This is the post in a series devoted to comparing different machine learning methods for predicting clothing categories from images using the Fashion MNIST data by Zalando.

First, we prepared the data for analysis and built a Python deep learning neural network model to predict the clothing categories of the Fashion MNIST data.

To start, we first set our seed to make sure the results are reproducible.

set.seed(1234)

Importing and exploring the data

The keras package contains the Fashion MNIST data, so we can easily import the data into RStudio from this package directly after installing it from Github and loading it.

**library**(devtools)

install.packages("keras")

*#devtools::install\_github("rstudio/keras")*

**library**(keras)

install\_keras()

fashion\_mnist <- keras::dataset\_fashion\_mnist()

The resulting object named fashion\_mnist is a nested list, consisting of lists train and test. Each of these lists in turn consists of arrays x and y. To look at the dimensions of these elements, we recursively apply the dim() function to the fashion\_mnist list.

rapply(fashion\_mnist, dim)

train.x1 train.x2 train.x3 train.y test.x1 test.x2 test.x3 test.y

60000 28 28 60000 10000 28 28 10000

From the result, we observe that the x array in the training data contains 28 matrices each of 60000 rows and 28 columns, or in other words 60000 images each of 28 by 28 pixels. The y array in the training data contains 60000 labels for each of the images in the x array of the training data. The test data has a similar structure but only contains 10000 images rather than 60000. For simplicity, we rename these lists elements to something more intuitive (where x now represents images and y represents labels):

c(train.images, train.labels) %<-% fashion\_mnist$train

c(test.images, test.labels) %<-% fashion\_mnist$test

Every image is captured by a 28 by 28 matrix, where entry [i, j] represents the opacity of that pixel on an integer scale from 0 (white) to 255 (black). The labels consist of integers between zero and nine, each representing a unique clothing category. As the category names are not contained in the data itself, we have to store and add them manually. Note that the categories are evenly distributed in the data.

cloth\_cats = data.frame(category = c('Top', 'Trouser', 'Pullover', 'Dress', 'Coat',

'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Boot'),

label = seq(0, 9))

To get an idea of what the data entail and look like, we plot the first ten images of the test data. To do so, we first need to reshape the data slightly such that it becomes compatible with ggplot2. We select the first ten test images, convert them to data frames, rename the columns into digits 1 to 28, create a variable named y with digits 1 to 28 and then we melt by variable y. We need package reshape2 to access the melt() function. This results in a 28 times 28 equals 784 by 3 (y pixels (= y), x pixels (= variable) and the opacity (= value)) data frame. We bind these all together by rows using the rbind.fill() function from the plyr package and add a variable Image, which is a unique string repeated 784 times for each of the nine images containing the image number and corresponding test set label.

**library**(reshape2)

**library**(plyr)

subarray <- apply(test.images[1:10, , ], 1, as.data.frame)

subarray <- lapply(subarray, **function**(df){

colnames(df) <- seq\_len(ncol(df))

df['y'] <- seq\_len(nrow(df))

df <- melt(df, id = 'y')

**return**(df)

})

plotdf <- rbind.fill(subarray)

first\_ten\_labels <- cloth\_cats$category[match(test.labels[1:10], cloth\_cats$label)]

first\_ten\_categories <- paste0('Image ', 1:10, ': ', first\_ten\_labels)

plotdf['Image'] <- factor(rep(first\_ten\_categories, unlist(lapply(subarray, nrow))),

levels = unique(first\_ten\_categories))

We then plot these first ten test images using package ggplot2. Note that we reverse the scale of the y-axis because the original dataset contains the images upside-down. We further remove the legend and axis labels and change the tick labels.

**library**(ggplot2)

ggplot() +

geom\_raster(data = plotdf, aes(x = variable, y = y, fill = value)) +

facet\_wrap(~ Image, nrow = 2, ncol = 5) +

scale\_fill\_gradient(low = "white", high = "black", na.value = NA) +

theme(aspect.ratio = 1, legend.position = "none") +

labs(x = NULL, y = NULL) +

scale\_x\_discrete(breaks = seq(0, 28, 7), expand = c(0, 0)) +

scale\_y\_reverse(breaks = seq(0, 28, 7), expand = c(0, 0))

[](https://rviews.rstudio.com/post/2019-10-31-a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/index_files/figure-html/unnamed-chunk-8-1.png)

Data Preparation

Next, it’s time to start the more technical work of predicting the labels from the image data. First, we need to reshape our data to convert it from a multidimensional array into a two-dimensional matrix. To do so, we vectorize each 28 by 28 matrix into a column of length 784, and then we bind the columns for all images on top of each other, finally taking the transpose of the resulting matrix. This way, we can convert a 28 by 28 by 60000 array into a 60000 by 784 matrix. We also normalize the data by dividing between the maximum opacity of 255.

train.images <- data.frame(t(apply(train.images, 1, c))) / max(fashion\_mnist$train$x)

test.images <- data.frame(t(apply(test.images, 1, c))) / max(fashion\_mnist$train$x)

We also create two data frames that include all training and test data (images and labels), respectively.

pixs <- 1:ncol(fashion\_mnist$train$x)^2

names(train.images) <- names(test.images) <- paste0('pixel', pixs)

train.labels <- data.frame(label = factor(train.labels))

test.labels <- data.frame(label = factor(test.labels))

train.data <- cbind(train.labels, train.images)

test.data <- cbind(test.labels, test.images)

Artificial Neural Network

Now, let’s continue by building a simple neural network model to predict our clothing categories. Neural networks are artificial computing systems that were built with human neural networks in mind. Neural networks contain nodes, which transmit signals amongst one another. Usually the input at each node is a number, which is transformed according to a non-linear function of the input and weights, the latter being the parameters that are tuned while training the model. Sets of neurons are collected in different layers; neural networks are referred to as ‘deep’ when they contain at least two hidden layers. If you’re not familiar with artificial neural networks, then this free online book is a good source to start learning about them.

In this post, I will show you how artificial neural networks with different numbers of hidden layers compare, and I will also compare these networks to a convolutional network, which often performs better in the case of visual imagery. I will show you some basic models and how to code these, but will not spend too much time on tuning neural networks, for example when it comes to choosing the right amount of hidden layers or the number of nodes in each hidden layer. In essence, what it comes down to is that these parameters largely depend on your data structure, magnitude and complexity. The more hidden layers one adds, the more complex non-linear relationships can be modelled. Often, in my experience, adding hidden layers to a neural network increases their performance up to a certain number of layers, after which the increase becomes non-significant while the computational requirements and interpretation become more infeasible. It is up to you to play around a bit with your specific data and test how this trade-off works.

Although neural networks can easily built in RStudio using TensorFlow and Keras, I really want to show you one of the incredible features of RStudio where you can run Python in RStudio. This can be done in two ways: either we choose “Terminal” on the top of the output console in RStudio and run Python via Terminal, or we use the base system2() function to run Python in RStudio.

For the second option, to use the system2() command, it’s important to first check what version of Python should be used. You can check which versions of Python are installed on your machine by running python --version in Terminal. Note that with RStudio 1.1 (1.1.383 or higher), you can run in Terminal directly from RStudio on the “Terminal” tab. You can also run python3 --version to check if you have Python version 3 installed. On my machine, python --version and python3 --version return Python 2.7.16 and Python 3.7.0, respectively. You can then run which python (or which python3 if you have Python version 3 installed) in Terminal, which will return the path where Python is installed. In my case, these respective commands return /usr/bin/python and /Library/Frameworks/Python.framework/Versions/3.7/bin/python3. As I will make use of Python version 3, I specify the latter as the path to Python in the use\_python() function from the reticulate package. We can check whether the desired version of Python is used by using the sys package from Python. Just make sure to change the path in the code below to what version of Python you desire using and where that version in installed.

**library**(reticulate)

use\_python(python = '/Library/Frameworks/Python.framework/Versions/3.7/bin/python3')

sys <- import("sys")

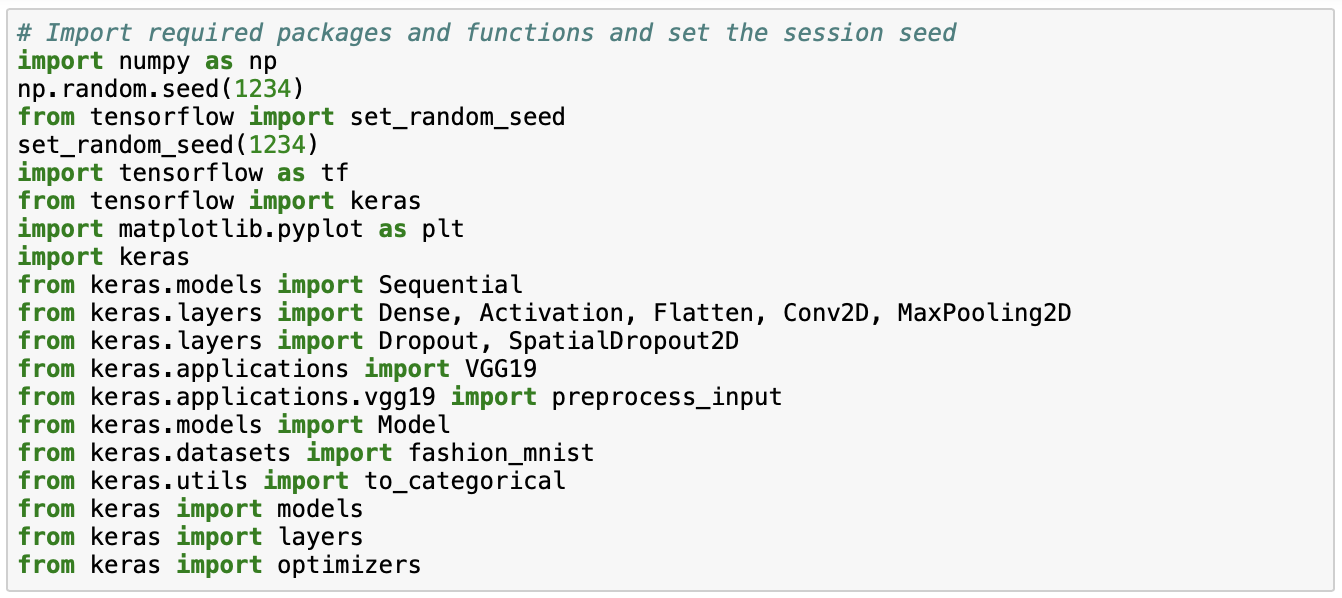
sys$version

Now that we’ve specified the correct version of Python to be used, we can run our Python script from RStudio using the system2() function. This function also takes an argument for the version of Python used, which in my case is Python version 3. If you are using an older version of Python, make sure to change "python3" in the command below to "python2".

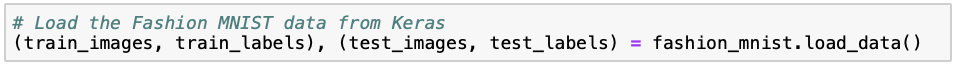
python\_file <- "simple\_neural\_network\_fashion\_mnist.py"

system2("python3", args = c(python\_file), stdout = NULL, stderr = "")

I will now guide you step by step through the script called in the command above. First, we load the required packages in Python and set the session seed for replicability.

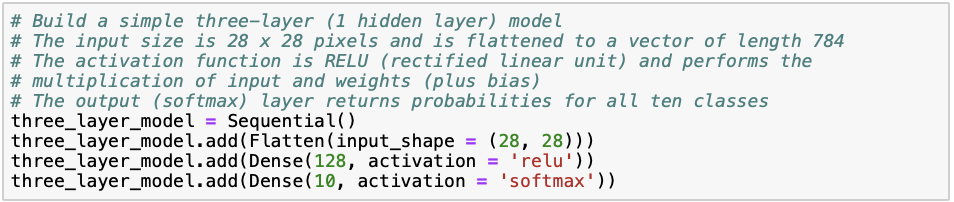
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic1.png)

We then load the fashion MNIST data from keras and we normalize the data by dividing by maximum opacity of 255.

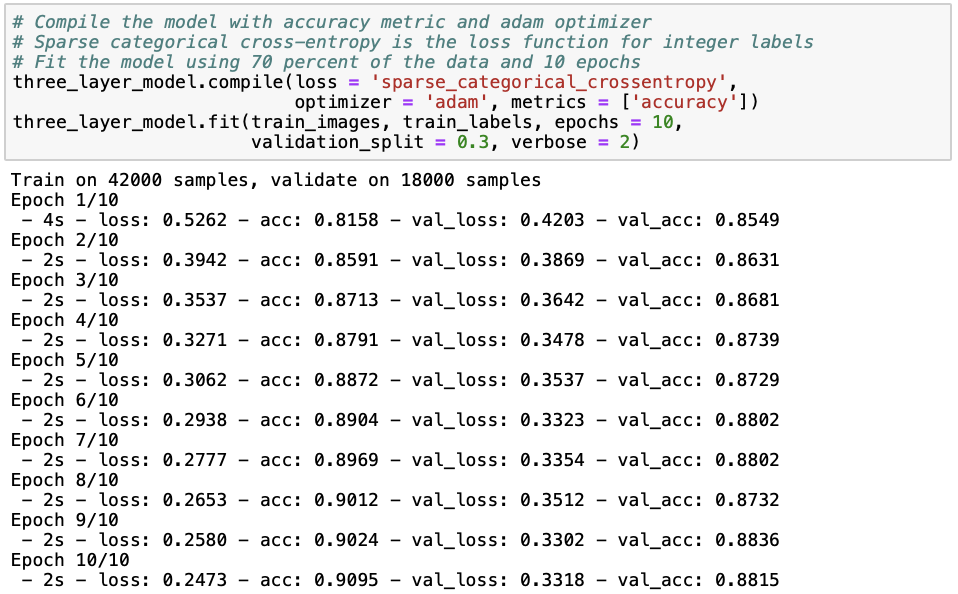
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic2.png)

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic3.png)

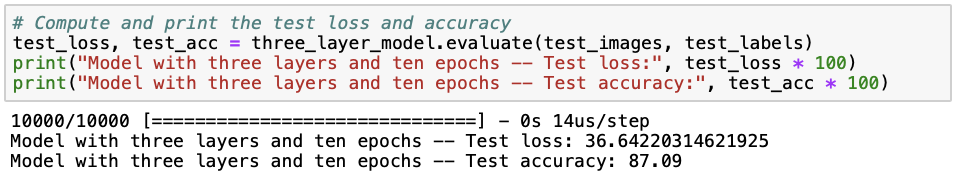
We start by building a simple neural network containing one hidden layer. Note that as here we use the untransformed but normalized data, we need to flatten the 28 by 28 pixels input first. We add one hidden densely-connected layer which performs output = relu(dot(input, kernel) + bias), where the rectified linear unit (relu) activation function has been proven to work well. We set the number of nodes equal to 128, because this seems to work well in our case. The number of nodes could essentially be any of the numbers 32, 64, 128, 256 and 512, as these are in a sequence of multiples between the number of nodes in the output (= 10) and input (= 784) layers. The softmax layer then assigns predicted probabilities to each of the ten clothing categories, which is also why there are ten nodes in this layer.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic4.png)

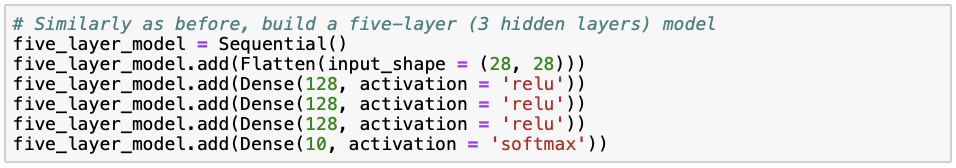
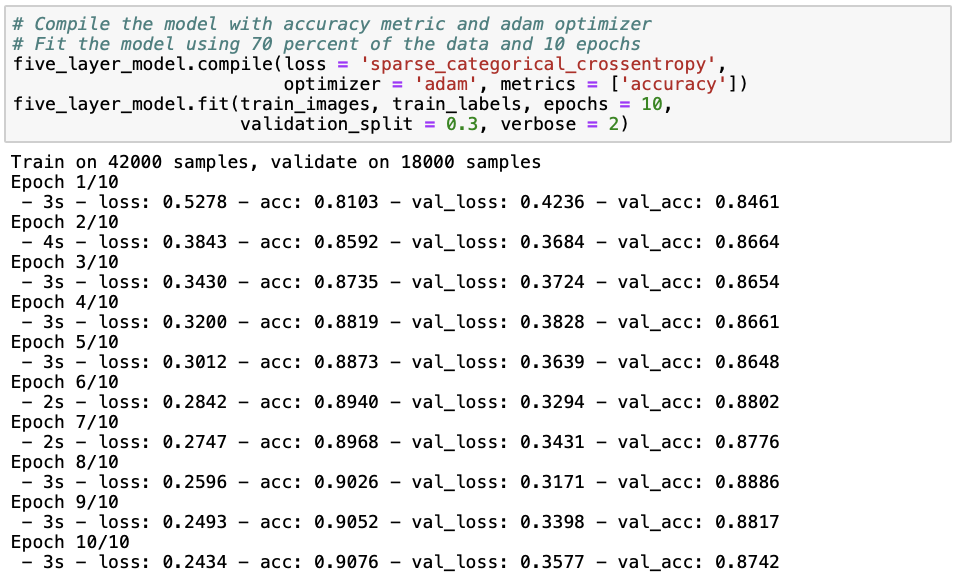
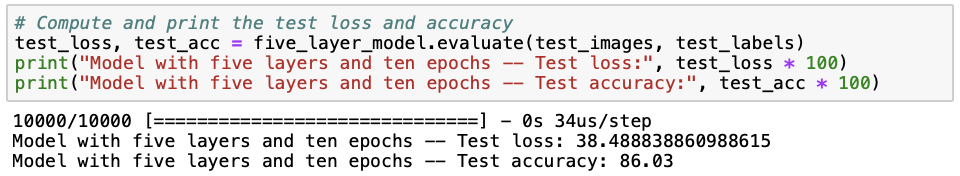
After building the neural network, we compile it. We specify sparse\_categorical\_crossentropy as the loss function, which is suitable for categorical multi-class responses. The optimizer controls the learning rate; adam (adaptive moment estimation) is similar to classical stochastic gradient descent and usually a safe choice for the optimizer. We set our metric of interest to be the accuracy, or the percentage of correctly classified images. Hereafter, we fit the model onto our training data set using ten iterations through the training data (“epochs”). Here, 70% is used for training and 30% is used for validation.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic5.png)

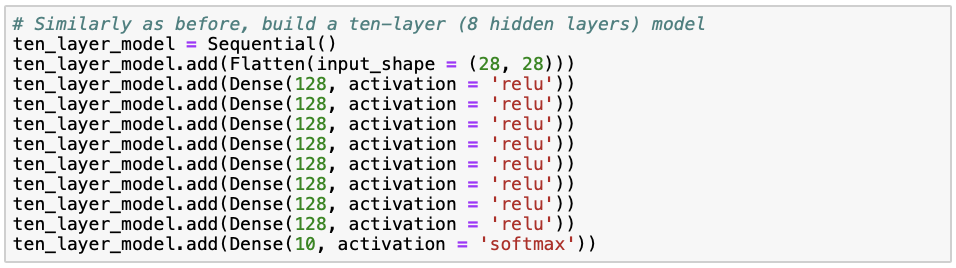
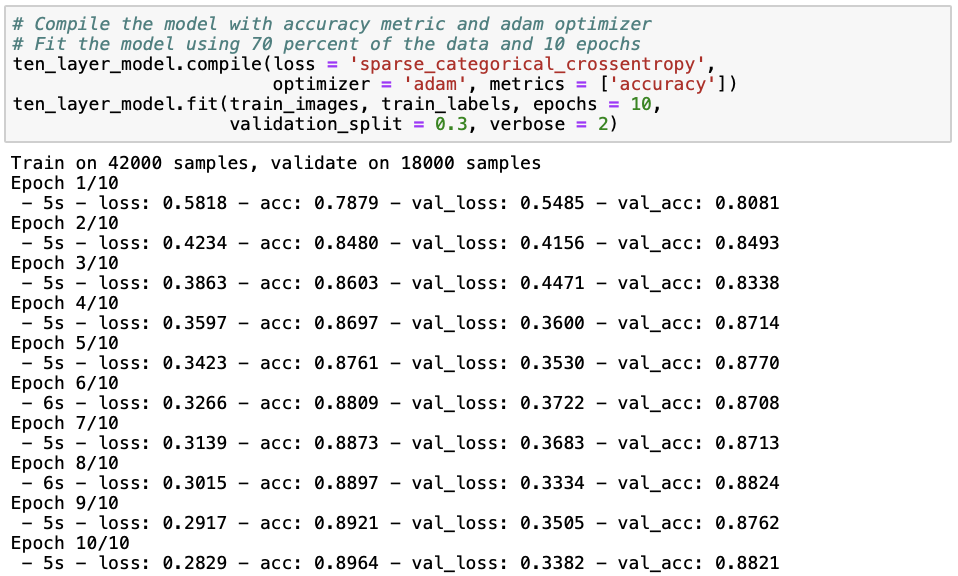
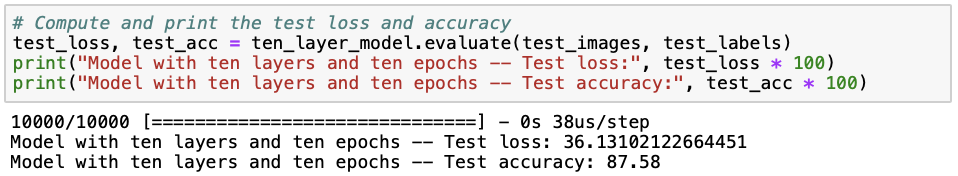
Next, we print the results of the model in terms of training and testing loss and accuracy.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic6.png)

We can see that the neural network with one hidden layer already performs relatively well with a test accuracy of 87.09%. However, it seems like we are slightly overfitting (i.e. the model is fitted too well to a particular data set and therefore does not well extend to other data sets), as the training set accuracy (88.15%) is slightly higher than the test set accuracy. There are several ways to avoid overfitting in neural networks, such as simplifying our model by reducing the number of hidden layers and neurons, adding dropout layers that randomly remove some of the connections between layers, and early stopping when validation loss starts to increase. Later on in this post, I will demonstrate some of these methods to you. Instead, to see whether a deep neural network performs better at predicting clothing categories, we build a neural network with three hidden layers in a similar way as before.

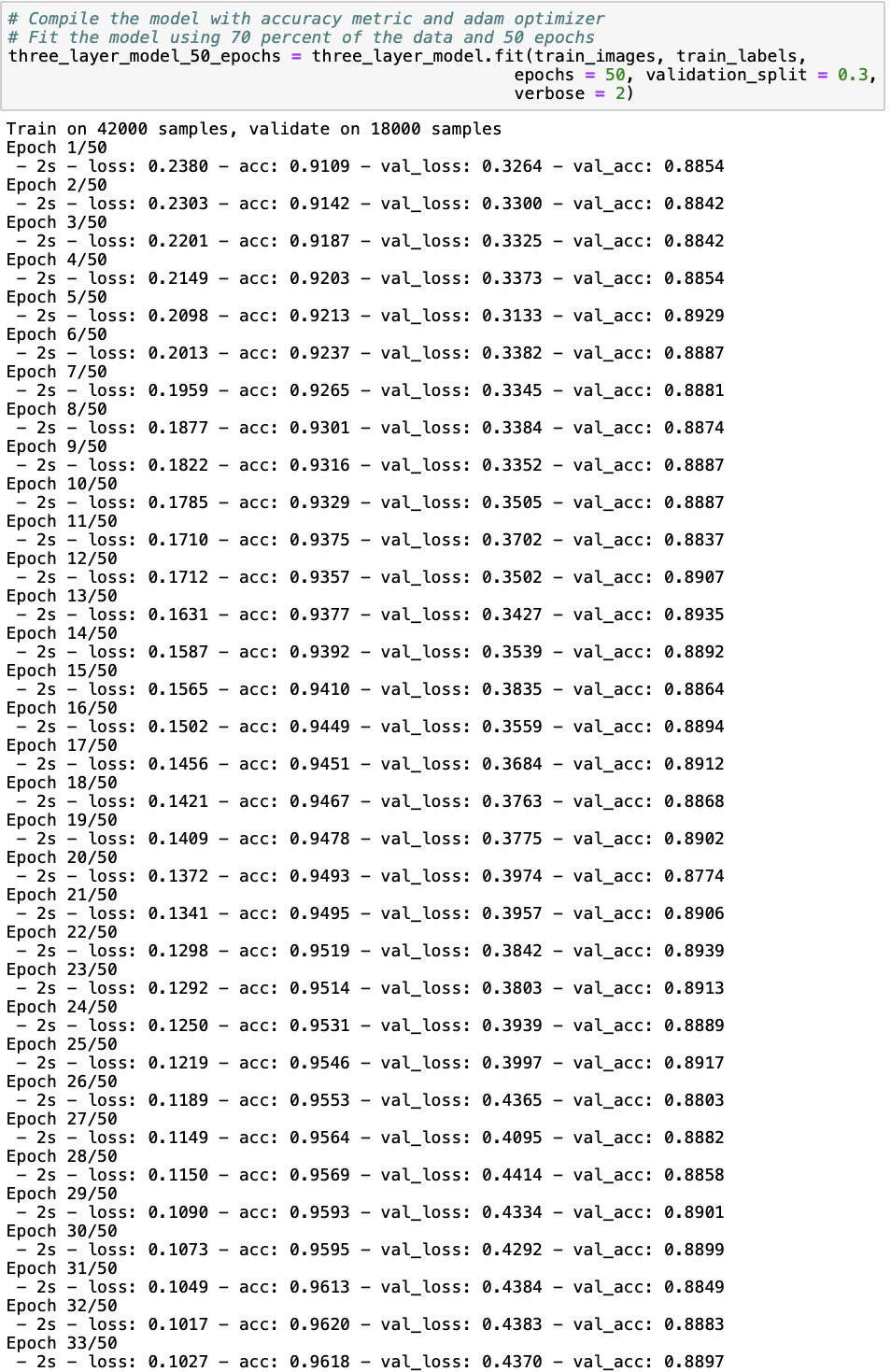
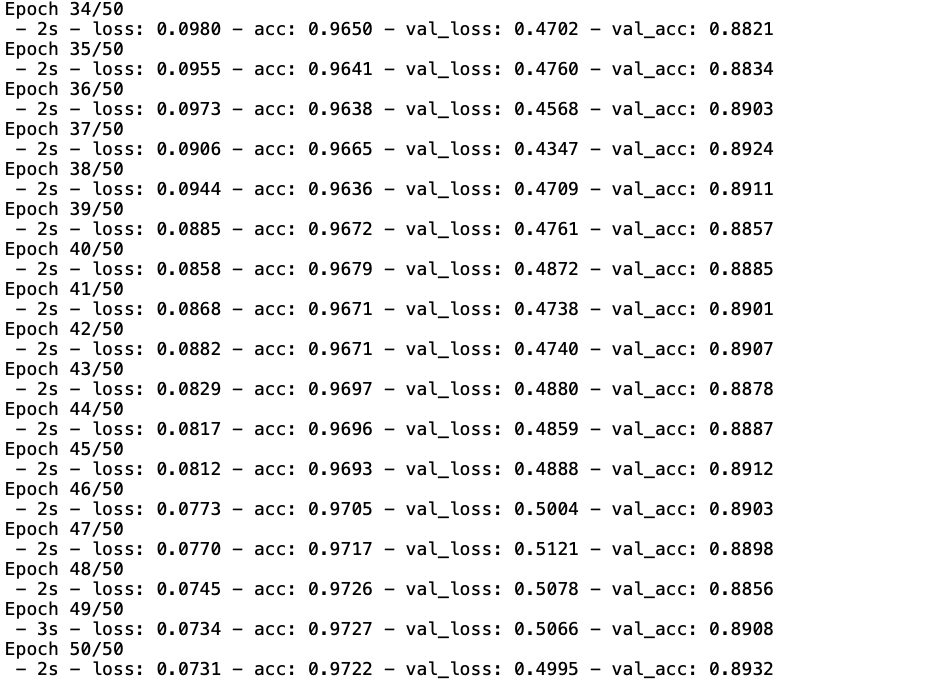
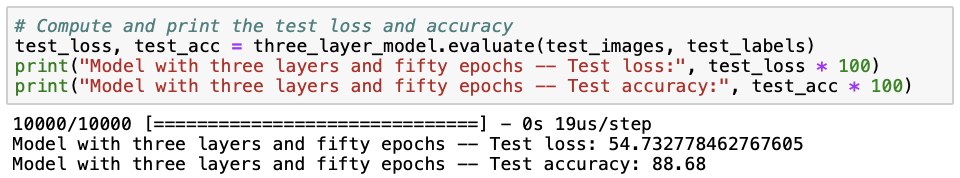
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic7.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic8.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic9.png)

It seems like the model with two additional layers does not perform better than the previous one with only one hidden layer, given that both the training (87.42%) and test set (86.03%) accuracies are lower and the loss (38.49) is higher. Let’s try whether adding another five hidden layers improves model performance, or whether we can include that increasing model complexity does not improve performance.

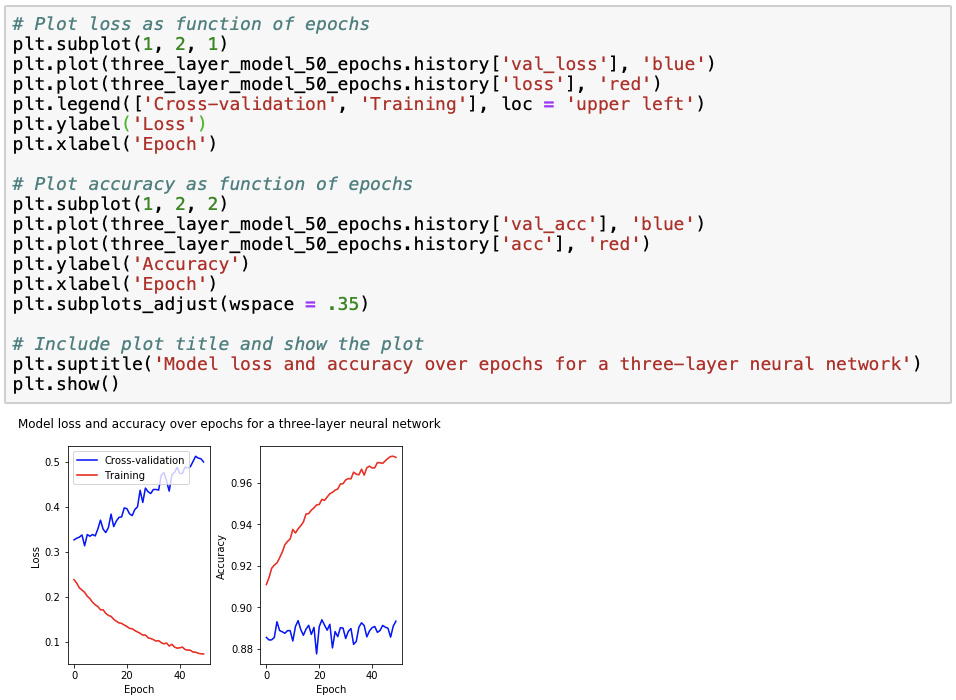
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic10.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic11.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic12.png)

The model with eight hidden layers performs best in terms of training (88.21%) and test (87.58%) accuracy as well as loss (36.12). Nevertheless, the difference in performance between the first model with one hidden layer and the current model with eight hidden layers is only quite small. Although it seems that with so many hidden layers, we can model additional complexity that improves the accuracy of the model, we must ask ourselves whether increasing model complexity at the cost of interpretability and computational feasibility is worth this slight improvement in accuracy and loss.

Now that we have seen how the number of hidden layers affects model performance, let’s try and see whether increasing the number of epochs (i.e. the number of times the model iterates through the training data) from ten to fifty improves the performance of our first neural network with one hidden layer.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic13.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic14.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic15.png)

The three-layer model trained with fifty epochs has the highest train (89.32%) and test (88.68%) accuracies we have seen so far. However, the loss (54.73) is also about a third larger than we have seen before. Additionally, the model is also less time-efficient, given that the increase in accuracy is not substantial but the model takes significantly longer to fit. To better understand the trade-off between minimizing loss and maximizing accuracy, we plot model loss and accuracy over the number of epochs for the training and cross-validation data.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic16.png)

We observe that for the training data, loss decreases to zero while accuracy increases to one, as a result of overfitting. This is why we also check how the model performs on the cross-validation data, for which we observe that loss increases with the number of epochs while accuracy remains relatively stable. Using this figure, we can select an “optimal” number of epochs such that accuracy is maximized while loss is minimized. Looking at the cross-validation data accuracy, we see that the accuracy peak lays at around 20 epochs, for which loss is approximately 0.4. However, similar accuracies but much lower losses and modelling time are achieved with around 6 and 12 epochs, and so we might rather choose to train our model with around 6 or 20 epochs.

Regarding the model output, the predictions returned are probabilities per class or clothing category. We can calculate the majority vote by taking class that has the maximum of predicted probabilities of all classes. We can print the first ten elements of the majority\_vote dictionary, which we can obtain as follows:

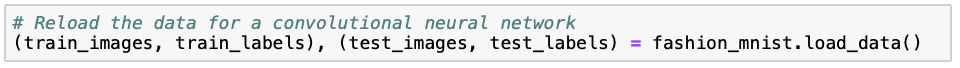
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic17.png)

All except the fifth (number 4) prediction are correct. In the fifth prediction, a shirt (category 6) is being misclassified as a top (category 0).

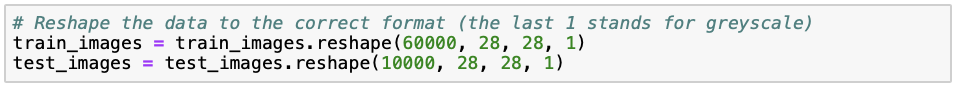
Convolutional Neural Network

I also wanted to show you how to build a convolutional neural network and compare its performance to the neural networks presented earlier, mostly because convolutional neural networks have generally been shown to perform better on visual image data. Essentially, what happens in a convolutional neural network is that a smaller matrix (the “filter matrix” or “kernel”) slides over the full image matrix, moving pixel by pixel, multiplies the filter matrix with the part of the full image matrix covered by the filter matrix on that moment, sums up these values and then repeats this until the full image matrix has been covered. For a more extensive explanation on how convolutional neural networks, I refer you to this page or this page.

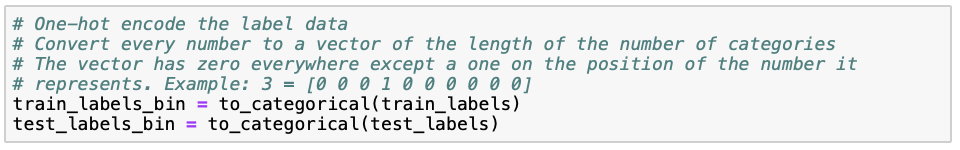
As we need to prepare our data slightly differently for a convolutional neural network, we reload the data and reshape the images to “flatten” them. The last “1” in the reshape dimensions stand for a greyscale, as we have images on a black-to-white scale. If we would have RGB images, we would change the “1” into a “3”.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic18.png)

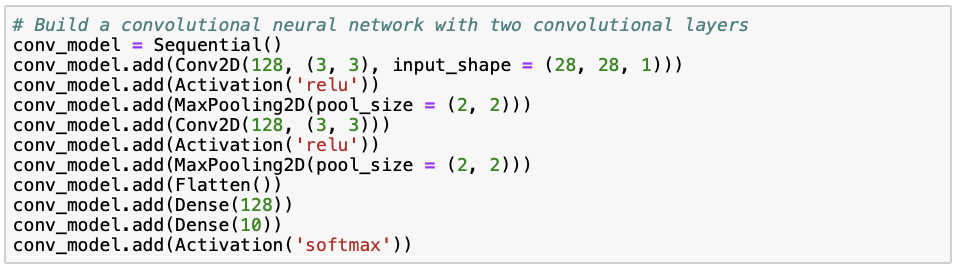
We make sure the the values of the pixels, ranging from zero to 255, are of the float type and then we normalize the values as before.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic19.png)[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic20.png)

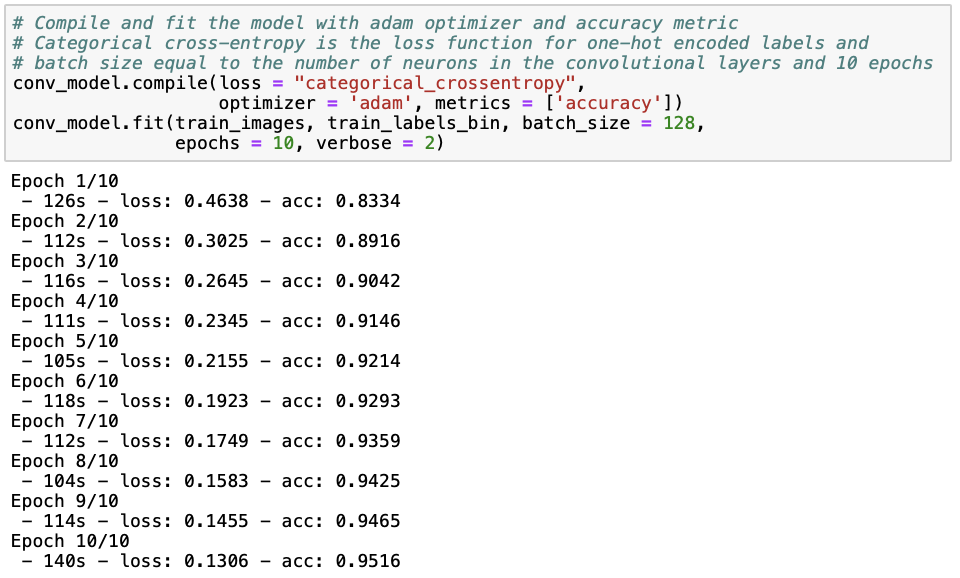
The convolutional neural network cannot deal with categorical labels. Therefore, we transform the labels to binary vectors, where all vectors have length ten (as there are ten categories), a “1” at the index of the category and zeros elsewhere. For example, category 3 and 8 would be coded as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] and [0, 0, 0, 0, 0, 0, 0, 0, 1, 0], respectively. This transformation is referred to as “one hot encoding”.

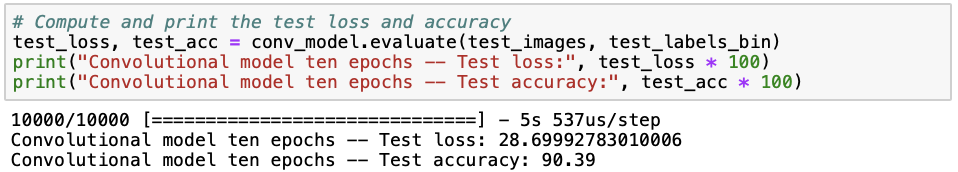
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic21.png)

Now, we can start building our convolutional neural network. The first layer Conv2D is a convolutional layer that takes a 2-dimensional matrix of 28 by 28 pixels in greyscale (1) as input. As before, we use 128 nodes in this layer, as the size of the data is not extremely large and we want to avoid making our model unnecessarily complex. The filter matrix is of size 3 by 3, which is quite standard. As before, we use the rectified linear (“relu”) activation function. The MaxPooling2D layer reduces the dimensionality (and thus required computational power) by outputting the maximum of the part of the input image that is captured by the filter matrix. The Flatten layer simply flattens the result from the previous layer into a vector. As we saw before, the softmax layer then assigns predicted probabilities to each of the ten clothing categories. Note that we use the same optimizer and metric as before, but that we now use “categorical\_crossentropy” as the loss function instead of “sparse\_categorical\_crossentropy”. The reason for this is that the former works for one-hot encoded labels, whereas the other works for categorical labels.

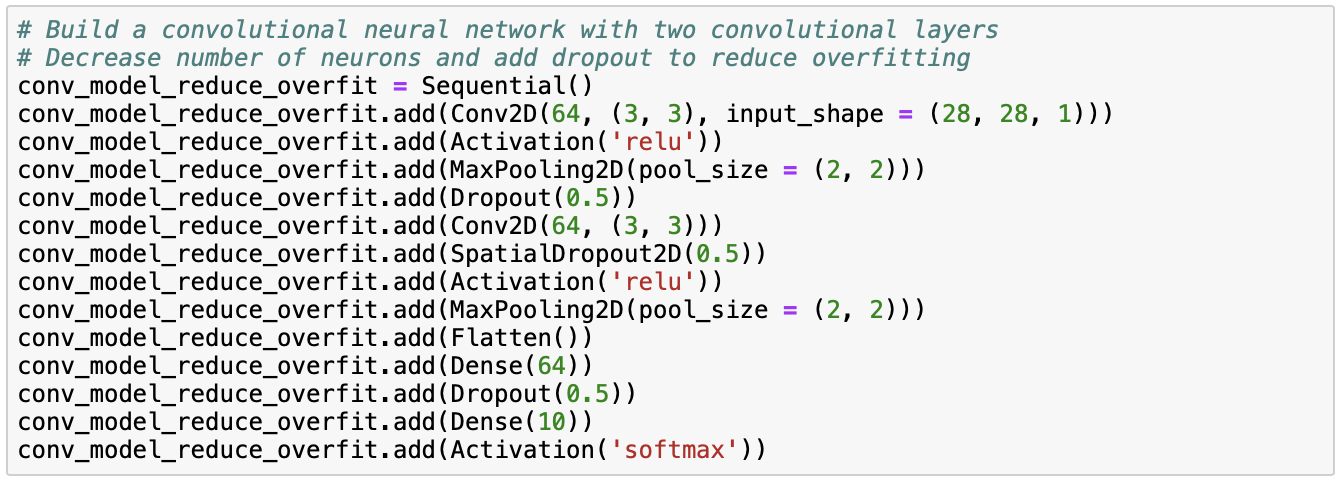
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic22.png)

We fit our model to the training data, where we set the batch\_size argument equal to the number of neurons in the convolutional layers (= 128).

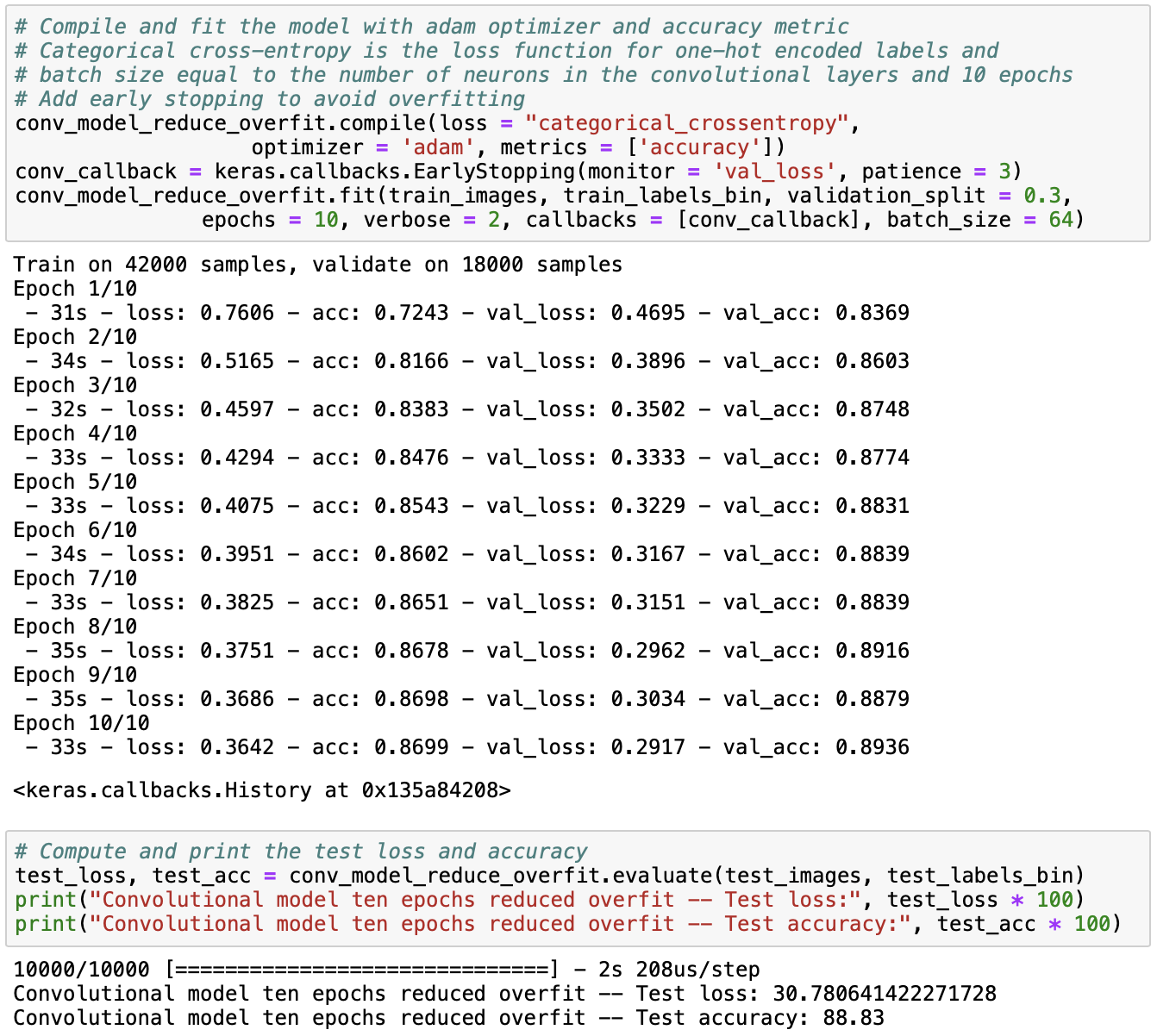
[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic23.png)

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic24.png)

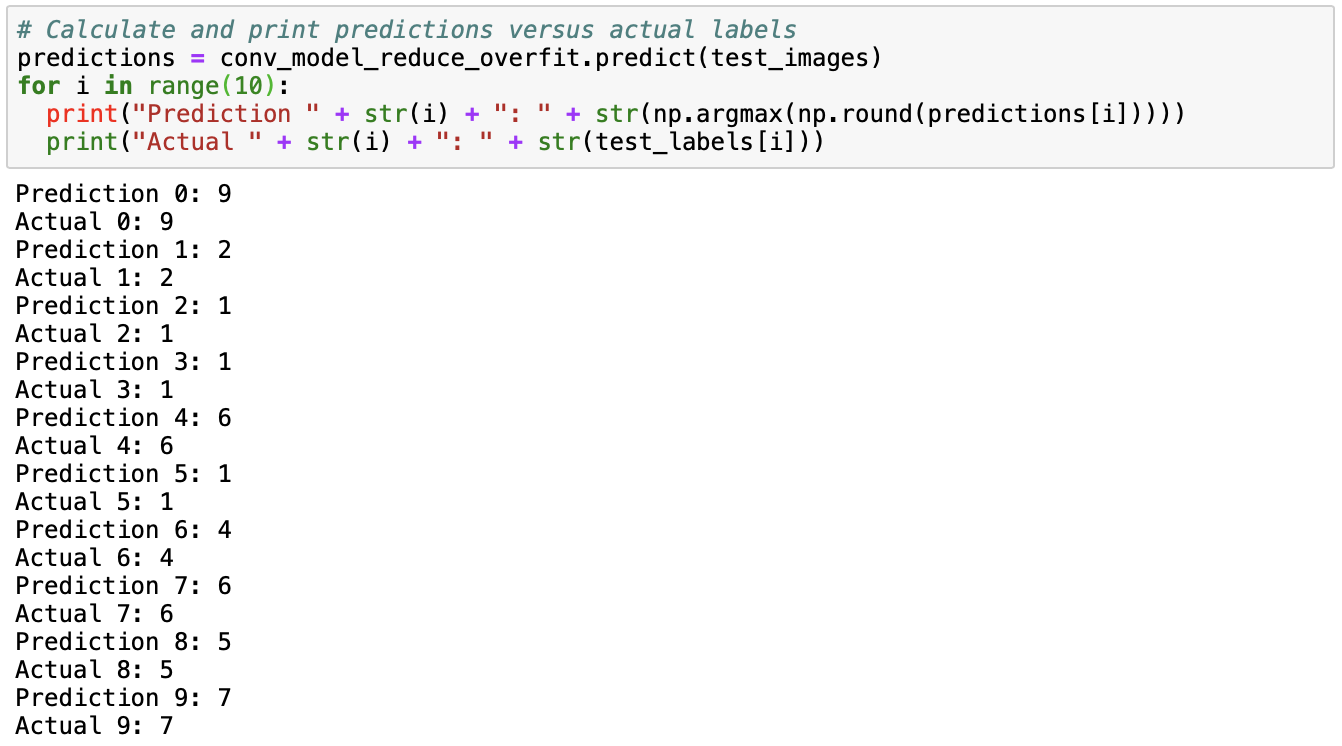
Although we are still overfitting, we observe that the convolutional neural network performs better than the neural networks we saw earlier, achieving a training set accuracy of 95.16% and a test set accuracy of 90.39%, and a lower loss of 28.70. This was to be expected, because convolutional neural networks have previously been shown to perform well on visual imagery data. Let’s see if we can reduce overfitting by reducing the number of neurons from 128 to 64, adding dropout layers and enabling early stopping. Note that the rate in the Dropout layer is the percentage of connections between layers that are being removed. the SpatialDropout2D is a special kind of dropout layer for convolutional neural networks, which drops certain multiplications of the filter matrix with parts of the original image before pooling across all movements over the original image.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic25.png)

When fitting our model, we also enable early stopping to reduce overfitting. Instead of going through all epochs specified, early stopping automatically stops the iterations through the epoch once it’s being noticed that the validation loss increases.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic26.png)

From the results, we observe that although the training and test accuracies have decreased, they are now much more similar than before. The test accuracy has not decreased substantially, but the training accuracy has, which means that overfitting is much less of a problem than before. Next, we can print the first ten predictions from the model and the first ten actual labels and compare them.

[](https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/screen_shots_python_code/pic27.png)

Comparing these predictions to the first ten labels in the data set, we observe that the first ten predictions are correct!

Python Script for Neural Network Fashion of MINST Database

|  |
| --- |
| # Import required packages and functions and set the session seed |
|  | import numpy as np |
|  | np.random.seed(1234) |
|  | from tensorflow import set\_random\_seed |
|  | set\_random\_seed(1234) |
|  | import tensorflow as tf |
|  | from tensorflow import keras |
|  | import matplotlib.pyplot as plt |
|  | import keras |
|  | from keras.models import Sequential |
|  | from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D |
|  | from keras.layers import Dropout, SpatialDropout2D |
|  | from keras.applications import VGG19 |
|  | from keras.applications.vgg19 import preprocess\_input |
|  | from keras.models import Model |
|  | from keras.datasets import fashion\_mnist |
|  | from keras.utils import to\_categorical |
|  | from keras import models |
|  | from keras import layers |
|  | from keras import optimizers |
|  |  |
|  | # Load the Fashion MNIST data from Keras |
|  | (train\_images, train\_labels), (test\_images, test\_labels) = fashion\_mnist.load\_data() |
|  |  |
|  | # Normalize the image data by dividing through the maximum pixel value (=255) |
|  | train\_images = train\_images / train\_images.max() |
|  | test\_images = test\_images / test\_images.max() |
|  |  |
|  | # Build a simple three-layer (1 hidden layer) model |
|  | # The input size is 28 x 28 pixels and is flattened to a vector of length 784 |
|  | # The activation function is RELU (rectified linear unit) and performs the |
|  | # multiplication of input and weights (plus bias) |
|  | # The output (softmax) layer returns probabilities for all ten classes |
|  | three\_layer\_model = Sequential() |
|  | three\_layer\_model.add(Flatten(input\_shape = (28, 28))) |
|  | three\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | three\_layer\_model.add(Dense(10, activation = 'softmax')) |
|  |  |
|  | # Compile the model with accuracy metric and adam optimizer |
|  | # Sparse categorical cross-entropy is the loss function for integer labels |
|  | # Fit the model using 70 percent of the data and 10 epochs |
|  | three\_layer\_model.compile(loss = 'sparse\_categorical\_crossentropy', |
|  | optimizer = 'adam', metrics = ['accuracy']) |
|  | three\_layer\_model.fit(train\_images, train\_labels, epochs = 10, |
|  | validation\_split = 0.3, verbose = 2) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = three\_layer\_model.evaluate(test\_images, test\_labels) |
|  | print("Model with three layers and ten epochs -- Test loss:", test\_loss \* 100) |
|  | print("Model with three layers and ten epochs -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Similarly as before, build a five-layer (3 hidden layers) model |
|  | five\_layer\_model = Sequential() |
|  | five\_layer\_model.add(Flatten(input\_shape = (28, 28))) |
|  | five\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | five\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | five\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | five\_layer\_model.add(Dense(10, activation = 'softmax')) |
|  |  |
|  | # Compile the model with accuracy metric and adam optimizer |
|  | # Fit the model using 70 percent of the data and 10 epochs |
|  | five\_layer\_model.compile(loss = 'sparse\_categorical\_crossentropy', |
|  | optimizer = 'adam', metrics = ['accuracy']) |
|  | five\_layer\_model.fit(train\_images, train\_labels, epochs = 10, |
|  | validation\_split = 0.3, verbose = 2) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = five\_layer\_model.evaluate(test\_images, test\_labels) |
|  | print("Model with five layers and ten epochs -- Test loss:", test\_loss \* 100) |
|  | print("Model with five layers and ten epochs -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Similarly as before, build a ten-layer (8 hidden layers) model |
|  | ten\_layer\_model = Sequential() |
|  | ten\_layer\_model.add(Flatten(input\_shape = (28, 28))) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(128, activation = 'relu')) |
|  | ten\_layer\_model.add(Dense(10, activation = 'softmax')) |
|  |  |
|  | # Compile the model with accuracy metric and adam optimizer |
|  | # Fit the model using 70 percent of the data and 10 epochs |
|  | ten\_layer\_model.compile(loss = 'sparse\_categorical\_crossentropy', |
|  | optimizer = 'adam', metrics = ['accuracy']) |
|  | ten\_layer\_model.fit(train\_images, train\_labels, epochs = 10, |
|  | validation\_split = 0.3, verbose = 2) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = ten\_layer\_model.evaluate(test\_images, test\_labels) |
|  | print("Model with ten layers and ten epochs -- Test loss:", test\_loss \* 100) |
|  | print("Model with ten layers and ten epochs -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Compile the model with accuracy metric and adam optimizer |
|  | # Fit the model using 70 percent of the data and 50 epochs |
|  | three\_layer\_model\_50\_epochs = three\_layer\_model.fit(train\_images, train\_labels, |
|  | epochs = 50, validation\_split = 0.3, |
|  | verbose = 2) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = three\_layer\_model.evaluate(test\_images, test\_labels) |
|  | print("Model with three layers and fifty epochs -- Test loss:", test\_loss \* 100) |
|  | print("Model with three layers and fifty epochs -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Plot loss as function of epochs |
|  | plt.subplot(1, 2, 1) |
|  | plt.plot(three\_layer\_model\_50\_epochs.history['val\_loss'], 'blue') |
|  | plt.plot(three\_layer\_model\_50\_epochs.history['loss'], 'red') |
|  | plt.legend(['Cross-validation', 'Training'], loc = 'upper left') |
|  | plt.ylabel('Loss') |
|  | plt.xlabel('Epoch') |
|  |  |
|  | # Plot accuracy as function of epochs |
|  | plt.subplot(1, 2, 2) |
|  | plt.plot(three\_layer\_model\_50\_epochs.history['val\_acc'], 'blue') |
|  | plt.plot(three\_layer\_model\_50\_epochs.history['acc'], 'red') |
|  | plt.ylabel('Accuracy') |
|  | plt.xlabel('Epoch') |
|  | plt.subplots\_adjust(wspace = .35) |
|  |  |
|  | # Include plot title and show the plot |
|  | plt.suptitle('Model loss and accuracy over epochs for a three-layer neural network') |
|  | plt.show() |
|  |  |
|  | # Calculate and print predictions versus actual labels |
|  | predictions = three\_layer\_model.predict(test\_images) |
|  | for i in range(10): |
|  | print("Prediction " + str(i) + ": " + str(np.argmax(np.round(predictions[i])))) |
|  | print("Actual " + str(i) + ": " + str(test\_labels[i])) |
|  |  |
|  | # Reload the data for a convolutional neural network |
|  | (train\_images, train\_labels), (test\_images, test\_labels) = fashion\_mnist.load\_data() |
|  |  |
|  | # Reshape the data to the correct format (the last 1 stands for greyscale) |
|  | train\_images = train\_images.reshape(60000, 28, 28, 1) |
|  | test\_images = test\_images.reshape(10000, 28, 28, 1) |
|  |  |
|  | # Convert the image data to numeric data and normalize them |
|  | train\_images = train\_images.astype('float32') |
|  | test\_images = test\_images.astype('float32') |
|  | train\_images = train\_images / train\_images.max() |
|  | test\_images = test\_images / test\_images.max() |
|  |  |
|  | # One-hot encode the label data |
|  | # Convert every number to a vector of the length of the number of categories |
|  | # The vector has zero everywhere except a one on the position of the number it |
|  | # represents. Example: 3 = [0 0 0 1 0 0 0 0 0 0] |
|  | train\_labels\_bin = to\_categorical(train\_labels) |
|  | test\_labels\_bin = to\_categorical(test\_labels) |
|  |  |
|  | # Build a convolutional neural network with two convolutional layers |
|  | conv\_model = Sequential() |
|  | conv\_model.add(Conv2D(128, (3, 3), input\_shape = (28, 28, 1))) |
|  | conv\_model.add(Activation('relu')) |
|  | conv\_model.add(MaxPooling2D(pool\_size = (2, 2))) |
|  | conv\_model.add(Conv2D(128, (3, 3))) |
|  | conv\_model.add(Activation('relu')) |
|  | conv\_model.add(MaxPooling2D(pool\_size = (2, 2))) |
|  | conv\_model.add(Flatten()) |
|  | conv\_model.add(Dense(128)) |
|  | conv\_model.add(Dense(10)) |
|  | conv\_model.add(Activation('softmax')) |
|  |  |
|  | # Compile and fit the model with adam optimizer and accuracy metric |
|  | # Categorical cross-entropy is the loss function for one-hot encoded labels and |
|  | # batch size equal to the number of neurons in the convolutional layers and 10 epochs |
|  | conv\_model.compile(loss = "categorical\_crossentropy", |
|  | optimizer = 'adam', metrics = ['accuracy']) |
|  | conv\_model.fit(train\_images, train\_labels\_bin, batch\_size = 128, |
|  | epochs = 10, verbose = 2) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = conv\_model.evaluate(test\_images, test\_labels\_bin) |
|  | print("Convolutional model ten epochs -- Test loss:", test\_loss \* 100) |
|  | print("Convolutional model ten epochs -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Build a convolutional neural network with two convolutional layers |
|  | # Decrease number of neurons and add dropout to reduce overfitting |
|  | conv\_model\_reduce\_overfit = Sequential() |
|  | conv\_model\_reduce\_overfit.add(Conv2D(64, (3, 3), input\_shape = (28, 28, 1))) |
|  | conv\_model\_reduce\_overfit.add(Activation('relu')) |
|  | conv\_model\_reduce\_overfit.add(MaxPooling2D(pool\_size = (2, 2))) |
|  | conv\_model\_reduce\_overfit.add(Dropout(0.5)) |
|  | conv\_model\_reduce\_overfit.add(Conv2D(64, (3, 3))) |
|  | conv\_model\_reduce\_overfit.add(SpatialDropout2D(0.5)) |
|  | conv\_model\_reduce\_overfit.add(Activation('relu')) |
|  | conv\_model\_reduce\_overfit.add(MaxPooling2D(pool\_size = (2, 2))) |
|  | conv\_model\_reduce\_overfit.add(Flatten()) |
|  | conv\_model\_reduce\_overfit.add(Dense(64)) |
|  | conv\_model\_reduce\_overfit.add(Dropout(0.5)) |
|  | conv\_model\_reduce\_overfit.add(Dense(10)) |
|  | conv\_model\_reduce\_overfit.add(Activation('softmax')) |
|  |  |
|  | # Compile and fit the model with adam optimizer and accuracy metric |
|  | # Categorical cross-entropy is the loss function for one-hot encoded labels and |
|  | # batch size equal to the number of neurons in the convolutional layers and 10 epochs |
|  | # Add early stopping to avoid overfitting |
|  | conv\_model\_reduce\_overfit.compile(loss = "categorical\_crossentropy", |
|  | optimizer = 'adam', metrics = ['accuracy']) |
|  | conv\_callback = keras.callbacks.EarlyStopping(monitor = 'val\_loss', patience = 3) |
|  | conv\_model\_reduce\_overfit.fit(train\_images, train\_labels\_bin, validation\_split = 0.3, |
|  | epochs = 10, verbose = 2, callbacks = [conv\_callback], batch\_size = 64) |
|  |  |
|  | # Compute and print the test loss and accuracy |
|  | test\_loss, test\_acc = conv\_model\_reduce\_overfit.evaluate(test\_images, test\_labels\_bin) |
|  | print("Convolutional model ten epochs reduced overfit -- Test loss:", test\_loss \* 100) |
|  | print("Convolutional model ten epochs reduced overfit -- Test accuracy:", test\_acc \* 100) |
|  |  |
|  | # Calculate and print predictions versus actual labels |
|  | predictions = conv\_model\_reduce\_overfit.predict(test\_images) |
|  | for i in range(10): |
|  | print("Prediction " + str(i) + ": " + str(np.argmax(np.round(predictions[i])))) |
|  | print("Actual " + str(i) + ": " + str(test\_labels[i]))  R Code for the above analysis   |  | | --- | | library(keras) | |  | install\_keras() | |  | fashion\_mnist = keras::dataset\_fashion\_mnist() | |  |  | |  | # Prepare data | |  | library(magrittr) | |  | c(train.images, train.labels) %<-% fashion\_mnist$train | |  | c(test.images, test.labels) %<-% fashion\_mnist$test | |  | train.images = data.frame(t(apply(train.images, 1, c))) / max(fashion\_mnist$train$x) | |  | test.images = data.frame(t(apply(test.images, 1, c))) / max(fashion\_mnist$train$x) | |  |  | |  | # Combine training and test images and labels | |  | pixs = ncol(fashion\_mnist$train$x) | |  | names(train.images) = names(test.images) = paste0('pixel', 1:(pixs^2)) | |  | train.labels = data.frame(label = factor(train.labels)) | |  | test.labels = data.frame(label = factor(test.labels)) | |  | train.data = cbind(train.labels, train.images) | |  | test.data = cbind(test.labels, test.images) | |  |  | |  | # Separate factor vectors with outcomes | |  | cloth\_cats = c('Top', 'Trouser', 'Pullover', 'Dress', 'Coat', | |  | 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Boot') | |  | train.classes = factor(cloth\_cats[as.numeric(as.character(train.labels$label)) + 1]) | |  | test.classes = factor(cloth\_cats[as.numeric(as.character(test.labels$label)) + 1]) | |  |  | |  | ################################################################################################ | |  | ############################################## PCA ############################################# | |  | ################################################################################################ | |  |  | |  | # Average pixel values over all images in training data set | |  | train.images.ave = data.frame(pixel = apply(train.images, 2, mean), | |  | x = rep(1:pixs, each = pixs), | |  | y = rep(1:pixs, pixs)) | |  |  | |  | # Plot average pixel values with custom plotting aesthetics | |  | library(ggplot2) | |  | my\_theme = function () { | |  | theme\_bw() + | |  | theme(axis.text = element\_text(size = 14), | |  | axis.title = element\_text(size = 14), | |  | strip.text = element\_text(size = 14), | |  | panel.grid.major = element\_blank(), | |  | panel.grid.minor = element\_blank(), | |  | panel.background = element\_blank(), | |  | legend.position = "bottom", | |  | strip.background = element\_rect(fill = 'white', colour = 'white')) | |  | } | |  | ggplot() + | |  | geom\_raster(data = train.images.ave, aes(x = x, y = y, fill = pixel)) + | |  | my\_theme() + | |  | labs(x = NULL, y = NULL, fill = "Average scaled pixel value") + | |  | ggtitle('Average image in Fashion MNIST training data') | |  |  | |  | # Obtain covariance matrix and principal components for training data | |  | library(stats) | |  | cov.train = cov(train.images) | |  | pca.train = prcomp(cov.train) | |  |  | |  | # Obtain cumulative proportion of variance against the number of principal components | |  | plotdf = data.frame(index = 1:(pixs^2), | |  | cumvar = summary(pca.train)$importance["Cumulative Proportion", ]) | |  | t(head(plotdf, 50)) | |  |  | |  | # Plot cumulative proportion of variance against the number of principal components | |  | ggplot() + | |  | geom\_point(data = plotdf, aes(x = index, y = cumvar), color = "red") + | |  | labs(x = "Index of primary component", y = "Cumulative proportion of variance") + | |  | my\_theme() + | |  | theme(strip.background = element\_rect(fill = 'white', colour = 'black')) | |  |  | |  | # Subset the first pca.dims principal components that explain at least 99.5% of the variance | |  | pca.dims = which(plotdf$cumvar >= .995)[1] | |  | pca.rot = pca.train$rotation[, 1:pca.dims] | |  |  | |  | # Multiply the image data by the rotation matrix to obtain transformed image data | |  | train.images.pca = data.frame(as.matrix(train.images) %\*% pca.rot) | |  | test.images.pca = data.frame(as.matrix(test.images) %\*% pca.rot) | |  |  | |  | # Combine image data with labels | |  | train.data.pca = cbind(train.images.pca, label = factor(train.data$label)) | |  | test.data.pca = cbind(test.images.pca, label = factor(test.data$label)) | |  |  | |  | ################################################################################################ | |  | ###################################### MODEL PERFORMANCE ####################################### | |  | ################################################################################################ | |  |  | |  | # Function to return model performance metrics | |  | model\_performance = function(fit, trainX, testX, trainY, testY, model\_name){ | |  |  | |  | # Predictions on train and test data for different types of models | |  | if (any(class(fit) == "rpart")){ | |  |  | |  | library(rpart) | |  | pred\_train = predict(fit, newdata = trainX, type = "class") | |  | pred\_test = predict(fit, newdata = testX, type = "class") | |  |  | |  | } else if (any(class(fit) == "train")){ | |  |  | |  | library(data.table) | |  | pred\_dt = as.data.table(fit$pred[, names(fit$bestTune)]) | |  | names(pred\_dt) = names(fit$bestTune) | |  | index\_list = lapply(1:ncol(fit$bestTune), function(x, DT, tune\_opt){ | |  | return(which(DT[, Reduce(`&`, lapply(.SD, `==`, tune\_opt[, x])), .SDcols = names(tune\_opt)[x]])) | |  | }, pred\_dt, fit$bestTune) | |  | rows = Reduce(intersect, index\_list) | |  | pred\_train = fit$pred$pred[rows] | |  | pred\_test = predict(fit, newdata = testX) | |  | trainY = fit$pred$obs[rows] | |  |  | |  | } else { | |  |  | |  | print(paste0("Error: Function evaluation unknown for object of type ", class(fit))) | |  | break | |  |  | |  | } | |  |  | |  | # Performance metrics on train and test data | |  | library(MLmetrics) | |  | df = data.frame(accuracy\_train = Accuracy(trainY, pred\_train), | |  | precision\_train = Precision(trainY, pred\_train), | |  | recall\_train = Recall(trainY, pred\_train), | |  | F1\_train = F1\_Score(trainY, pred\_train), | |  | accuracy\_test = Accuracy(testY, pred\_test), | |  | precision\_test = Precision(testY, pred\_test), | |  | recall\_test = Recall(testY, pred\_test), | |  | F1\_test = F1\_Score(testY, pred\_test), | |  | model = model\_name) | |  |  | |  | print(df) | |  |  | |  | return(df) | |  | } | |  |  | |  | ################################################################################################ | |  | ######################################### SINGLE TREES ######################################### | |  | ################################################################################################ | |  |  | |  | # Single tree | |  | library(rpart) | |  | set.seed(1234) | |  | tree = rpart(label ~., method = "class", data = train.data.pca) | |  | plotcp(tree) | |  | printcp(tree) | |  | mp.single.tree = model\_performance(tree, train.images.pca, test.images.pca, | |  | train.data.pca$label, test.data.pca$label, "single\_tree") | |  | save(tree, file = "saved\_objects/single\_tree\_pca.Rdata") | |  |  | |  | # Pruned tree | |  | set.seed(1234) | |  | prune.tree = prune(tree, cp = tree$cptable[which.min(tree$cptable[, "xerror"]), "CP"]) | |  | mp.single.tree = model\_performance(prune.tree, train.images.pca, test.images.pca, | |  | train.data.pca$label, test.data.pca$label, "pruned\_tree") | |  | save(prune.tree, file = "saved\_objects/prune\_tree\_pca.Rdata") | |  |  | |  | # Plot single and pruned trees | |  | par(mfrow = c(1, 2)) | |  | plot(tree, uniform = TRUE, main = "Classification Tree") | |  | text(tree, cex = 0.5) | |  | plot(prune.tree, uniform = TRUE, main = "Pruned Classification Tree") | |  | text(prune.tree, cex = 0.5) | |  | par(mfrow = c(1, 1)) | |  |  | |  | ################################################################################################ | |  | ######################################## RANDOM FORESTS ######################################## | |  | ################################################################################################ | |  |  | |  | # Random forest - random search for mtry acoss pca.dims values with 5-fold cross-validation | |  | library(caret) | |  | rf\_rand\_control = trainControl(method = "repeatedcv", | |  | search = "random", | |  | number = 5, | |  | repeats = 5, | |  | allowParallel = TRUE, | |  | savePredictions = TRUE) | |  | set.seed(1234) | |  | rf\_rand = train(x = train.images.pca, | |  | y = train.data.pca$label, | |  | method = "rf", | |  | ntree = 200, | |  | metric = "Accuracy", | |  | trControl = rf\_rand\_control, | |  | tuneLength = pca.dims) | |  | save(rf\_rand, file = "saved\_objects/rf\_rand\_pca.Rdata") | |  | print(rf\_rand) | |  | plot(rf\_rand) | |  | mp.rf.rand = model\_performance(rf\_rand, train.images.pca, test.images.pca, | |  | train.data.pca$label, test.data.pca$label, "random\_forest\_random") | |  |  | |  | # Random forest - grid search for mtry acoss values 1:pca.dims with 5-fold cross-validation | |  | rf\_grid\_control = trainControl(method = "repeatedcv", | |  | search = "grid", | |  | number = 5, | |  | repeats = 5, | |  | allowParallel = TRUE, | |  | savePredictions = TRUE) | |  | set.seed(1234) | |  | rf\_grid = train(x = train.images.pca, | |  | y = train.data.pca$label, | |  | method = "rf", | |  | ntree = 200, | |  | metric = "Accuracy", | |  | trControl = rf\_grid\_control, | |  | tuneGrid = expand.grid(.mtry = c(1:pca.dims))) | |  | save(rf\_grid, file = "saved\_objects/rf\_grid\_pca.Rdata") | |  | print(rf\_grid) | |  | plot(rf\_grid) | |  | mp.rf.grid = model\_performance(rf\_grid, train.images.pca, test.images.pca, | |  | train.data.pca$label, test.data.pca$label, "random\_forest\_grid") | |  |  | |  | # Obtain best model with highest accuracy | |  | rf\_models = list(rf\_rand$finalModel, rf\_grid$finalModel) | |  | rf\_accs = unlist(lapply(rf\_models, function(x){ sum(diag(x$confusion)) / sum(x$confusion) })) | |  | rf\_best = rf\_models[[which.max(rf\_accs)]] | |  |  | |  | # Plot forest size versus error and variable importance for best model | |  | library(randomForest) | |  | plot(rf\_best, main = "Relation between error and random forest size") | |  | varImpPlot(rf\_best) | |  |  | |  | # Obtain predicitons and confustion matrix from best model | |  | library(reshape2) | |  | pred = predict(rf\_best, test.images.pca, type = "class") | |  | conf = table(true = cloth\_cats[as.numeric(test.data.pca$label)], | |  | pred = cloth\_cats[as.numeric(pred)]) | |  | conf = data.frame(conf / rowSums(conf)) | |  |  | |  | # Plot the confusion matrix for a visual representation of model performance per class | |  | ggplot() + | |  | geom\_tile(data = conf, aes(x = true, y = pred, fill = Freq)) + | |  | labs(x = "Actual", y = "Predicted", fill = "Proportion") + | |  | my\_theme() + | |  | scale\_fill\_continuous(breaks = seq(0, 1, 0.25)) + | |  | theme(axis.text.x = element\_text(angle = 90, hjust = 1)) + | |  | coord\_fixed() | |  |  | |  | # Obtain required data to plot the ROC curves | |  | library(ROCR) | |  | library(plyr) | |  | pred\_roc = predict(rf\_best, test.images.pca, type = "prob") | |  | classes = unique(test.data.pca$label) | |  | classes = classes[order(classes)] | |  | plot\_list = list() | |  | for (i in 1:length(classes)) { | |  | actual = ifelse(test.data.pca$label == classes[i], 1, 0) | |  | pred = prediction(pred\_roc[, i], actual) | |  | perf = performance(pred, "tpr", "fpr") | |  | plot\_list[[i]] = data.frame(matrix(NA, nrow = length(perf@x.values[[1]]), ncol = 2)) | |  | plot\_list[[i]]['x'] = perf@x.values[[1]] | |  | plot\_list[[i]]['y'] = perf@y.values[[1]] | |  | } | |  | plotdf = rbind.fill(plot\_list) | |  | plotdf["Class"] = rep(cloth\_cats, unlist(lapply(plot\_list, nrow))) | |  |  | |  | # Plot the ROC curves | |  | ggplot() + | |  | geom\_line(data = plotdf, aes(x = x, y = y, color = Class)) + | |  | labs(x = "False positive rate", y = "True negative rate", color = "Class") + | |  | ggtitle("ROC curve per class") + | |  | theme(legend.position = c(0.85, 0.35)) + | |  | coord\_fixed() + | |  | my\_theme() | |  |  | |  | ################################################################################################ | |  | ########################################### BOOSTING ########################################### | |  | ################################################################################################ | |  |  | |  | # Boosting - grid search for parameters with 5-fold cross-validation | |  | xgb\_control = trainControl( | |  | method = "repeatedcv", | |  | number = 5, | |  | repeats = 5, | |  | classProbs = TRUE, | |  | allowParallel = TRUE, | |  | savePredictions = TRUE | |  | ) | |  |  | |  | xgb\_grid = expand.grid( | |  | nrounds = c(50, 100), | |  | max\_depth = seq(5, 15, 5), | |  | eta = c(0.002, 0.02, 0.2), | |  | gamma = c(0.1, 0.5, 1.0), | |  | colsample\_bytree = 1, | |  | min\_child\_weight = c(1, 2, 3), | |  | subsample = c(0.5, 0.75, 1) | |  | ) | |  |  | |  | set.seed(1234) | |  | xgb\_tune = train(x = train.images.pca, | |  | y = train.classes, | |  | method = "xgbTree", | |  | trControl = xgb\_control, | |  | tuneGrid = xgb\_grid | |  | ) | |  | xgb\_tune | |  | save(xgb\_tune, file = "saved\_objects/xgb\_tune\_pca.Rdata") | |  |  | |  | # Model performance metrics and results for boosting | |  | xgb\_tune$results[which.max(xgb\_tune$results$Accuracy), ] | |  | mp.xgb = model\_performance(xgb\_tune, train.images.pca, test.images.pca, | |  | train.classes, test.classes, "xgboost") | |  |  | |  | # Display the confusion matrix for the test data | |  | table(pred = predict(xgb\_tune, test.images.pca), | |  | true = test.classes) | |  |  | |  | ################################################################################################ | |  | #################################### SUPPORT VECTOR MACHINE #################################### | |  | ################################################################################################ | |  |  | |  | # Radial SVM - random search for C using 5-fold cross-validation and multi-class classification | |  | library(MLmetrics) | |  | svm\_control = trainControl(method = "repeatedcv", | |  | number = 5, | |  | repeats = 5, | |  | classProbs = FALSE, | |  | allowParallel = TRUE, | |  | summaryFunction = multiClassSummary, | |  | savePredictions = TRUE) | |  |  | |  | set.seed(1234) | |  | svm\_rand\_radial = train(label ~ ., | |  | data = cbind(train.images.pca, label = train.classes), | |  | method = "svmRadial", | |  | trControl = svm\_control, | |  | tuneLength = pca.dims, | |  | metric = "Accuracy") | |  | svm\_rand\_radial | |  | save(svm\_rand\_radial, file = "saved\_objects/svm\_rand\_radial\_pca.Rdata") | |  | mp.svm.rand.radial = model\_performance(svm\_rand\_radial, train.images.pca, test.images.pca, | |  | train.classes, test.classes, "svm\_random\_radial") | |  |  | |  | # Obtain predictions to plot confusion matrix for svm\_rand\_radial | |  | library(data.table) | |  | pred\_dt = as.data.table(svm\_rand\_radial$pred[, names(svm\_rand\_radial$bestTune)]) | |  | names(pred\_dt) = names(svm\_rand\_radial$bestTune) | |  | index\_list = lapply(1:ncol(svm\_rand\_radial$bestTune), function(x, DT, tune\_opt){ | |  | return(which(DT[, Reduce(`&`, lapply(.SD, `==`, tune\_opt[, x])), .SDcols = names(tune\_opt)[x]])) | |  | }, pred\_dt, svm\_rand\_radial$bestTune) | |  | rows = Reduce(intersect, index\_list) | |  | pred\_train = svm\_rand\_radial$pred$pred[rows] | |  | trainY = svm\_rand\_radial$pred$obs[rows] | |  | conf = table(pred\_train, trainY) | |  |  | |  | # Plot the confusion matrix in a tile plot | |  | conf = data.frame(conf / rowSums(conf)) | |  | ggplot() + | |  | geom\_tile(data = conf, aes(x = trainY, y = pred\_train, fill = Freq)) + | |  | labs(x = "Actual", y = "Predicted", fill = "Proportion") + | |  | my\_theme() + | |  | theme(axis.text.x = element\_text(angle = 90, hjust = 1)) + | |  | scale\_fill\_continuous(breaks = seq(0, 1, 0.25)) + | |  | coord\_fixed() | |  |  | |  | # Radial SVM - grid search for C and sigma using 5-fold cross-validation and multi-class classification | |  | svm\_grid\_radial = expand.grid(sigma = c(.01, 0.04, 0.1), C = c(0.01, 10, 32, 70, 150)) | |  | set.seed(1234) | |  | svm\_grid\_radial = train(label ~ ., | |  | data = cbind(train.images.pca, label = train.classes), | |  | method = "svmRadial", | |  | trControl = svm\_control, | |  | tuneGrid = svm\_grid\_radial, | |  | metric = "Accuracy") | |  | svm\_grid\_radial | |  | save(svm\_grid\_radial, file = "saved\_objects/svm\_grid\_radial\_pca.Rdata") | |  | mp.svm.grid.radial = model\_performance(svm\_grid\_radial, train.images.pca, test.images.pca, | |  | train.classes, test.classes, "svm\_grid\_radial") | |  |  | |  | # Plot accuracy as a function of cost, lines colored by sigma | |  | ggplot() + | |  | my\_theme() + | |  | geom\_line(data = svm\_grid\_radial$results, aes(x = C, y = Accuracy, color = factor(sigma))) + | |  | geom\_point(data = svm\_grid\_radial$results, aes(x = C, y = Accuracy, color = factor(sigma))) + | |  | labs(x = "Cost", y = "Cross-Validation Accuracy", color = "Sigma") + | |  | ggtitle('Relationship between cross-validation accuracy and values of cost and sigma') | |  |  | |  | # Linear SVM - grid search for C using 5-fold cross-validation and multi-class classification | |  | svm\_grid\_linear = expand.grid(C = c(1, 10, 32, 75, 150)) | |  | set.seed(1234) | |  | svm\_grid\_linear = train(label ~ ., | |  | data = cbind(train.images.pca, label = train.classes), | |  | method = "svmLinear", | |  | trControl = svm\_control, | |  | tuneGrid = svm\_grid\_linear, | |  | metric = "Accuracy") | |  | svm\_grid\_linear | |  | save(svm\_grid\_linear, file = "saved\_objects/svm\_tune\_grid\_linear\_pca.Rdata") | |  | mp.svm.grid.linear = model\_performance(svm\_grid\_linear, train.images.pca, test.images.pca, | |  | train.classes, test.classes, "svm\_grid\_linear") | |  |  | |  | # Boxplots of resampled accuracy for linear and radial Kernel SVMs | |  | resamp\_val = resamples(list(svm\_radial = svm\_grid\_radial, svm\_linear = svm\_grid\_linear)) | |  | plotdf = data.frame(Accuracy = c(resamp\_val$values$`svm\_radial~Accuracy`, resamp\_val$values$`svm\_linear~Accuracy`), | |  | Model = rep(c("Radial Kernel", "Linear Kernel"), rep(nrow(resamp\_val$values), 2))) | |  | ggplot() + | |  | geom\_boxplot(data = plotdf, aes(x = Model, y = Accuracy)) + | |  | ggtitle('Resample accuracy for SVM with linear and radial Kernel') + | |  | my\_theme() | |  |  | |  | ################################################################################################ | |  | #################################### WRAPPING EVERYTHING UP #################################### | |  | ################################################################################################ | |  |  | |  | # Data for resampled accuracies | |  | model\_list = list(rf\_rand, rf\_grid, xgb\_tune, svm\_rand\_radial, svm\_grid\_radial, svm\_grid\_linear) | |  | names(model\_list) = c(paste0('Random forest ', c("(random ", "(grid "), "search)"), "Gradient-boosted trees", | |  | paste0('Radial support vector machine ', c("(random ", "(grid "), "search)"), | |  | 'Linear support vector machine (grid search)') | |  | resamp = resamples(model\_list) | |  | accuracy\_variables = names(resamp$values)[grepl("Accuracy", names(resamp$values))] | |  | plotdf = melt(resamp$values[, c('Resample', accuracy\_variables)], | |  | id = "Resample", value.name = "Accuracy", variable.name = "Model") | |  | plotdf$Model = gsub("~.\*","", plotdf$Model) | |  |  | |  | # Box plot of resampled accuracies | |  | ggplot() + | |  | geom\_boxplot(data = plotdf, aes(x = Model, y = Accuracy, color = Model)) + | |  | ggtitle('Resampled accuracy for machine learning models estimated') + | |  | my\_theme() + | |  | theme(axis.text.x = element\_text(angle = 45, hjust = 1)) + | |  | labs(x = NULL, color = NULL) + | |  | guides(color = FALSE) | |  |  | |  | # Combine all model performance metrics for all models to obtain overview | |  | mp.df = rbind(mp.single.tree, mp.pruned.tree, mp.rf.rand, mp.rf.grid, mp.xgb, | |  | mp.svm.rand.radial, mp.svm.grid.radial, mp.svm.grid.linear) | |  | mp.df[order(mp.df$accuracy\_test, decreasing = TRUE), ] | |

Next, we used principal components analysis (PCA) to compress the clothing image data down from 784 to just 17 pixels.

**library**(keras)

**library**(magrittr)

**library**(ggplot2)

### **Data Preparation**

Let’s fetch the data again and prepare the training and test sets.

install\_keras()

fashion\_mnist = keras::dataset\_fashion\_mnist()

c(train.images, train.labels) %<-% fashion\_mnist$train

c(test.images, test.labels) %<-% fashion\_mnist$test

Next, we normalize the image data by dividing the pixel values by the maximum value of 255.

train.images = data.frame(t(apply(train.images, 1, c))) / max(fashion\_mnist$train$x)

test.images = data.frame(t(apply(test.images, 1, c))) / max(fashion\_mnist$train$x)

Now, we combine the training images train.images and labels train.labels as well as test images test.images and labels test.labels in separate data sets, train.data and test.data, respectively.

pixs = ncol(fashion\_mnist$train$x)

names(train.images) = names(test.images) = paste0('pixel', 1:(pixs^2))

train.labels = data.frame(label = factor(train.labels))

test.labels = data.frame(label = factor(test.labels))

train.data = cbind(train.labels, train.images)

test.data = cbind(test.labels, test.images)

As train.labels and test.labels contain integer values for the clothing category (i.e. 0, 1, 2, etc.), we also create objects train.classes and test.classes that contain factor labels (i.e. Top, Trouser, Pullover etc.) for the clothing categories. We will need these for some of the machine learning models later on.

cloth\_cats = c('Top', 'Trouser', 'Pullover', 'Dress', 'Coat',

'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Boot')

train.classes = factor(cloth\_cats[as.numeric(as.character(train.labels$label)) + 1])

test.classes = factor(cloth\_cats[as.numeric(as.character(test.labels$label)) + 1])

### **Principal Components Analysis**

Our training and test image data sets currently contain 784 pixels or variables. We may expect a large share of these pixels, especially those towards the boundaries of the images, to have relatively small variance, because most of the fashion items are centered in the images. In other words, there may be quite some redundant pixels in our data set. To check whether this is the case, let’s plot the average pixel value on a 28 by 28 grid. We first obtain the average pixel values and store these in train.images.ave, after which we plot these values on the grid. We also define a custom plotting theme, my\_theme, to make sure all our figures have the same aesthetics. Note that in the plot created, a higher cell (pixel) value means that the average value of that pixel is higher, and thus that the pixel is darker on average (as a pixel value of 0 refers to white and a pixel value of 255 refers to black).

train.images.ave = data.frame(pixel = apply(train.images, 2, mean),

x = rep(1:pixs, each = pixs),

y = rep(1:pixs, pixs))

my\_theme = **function** () {

theme\_bw() +

theme(axis.text = element\_text(size = 14),

axis.title = element\_text(size = 14),

strip.text = element\_text(size = 14),

panel.grid.major = element\_blank(),

panel.grid.minor = element\_blank(),

panel.background = element\_blank(),

legend.position = "bottom",

strip.background = element\_rect(fill = 'white', colour = 'white'))

}

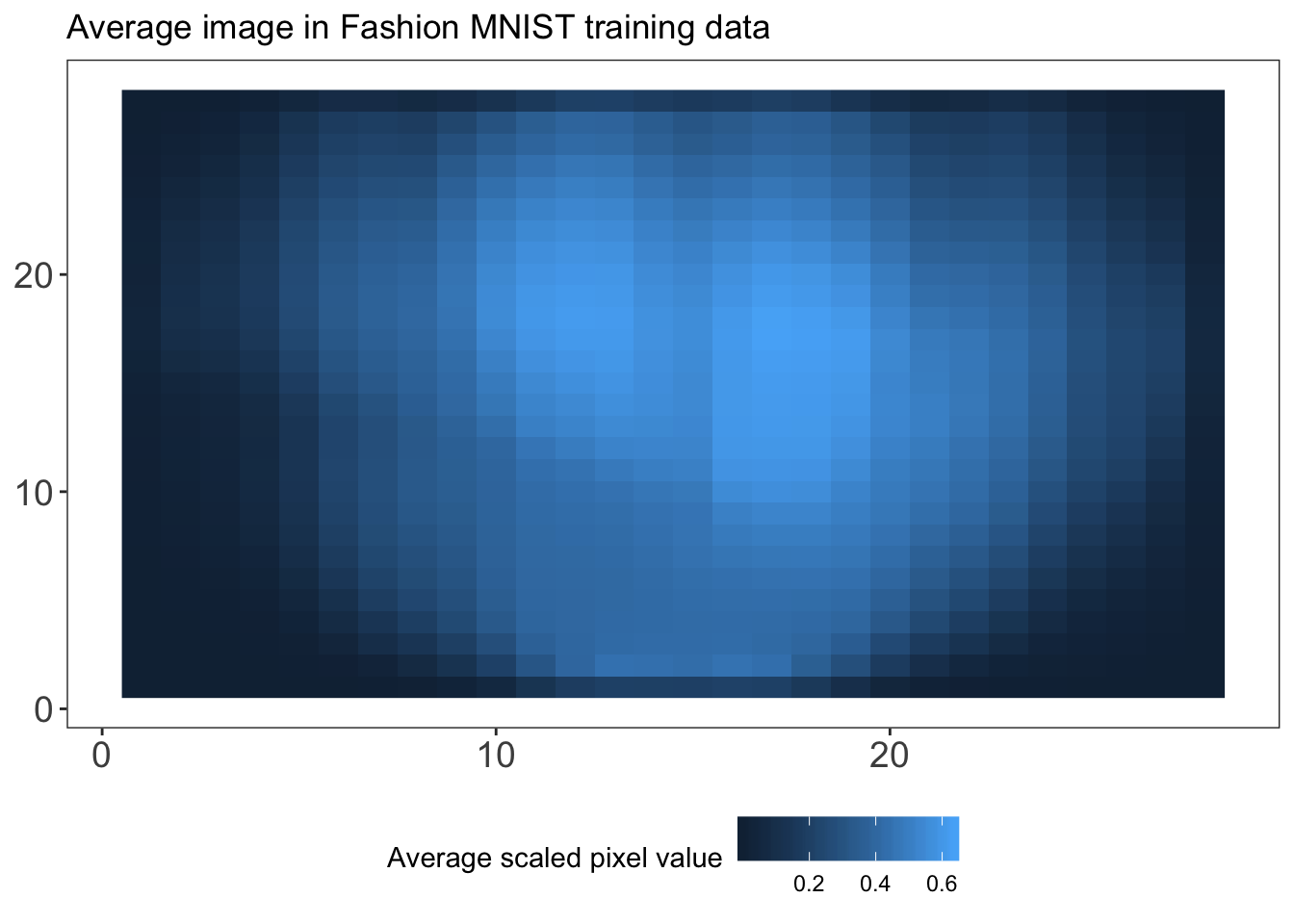
ggplot() +

geom\_raster(data = train.images.ave, aes(x = x, y = y, fill = pixel)) +

my\_theme() +

labs(x = NULL, y = NULL, fill = "Average scaled pixel value") +

ggtitle('Average image in Fashion MNIST training data')

[](https://rviews.rstudio.com/post/2020-02-24-predicting-clothing-classes-part-2/index_files/figure-html/unnamed-chunk-6-1.png)

As we can see from the plot, there are many pixels with a low average value, meaning that they are white in most of the images in our training data. These pixels are mostly redundant, while they do contribute to computational costs and sparsity. Therefore, we might be better off reducing the dimensionality in our data to reduce redundancy, overfitting and computational cost. One method to do so is principal components analysis (PCA). Essentially, PCA statistically reduces the dimensions of a set of correlated variables by transforming them into a smaller number of linearly uncorrelated variables. The resulting “principal components” are linear combinations of the original variables. The first principal component explains the largest part of the variance, followed by the second principal component and so forth. For a more extensive explanation of PCA, I refer you to James et al. (2013).

Let’s have a look at how many variables can explain which part of the variance in our data. We compute the 784 by 784 covariance matrix of our training images using the cov() function, after which we execute PCA on the covariance matrix using the prcomp() function in the stats library. Looking at the results, we observe that 50 principal components in our data explain 99.902% of the variance in the data. This can be nicely shown in a plot of the cumulative proportion of variance against component indices. Note that the component indices here are sorted by their ability to explain the variance in our data, and not based on their pixel position in the 28 by 28 image.

**library**(stats)

cov.train = cov(train.images)

pca.train = prcomp(cov.train)

plotdf = data.frame(index = 1:(pixs^2),

cumvar = summary(pca.train)$importance["Cumulative Proportion", ])

t(head(plotdf, 50))

## PC1 PC2 PC3 PC4 PC5 PC6 PC7 PC8 PC9

## index 1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000

## cumvar 0.6491 0.8679 0.9107 0.9421 0.9611 0.9759 0.9816 0.9862 0.9885

## PC10 PC11 PC12 PC13 PC14 PC15 PC16 PC17

## index 10.0000 11.0000 12.0000 13.0000 14.0000 15.0000 16.000 17.0000

## cumvar 0.9906 0.9918 0.9928 0.9935 0.9941 0.9945 0.995 0.9954

## PC18 PC19 PC20 PC21 PC22 PC23 PC24 PC25

## index 18.0000 19.000 20.0000 21.0000 22.0000 23.0000 24.000 25.0000

## cumvar 0.9957 0.996 0.9962 0.9965 0.9967 0.9969 0.997 0.9972

## PC26 PC27 PC28 PC29 PC30 PC31 PC32 PC33

## index 26.0000 27.0000 28.0000 29.0000 30.0000 31.000 32.000 33.0000

## cumvar 0.9974 0.9975 0.9976 0.9978 0.9979 0.998 0.998 0.9981

## PC34 PC35 PC36 PC37 PC38 PC39 PC40 PC41

## index 34.0000 35.0000 36.0000 37.0000 38.0000 39.0000 40.0000 41.0000

## cumvar 0.9982 0.9983 0.9984 0.9984 0.9985 0.9986 0.9986 0.9987

## PC42 PC43 PC44 PC45 PC46 PC47 PC48 PC49

## index 42.0000 43.0000 44.0000 45.0000 46.0000 47.0000 48.000 49.000

## cumvar 0.9987 0.9988 0.9988 0.9989 0.9989 0.9989 0.999 0.999

## PC50

## index 50.000

## cumvar 0.999

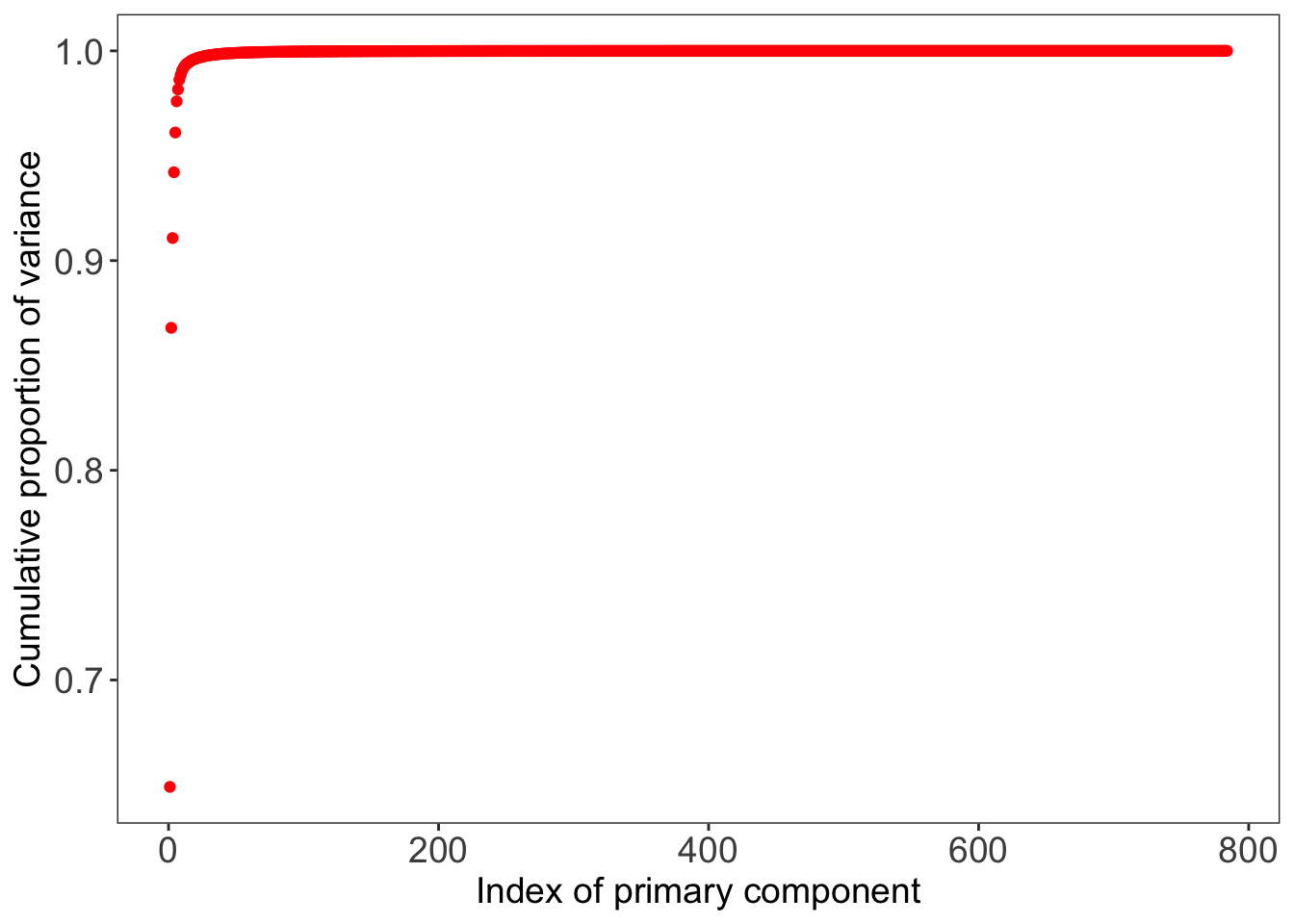
ggplot() +

geom\_point(data = plotdf, aes(x = index, y = cumvar), color = "red") +

labs(x = "Index of primary component", y = "Cumulative proportion of variance") +

my\_theme() +

theme(strip.background = element\_rect(fill = 'white', colour = 'black'))

[](https://rviews.rstudio.com/post/2020-02-24-predicting-clothing-classes-part-2/index_files/figure-html/unnamed-chunk-7-1.png)

We also observe that 99.5% of the variance is explained by only 17 principal components. As 99.5% is already a large share of the variance, and we want to reduce the number of pixels (variables) by as many as we can to reduce computation time for the models coming up, we choose to select these 17 components for further analysis. (Although this is unlikely to influence our results hugely, if you have more time I’d suggest you select the 50 components explaining 99.9% of the data, or execute the analyses on the full data set.)

We also save the relevant part of the rotation matrix created by the prcomp() function and stored in pca.train, such that its dimensions become 784 by 17. We then multiply our training and test image data by this rotation matrix called pca.rot. We further combine the transformed image data (train.images.pca and test.images.pca) with the integer labels for the clothing categories in train.data.pca and test.data.pca. We will use these reduced data in our further analyses to decrease computational time.

pca.dims = which(plotdf$cumvar >= .995)[1]

pca.rot = pca.train$rotation[, 1:pca.dims]

train.images.pca = data.frame(as.matrix(train.images) %\*% pca.rot)

test.images.pca = data.frame(as.matrix(test.images) %\*% pca.rot)

train.data.pca = cbind(train.images.pca, label = factor(train.data$label))

test.data.pca = cbind(test.images.pca, label = factor(test.data$label))

Next we saw that gradient-boosted trees and random forests achieve relatively high accuracy on dimensionality-reduced data, although not as high as the neural network.

Random forests use bootstrap aggregating to reduce the variance of the outcomes. In the first step, bootstrapping (sampling with replacement) is used to create B training sets from the population with the same size as the original training set. Hereafter, a separate tree for each of these training sets is built. Trees are grown using recursive binary splitting on the training data until a node reaches some minimum number of observations. The idea is that the tree should go from impure (equal mixing of classes) to pure (each leaf corresponds to one class exactly). The splits are determined such that they decrease variance, error and impurity. Random forests decorrelate the trees by considering only m of all p predictors as split candidates, whereby often m = sqrt(p).

Classification trees predict that each observation belongs to the most commonly occurring class (i.e. majority vote) of training observations in the region to which it belongs. The classification error rate is the fraction of the number of misclassified observations and the total number of classified observations. The Gini index and cross-entropy measures determine the level of impurity in order to decide on the best split at each node. In the final step, the average of the classification prediction results of all B trees is computed from the majority vote. The accuracy is computed as the out-of-bag (OOB) error and/or the test set error.

As each bootstrap samples from the training set with replacement, about 2⁄3 of the observations are not sampled and some are sampled multiple times. In the case of B trees in the forest, each observation is left out of approximately B/ 3 trees. The non-sampled observations are used as test set and the B/ 3 trees are used for out-of-sample predictions. In random forests, pruning is not needed as potential over-fitting is (partially) mitigated by the usage of bootstrapped samples and multiple decorrelated random trees.

We start by tuning the number of variables that are randomly sampled as candidates at each split,mtry. We make use of the caret framework, which makes it easy to train and evaluate a large number of different types of models. For random forests, we have the repeatedcv method perform five-fold cross-validation with five repetitions. For now, we build a random forest containing 200 trees because previous analyses with these data showed that the error does not decrease substantially when the number of trees is larger than 200, while a larger number of trees does require more computational power. We will see later on that 200 trees is indeed sufficient for this analysis. We let the algorithm determine what the best model is based on the accuracy metric, and we ask the algorithm to run the model for pca.dims (= 17) different values of mtry. We first specify the controls in rf\_rand\_control: we perform 5-fold cross-validation with 5 repeats (method = "cv", number = 5 and repeats = 5), allow parallel computation (allowParallel = TRUE) and save the predicted values (savePredictions = TRUE).

**library**(caret)

rf\_rand\_control = trainControl(method = "repeatedcv",

search = "random",

number = 5,

repeats = 5,

allowParallel = TRUE,

savePredictions = TRUE)

set.seed(1234)

rf\_rand = train(x = train.images.pca,

y = train.data.pca$label,

method = "rf",

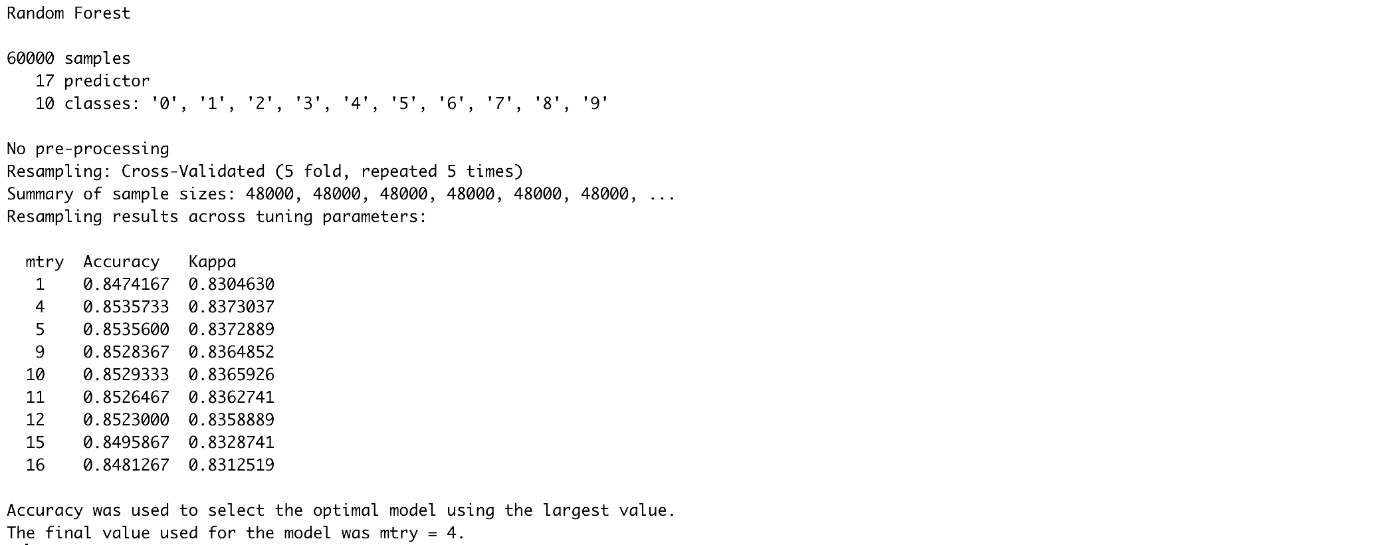
ntree = 200,

metric = "Accuracy",

trControl = rf\_rand\_control,

tuneLength = pca.dims)

print(rf\_rand)

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_rand_print.png)

We can check the model performance on both the training and test sets by means of different metrics using a custom function, model\_performance, which can be found on my Github.

mp.rf.rand = model\_performance(rf\_rand, train.images.pca, test.images.pca,

train.data.pca$label, test.data.pca$label, "random\_forest\_random")

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_rand_mp.png)

We can also use the caret framework to perform a grid search with pre-specified values for mtry rather than a random search as above.

rf\_grid\_control = trainControl(method = "repeatedcv",

search = "grid",

number = 5,

repeats = 5,

allowParallel = TRUE,

savePredictions = TRUE)

set.seed(1234)

rf\_grid = train(x = train.images.pca,

y = train.data.pca$label,

method = "rf",

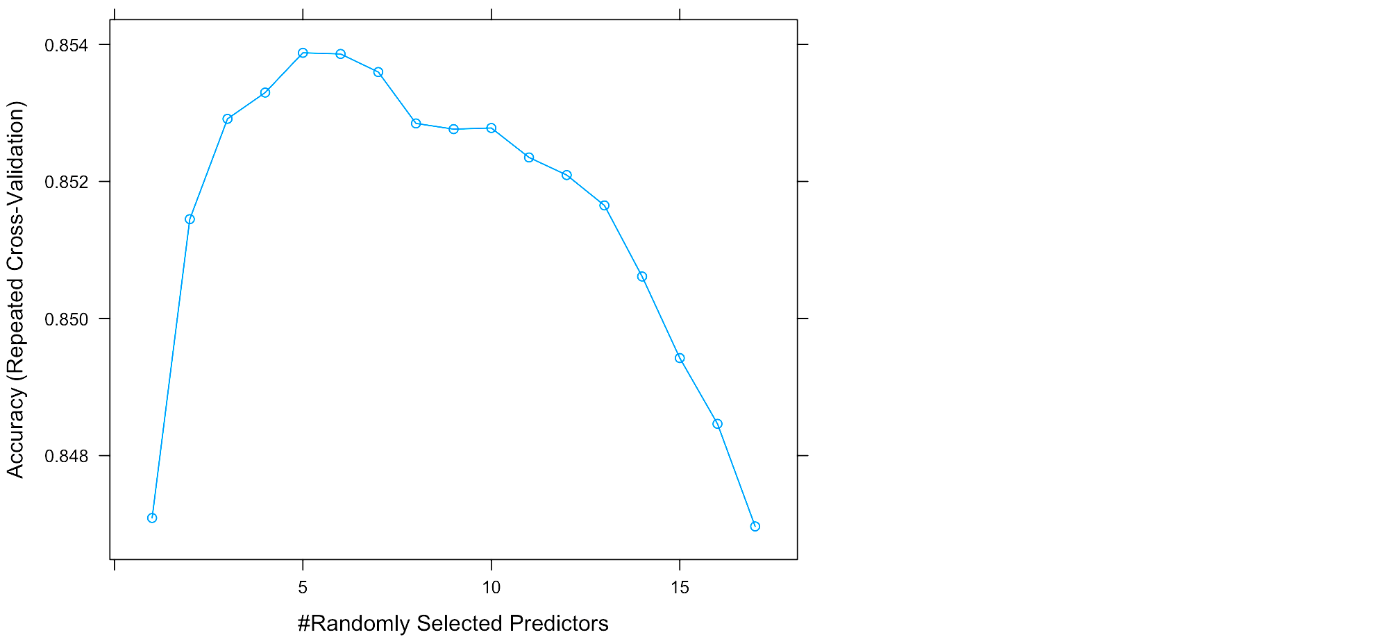
ntree = 200,

metric = "Accuracy",

trControl = rf\_grid\_control,

tuneGrid = expand.grid(.mtry = c(1:pca.dims)))

plot(rf\_grid)

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_grid_plot.png)

mp.rf.grid = model\_performance(rf\_grid, train.images.pca, test.images.pca,

train.data.pca$label, test.data.pca$label, "random\_forest\_grid")

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_grid_mp.png)

As shown by the results, the random search selects mtry=4 as the optimal parameter, resulting in 85% training and test set accuracies. The grid search selects mtry=5 and achieves similar accuracies for both values of 4 and 5 for mtry. We can see from the results that according to rf\_rand, mtry values of 4 and 5 lead to very similar results, which also goes for mtry values of 5 and 6 for rf\_grid. Although the results of rf\_rand and rf\_grid are very similar, we choose the best model on the basis of accuracy and save this in rf\_best. For this model, we’ll look at the relationship between the error and random forest size as well as the receiver operating characteristic (ROC) curves for every class. Let’s start by subtracting the best performing model from rf\_rand and rf\_grid.

rf\_models = list(rf\_rand$finalModel, rf\_grid$finalModel)

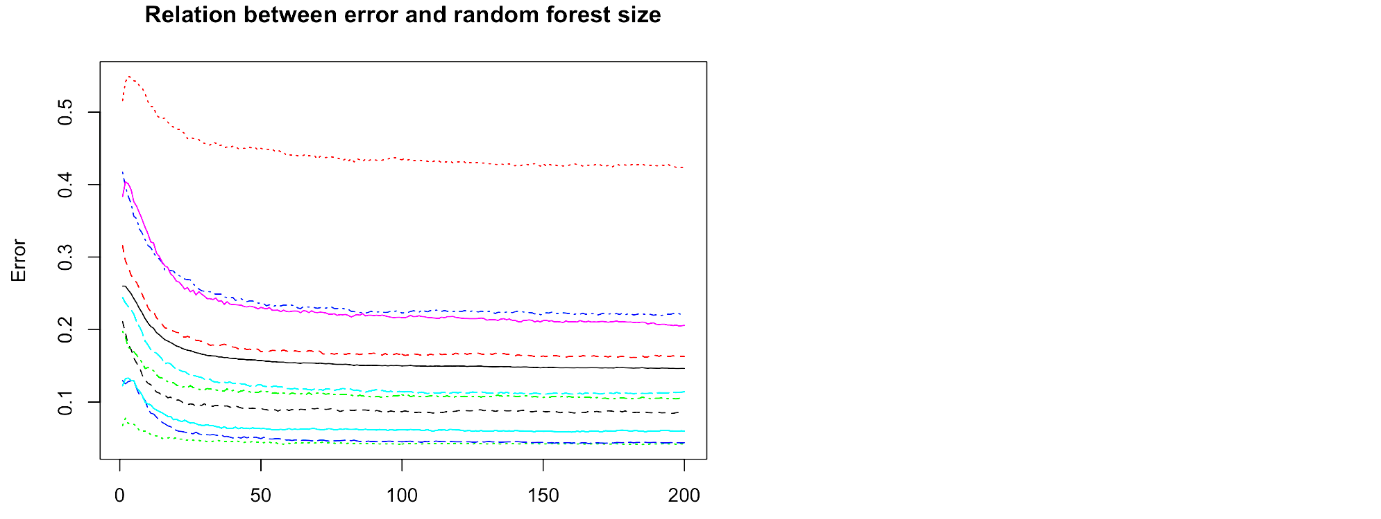
rf\_accs = unlist(lapply(rf\_models, **function**(x){ sum(diag(x$confusion)) / sum(x$confusion) }))

rf\_best = rf\_models[[which.max(rf\_accs)]]

Next, we plot the relationship between the size of the random forest and the error using the plot() function from the randomForest package.

**library**(randomForest)

plot(rf\_best, main = "Relation between error and random forest size")

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_error_trees.png)

We observe from this plot that the error does not decrease anymore for any of the classes after about 100 trees, and so we can conclude that our forest size of 200 is sufficient. We can also use the varImpPlot() function from the randomForest package to plot the importance for each variable. I will not show that here because it’s not as meaningful given that our variables are principal components of the actual pixels, but it’s good to keep in mind when extending these analyses to other data.

Finally, we plot the ROC curves for every class. On the x-axis of an ROC plot, we usually have the false positive rate (false positive / (true negative + false positive)) and on the y-axis the true positive rate (true positive / (true positive + false negative)). Essentially, the ROC plot helps us to compare the performance of our model with respect to predicting different classes. The area underneath each curve is the proportion of correct classifications for that particular class. Therefore, the further the curve is “drawn” towards the top left from the 45 degrees line, the better the classification for that class. We first need to obtain the data for the ROC curve for every class (or clothing category) in our data, which we bind together by rows, including a label for the classes.

**library**(ROCR)

**library**(plyr)

pred\_roc = predict(rf\_best, test.images.pca, type = "prob")

classes = unique(test.data.pca$label)

classes = classes[order(classes)]

plot\_list = list()

**for** (i **in** 1:length(classes)) {

actual = ifelse(test.data.pca$label == classes[i], 1, 0)

pred = prediction(pred\_roc[, i], actual)

perf = performance(pred, "tpr", "fpr")

plot\_list[[i]] = data.frame(matrix(NA, nrow = length(perf@x.values[[1]]), ncol = 2))

plot\_list[[i]]['x'] = perf@x.values[[1]]

plot\_list[[i]]['y'] = perf@y.values[[1]]

}

plotdf = rbind.fill(plot\_list)

plotdf["Class"] = rep(cloth\_cats, unlist(lapply(plot\_list, nrow)))

Next, we plot the ROC curves for every class. Note that we use the custom plotting theme my\_theme() as defined in the the second blog post of this series.

ggplot() +

geom\_line(data = plotdf, aes(x = x, y = y, color = Class)) +

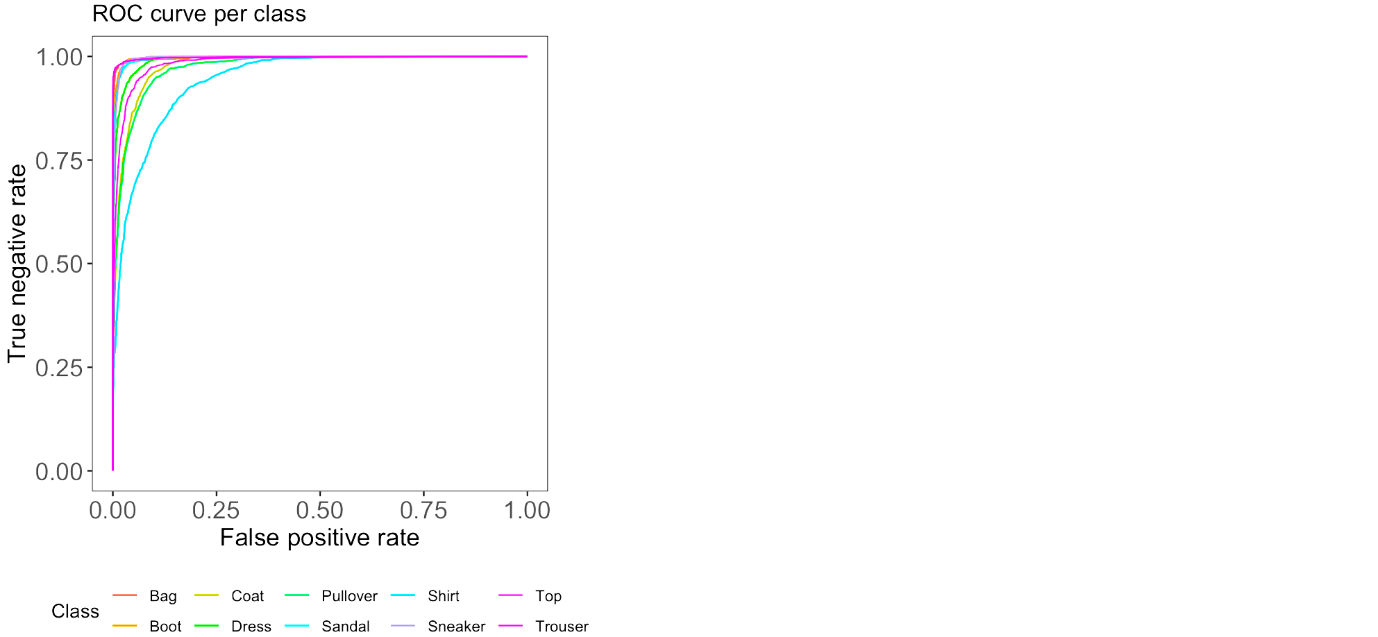
labs(x = "False positive rate", y = "True negative rate", color = "Class") +

ggtitle("ROC curve per class") +

theme(legend.position = c(0.85, 0.35)) +

coord\_fixed() +

my\_theme()

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/rf_roc.png)

We observe from the ROC curves that shirts and pullovers are most often misclassified, whereas trousers, bags, boots and sneakers are most often correctly classified. A possible explanation for this could be that shirts and pullovers can be very similar in shape to other categories, such as tops, coats and dresses; whereas bags, trousers, boots and sneakers are more dissimilar to other categories in the data.

## Gradient-Boosted Trees

While in random forests each tree is fully grown and trained independently with a random sample of data, in boosting every newly built tree incorporates the error from the previously built tree. That is, the trees are grown sequentially on an adapted version of the initial data, which does not require bootstrap sampling. Because of this, boosted trees are usually smaller and more shallow than the trees in random forests, improving the tree where it does not work well enough yet. Boosting is often said to outperform random forests, which is mainly because the approach learns slowly. This learning rate can be controlled by the shrinkage parameter, which we’ll tune later.

In boosting, it’s important to tune the parameters well and play around with different values of the parameters, which can easily be done using the caret framework. These parameters include the learning rate, eta, the minimal required loss reduction to further partition on a leaf node of the tree, gamma, the maximal depth of a tree max\_depth, the number of trees in the forest, nrounds, the minimum number of observations in the trees’ nodes, min\_child\_weight, the fraction of the training set observations randomly selected to grow trees, subsample, and the proportion of independent variables to use for each tree, colsample\_bytree. An overview of all parameters can be found here. Again, we use the caret framework to tune our boosting model.

xgb\_control = trainControl(

method = "cv",

number = 5,

classProbs = TRUE,

allowParallel = TRUE,

savePredictions = TRUE

)

Next, we define the possible combinations of the tuning parameters in the form of a grid, named xgb\_grid.

xgb\_grid = expand.grid(

nrounds = c(50, 100),

max\_depth = seq(5, 15, 5),

eta = c(0.002, 0.02, 0.2),

gamma = c(0.1, 0.5, 1.0),

colsample\_bytree = 1,

min\_child\_weight = c(1, 2, 3),

subsample = c(0.5, 0.75, 1)

)

We set the seed and then train the model onto the transformed principal components of the training data using xgb\_control and xgb\_grid as specified earlier. Note that because of the relatively large number of tuning parameters, and thus the larger number of possible combinations of these parameters (nrow(xgb\_grid) = 486), this may take quite a long time to run.

set.seed(1234)

xgb\_tune = train(x = train.images.pca,

y = train.classes,

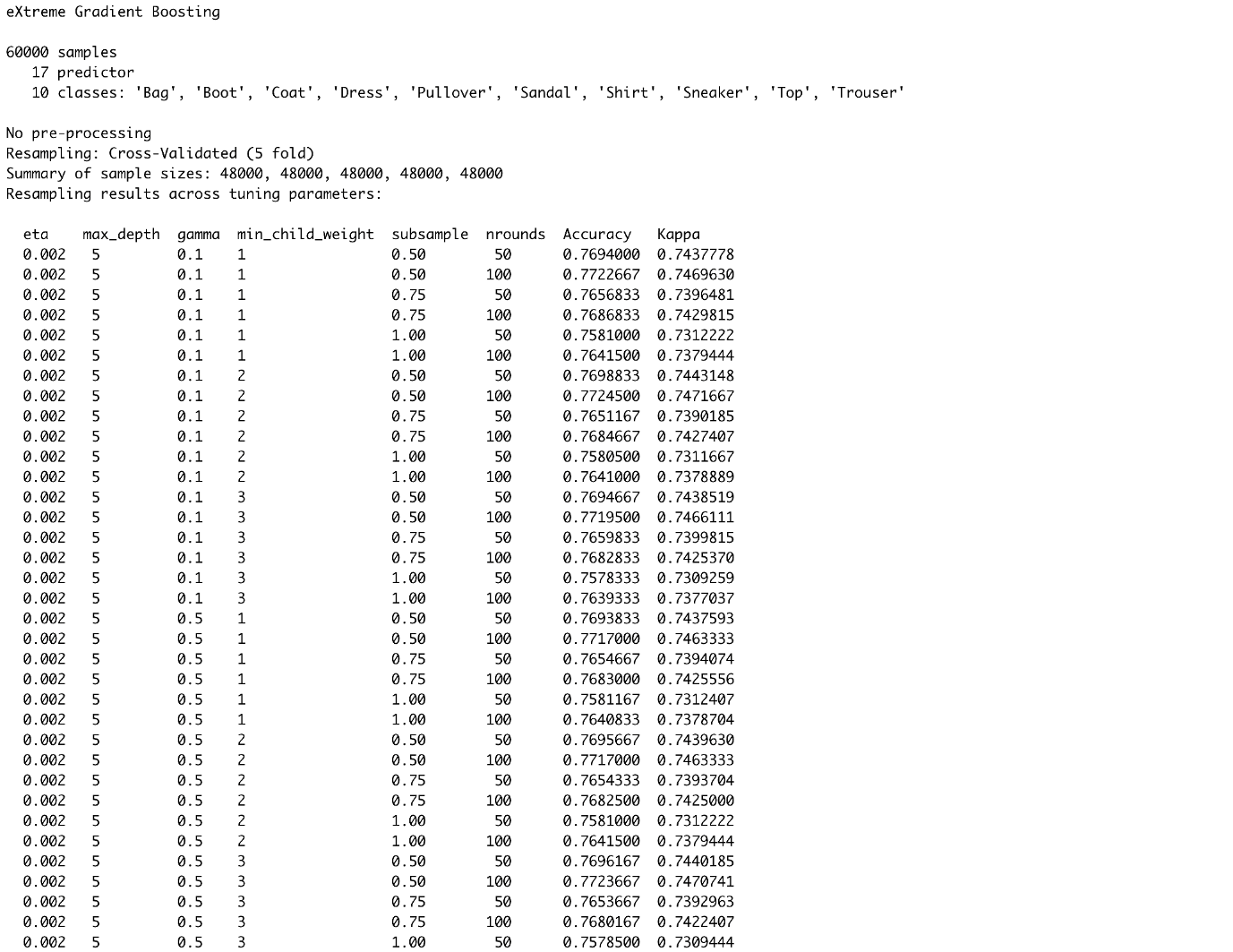
method = "xgbTree",

trControl = xgb\_control,

tuneGrid = xgb\_grid

)

xgb\_tune

(Note that the output of xgb\_tune has been truncated for this post.)[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/xgb_tune_print_1.png)

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/xgb_tune_print_4.png)

Let’s have a look at the tuning parameters resulting in the highest accuracy, and the model performance overall.

xgb\_tune$results[which.max(xgb\_tune$results$Accuracy), ]

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/xgb_highest_accuracy.png)

mp.xgb = model\_performance(xgb\_tune, train.images.pca, test.images.pca,

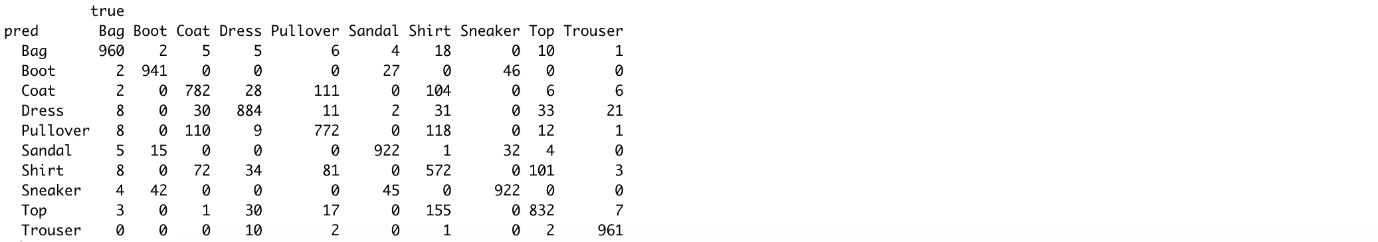
train.classes, test.classes, "xgboost")

[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/xgb_mp.png)

The optimal combination of tuning parameter values resulted in 86.2% training and 85.5% testing accuracies. Although there may be some slight overfitting going on, the model performs a bit better than the random forest, as was expected. Let’s have a look at the confusion matrix for the test set predictions to observe what clothing categories are mostly correctly or wrongly classified.

table(pred = predict(xgb\_tune, test.images.pca),

true = test.classes)

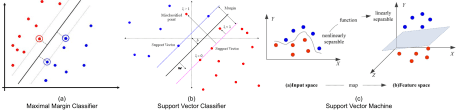
[](https://rviews.rstudio.com/2020/03/10/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-3/xgb_confusion.png)

As we saw with the random forests, pullovers, shirts and coats are most often mixed up, while trousers, boots, bags and sneakers are most often correctly classified.

In this post, we will fit a support vector machine, compare the findings from all models we have built and discuss the results. The R code for this post can be found on my [Github](https://github.com/fverkroost/RStudio-Blogs/blob/master/machine_learning_fashion_mnist_post234.R) repository.

**Support Vector Machine**

Support vector machines (SVMs) provide another method for classifying the clothing categories in the Fashion MNIST data. To better understand what SVMs entail, we’ll have to go through some more complex explanations –mainly summarizing [James et. al. (2013)](http://faculty.marshall.usc.edu/gareth-james/ISL/)– so please bear with me! The figure below might help you in understanding the different classifiers I will discuss in the next sections



For an \(n \times p\) data matrix and binary outcome variable \(y\_i \in \{-1, 1\}\), a hyperplane is a flat affine subspace of dimension \(p – 1\) that divides the \(p\)-dimensional space into two halves, defined by \(\beta\_0 + \beta\_1 X\_1 + \dots + \beta\_p X\_p\). An observation in the test data is assigned an outcome class depending on which side of the perfectly separating hyperplane it lies, assuming that such a hyperplane exists. Cutoff \(t\) for an observation’s score \(\hat{f}(X) = \hat{\beta}\_1 X\_1 + \hat{\beta}\_2 X\_2 + \dots + \hat{\beta}\_p X\_p\) determines which class it will be assigned to. The further an observation is located from the hyperplane at zero, the more confident the classifier is about the class assignment. If existent, an infinite number of separating hyperplanes can be constructed. A good option in this case would be to use the maximal margin classifier (MMC), which maximizes the margin around the midline of the widest strip that can be inserted between the two outcome classes.

If a perfectly separating hyperplane does not exist, “almost separating” hyperplanes can be used by means of the support vector classifier (SVC). The SVC extends the MMC as it does not require classes to be separable by a linear boundary by including slack variables \(\epsilon\_i\) that allow some observations to be on the incorrect side of the margin or hyperplane. The extent to which incorrect placements are done is determined by tuning parameter cost \(C \geq \sum\_{i=1}^{n} \epsilon\_i\), which thereby controls the bias-variance trade-off. The SVC is preferable over the MMC as it is more confident in class assignments due to the larger margins and ensures greater robustness as merely observations on the margin or violating the margin affect the hyperplane (James et al., 2013).

Both MMCs and SVCs assume a linear boundary between the two classes of the outcome variable. Non-linearity can be addressed by enlarging the feature space using functions of predictors. Support vector machines combine SVCs with non-linear (e.g. radial, polynomial or sigmoid) Kernels \(K(x\_i, x\_{i'})\) to achieve efficient computations. Kernels are generalizations of inner products that quantify the similarity of two observations (James et al., 2013). Usually, the radial Kernel is selected for non-linear models as it provides a good default Kernel in the absence of prior knowledge of invariances regarding translations. The radial Kernel is defined as \(K(x\_i, x\_{i'})= \exp{(-\sigma \sum\_{j=1}^{p} (x\_{ij} – x\_{i'j})^2)}\), where \(\sigma\) is a positive constant that makes the fit more non-linear as it increases. Tuning \(C\) and \(\sigma\) is necessary to find the optimal trade-off between reducing the number of training errors and making the decision boundary more irregular (by increasing C). As SVMs only require the computation of \(\bigl(\begin{smallmatrix} n\\ 2 \end{smallmatrix}\bigr)\) Kernels for all distinct observation pairs, they greatly improve efficiency.

As aforementioned, the parameters that need to be tuned are cost C and, in the case of a radial Kernel, non-linearity constant sigma. Let’s start by tuning these parameters using a random search algorithm, again making use of the caret framework. We set the controls to perform 5-fold cross-validation and we use the multiClassSummary() function from the MLmetrics library to perform multi-class classification. We specify a radial Kernel, use accuracy as the performance metric[1](https://rviews.rstudio.com/2020/03/24/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-4/#fn1) and let the algorithm perform a random search for the cost parameter C over pca.dims (=17) random values. Note that the random search algorithm only searches for values of C while keeping a constant value for sigma. Also, contrary to previous calls to trainControl(), we now set classProbs = FALSE because the base package used for estimating SVMs in caret, kernlab, leads to lower accuracies when specifying classProbs = TRUE due to using a secondary regression model.

We begin with training the support vector machine using the PCA reduced training and test data sets train.images.pca and test.images.pca

library(MLmetrics)

svm\_control = trainControl(method = "repeatedcv",

number = 5,

repeats = 5,

classProbs = FALSE,

allowParallel = TRUE,

summaryFunction = multiClassSummary,

savePredictions = TRUE)

set.seed(1234)

svm\_rand\_radial = train(label ~ .,

data = cbind(train.images.pca, label = train.classes),

method = "svmRadial",

trControl = svm\_control,

tuneLength = pca.dims,

metric = "Accuracy")

svm\_rand\_radial$results[, c("sigma", "C", 'Accuracy')]

We can check the model performance on both the training and test sets by means of different metrics using a custom function, model\_performance,.

mp.svm.rand.radial = model\_performance(svm\_rand\_radial, train.images.pca, test.images.pca,

train.classes, test.classes, "svm\_random\_radial")

The results show that the model is achieving relatively high accuracies of 88% and 87% on the training and test sets respectively, selecting sigma = 0.040 and C = 32 as the optimal parameters. Let’s have a look at which clothing categories are best and worst predicted by visualizing the confusion matrix. First, let’s compute the predictions for the training data. We need to use the out-of-bag predictions contained in the model object (svm\_rand\_radial$pred) rather than the manually computed in-sample (non-out-of-bag) predictions for the training data computed using the predict() function. Object svm\_rand\_radial$pred contains the predictions for all tuning parameter values specified by the user. However, we only need those predictions belonging to the optimal tuning parameter values. Therefore, we subset svm\_rand\_radial$pred to only contain those predictions and observations in indices rows. Note that we convert svm\_rand\_radial$pred to a data.table object to find these indices as computations on data.table objects are much faster for large data (e.g. svm\_rand\_radial$pred has 4.5 million rows).

library(data.table)

pred\_dt = as.data.table(svm\_rand\_radial$pred[, names(svm\_rand\_radial$bestTune)])

names(pred\_dt) = names(svm\_rand\_radial$bestTune)

index\_list = lapply(1:ncol(svm\_rand\_radial$bestTune), function(x, DT, tune\_opt){

return(which(DT[, Reduce(`&`, lapply(.SD, `==`, tune\_opt[, x])), .SDcols = names(tune\_opt)[x]]))

}, pred\_dt, svm\_rand\_radial$bestTune)

rows = Reduce(intersect, index\_list)

pred\_train = svm\_rand\_radial$pred$pred[rows]

trainY = svm\_rand\_radial$pred$obs[rows]

conf = table(pred\_train, trainY)

Next, we reshape the confusion matrix into a data frame with three columns: one for the true categories (trainY), one for the predicted categories (pred\_train), and one for the proportion of correct predictions for the true category (Freq). We plot this as a tile plot with a blue color scale where lighter values indicate larger proportions of matches between a particular combination of true and predicted categories, and darker values indicate a small proportion of matches between them.

conf = data.frame(conf / rowSums(conf))

ggplot() +

geom\_tile(data = conf, aes(x = trainY, y = pred\_train, fill = Freq)) +

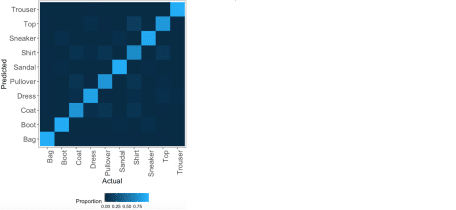
labs(x = "Actual", y = "Predicted", fill = "Proportion") +

my\_theme() +

theme(axis.text.x = element\_text(angle = 90, hjust = 1)) +

scale\_fill\_continuous(breaks = seq(0, 1, 0.25)) +

coord\_fixed()



We observe from this plot that most of the classes are predicted accurately as the light blue (high percentages of correct predictions) are on the diagonal of the tile plot. We can also observe that the categories that are most often mixed up include shirts, tops, pullovers and coats, which makes sense because these are all mostly upper body clothing parts having similar shapes. The model predicts trousers, bags, boots and sneakers well, given that these rows and columns are particularly dark except for the diagonal element.

Next, we repeat the above process for fitting a support vector machine but instead of a random search for the optimal parameters, we perform a grid search. As such, we can prespecify values to evaluate the model at, not only for C but also for sigma. We define the grid values in grid\_radial.

grid\_radial = expand.grid(sigma = c(.01, 0.04, 0.1), C = c(0.01, 10, 32, 70, 150))

set.seed(1234)

svm\_grid\_radial = train(label ~ .,

data = cbind(train.images.pca, label = train.classes),

method = "svmRadial",

trControl = svm\_control,

tuneGrid = grid\_radial,

metric = "Accuracy")

svm\_grid\_radial$results[, c("sigma", "C", 'Accuracy')]

mp.svm.grid.radial = model\_performance(svm\_grid\_radial, train.images.pca, test.images.pca,

train.classes, test.classes, "svm\_grid\_radial")

The grid search selects the same optimal parameter values as the random search (C=32 and sigma = 0.040), therefore also resulting in 88% and 87% training and test accuracies. To get an idea on how C and sigma influence the training set accuracy, we plot the cross-validation accuracy as a function of C, with separate lines for each value of sigma.

ggplot() +

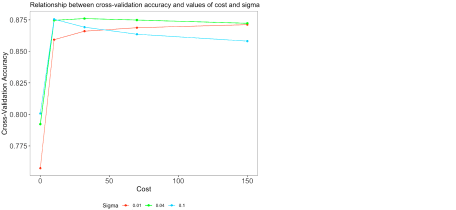
my\_theme() +

geom\_line(data = svm\_grid\_radial$results, aes(x = C, y = Accuracy, color = factor(sigma))) +

geom\_point(data = svm\_grid\_radial$results, aes(x = C, y = Accuracy, color = factor(sigma))) +

labs(x = "Cost", y = "Cross-Validation Accuracy", color = "Sigma") +

ggtitle('Relationship between cross-validation accuracy and values of cost and sigma')



The plot shows that the green line (sigma = 0.04) has the highest cross-validation accuracy for all values of C except for smaller values of C such as 0.01 and 10. Although the accuracy at C=10 and sigma = 0.1 (blue line) comes close, the highest overall accuracy achieved is for C=32 and sigma=32 (green line).

**Wrapping Up**

To compare the models we have estimated throughout this series of blog posts, we can look at the resampled accuracies of the models. We can do this in our case because we set the same seed of 1234 before training each model.[2](https://rviews.rstudio.com/2020/03/24/comparing-machine-learning-algorithms-for-predicting-clothing-classes-part-4/#fn2) Essentially, resampling is an important tool to validate our models, and to what extent they are generalizeable onto data they have not been trained on. We used five repeats of five-fold cross-validation, which means that the training data was divided into five random subsets, and that throughout five iterations (“folds”) the model was trained on four of these subsets and tested on the remaining subset (changing with every fold), and that this whole process has been repeated five times. The goal of these repetitions of k-fold cross-validation is to reduce the bias in the estimator, given that the folds in non-repeated cross-validation are not independent (as data used for training at one fold is used for testing at another fold). As we performed five repeats of five-fold cross-validation, we can essentially obtain 5\*5=25 accuracies per model. Let’s compare these resampled accuracies visually by means of a boxplot. First, we create a list of all models estimated, including the random forests, gradient-boosted trees and support vector machines. We then compute the resampled accuracies using the resamples() function from the caret package. From the resulting object, resamp, we only keep the columns containing the resample unit (e.g. Fold1.Rep1) and the five columns containing the accuracies for each of the five models. We melt this into a long format and from the result, plotdf, we remove the ~Accuracy part from the strings in column Model.

library(reshape2)

model\_list = list(rf\_rand, rf\_grid, xgb\_tune, svm\_rand\_radial, svm\_grid\_radial)

names(model\_list) = c(paste0('Random forest ', c("(random ", "(grid "), "search)"), "Gradient-boosted trees",

paste0('Support vector machine ', c("(random ", "(grid "), "search)"))

resamp = resamples(model\_list)

accuracy\_variables = names(resamp$values)[grepl("Accuracy", names(resamp$values))]

plotdf = melt(resamp$values[, c('Resample', accuracy\_variables)],

id = "Resample", [value.name](http://value.name) = "Accuracy", [variable.name](http://variable.name) = "Model")

plotdf$Model = gsub("~.\*","", plotdf$Model)

Next, we create a boxplot with the estimated models on the x-axis and the accuracy on the y-axis.

ggplot() +

geom\_boxplot(data = plotdf, aes(x = Model, y = Accuracy, color = Model)) +

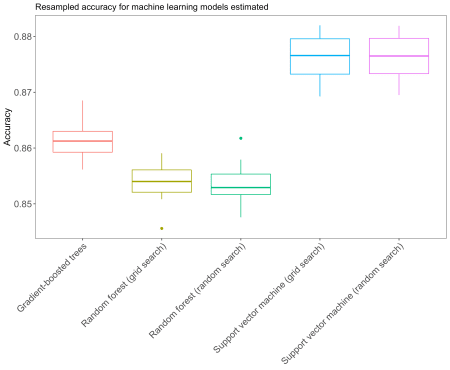
ggtitle('Resampled accuracy for machine learning models estimated') +

my\_theme() +

theme(axis.text.x = element\_text(angle = 45, hjust = 1)) +

labs(x = NULL, color = NULL) +

guides(color = FALSE)



We observe from these box plots that the support vector machines perform best, followed by the gradient-boosted trees and the random forests. Let’s also take a look at the other performance metrics from all models we have looked at.

mp.df = rbind(mp.rf.rand, mp.rf.grid, mp.xgb, mp.svm.rand.radial, mp.svm.grid.radial, mp.svm.grid.linear)

mp.df[order(mp.df$accuracy\_test, decreasing = TRUE), ]