Assignment 4

Martin Lindqvist

2024-03-15

Task 1

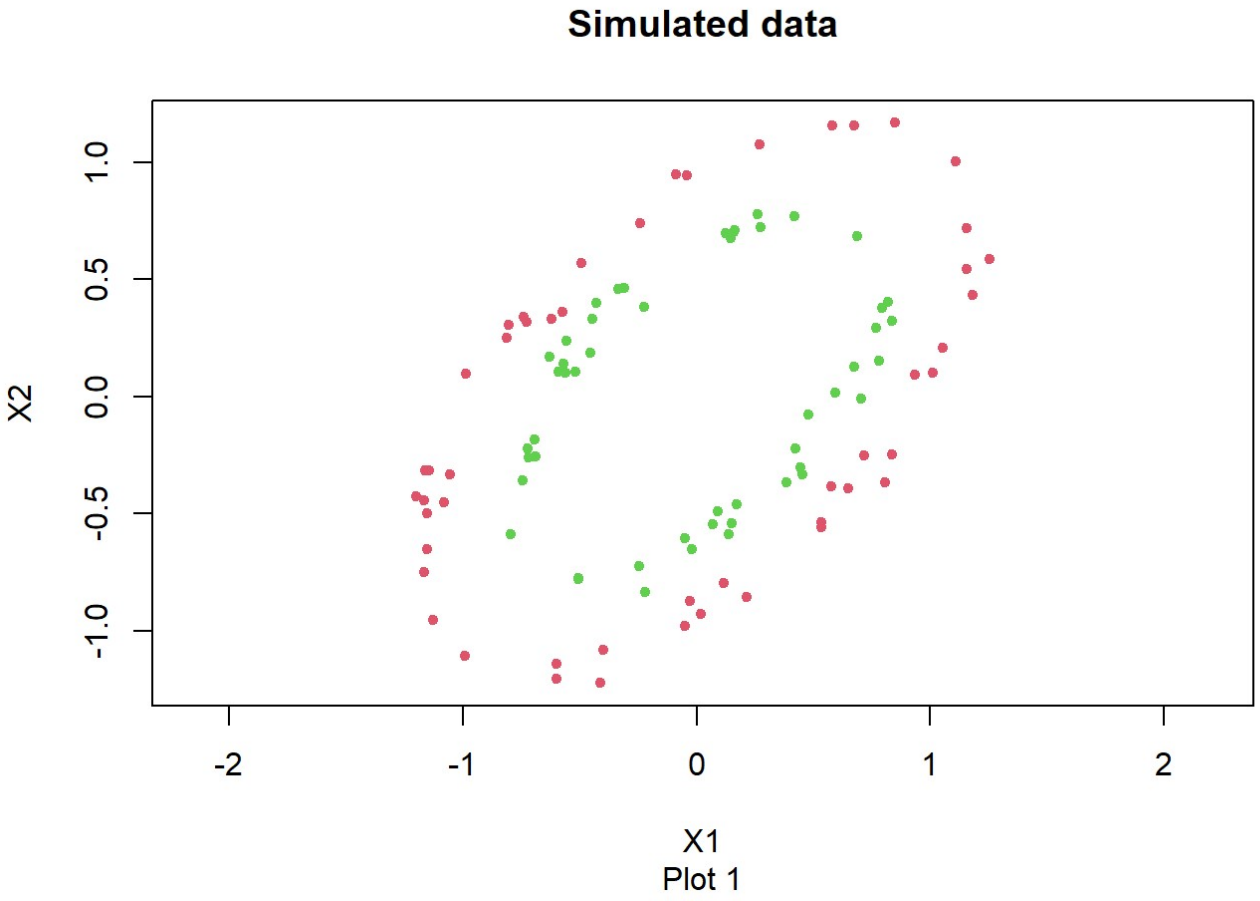
Introduction

Principal Component Analysis is a common dimensionality reduction technique used to create uncorrelated components that keeps as much of the variance as possible, while significantly fewer variables. These components can be used for exploratory data analysis or as components in machine learning models. In this assignment we will explore two different approaches to finding principal components that makes the data linearly separable.

We begin by simulating data with complex non-linear relationships. We then demonstrate two methods: feature mapping, where we transform the data to a higher dimensional space for linear separation, and the kernel trick which manipulates the data without transforming it to a higher dimensional space.

Data

The data was generated such that it is not linearly separable. The data consists of two groups, the distribution of both groups can be seen as oval shapes with one group laying inside the other as seen in plot 1.



Plot 1

Feature mapping

The original data consists of two features and a response class. The data is initially transformed via the following feature mapping:

$$\phi(\mathbf{x}_1, \mathbf{x}_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

Then the new features are saved in a new matrix H.

$$H = \begin{bmatrix} h1_1 & h2_1 & h3_1 \\ \vdots & \vdots & \vdots \\ h1_{100} & h2_{100} & h3_{100} \end{bmatrix}$$

The matrix is centred by subtracting the column means and saved in a new matrix HH.

$$HH = \begin{bmatrix} h1_1 & h2_1 & h3_1 \\ \vdots & \vdots & \vdots \\ h1_{100} & h2_{100} & h3_{100} \end{bmatrix} - \begin{bmatrix} \bar{h1} & \bar{h2} & \bar{h3} \\ \vdots & \vdots & \vdots \\ \bar{h1} & \bar{h2} & \bar{h3} \end{bmatrix}$$

The transpose of HH is multiplied by HH and saved in matrix S.

$S = HH^T HH$

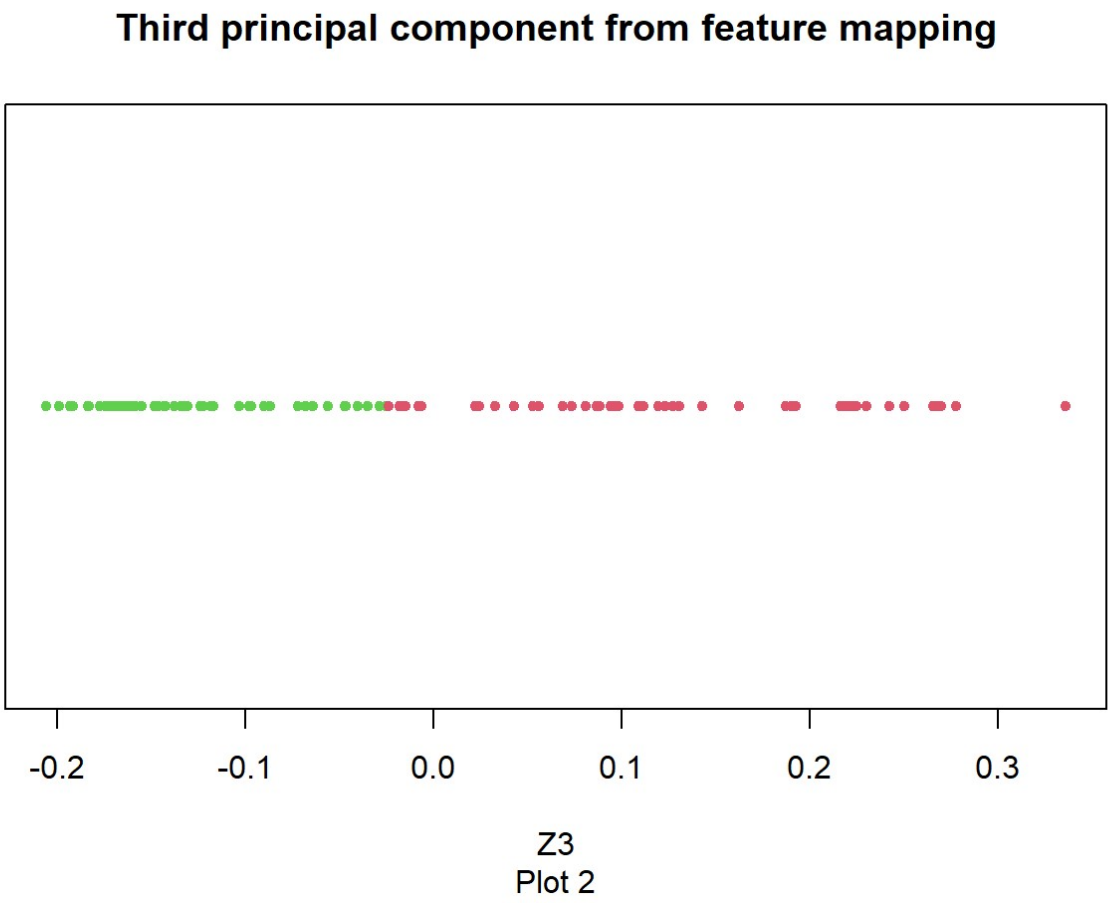
We then use the `eigen` function to get the eigenvectors that correspond to the principal components and save it as P.

$$P = \begin{bmatrix} 0.5217520 & -0.6628223 & 0.5370675 \\ 0.7911687 & 0.1404677 & -0.5952486 \\ 0.3191034 & 0.7354832 & 0.5976935 \end{bmatrix}$$

We then project the data onto the principal components and save it as z.

$$z = HHP$$

The third principal component of z is plotted below in plot 2. We can see the the two classes are linearly separable using the third principal component.



Plot 2

Kernel trick

To get the same results without feature mapping we use the kernel trick.

We choose the polynomial kernel $\kappa_\phi(x_i, x_j) = (c + \xi x_i^\top x_j)^M$ and set $M = 2$, $\xi = 1$ and $c = 0$ to get $\kappa_\phi(x_i, x_j) = (x_i^\top x_j)^2$ which corresponds to the earlier feature mapping $\phi(\mathbf{x}_1, \mathbf{x}_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$.

This means that instead of transforming the whole dataset we can manipulate the inner product to achieve the same results.

We start by computing the inner product of the rows of X and square each of these values, saving the result in a new matrix K.

$$K = (XX^T)^2$$

We then centralise the data to obtain the Gram matrix.

$$K^* = K - CK - KC + CKC$$

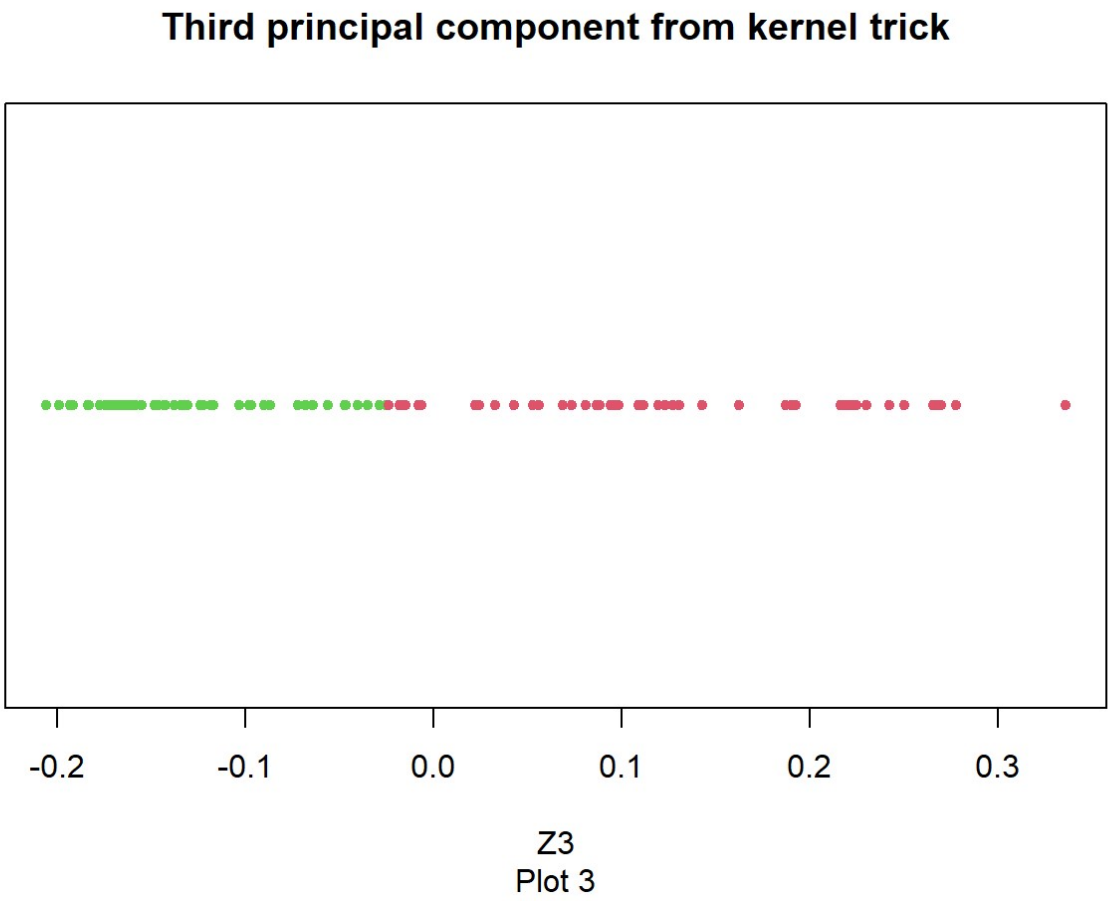
Where C is a 100 by 100 matrix:

$$C = \begin{bmatrix} \frac{1}{100} & \cdots & \frac{1}{100} \\ \vdots & \ddots & \vdots \\ \frac{1}{100} & \cdots & \frac{1}{100} \end{bmatrix}$$

We then use the `eigen` function to do eigen decomposition on K^* to get the eigen values and eigen vectors.

To get the third principal component we multiply the square root of the third eigen value by the third eigen vector.

$$\sqrt{\lambda}_3 e_3$$



Plot 3

Discussion

We can see in **plot 2** and **plot 3** that both methods can find the third principal component that makes the data linearly separable. The kernel trick does not need to transform the data in to a higher dimensional space. Instead it computes the inner products in the new feature space, which is much more computationally efficient. It also bypasses the need to store this higher dimensional data, which makes it much more memory efficient. These two benefits makes this method more scalable on larger datasets. Given these advantages, I would opt for the kernel trick.

Task 2

Introduction

Three different methods have been trained for image classification: Support Vector Machine (SVM), Artificial Neural Network (ANN) and Convolutional Neural Network. We will start by exploring the data before looking at each model separately and finally compare the performance.

Data

The data is a subset of the “Fashion-MNIST” data and consists of 30,000 observations of images belonging to one of 10 classes:

0 = T-shirt/top

1 = Trouser

2 = Pullover

3 = Dress

4 = Coat

5 = Sandal

6 = Shirt

7 = Sneaker

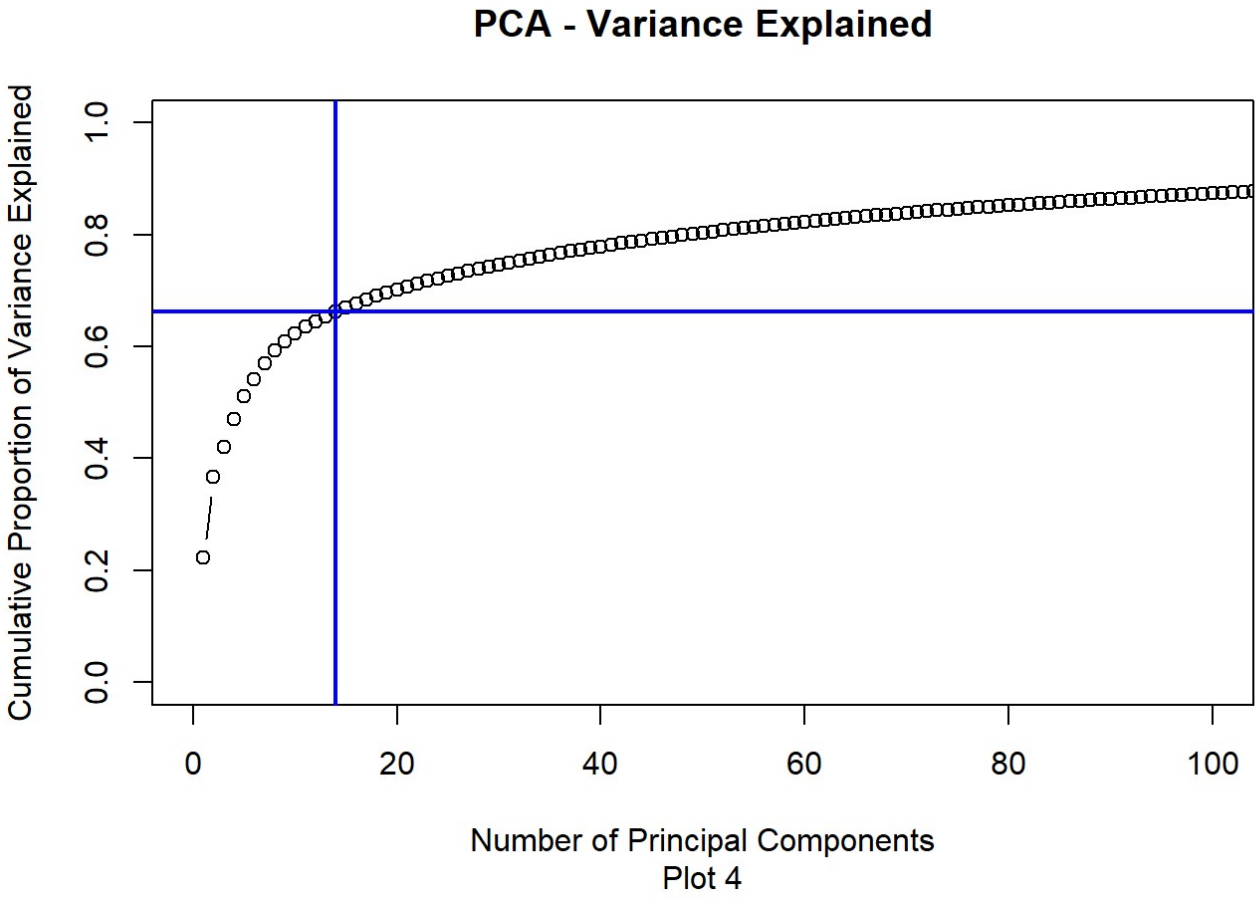
8 = Bag

9 = Ankle boot

The data is then randomly split in to training (80%) and testing (20%) data. To prepare the data for SVM we separate the features and the labels, and set the labels to factors. To prepare the data for ANN and CNN we do one-hot encoding for the categorical variables and normalise the features.

SVM

Support Vector Machines (SVM) use the kernel trick to separate the data in to different classes. SVMs are known to be computationally intensive. We therefore calculate the principal components and use a few of them as inputs to speed up the training process.



Plot 4

In **plot 4** we can see the variance explained as we use more principal components. The amount of variance explained for each additional component decrease as the number of components increase, meaning that

we get the most effect/component for the first few components. The number of principal components are set to 14 (the reasoning for this will be explained in the tuning section).

Tuning and training

To train the model the `e1071` package's `svm` function was used. Type was set to `C-classification` since this is a classification problem. For the kernel function it was set to `radial` (radial basis function) since it is often a good fit for both simpler and more complex relationships within the data. It only has one hyper parameter, which makes the model easier to tune. The model uses a soft margin classifier by default since using a hard margin classifier would likely overfit the model to the training data. This leaves us with two hyper parameters to tune: the `gamma` from the `radial` kernel, and the `cost` function that indicates how severely misclassifications should be treated.

Due to the time consuming nature of tuning SVM's and my lack of computational power some guessing had to be done in tuning the model. The number of principal components was initially set to 10. Then the cost and gamma was tuned separately (due to my lack of computational power) using the `svm.fit` function for `e1071` using 5 fold cross validation. After each iteration the best value was choose and a new range of values to try was set for the other parameter. After a few iterations the best values was `cost` = 2.5 and `gamma` = 0.3. This model was then trained with different number of principal components. Out of the range from 5 to 25, 14 gave the best results on the testing data.

The final model gave a training accuracy of 91.56% and a test accuracy of 84.65%.

ANN

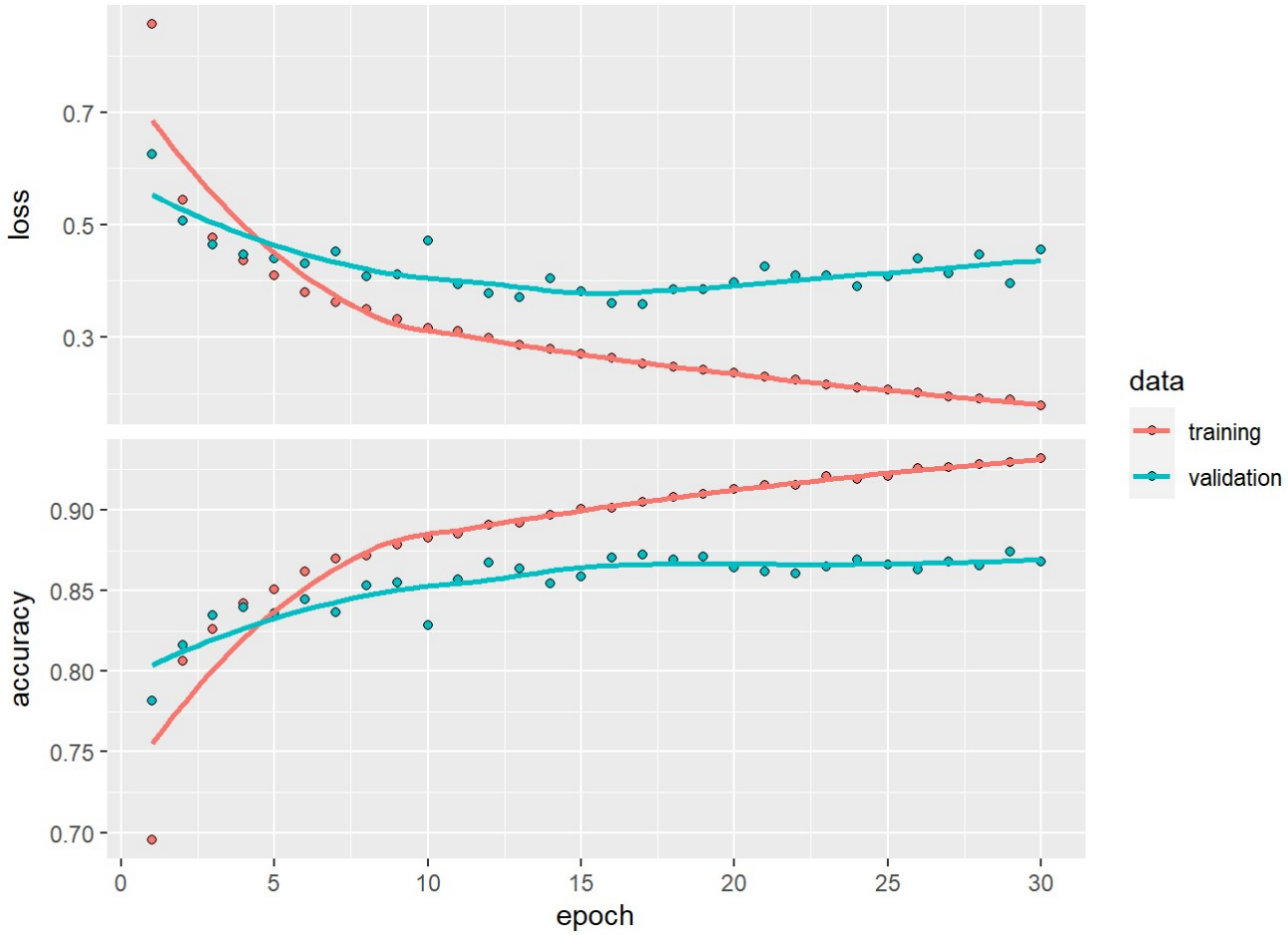
Artificial Neural Networks (ANN) uses one or multiple layers of nodes often called neurons for a number of different tasks, including regression and classification problems. Each node in the network performs simple computations and passes its output to the subsequent layer. During the training process the weights between the nodes are adjusted to minimise the difference between the predictions and the target.

To train this model the `keras` package was used. The model consists of three hidden layers with 128, 24 and 24 nodes in each layer, respectively. The activation function for the three hidden layers is set to `relu` (Rectified Linear Unit) as $f(x) = max(0, x)$. If x is negative the function will return zero, else it will return x. This function is chosen for its ability to reduce vanishing gradient problem. Following hidden layers, comes an output layer with 10 nodes (since we have 10 classes). The activation function for the output layer is set to `softmax` which gives each class an probability of occurring. The class with the highest probability is then chosen as the models final prediction.

The model was compiled with the loss function set to `categorical_crossentropy` which is commonly used for classification problems with multiple classes. The optimiser was set to `optimizer_rmsprop()` (Root Mean Square Propagation). The metrics used to evaluate the model performance was set to `accuracy` .

The model was trained over the entire dataset for 30 iterations (`epochs`) with `batch_size` of 128 meaning that the model samples 128 observations per gradient update. The `validation_split` was set to 0.2, meaning that for each iteration the model trained on 80% of the data, and evaluated the loss function and accuracy on the remaining 20%.

In **plot 5** we can see how the how the accuracy increases for each training iteration. After approximately 15 iterations the validation accuracy stops increasing while the training accuracy keeps increasing, indicating overfitting. This can be seen in the loss function as well, where the loss starts to increase after approximately 15 iterations.



Plot 5

The ANN model achieves a test accuracy of 86.65%.

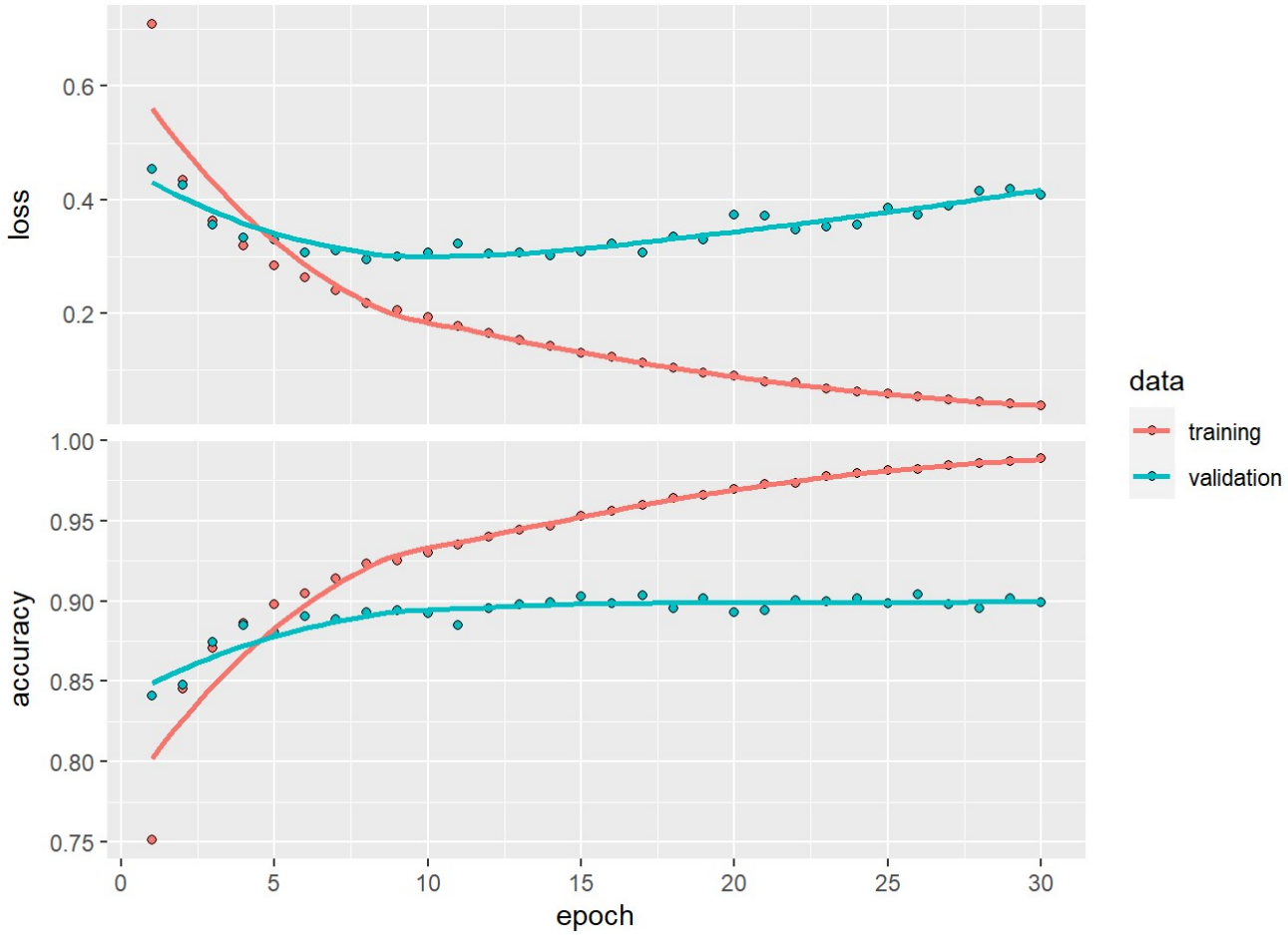
CNN

Convolutional Neural Networks (CNN) works similarly to ANNs, but are especially made for grid data such as images. The main difference is that one or multiple convolutional layers are added in addition to the hidden layers. These convolutional layers add smaller `filters` that are slid along the image to detect patterns such as edges. These layers improves the networks ability to detect and classify images.

The model consists of one convolutional layer with 32 `filters` of size 3x3. The `padding` is set to `same` to ensure the output has the same size as the input. The `relu` activation is used again. Following the convolutional layer a max pooling layer is used to reduce the spatial dimensions of the output by taking the max value of 2x2 non-overlapping girds. Then a flattening layer is used to convert the data from a grid to a vector before sending it to the hidden layer with 100 neurons. Finally we have the output layer with 10 classes that once again uses the `softmax` activation function.

The same process with the same parameters was used for model compilation and training.

In **plot 6** is see little to no improvement in accuracy after 10 iterations, with an even more pronounced increase in the loss function compared to the ANN model in **plot 5**.



Plot 6

The CNN model achieves a test accuracy of 90.2%.

Discussion

Out of the three models SVM showed the worst results on the training data. It was also by far the most cumbersome to work with. Even though the dataset was small compared to how large datasets for machine learning projects can be in the industry, model fitting and tuning took a long time, even though PCA was used. ANN provided a decent improvement in accuracy, but a huge improvement in the ease it was to work with. The training and tuning of parameters was a lot faster. To no one's surprise the CNN provided the most accurate classifications on the image data. This is because it uses a convolutional layer especially made for this task. The CNN also had the same benefits as ANN in training time. Going forward I would prefer working with ANNs and CNNs for classification tasks on large datasets due to both their performance in classification accuracy and speed of training.