, cutPredictors)

  # apply PCA
  train.pca <- applyPCA(train.scale)

  # represent the training and testing sets with principal components
  train_as_pca <- getDataAsPCA(train.scale, train.pca, numComponents)
  test_as_pca <- getDataAsPCA(test.scale, train.pca, numComponents)

  # make predictions
  pred = knn(train = train_as_pca[,-1], test = test_as_pca[,-1], cl = train_
      as_pca$Digit, k = k)
  return(pred)
}

# cross validation including the use of principal components
cross_validate_pca <- function(data, num_runs, numFolds, numComponents, k){
  avg_errors <- c()
  for (run in 1:num_runs) {

    shuffle <- sample(1:nrow(data),length(1:nrow(data)))
```

```
54      folds <- split(shuffle, 1:numFolds)
55
56      errors <- c()
57      for (fold in folds){
58        cv.test = data[fold,]
59        cv.train = data[-fold,]
60
61        pred <- executeKNN(cv.train, cv.test, k, numComponents)
62        err = 1 - confusionMatrix(pred, cv.test$Digit)$overall["Accuracy"]
63
64        errors <- c(errors, err)
65      }
66      avg_errors <- c(avg_errors, mean(errors))
67
68    }
69    return(mean(avg_errors))
70  }
71
72  # ==================== #
73  # END OF HELP FUNCTIONS #
74  # ==================== #
75
76
77  # find the optimal value for k using cross validation
78  numComponents = 25
79  k_df = data.frame(k = seq(1,101,by=2), err = NA)
80  for (i in 1:nrow(k_df)){
81    k = k_df$k[i]; print(k)
82    k_df$err[i] = cross_validate_pca(data = train, num_runs = 1, numFolds = 5,
          numComponents = numComponents, k = k)
83  }
84
85  # plot the CV error for every K
86  pl <- ggplot(k_df) +
87    geom_line(aes(x = k, y = err)) +
88    geom_point(aes(x = k, y = err)) +
89    xlab("K") + ylab("CV error") +
90    theme_bw()
91  pl
92  exportPlotToLatex(pl, exportspath, "knn_pca_cverr.tex")
93
94  # choose the best k
95  optimalK = k_df$k[which.min(k_df$err)]
96
97  # make new predictions with the optimal K
98  pred <- executeKNN(train, test, optimalK, numComponents)
99  pred_df <- data.frame(pred = pred, true = test$Digit)
100 confM <- makeConfusionMatrix(pred, test$Digit); print(confM)
101 exportTableToLatex(confM, exportspath, "knn_pca_confusionmatrix.tex")
102
103 # write to .csv-files
104 write.table(k_df, file = paste0(exportspath, "knn_pca_cv_error.csv"), row.
      names=F, col.names=T, sep = ",", append=F)
105 write.table(pred_df, file = paste0(exportspath, "knn_pca_predictions.csv"),
      row.names=F, col.names=T, sep = ",", append=F)
```

r_files/KNNwithPCA.R

A.7 Support Vector Machines

```
rm(list=ls())
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(caret)
library(tikzDevice)
library(xtable)
library(e1071)

source("HelpFunctions.R")
exportspath <- "../exports/svm/"
exportspath <- "../test/"

# load data and split into train/test
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]




# ========================= #
# Preprocessing of the data #
# ========================= #

# find near zero variance predictors in training set
cutPredictors <- getNZVPredictors(data = train)

# remove near zero variance predictors from the data, then normalize.
train.scaled <- getScaledAndTrimmedData(data = train, cutPredictors)
test.scaled <- getScaledAndTrimmedData(data = test, cutPredictors)




# ============ #
# Applying PCA #
# ============ #

# apply PCA to training data
train.pca <- applyPCA(train.scaled)

# represent the training and testing sets with principal components
train_as_pca <- getDataAsPCA(train.scaled, train.pca, 50)
test_as_pca <- getDataAsPCA(test.scaled, train.pca, 50)

# plot variance explained per included principal components
varExpl <- summary(train.pca)$importance[2,]
varExplSummed <- cumsum(varExpl)
```

```r
varExpl_df <- data.frame(Comp = 1:length(train.pca$sdev), VarExpl = varExpl,
    CumVarExpl = varExplSummed)

pl <- ggplot(varExpl_df[1:100,]) +
  geom_point(aes(x = Comp, y = CumVarExpl)) +
  scale_y_continuous(limits=c(0,1)) +
  theme_bw() + xlab("Number of components") + ylab("Summed variance
      explained")
exportPlotToLatex(pl, exportspath, "svm_PCA_varExplainedTop100.tex")

# plot the data described by the first two principal components
namePC1 <- paste("PC1 (",round(varExpl[1],4)*100,"\\%)",sep="")
namePC2 <- paste("PC2 (",round(varExpl[2],4)*100,"\\%)",sep="")
pl <- ggplot(data = train_as_pca) +
  geom_point(aes(x = PC1, y = PC2, col=Digit)) +
  scale_color_discrete(name = "Digit") +
  xlab(namePC1) + ylab(namePC2) +
  theme_bw()
exportPlotToLatex(pl, exportspath, "svm_digit_twoPCA.tex")




# ============================================== #
# Finding optimal number of components to include #
# ============================================== #
number_of_runs = 10
components_vec = 2:33

evaluateCVofComponents <- function(number_of_runs, components_vec){
  error_df <- data.frame(components = components_vec, cv_err = NA)
  for (i in 1:length(components_vec)){
    chosenNumberOfComponents = components_vec[i]

    # train the model
    train_as_pca = getDataAsPCA(scaledData = train.scaled, pca = train.pca,
        numComponents = chosenNumberOfComponents)

    # tune and fit a model on the training set
    cv_errors <- c()
    for (run in 1:number_of_runs){
      svm.tune <- tune.svm(Digit ~ ., data = train_as_pca, cost = 2, kernel
          = "radial")
      cv_errors <- c(cv_errors, svm.tune$best.performance)
    }

    # save the mean error measure to the error_df
    error_df$cv_err[i] <- mean(cv_errors)

    # print progress to console
    print(sprintf("Number of components: %.0f", chosenNumberOfComponents))
    print(sprintf("CV Error: %.5f", error_df$cv_err[i]))
    print("", quote = F)

  }
```

```
  return(error_df)
}

error_df <- evaluateCVofComponents(number_of_runs = number_of_runs, components_vec = components_vec)
error_df$cv_err <- as.numeric(error_df$cv_err)

# plot the CV error per included component
pl <- ggplot(error_df) +
  geom_point(aes(x = components, y = cv_err)) +
  geom_line(aes(x = components, y = cv_err)) +
  scale_x_continuous(breaks = c(seq(0,nrow(error_df),5))) +
  theme_bw() +
  xlab("Number of components") + ylab("Error") + ggtitle("")
exportPlotToLatex(pl, exportspath, "svm_errors_per_components.tex")

# based on the plot, choosing to include 25 of principal components in the model
chosenNumberOfComponents = 25

# retrain the model
train_as_pca <- getDataAsPCA(train.scaled, train.pca, chosenNumberOfComponents)
test_as_pca <- getDataAsPCA(test.scaled, train.pca, chosenNumberOfComponents)




# ================================== #
# Finding the optimal regularization C #
# ================================== #
# will fit a model N times for a given C and return the mean cv error for each C.

number_of_runs = 10
trainingData = train_as_pca
cost_params = seq(0.5,15,by=0.5)

compareCostParameter <- function(number_of_runs, trainingData = train, cost_params){

  errors <- matrix(0L, nrow = length(cost_params), ncol = number_of_runs, byrow=T,
                   dimnames = list(paste0("C",cost_params),paste0("run",1:number_of_runs)))
  for (run in 1:number_of_runs){
    print(run)
    grid <- tune.svm(Digit ~ ., data = trainingData, cost = cost_params, kernel ="radial")
    df <- data.frame(cost = grid$performances$cost, cv_err = grid$performances$error)
    errors[,run] = df$cv_err
  }

```

```r
  avg_errors <- data.frame(cost = cost_params, cv_err = rowMeans(errors))
  bestCost = avg_errors$cost[which(avg_errors$cv_err == min(avg_errors$cv_
      err))]

  return(list(avg_errors = avg_errors, bestCost = bestCost))
}

cost_errors <- compareCostParameter(number_of_runs, trainingData, cost_
    params)
bestCost = cost_errors$bestCost

# plot the expected CV error per C
pl <- ggplot(cost_errors$avg_errors) +
  geom_line(aes(x = cost, y = cv_err)) +
  geom_point(aes(x = cost, y = cv_err)) +
  geom_point(aes(x = bestCost, y = cv_err[which(cv_err == min(cv_err))]),
      col="red") +
  geom_vline(xintercept = bestCost, linetype = "dotted", col ="red") +
  scale_x_continuous(breaks=c(bestCost,seq(0,15,by=5))) +
  xlab("C") + ylab("CV Error") + ggtitle("") +
  theme_bw()
exportPlotToLatex(pl, exportspath, "svm_CvsCVerr.tex")









# ====================================== #
# Fitting a new model with best parameters #
# ====================================== #

# fit a model with the best cost parameter
svm.fit <- svm(Digit ~ ., data = train_as_pca, cost = bestCost, kernel="
    radial")

# evaluate the fit on the testing set
pred <- predict(svm.fit, test_as_pca, type="response")
pred_df <- data.frame(pred = pred, true = test$Digit)

confM <- makeConfusionMatrix(pred,test$Digit)
exportTableToLatex(confM, exportspath, "svm_confusionmatrix.tex")


# write csv files to store the data
write.table(error_df, file = paste(exportspath, "svm_cv_error.csv", sep=""),
     row.names=F, col.names=T, sep = ",", append=F)
write.table(cost_errors$avg_errors, file = paste(exportspath, "svm_cost_
    error.csv", sep=""), row.names=F, col.names=T, sep = ",", append=F)
write.table(pred_df, file = paste(exportspath, "svm_predictions.csv", sep=""
    ), row.names=F, col.names=T, sep = ",", append=F)
```

r_files/SVM.R

A.8 Artificial Neural Networks

```
rm(list=ls())
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(ggplot2)
library(h2o)
library(tikzDevice)
library(caret)
library(xtable)

source("HelpFunctions.R")
exportspath = "../exports/NN/"

# read in data
data <- read.csv("../data/Train_Digits_20171108.csv")
data$Digit <- as.factor(data$Digit)

sample <- sample(1:nrow(data), 0.8*nrow(data))
train <- data[sample,]
test <- data[-sample,]


# initialize h2o
localH2O = h2o.init(nthreads = -1, max_mem_size = '4G')

# add data to h2o environment
train.h2o <- as.h2o(train)
test.h2o <- as.h2o(test)

# define predictors and response variables
response = colnames(train)[1]
predictors = colnames(train)[2:ncol(train)]


# using a grid search to find the best combination of parameter tunings
epochs_opt = c(100)
hidden_opt <- list(c(100), c(200), c(400), c(400, 200), c(500, 100), c
    (300,300))
rate_opt <- c(0.1, 0.01, 0.001)
stopping_tolerance_opt <- c(0.0001)
hyper_params = list(epochs = epochs_opt,
                    hidden = hidden_opt,
                    rate = rate_opt,
                    stopping_tolerance = stopping_tolerance_opt)

nn.grid <- h2o.grid("deeplearning",
                    x = predictors, y = response,
                    training_frame = train.h2o,
                    nfolds = 5,
                    seed = 1000,
                    input_dropout_ratio = 0.20,
                    activation = "Tanh",
                    stopping_metric = "misclassification",
                    stopping_rounds = 2,
```

```
                    l1 = 1e-5,
                    hyper_params = hyper_params)

# sort grid by increasing CV error and extract the best model
# nn.grid.ordered <- h2o.getGrid(nn.grid@grid_id, sort_by = "err",
    decreasing=FALSE);
nn.grid.ordered <- h2o.getGrid(nn.grid@grid_id, sort_by = "mse", decreasing
    = FALSE);
nn.grid.ordered

# plot the grid sorted by mse
pl <- ggplot(nn.grid.ordered@summary_table, aes(x=1:length(nn.grid.
    ordered@model_ids), y=as.numeric(err))) +
  geom_point() +
  geom_point(colour = "red", aes(x = 1, y = as.numeric(err[1]))) + # colour
      the best model in red
  xlab("Nth model with lowest CV Error") + ylab("CV error") + ggtitle("") +
  theme_bw()
exportPlotToLatex(pl, exportspath, "ANN_cverr_gridmodel.tex")


# extract the best model
best_model <- h2o.getModel(nn.grid.ordered@model_ids[[1]])
summary(best_model)

# metrics of best model on TRAINING set
h2o.mse(best_model, xval=TRUE) # mean CV error
best_model@model$cross_validation_metrics_summary # All CV error data

# make predictions and write confusion matrix to file
pred <- h2o.predict(best_model, newdata = test.h2o)$predict
pred <- as.numeric(as.matrix(pred))

confM <- makeConfusionMatrix(pred, test$Digit)
exportTableToLatex(confM, exportspath, "ANN_confusionmatrix.tex")


# write parameters to tex-file
grid_df <- as.data.frame(nn.grid.ordered@summary_table)[,-5]
grid_df$epochs <- as.numeric(grid_df$epochs)
grid_df$mse <- as.numeric(grid_df$mse)

# write grid to tex-file
print(xtable(grid_df,
           display = c("d","f", "s", "f", "e", "f"),
           digits = c(0,2,0,2,1,5)),
           align = c(rep("c",ncol(grid_df)+1)),
    file = paste0(exportspath, "ANN_gridresults.tex"),
    only.contents = T,
    include.rownames = T)


h2o.shutdown(prompt = F)
```

r_files/ANN.R

A.9 Convolutional Neural Networks

```
rm(list=ls())
set.seed(1000)
Sys.setenv(TZ="Africa/Johannesburg")

library(mxnet)
library(ggplot2)
library(xtable)
library(tikzDevice)
library(caret)

source("HelpFunctions.R")
exportspath <- "../exports/NN/"

# read in data
alldata <- read.csv("../data/Train_Digits_20171108.csv")
alldata$Digit <- as.factor(alldata$Digit)

sample <- sort(sample(1:nrow(alldata), 0.8*nrow(alldata)))
train <- alldata[sample[1:(0.8*length(sample))],]
val <- alldata[sample[(nrow(train)+1):length(sample)],]
test <- alldata[-sample,]

train.m <- data.matrix(train)
val.m <- data.matrix(val)
test.m <- data.matrix(test)

# define predictors and response for training, validation and testing
train.x <- t(train.m[,-1])
train.y <- train.m[,1] - 1
train.array <- train.x
dim(train.array) <- c(28, 28, 1, ncol(train.x))

val.x <- t(val.m[,-1])
val.y <- val.m[,1] - 1
val.array <- val.x
dim(val.array) <- c(28, 28, 1, ncol(val.x))


test.x <- t(test.m[,-1])
test.y <- test.m[,1]
test.array <- test.x
dim(test.array) <- c(28, 28, 1, ncol(test.x))









# ========================== #
# Setup the CNN architecture #
# ========================== #
data <- mx.symbol.Variable('data')

# first convolution
```

```
conv1 <- mx.symbol.Convolution(data=data, kernel=c(5,5), num_filter=20)
tanh1 <- mx.symbol.Activation(data=conv1, act_type="tanh")
pool1 <- mx.symbol.Pooling(data=tanh1, pool_type="max",
                           kernel=c(2,2), stride=c(2,2))

# second convolution
conv2 <- mx.symbol.Convolution(data=pool1, kernel=c(5,5), num_filter=50)
tanh2 <- mx.symbol.Activation(data=conv2, act_type="tanh")
pool2 <- mx.symbol.Pooling(data=tanh2, pool_type="max",
                           kernel=c(2,2), stride=c(2,2))

# first fully-connected layer
# use data from convolutions (flatten)
flatten <- mx.symbol.Flatten(data=pool2)
fc1 <- mx.symbol.FullyConnected(data=flatten, num_hidden=500)
tanh3 <- mx.symbol.Activation(data=fc1, act_type="tanh")


# second fully-connected layer
fc2 <- mx.symbol.FullyConnected(data = tanh3, num_hidden=10)

# the output will be a 10-way softmax.
softmax <- mx.symbol.SoftmaxOutput(data = fc2)

# use CPU device
devices <- mx.cpu()
mx.set.seed(1000)

# define a logger to keep track of errors
logger <- mx.metric.logger$new()




# ============================== #
# Hyperparameter search in a grid #
# ============================== #

# perform a grid search for the best parameters
grid <- data.frame(id = 1:6, rate = c(0.01,0.01,0.001,0.001,0.005,0.005),
    array_size = c(50,100), e_val = NA)

for (i in 1:nrow(grid)){
  rate = grid$rate[i]
  size = grid$array_size[i]

  cnn.model <- mx.model.FeedForward.create(softmax,
                                           X = train.array, y = train.y,
                                           eval.data = list(data=val.array,
                                               label=val.y),
                                           ctx=devices,
                                           num.round=40,
                                           array.batch.size=size,
                                           learning.rate=rate,
                                           momentum=0.9,
                                           eval.metric=mx.metric.accuracy,
```

```
                                            optimizer = "sgd",
                                            epoch.end.callback=mx.callback.
                                                log.train.metric(5,logger))

  grid$e_val[i] = 1-logger$eval[length(logger$eval)]

}

# export grid to latex
print(xtable(grid,
             display = c("d", "d", "f", "d", "f"),
             digits = c(0, 0, 3, 0, 4)),
      file = paste0(exportspath, "cnn_gridsearch.tex"),
      only.contents = T,
      include.rownames = F)

# plot the grid
pl<-ggplot(grid) +
  geom_point(aes(x = id, y = e_val)) +
  theme_bw() + xlab("Model") + ylab("$E_{val}$")
exportPlotToLatex(pl, exportspath, "cnn_griderrors.tex")




# ======================================== #
# Train a new CNN with the best parameters #
# ======================================== #

# just a run-through to plot the validation error as a function of epochs
sortedGrid <- grid[order(grid$e_val),]
bestParameters <- sortedGrid[1,c(2,3)]
bestParameters <- data.frame(rate = 0.01, array_size = 100)

logger <- mx.metric.logger$new()
best.model.val <- mx.model.FeedForward.create(softmax,
                                          X = train.array, y = train.y,
                                          eval.data = list(data=val.array,
                                              label=val.y),
                                          ctx=devices,
                                          num.round=40,
                                          array.batch.size=bestParameters$
                                              array_size,
                                          learning.rate=bestParameters$rate,
                                          momentum=0.9,
                                          optimizer = "sgd",
                                          eval.metric=mx.metric.accuracy,
                                          epoch.end.callback=mx.callback.log.
                                              train.metric(5,logger))


err_df <- data.frame(runs = 1:40, e_val = 1 - logger$eval)
pl <- ggplot(err_df) +
  geom_line(aes(x = runs, y = e_val)) +
```

```
  theme_bw() + xlab("Epochs") + ylab("$E_{val}$")
exportPlotToLatex(pl, exportspath, "cnn_error.tex")


# train A NEW model with THE WHOLE training data, no validation set
train <- alldata[sample,]
train.m <- data.matrix(train)
train.x <- t(train.m[,-1])
train.y <- train.m[,1] - 1
train.array <- train.x
dim(train.array) <- c(28, 28, 1, ncol(train.x))

best.model <- mx.model.FeedForward.create(softmax,
                                          X = train.array, y = train.y,
                                          ctx=devices,
                                          num.round=40,
                                          array.batch.size=bestParameters$
                                              array_size,
                                          learning.rate=bestParameters$rate,
                                          momentum=0.9,
                                          optimizer = "sgd",
                                          eval.metric=mx.metric.accuracy,
                                          epoch.end.callback=mx.callback.log.
                                              train.metric(100))


# predict on the test set
pred <- predict(best.model, test.array)
pred_digits <- max.col(t(pred)) - 1

confM <- makeConfusionMatrix(pred_digits,test$Digit)
print(confM)
exportTableToLatex(confM, exportspath, "CNN_confusionmatrix.tex")









# ================================= #
# MAKE PREDICITONS ON REAL TEST SET #
# ================================= #
# As this method was the one who produced the best results of all ML
    techniques, use this
# to predict on the final test set

# read in data again
train <- read.csv("../data/Train_Digits_20171108.csv")
test <- read.csv("../data/Test_Digits_20171108.csv")
train$Digit = as.factor(train$Digit)
test$Digit = as.factor(test$Digit)

train.m <- data.matrix(train)
test.m <- data.matrix(test)

```

```
214 # define predictors and response
215 train.x <- t(train.m[,-1])
216 train.y <- train.m[,1] - 1
217 train.array <- train.x
218 dim(train.array) <- c(28, 28, 1, ncol(train.x))
219
220 test.x <- t(test.m[,-1])
221 test.array <- test.x
222 dim(test.array) <- c(28, 28, 1, ncol(test.x))
223
224 # train model on whole testing data
225 # use the parameters found in validation
226 final.model <- mx.model.FeedForward.create(softmax,
227                                            X = train.array, y = train.y,
228                                            ctx=devices,
229                                            num.round=40,
230                                            array.batch.size=bestParameters$
                                                array_size,
231                                            learning.rate=bestParameters$rate,
232                                            momentum=0.9,
233                                            optimizer = "sgd",
234                                            eval.metric=mx.metric.accuracy,
235                                            epoch.end.callback=mx.callback.log
                                                .train.metric(100))
236
237
238 # predict on the test set
239 pred <- predict(final.model, test.array)
240 pred_digits <- max.col(t(pred)) - 1
241 pred_isUneven <- pred_digits %% 2
242
243 pred_df <- data.frame(row_id = 1:2500, pred_digit = pred_digits, pred_
        isUneven = pred_isUneven)
244 write.table(pred_df, file = "../exports/finalpredictions.csv", row.names=F,
        col.names=T, sep = ",", append=F)
```

r_files/CNN.R