Movement Prediction Project

44Yeti

2020 06 03

Contents

0.1 Executive Summary

This is the final challenge in the "HarvardX Professional Certificate in Data Science" course. For this project we had the task to apply machine learning techniques which go beyond the standard linear regression we applied in the Movielens project before. Even so we shall use a dataset which we not used already in the previous courses (movielens, mnist, iris...). As potential sources *UCI Machine Learning Repository*[1] or *Kaggle*[2] were recommended. After making me some thoughts about the assignment I defined the following criteria which the challenge shall provide to me:

1. The raw data must be split up in several files, so there is a challenge to bring all this single pieces together in one usable dataset
2. Raw data shall not be pre-processed in a way that the features are "ready for use", so some pre-processing must be done by me
3. The machine learning challenge shall be a classification challenge with more than two classes and more then 10 different predictors

I have defined these challenges as they are topics we did not touch too much during the courses and I therefore saw the need for some practise. Especially the first two point were important to me as these skills seem to me something which will be needed quite often in real-life cases. As I have defined these challenges as key for my work, they will also get some weight in this report, therefore the chapter "Generate the Feature Matrix" will be quite detailed compared to others. After some searching on the UCI and Kaggle platform I found the *Daily and Sports Activities Data Set*[3] on UCI. For the creation of this dataset 8 person had to fulfil 19 different activities, while their movements got measured by sensors. The raw data is stored in 60 different files (each file consists out of a 125 * 45 matrix) per person per activity, which means that we must handle $60 * 8 * 19 = 9120$ different files. Even so the features for the machine learning model had to be calculated from the raw data - the raw data did not provide "usable" features directly. And as the machine learning challenge was a multiclass classification problem, all requirements I set as my project challenge were given with this dataset. So, my choice was done. During my project the study "Comparative study on classifying human activities with miniature inertial and magnetic sensors" from Kerem Altun, Bilur Barshan and Orkun Tuncel[4] was a good support, as they used the same dataset for their studies. As I am not exactly a subject matter expert in the domain of sensor-data, their work was especially helpful by the definition of the different features. Even so it helped me to set an accuracy target for my machine learning model: They reached with their most successful model an accuracy of over 99%, so this was my goal too. But as I wanted to prevent that it gets too much of a "copy" of their study, I skipped most of the models they applied, including their most successful one (Bayesian decision making (BDM)). My set of models consists out of the following six algorithms: Penalized Multinomial Regression (PMR), k-Nearest Neighbours (k-NN), naive Bayes (nBayes), Support Vector Machines (SVM), Gradient Boosting linear (GBl) and Gradient Boosting tree (GBt). Beside SVM and k-NN algorithm these were all models not used by the study from Altun/Barshan/Tuncel. The best result I achieved with the "Gradient Boosting (tree)" algorithm with an accuracy of **99,49%**. Thus I have reached my set goal. To get this result I followed the following steps:

1. Analyse and understand the raw dataset
2. Load the different raw data files, transform them into a feature set and generate a "Feature Matrix" as our base for all next steps
3. Generate a train- and test dataset
4. Analyse the train dataset and do a feature reduction
5. Train all six models with the train dataset and apply the trained models against the test dataset
6. Analyse and discuss the results

Along these steps I have also done the structure of this document plus I added a chapter about my personal conclusion and potential future work at the end.

[1] https://archive.ics.uci.edu/ml/index.php

[2] https://www.kaggle.com/datasets

[3] https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities

[4] https://www.researchgate.net/publication/223300172_Comparative_study_on_classifying_human_activities_with_miniature_inertial_and_magnetic_sensors

0.2 The raw data dataset

The raw dataset one can find under following link: https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities. To generate this data 8 people (4 women, 4 men) got equipped with 5 sensor units each. The sensor units were positioned at the torso (T), right arm (RA), left arm (LA), right leg (RL) and left leg (LL). Each of these sensor units included three different sensors, of which each delivered values for the x-, y- and z-axis. Therefore, for all measure points in time we get three values from the accelerometer- (xacc, yacc, zacc), gyroscope- (xgyro, ygyro, zgyro) and magnetometer-sensors (xmag, ymag, zmag) each. This is how such a sensor unit is looking like:

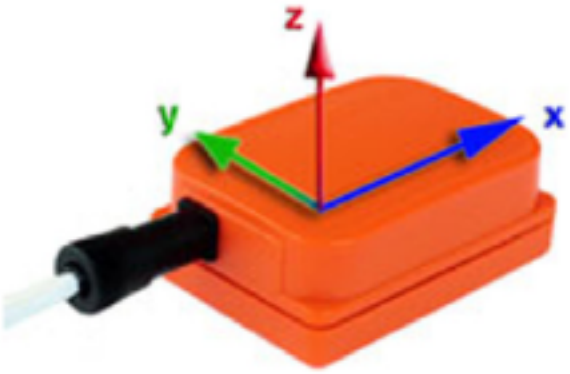

Figure 1: MTx 3-DOF orientation tracker (reprinted from http://www.xsens.com/en/general/mtx)

As the measurements get done in a pace of 25 Hertz this means every second we get the result of 25 measure points, and as each measure point includes 45 values (5 sensor units * 3 sensors * 3 axis) all second we get $25 * 45 = 1125$ values. In the dataset this data got captured as follows: per each 5 second they generated a txt-file in which the values are stored as a 125 * 45 matrix (125 rows as we have 25 * 5 "capture points"). The build-up logic of the files is always the same and follows the following pattern: Columns 1-45 correspond to:

T_xacc, T_yacc, T_zacc, T_xgyro, ..., T_ymag, T_zmag, RA_xacc, RA_yacc, RA_zacc, RA_xgyro, ..., RA_ymag, RA_zmag, LA_xacc, LA_yacc, LA_zacc, LA_xgyro, ..., LA_ymag, LA_zmag, RL_xacc, RL_yacc, RL_zacc, RL_xgyro, ..., RL_ymag, RL_zmag, LL_xacc, LL_yacc, LL_zacc, LL_xgyro, ..., LL_ymag, LL_zmag

Therefore:

- columns 1-9 correspond to the sensors in unit 1 (T)
- columns 10-18 correspond to the sensors in unit 2 (RA)
- columns 19-27 correspond to the sensors in unit 3 (LA)
- columns 28-36 correspond to the sensors in unit 4 (RL)
- columns 37-45 correspond to the sensors in unit 5 (LL)

Now as we know how the single input files are structured let us zoom out and have a look at the activities the participants were measured at. Each participant had to perform the following 19 activities for exact 5 minutes:

- sitting (A1)
- standing (A2)
- lying on back and on right side (A3 and A4)
- ascending and descending stairs (A5 and A6)
- standing in an elevator still (A7)

- and moving around in an elevator (A8)
- walking in a parking lot (A9)
- walking on a treadmill with a speed of 4 km/h (in flat and 15 deg inclined positions) (A10 and A11)
- running on a treadmill with a speed of 8 km/h (A12)
- exercising on a stepper (A13)
- exercising on a cross trainer (A14)
- cycling on an exercise bike in horizontal and vertical positions (A15 and A16)
- rowing (A17)
- jumping (A18)
- and playing basketball (A19)

In addition, it is to say that the subjects did not get any guidance how to perform the activities. So, it is likely that we will see some "inter-subject" variations.

Now we have all components together and can have a look how the dataset is delivered. First of all it is a zip.file called "data.zip". Within this zip.file we find a folder per activity (folder a01 till a19). Within each of these folders there is a folder per subject (folder p1 till p8). And finally, within these subject folders are always 60 input files (files s01.txt till s60.txt) - explanation: 5 seconds are captured in one file, each activity gets measured for 5min equals 60 files. In a classical Windows Explorer view that looks like this:

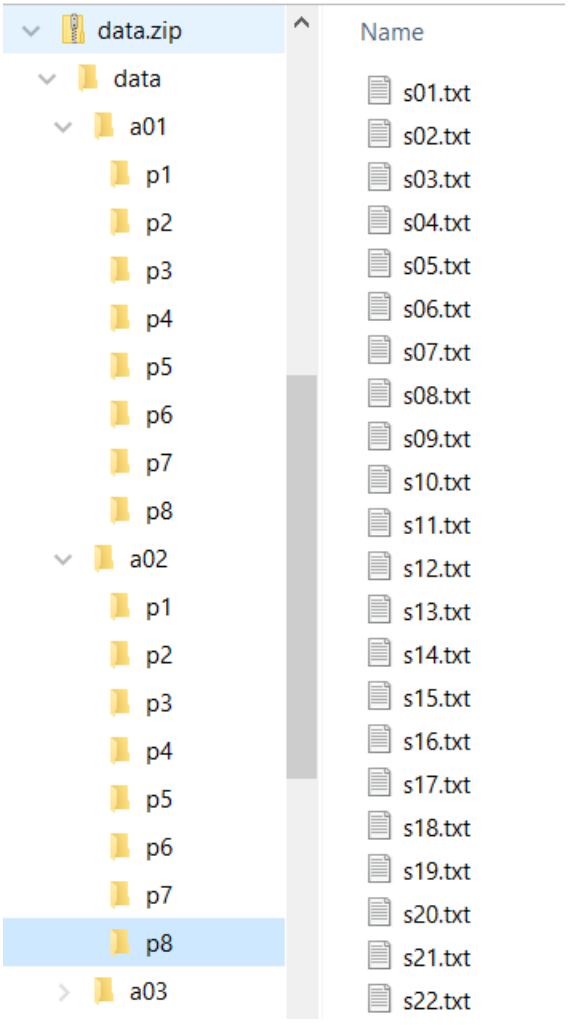

Figure 2: Visualisation of data structure

If we now have a look at the measured values itself, then we realise that a) there are a lot of values to deal with (51.3 millions) and b) the single values do not really have a "meaning" they are "just" single points on an axis. To get a clear and somehow meaningful set of features we must bring this single values somehow in a context, either by analysing them in a time series, by its distribution or by some statistical performance indicators. And this leads me over to the next chapter where we will see how we produced the final "Feature Vector" with which we then fed the different algorithms.

0.3 Method/Analysis

0.3.1 Generate the "Feature Matrix"

So, here I was with 9'120 single files and something over 50 million values which have to be transformed and merged into one single matrix, where each column is a meaningful feature-vector. First I did not really have a clue how to tackle this topic. As chances were low that I will get the master plan by just thinking it through, I started with addressing just one of the topics to be solved: How to load the data from the single files into one R data object? My initial idea was to load all raw data in a matrix per activity and all these 19 matrixes are in turn part of an array. As a first step I had to define how to extract the data of one single file. The files follow a structure where there are 45 values per row separated by a comma and there are 125 such rows in each file. So, the structure of the future 125*45 matrix is already given, therefore this step should not be too hard. To simplify things even further I stored the raw dataset on a local drive in a unziped format. This is how the first "extraction code" looked like:

```
# This code is not part of the final code script - as it was just an intermediate step
file1 <- paste0("C:/Users/XXX/XXX/Projects/R/TempRohDaten/data/a01/p1/s01.txt")
            a <- as.matrix(read_delim(file1, delim = ",", col_names = ColNames))
```

Then I started with loops. First just for one person in one activity, then for a whole activity and then for the whole raw data set. Therefore, I had to start working with variables within the file path, which I had to define first. Even so I defined the column names. As loop function I used the for-loop

```
# This code is not part of the final code script - as it was just an intermediate
# step Label for columns:
ColNames <- c("T_xacc", "T_yacc", "T_zacc", "T_xgyro", "T_ygyro", "T_zgyro", "T_xmag",
    "T_ymag", "T_zmag", "RA_xacc", "RA_yacc", "RA_zacc", "RA_xgyro", "RA_ygyro",
    "RA_zgyro", "RA_xmag", "RA_ymag", "RA_zmag", "LA_xacc", "LA_yacc", "LA_zacc",
    "LA_xgyro", "LA_ygyro", "LA_zgyro", "LA_xmag", "LA_ymag", "LA_zmag", "RL_xacc",
    "RL_yacc", "RL_zacc", "RL_xgyro", "RL_ygyro", "RL_zgyro", "RL_xmag", "RL_ymag",
    "RL_zmag", "LL_xacc", "LL_yacc", "LL_zacc", "LL_xgyro", "LL_ygyro", "LL_zgyro",
    "LL_xmag", "LL_ymag", "LL_zmag")

# Define the variables for the loop:
activityNames <- c("a01", "a02", "a03", "a04", "a05", "a06", "a07", "a08", "a09",
    "a10", "a11", "a12", "a13", "a14", "a15", "a16", "a17", "a18", "a19")
personNumber <- c("p1", "p2", "p3", "p4", "p5", "p6", "p7", "p8")
fileNames <- sprintf("%02d", seq(1:60))

# Build the loops:
RawData_list <- vector("list", 1)  #build an empty list to be filled in the loops

for (z in activityNames) {
    # Loop for the activities


    for (y in personNumber) {
        # Loop for the person
```

```
        for (x in fileNames) {
            # Loop for the files; first file has to be treated differently therefore the
            # if-else-statement
            if (exists("a") == FALSE) {
                file1 <- paste0("C:/Users/XXX/XXX/Projects/R/TempRohDaten/data/",
                  z, "/p1/s01.txt")
                a <- as.matrix(read_delim(file1, delim = ",", col_names = ColNames))
            } else {
                file <- paste0("C:/Users/XXX/XXX/Projects/R/TempRohDaten/data/",
                  z, paste0("/", y, paste0("/s", x, ".txt")))
                a <- as.matrix(cbind(a, read_delim(file, delim = ",", col_names = ColNames)))
                a
            }
        }
    }
    RawData_list[[z]] <- a
    rm(a)
}

RawData_array <- abind(RawData_list, rev.along = 0)
```

Already while I was building up this solution, I realised, that this will not be the solution, I will use for my code. Thus, because it seems not to be efficient to load everything in a R-object first and then transform it into a new dataset with generating the feature-set. But nevertheless, I pushed this approach forward to a working solution as I was convinced that at least the principal "philosophy" of the loops and the logic behind I will be able to reuse also in an adapted approach. So, my new goal was to combine the feature building- with the data loading process. But before we can start looking at this code, we first must discuss the feature set. To define a meaningful feature-set I oriented myself at the work of Altun/Barshan/Tuncel.They suggested the following metrics:

- Minimal value per raw data column/sensor data-axis
- Maximal value per raw data column/sensor data-axis
- Mean per raw data column/sensor data-axis
- Variance per raw data column/sensor data-axis
- Skewness per raws data column/sensor data-axis
- Kurtosis per raw data column/sensor data-axis
- Discrete Fourier Transformation per raw data column/sensor data-axis
- Autocorrelation per raw data column/sensor data-axis

After analysing these different parameters I decided to include all of them in my work too. This mainly because of two reasons: First this set seems reasonable to me as it covers different perspectives on the raw dataset - from metrics which help to explain the data distribution (Minimum, Maximum, Mean, Variance, Skewness and Kurtosis) to metrics which look at the data as a time series (Discrete Fourier Transformation and Autocorrelation). My credo was: As more different perspectives we get on our input data, as higher are chances that we find unique patterns per activity. Second Altur/Barshan/Tuncel achieved particularly good results with this feature set (high accuracies), so just out of a practical reason there is no need for change... While most of the above listed features are straight forward, we nevertheless have to discuss Discrete Fourier Transform and Autocorrelation in more depth.

Discrete Fourier Transform (DFT)

I did not really know DFT before. Fast it got clear that this is a big topic with lots of domains where it gets applicated but also quite a complex topic. Thanks to the first lessons of an online course from Mike X Cohen I could build up enough understanding to fulfil the tasks needed for this analysis[5]. Neverthless, I

[5] https://www.udemy.com/course/fourier-transform-mxc/

waive explaining the DFT concept in this report as this is a huge topic for itself and there are experts out there which are by far better qualified to do so. But we must discuss what of the DFT results we want to include in the feature-set. With DFT, or better said FFT (Fast Fourier Transform) as this is the formula we apply, we get again 125 values back per raw data column. So, if we would use all of those values our feature-set would "explode". Therefore Altun/Barshan/Tuncel suggested to use only the top 5 magnitude peaks and its corresponding frequencies per sensor axis (columns in the raw data files) - this would add $5 * 45 * 2 = 450$ feature columns (225 for the magnitude peaks and 225 for the corresponding frequencies). After some testing with replacing the corresponding frequency with the corresponding phases (angles) instead, I then decided to follow again their suggestion. The reasoning behind is again very much "hands on": I have trained the models with both options and I just got the better results... Two things I was not so sure about how Altun/Barshan/Tuncel treated this topic in their study:

- First, I just used half of the results per columns as the second half is like the mirror image of the first. Which means that I used the values from frequency 0 till the Nyquist frequency of 12.5Hz. This mentioned symmetry we can see for example here:

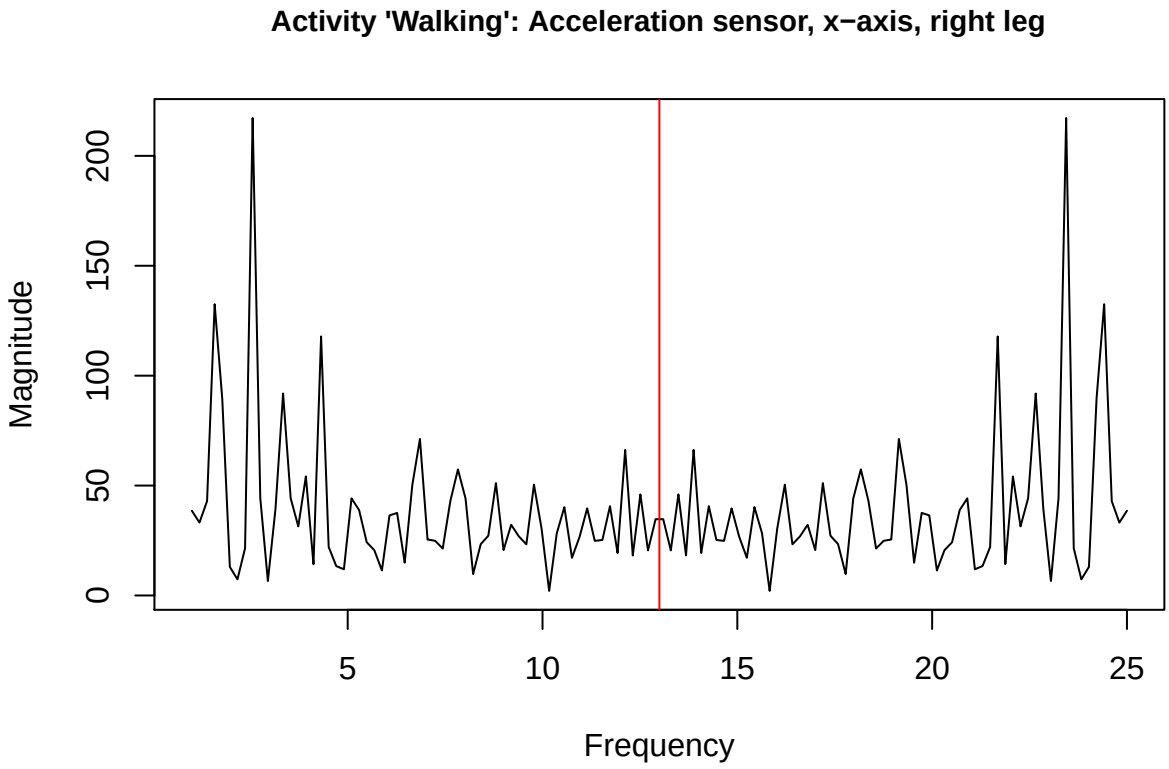

Figure 3: Visualisation of the symmetry of the magnitude values (without value for Freq 0)

- Second, I was not 100% sure what to do with the DC value (so the Magnitude at Frequency 0) as often this value was much higher than the other values and looked like an "outlier". In addition, we have already the column mean in our feature-set to which the DC is highly correlated (in fact it is the sum of the column values of the raw data). So I run my script with and without the DC value and to my surprise the DC value seems to add valuable information for the models, as the Accuracies were clearly higher with the DC... Therefore, the for this report used feature-set includes it. To visualize the effect of DC I show the same graph as before but this time including frequency 0:

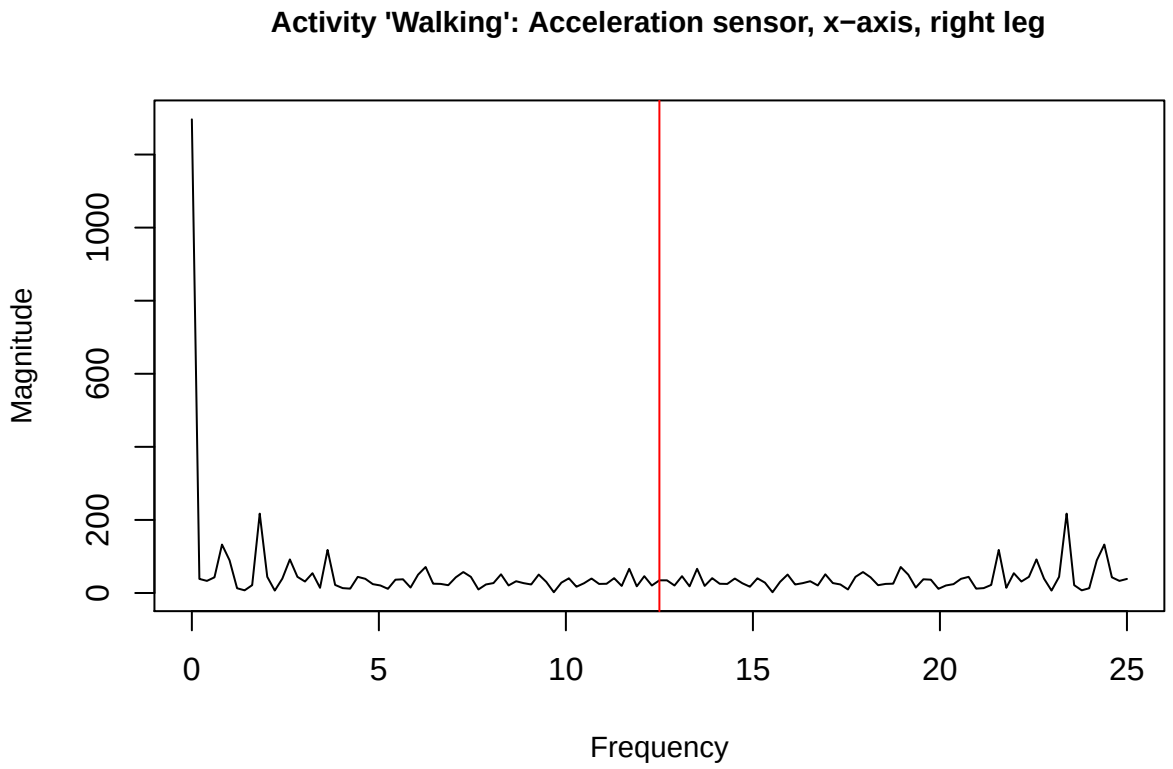

Figure 4: Visualise the impact on the y-axis scale because of the DC value

Autocorrelation

Like with DFT we must decide which values of an autocorrelation analysis we want to include in our feature-set. For this we start with two examples of autocorrelation with a lag of 125:

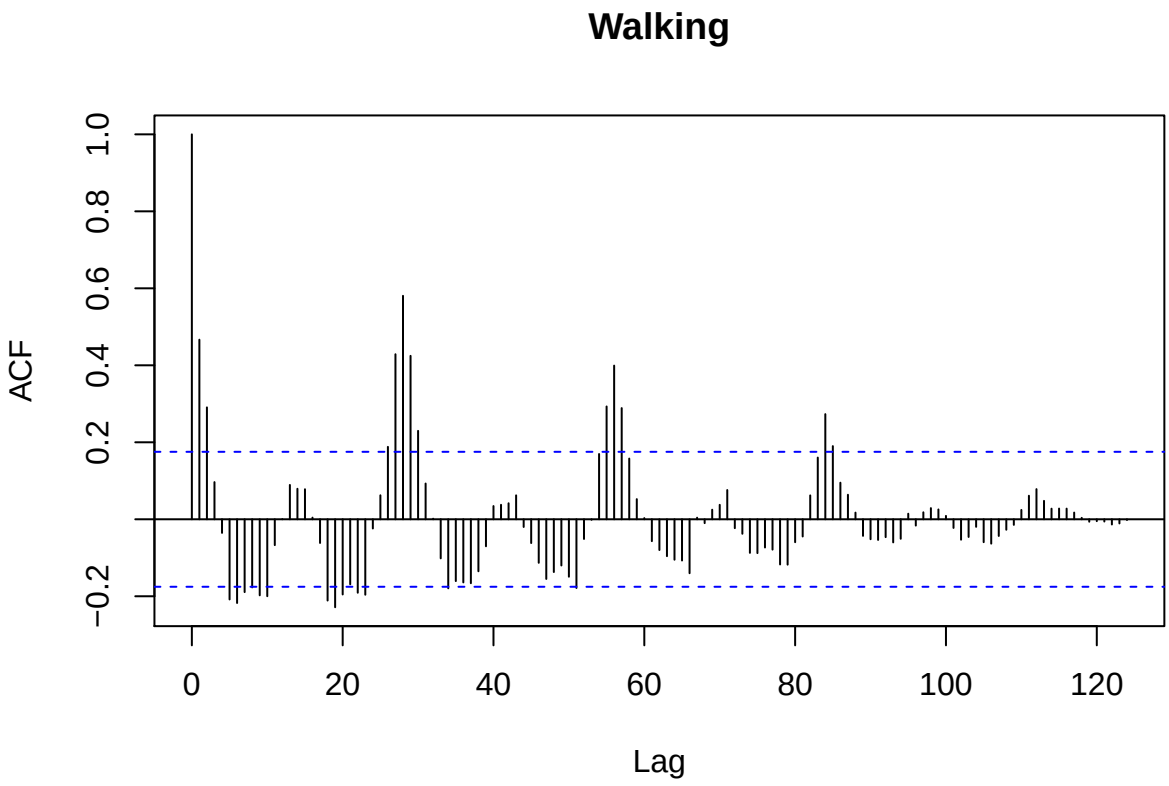

Figure 5: Example of autocorrelation with lag = 125 (Activity: Walking)

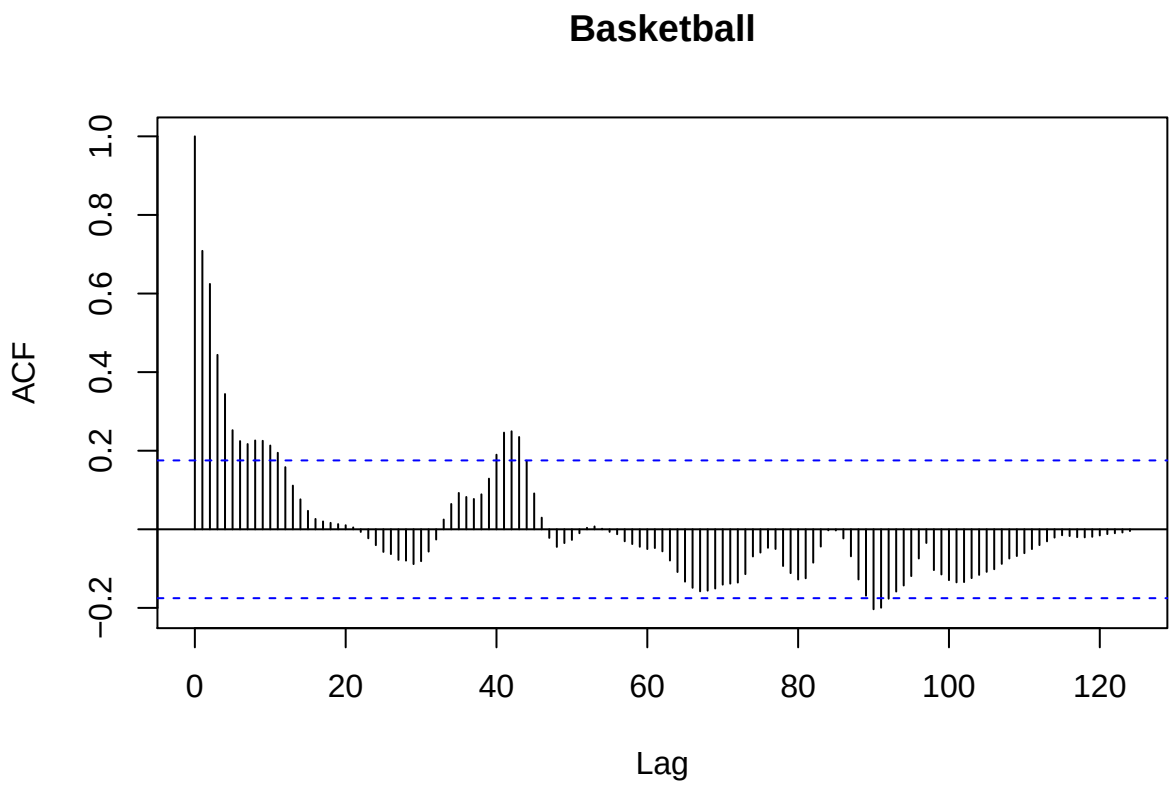

Figure 6: Example of autocorrelation with lag = 125 (Activity: Basketball)

After looking at some of these examples I got to the following conclusions:

- lag = 1 is the "comparison" with itself so this value is always 1 and has no additional value
- Beside this the first 5 lags seem always to have the highest correlations
- Otherwise there is no 100% clear pattern to take into consideration

Based on these findings I deviated a bit from the selection of Altun/Barshan/Tuncel. Nevertheless I stuck to their suggestion to include 11 values per raw data column - my selection was the following: Values of lag 2 to 5, values of lag 23 to 26 (around 1 second after initial signal), values of lag 48:50 (around 2 seconds after initial signal)

So, we have our feature-set defined and it consists out of **1215 features**: 6 times 45 (min, max, mean, var, skew., kurt.) + 2 times 225 (DFT mag., DFT freq) + 11*45 (Autocor.). Just for completeness: Altun/Barshan/Tuncel used "just" 1170 features at the end, they seemed to have skipped one variable out of the list of the first six features. But as this has no specific relevance for my project, I did not investigate further.

Now we can go back to our code. As stated above the new plan is to involve the "feature calculation" in the loading process. Therefore, I created a function "Features" which does all the calculations and then involve it in the if-loop process showed above. Here the code of this function:

```
Features <- function(f) {
    # Maximum/Minimum, mean, variance, skewness, kurtosis per column
    f_min <- apply(f, 2, min)
    f_max <- apply(f, 2, max)
    f_mean <- colMeans(f)
```

```
f_var <- colVars(f)
f_skew <- apply(f, 2, skewness)
f_kurt <- apply(f, 2, kurtosis)

#----------Discrete Fourier Transformation (with fast fourier transform)---------
# Do Fast Fourier transformation (fft)
f_DFT_temp <- apply(f, 2, fft)

# Get the peaks and the phases
f_DFT_Mod_temp <- apply(f_DFT_temp, 2, Mod)

# Choose just half of the values (seconde half is like a mirror of the
# first)
f_DFT_Mod_temp_Half <- f_DFT_Mod_temp[1:(length(f_DFT_Mod_temp[, 1])/2 +
    1), ]

# Double all values beside for Frequency 0 (first index value)
f_DFT_Mod_temp_Half[2:(length(f_DFT_Mod_temp[, 1])/2), ]
    <- f_DFT_Mod_temp_Half[2:(length(f_DFT_Mod_temp[, 1])/2), ] * 2

# Calculate Frequency:
Freq <- seq(0, (25/2), length = length(f_DFT_Mod_temp[, 1])/2 + 1)

# Select the top 5 Peaks
f_DFT_Mod <- apply(f_DFT_Mod_temp_Half, 2, function(x) {
    head(sort(x, decreasing = TRUE), 5)
})

# Find corresponding Frequency
for (i in 1:45) {

    if (exists("Index_vec") == FALSE) {
        Index_vec <- match(f_DFT_Mod[, i], f_DFT_Mod_temp_Half[, i])
    } else {
        Index_vec <- c(Index_vec, match(f_DFT_Mod[, i], f_DFT_Mod_temp_Half[,
            i]))
    }
}

Index_matrix <- matrix(Index_vec, 5, 45)

for (i in 1:45) {
    if (exists("f_DFT_Freq") == FALSE) {
        f_DFT_Freq <- Freq[Index_matrix[, i]]
    } else {
        f_DFT_Freq <- c(f_DFT_Freq, Freq[Index_matrix[, i]])
    }
}
#------------------------------------ Autocorrelation -------------------
# Get the Autocorrrelation for lag 50 (corresponds to 2 sek)
f_Autoc_temp <- apply(f, 2, function(x) {
    acf(x, lag.max = 50, plot = FALSE)
})
```

```
    # Get 11 values out of the 50 per 'column' (here more 'list section') -
    # First value always =1 so not a lot of info from this, then I chose 4
    # correlation values of the beginning, 4 values around 1 sek distance
    # and 3 values at the end (=around 2sek difference)
    for (i in 1:45) {
        if (exists("f_Autoc") == FALSE) {
            f_Autoc <- f_Autoc_temp[[i]]$acf[c(2:5, 23:26, 48:50)]
        } else {
            f_Autoc <- c(f_Autoc, f_Autoc_temp[[i]]$acf[c(2:5, 23:26, 48:50)])
        }
    }
    #---------------------Building the Feature Vector ----------------------------------
    Feature_Vector <<- c(f_min, f_max, f_mean, f_var, f_skew, f_kurt, as.vector(f_DFT_Mod),
        f_DFT_Freq, f_Autoc)  #global Variable '<<-'
}
```

As hoped, I could reuse most of the logic, I used for my initial data loading process and it was not such a big thing to include the Features function. In addition, there was one other step I skipped right away and this was the conversion into an array as this made no sense at all. The new "strategy" was to load everything in a list, get one long vector out of it and at the very end of the loading/conversion process transform this vector in a matrix where the columns represent the 1215 features. After being happy to have solved this topic I faced another issue with this whole process: It takes more than 40minutes to run it and this with the raw data unziped and locally stored... I knew that there has to be room for improvement concerning this problem. Nevertheless, as I was curious about the results I get out of the algorithms, I accepted this issue for the moment and proceeded like this (to be concrete I stored the generated Feature-Matrix locally and usually started the script from there). It was only after seeing the results that I came back to this challenge. But for the sake of the structure of this report I describe the final step of this loading/conversion process now. To come directly to the point: I exchanged the for-loop function by the `lapply()` function and with some slight adjustments the whole process needed less than 10minutes.

This changed again when I switched the process from locally stored, unziped raw data to ziped, online data (process runs for over 45minutes). This because I first used the logic of the code we got as starting point for the "Movielens" challenge. As I have not just one file but a whole dataset and by using this template in every loop the new file had first to be unziped it was a very inefficient process. So, the solution was to first define a temporary directory where I can unzip the whole dataset for further use. Like this the whole process went down to just over 12 minutes again. One topic I want to bring up here, as this could be a reason why my code is not running well on another system: I stumbled over a problem with the path of the temporary directory - I found a solution for it on my Windows-System but I am not 100% sure if this correction will now prevent my code from running properly on a non-Windows system... All in all, the data download and unzip code look like this:

```
# Download the raw data
td <- tempdir()
tf = tempfile(tmpdir = td, fileext = ".zip")
download.file("https://publicdatarfu.blob.core.windows.net/publicdatarfucontainer/data.zip",
    tf)
unzip(tf, exdir = td)
# I had '\\' as separater on the path which did not work on my windows system,
# to correct this I found the following correction option, which I hope will
# solve it in a way that my code is also running on a non windows system without
# any problems. In worst case please adjust the separaters in fpath manually...
fpath <- normalizePath(td, winslash = "/", mustWork = NA)
```

And the final code for the data loading/conversion process to get the Feature Vector is the following:

```
# Loop for the 19 activities
FV2 <- lapply(z <- activityNames, function(z) {

    # Loop for the 8 person
    FV1 <- lapply(y <- personNumber, function(y) {

        # Loop for the 60 time-series files
        FV <- lapply(x <- fileNames, function(x) {
            file <- paste0(fpath, "/data/", z, "/", y, "/s", x, ".txt")
            f <- as.matrix(read_delim(file, delim = ",", col_names = ColNames))
            if (exists("FV") == FALSE)
                {
                  FV <- vector("list", 1)
                  FV[[x]] <- Features(f)
                }  #calling the Feature function
 else {
                FV[[x]] <- Features(f)
            }  #calling the Feature function
        })

        if (exists("FV1") == FALSE) {
            FV1 <- vector("list", 1)
            FV1[[y]] <- Reduce(c, FV)
        } else {
            FV1[[y]] <- Reduce(c, FV)
        }
    })
    if (exists("FV2") == FALSE) {
        FV2 <- vector("list", 1)
        FV2[[z]] <- Reduce(c, FV1)
    } else {
        FV2[[z]] <- Reduce(c, FV1)
    }
})

Feature_Vector <- Reduce(c, FV2)
```

And finally this Feature Vector will be transformed into the Feature_Matrix variable. This will then be the R object with which we will go on in the next steps of the script, all the rest we remove. The challenge of this final step was to label all columns and rows correctly. But since it is not so decisive, I renounce the representation of the code in favour of the readability of this report. In this sense let us go on to the next chapter which will be a bit a shorter one about building a train- and test dataset out of the Feature_Matrix.

0.3.2 Train- and Test dataset

To get to a train- and test dataset we use the `createDataPartition` function from the caret package. I set the parameters in a way that 85% of the Feature_Matrix shall be in the Feature_train- and the remaining 15% in the Feature_test dataset.

With the following control code we can check if the split got executed as expected:

```
#show dimensions of Matrix and ratio between train and test set
dim(Feature_train)
```

```
## [1] 7752 1215
```

```
dim(Feature_test)
```

```
## [1] 1368 1215
```

```
length(Feature_train[,1])/length(Feature_Matrix[,1])
```

```
## [1] 0.85
```

As we can see the Feature_train dataset has the dimensions/ratio which we were expecting, therefore we go on by examining the dataset more in depth.

0.3.3 Analyse train dataset and do feature reduction

We start with some basic controls by looking at the structure of Feature_train/-_test datasets and if there is any NA value in it:

```
#structure of Feature_train
str(Feature_train)
```

```
##  num [1:7752, 1:1215] 7.68 7.85 7.85 7.69 7.83 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:7752] "a01" "a01" "a01" "a01" ...
##   ..$ : chr [1:1215] "min_T_xacc" "min_T_yacc" "min_T_zacc" "min_T_xgyro" ...
```

```
str(Feature_test)
```

```
##  num [1:1368, 1:1215] 7.85 7.79 7.77 7.75 7.77 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:1368] "a01" "a01" "a01" "a01" ...
##   ..$ : chr [1:1215] "min_T_xacc" "min_T_yacc" "min_T_zacc" "min_T_xgyro" ...
```

```
#Are there any NA values?
sum(is.na(Feature_train))
```

```
## [1] 0
```

```
sum(is.na(Feature_test))
```

```
## [1] 0
```

This first glance shows that column- and row names seem to be correct so the labelling worked out well. Even so there are no NA values and as we have seen in the previous chapter the dataset's dimensions are as expected too, so it looks like the whole data loading and data splitting process went well.

As a next step we will have a look at the data of the Feature_train dataset. We start with a comparison of the three different activities "sitting" (a01), "walking" (a10) and "basketball" (a19). As there are too many features I reduced it to a selection of the following six features: Magnitude 1 (DC value) of the x,y,z-axis, of the acceleration sensor of the right leg (RL) sensor unit and the same selection for the variance. It will certainly be interesting to see the differences between the different activities:

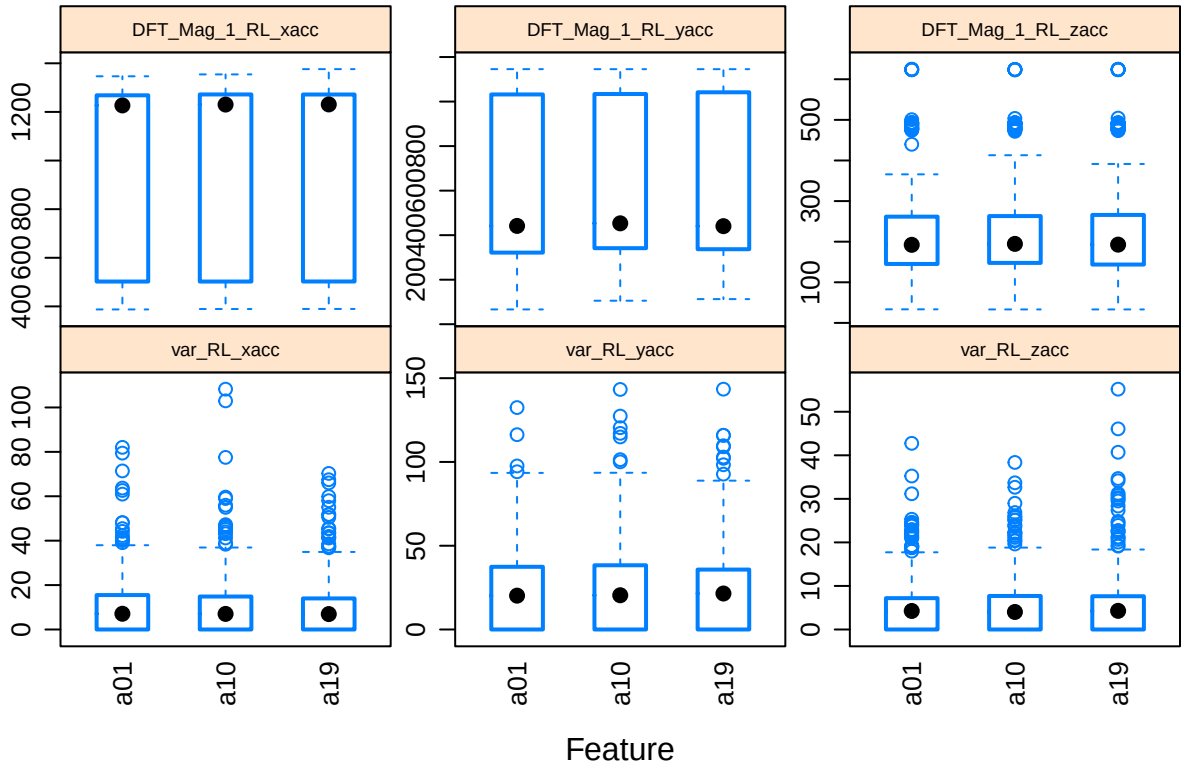

Figure 7: Comparison of activity sitting (a01), walking (a10), and basketball (a19) by selected features

To my astonishment there is far less difference between the activities as I had expected, then the different distributions seem to be remarkably similar. Only when you zoom in heavily one can see some deviations. This made me a bit nervous as when there are no activity specific patterns how an algorithm shall be able to distinguish them? Therefore, let us do another analysis with a different perspective: This time we look at the relation between the different features per activity. To do so we will calculate the correlation matrix per activity with the same feature set as before.

```
#activity 01 (sitting):
c1 <- cor(Feature_train[1:408,x8[c(43:45,79:81)]])
c1
```

```
##                     var_RL_xacc var_RL_yacc var_RL_zacc DFT_Mag_1_RL_xacc
## var_RL_xacc         1.000000000  0.84946039  0.77557738       -0.03483803
## var_RL_yacc         0.849460393  1.00000000  0.62524172       -0.03020508
## var_RL_zacc         0.775577379  0.62524172  1.00000000       -0.01239258
## DFT_Mag_1_RL_xacc -0.034838025 -0.03020508 -0.01239258        1.00000000
## DFT_Mag_1_RL_yacc -0.009932407 -0.06212439 -0.00986361       -0.44451652
## DFT_Mag_1_RL_zacc  0.030823379  0.09356388  0.03133326       -0.04902379
##                   DFT_Mag_1_RL_yacc DFT_Mag_1_RL_zacc
## var_RL_xacc          -0.009932407        0.03082338
## var_RL_yacc          -0.062124393        0.09356388
## var_RL_zacc          -0.009863610        0.03133326
## DFT_Mag_1_RL_xacc    -0.444516517       -0.04902379
## DFT_Mag_1_RL_yacc     1.000000000       -0.84511084
```

```
## DFT_Mag_1_RL_zacc      -0.845110837        1.00000000
```

```
#activity 10 (walking):
c2 <- cor(Feature_train[3673:4080,x8[c(43:45,79:81)]])
c2
```

```
##                   var_RL_xacc var_RL_yacc var_RL_zacc DFT_Mag_1_RL_xacc
## var_RL_xacc         1.0000000   0.8688555  0.50447803        0.22188587
## var_RL_yacc         0.8688555   1.0000000  0.23618050        0.12767058
## var_RL_zacc         0.5044780   0.2361805  1.00000000        0.20132712
## DFT_Mag_1_RL_xacc   0.2218859   0.1276706  0.20132712        1.00000000
## DFT_Mag_1_RL_yacc   0.8007373   0.8590674  0.27617114        0.08609979
## DFT_Mag_1_RL_zacc   0.1475990   0.2290144  0.04639445       -0.42284015
##                   DFT_Mag_1_RL_yacc DFT_Mag_1_RL_zacc
## var_RL_xacc              0.80073726        0.14759898
## var_RL_yacc              0.85906742        0.22901444
## var_RL_zacc              0.27617114        0.04639445
## DFT_Mag_1_RL_xacc        0.08609979       -0.42284015
## DFT_Mag_1_RL_yacc        1.00000000        0.22920928
## DFT_Mag_1_RL_zacc        0.22920928        1.00000000
```

```
#activity 19 (basketball):
c3 <- cor(Feature_train[7345:7752,x8[c(43:45,79:81)]])
c3
```

```
##                   var_RL_xacc var_RL_yacc var_RL_zacc DFT_Mag_1_RL_xacc
## var_RL_xacc         1.0000000   0.8001076   0.7001684         0.6871837
## var_RL_yacc         0.8001076   1.0000000   0.7447831         0.7325013
## var_RL_zacc         0.7001684   0.7447831   1.0000000         0.5972789
## DFT_Mag_1_RL_xacc   0.6871837   0.7325013   0.5972789         1.0000000
## DFT_Mag_1_RL_yacc   0.6503777   0.8400602   0.5894472         0.7230972
## DFT_Mag_1_RL_zacc   0.3024238   0.3736950   0.5121457         0.1091070
##                   DFT_Mag_1_RL_yacc DFT_Mag_1_RL_zacc
## var_RL_xacc               0.6503777         0.3024238
## var_RL_yacc               0.8400602         0.3736950
## var_RL_zacc               0.5894472         0.5121457
## DFT_Mag_1_RL_xacc         0.7230972         0.1091070
## DFT_Mag_1_RL_yacc         1.0000000         0.3465359
## DFT_Mag_1_RL_zacc         0.3465359         1.0000000
```

As we can see in these 3 matrices the values are completely different per activity now. To illustrate this let us compare the correlation and the mean of the last column of these correlation matrices (Feature: Magnitude_1, z-axis):

Cor.a01/a10	Cor.a01/a19	Cor. a10/a19
-0.2605215	0.021387	0.7041768

mean sitting	mean walking	mean basketball
-0.1476828	0.0458754	0.3287815

Even though there is still some correlation between the correlations of basketball and walking (0.704) I think it is safe to say that the relation between the Feature are quite different per activity. So at least this perspective gives some evidence that there are specific patterns per activity with which we "can work with".

After this deep dive into the data we now tackle the challenge of feature reduction. This is part of every data pre-processing but in this specific case it also has a very practical reason: With a feature set of 1215 features I would hardly be able to calculate any model on my laptop... For the feature reduction I followed the steps as they are described in the caret package[6]. We start with the "near zero variance" analysis:

```
#As a first step let's check how much of a reduction we get from "near zero variance" analysis:
nzv <- nearZeroVar(Feature_train)
length(nzv)
```

```
## [1] 16
```

```
#remove these Features from Feature_train
Feature_train <- Feature_train[,-nzv]
#new dimension of Feature_train dataset:
dim(Feature_train)
```

```
## [1] 7752 1199
```

There were 16 Features which had a variance close to zero. As we will use a tree-based model and they do not like this type of predictors at all I have removed them from the Feature_train dataset. So, we are down to 1199 Features - still far too many for the models I want to work with. Next step is the analysis of the correlation between the predictors:

```
#Find highly correlated predictors:
FeatureCor <-  cor(Feature_train)
summary(FeatureCor[upper.tri(FeatureCor)])
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.965458 -0.084893  0.004298  0.030766  0.116634  0.992255
```

```
length(findCorrelation(FeatureCor, cutoff = 0.9)) # (3 Features when >= 99%)
```

```
## [1] 355
```

When we set the cut off at 0.9 we get 355 Features with an equal or higher correlation. As I need a way to reduce the feature-set it is tempting to remove them from the dataset. But I decided against it. First also in the caret documentation it is written that some models need highly correlated data and second the data analysis above shows differences in the correlations per activity which helps for the "pattern finding". I will keep this in mind and would come back to this decision when I have problems to generate accurate models. Linear Dependencies are another potential source for feature reductions:

```
#Find linear dependencies:
findLinearCombos(Feature_train)
```

```
## $linearCombos
## list()
##
## $remove
## NULL
```

No luck with this one - there are no linear dependencies between the features...
As we still have 1199 features, we will go for a PCA decomposition next. While applying PCA I got unsure about one topic: prcomp() is centring the values by default so no need to do it before, but what to do with the "scale." parameter? To show the effect of this parameter on the PCA results we will run both options and show for each version...

... the development of the cumulative sum in a graph ... at which PC the cumulative sum explains 99% of the variance *... the 8 Feature-Loadings which contribute the most to PC1

[6] http://topepo.github.io/caret

First will follow the values for the PCA decomposition with scale.=FALSE.

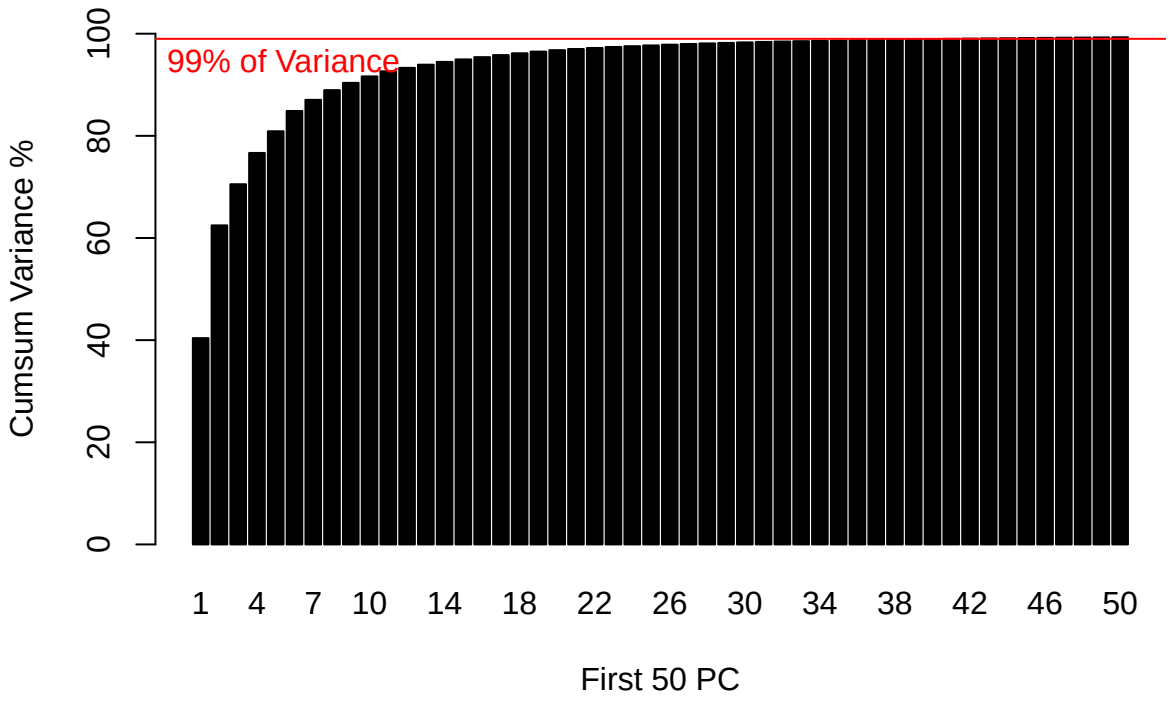

Figure 8: Cumulative sum of PCA variance for PCA analysis scale.=FALSE

```
#Find # of PC which explain more dann 99% of Variance
PC_99 <- match("TRUE",cumsum(variance/sum(variance))>=0.99)-1
PC_99
```

```
## [1] 41
```

```
#Show top loadings in PC1:
TopLoadings <- round(loading, 2)[,1:10]
Selection <- head(sort(abs(TopLoadings[,1]),decreasing = TRUE),8)
Selection
```

```
## DFT_Mag_1_LL_xacc DFT_Mag_1_RL_xacc  DFT_Mag_2_T_xacc DFT_Mag_2_LL_xacc
##              0.29              0.27              0.25              0.25
## DFT_Mag_2_RL_xacc  DFT_Mag_1_T_xacc DFT_Mag_1_LA_xacc DFT_Mag_1_RA_xacc
##              0.24              0.19              0.19              0.18
```

Now the same for the PCA decomposition with scale.=TRUE:

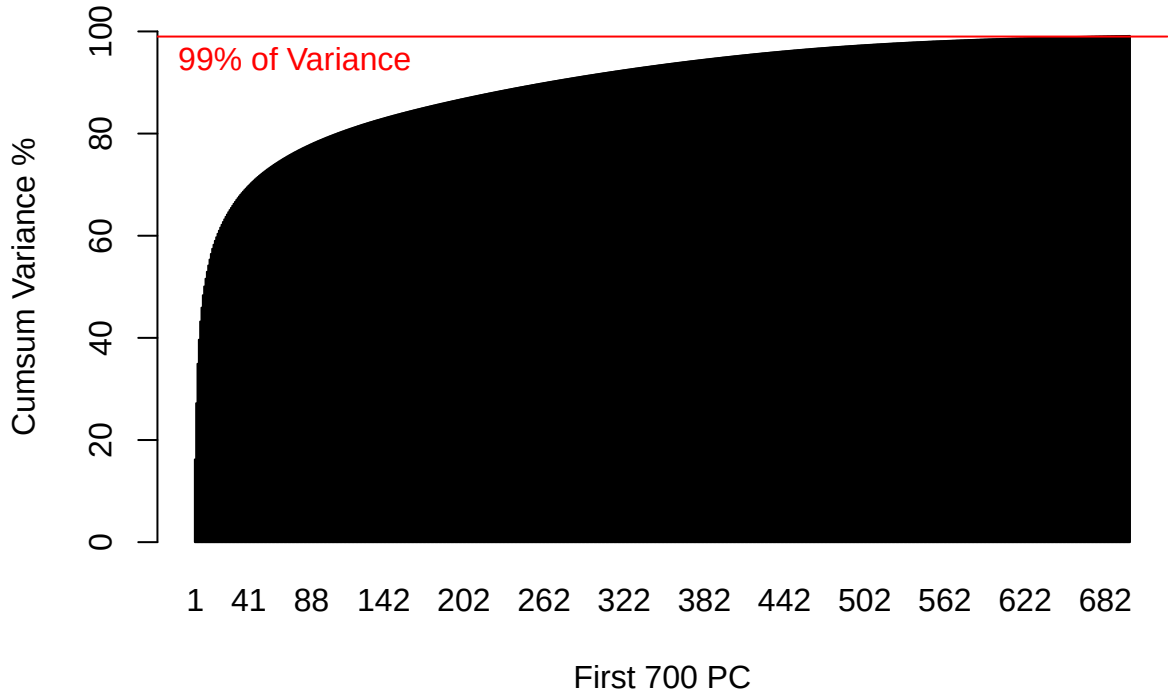

Figure 9: Cumulative sum of PCA variance for PCA analysis scale.=TRUE

```
#Find # of PC which explain more dann 99% of Variance
PC_991 <- match("TRUE",cumsum(variance1/sum(variance1))>=0.99)-1
PC_991
```

```
## [1] 676
```

```
#Show top loadings in PC1:
TopLoadings1 <- round(loading1, 2)[,1:10]
Selection1 <- head(sort(abs(TopLoadings1[,1]),decreasing = TRUE),8)
Selection1
```

```
##      min_RA_zgyro      min_LL_xgyro      min_LL_zgyro      max_RA_ygyro
##              0.06              0.06              0.06              0.06
##      max_LA_zgyro  DFT_Mag_4_T_yacc  DFT_Mag_5_T_yacc DFT_Mag_3_T_zgyro
##              0.06              0.06              0.06              0.06
```

The two PCA decompositions lead to vastly different results. This mainly because of the Magnitude values of the DFT calculation, as they are in a higher scale then most of the other predictors. As Steven M. Holland explains in his PCA tutorial[7] if "scale.=FALSE" is set then the whole decomposition will be based on the variance-covariance matrix and higher values will "dominate" the results. In contrast when "scale.=TRUE" PCA is based on a correlation matrix and higher scaled values do not influence the results. This fact gets proven when we look at the top 8 features of PC1. While on the PCA with "scale.=FALSE" this top 8 list is completely dominated by the Magnitude features the top 8 feature list for PCA "scale.=TRUE" has a broader variety (but includes some "Magnitude-Features" too).

[7] https://strata.uga.edu/software/pdf/pcaTutorial.pdf

Now, which option to choose? My initial thought was to set scale.=TRUE as the Magnitude values - especially for DC1 values - are really "out of scale" compared to the others. But then I would have to take 676 PC (features) into my model if I wanted to have the same level of 99% of variance explained (or 264 for variance >=90%). So, again the practical reasons dominated and I decided to go with the version where I set the parameter on "scale.=FALSE", as there I have only to consider 41 Features. A bit of a reassurance for this decision I got from the study of Altun/Barshan/Tuncel, as they mentioned 30 PC they considered for their models - in a PCA with scale.=TRUE this would be a selection which explains only around 60% of the variance, so I assume they set scale. on "FALSE" too. Nevertheless, we will handle it as with the "highly correlated features" above: When we do not get appropriate results, we will come back to this decision.
To get a feeling for the results of the PCA calculation (now all to follow with "scale.=FALSE") let's have a quick look at two graphs. The first graph is a biplot which shows the distribution of the "sample scores" (activities) and the Feature loadings of the top 8 contributors of the PC1 for the first two PCs:

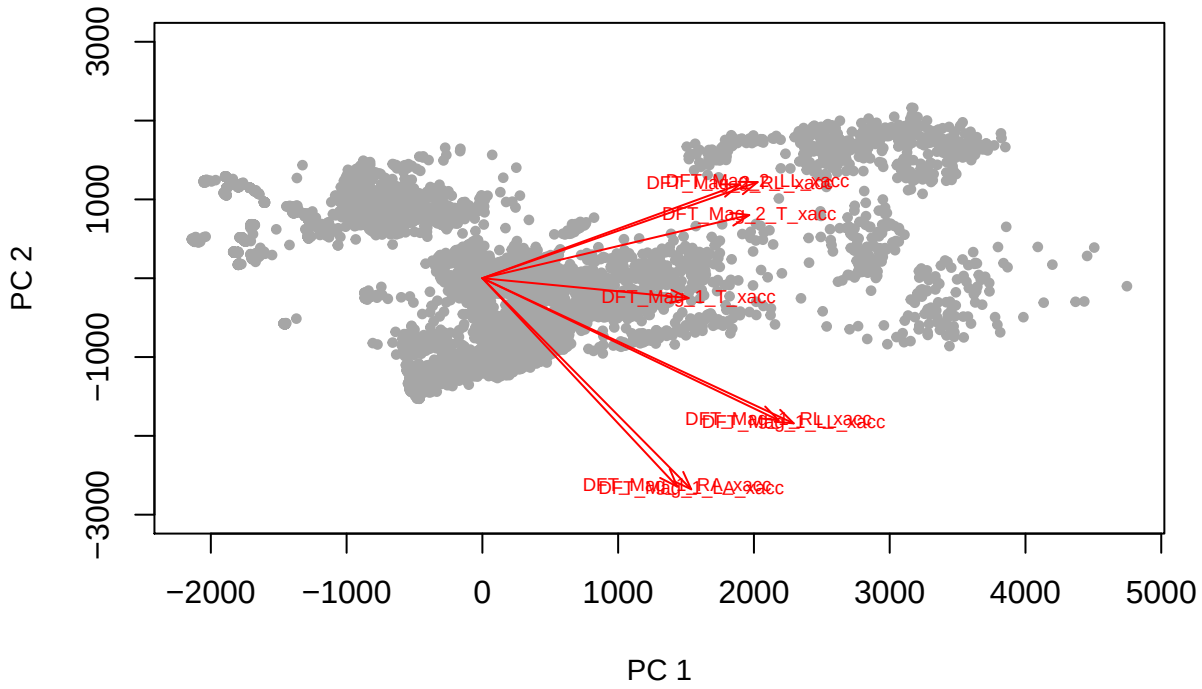

Figure 10: Biplot for PC1 and PC2 (Loading values: Top 8 contributors for PC1)

The second graph zooms into the sample scores and shows the distribution of the three activities we used for our data analysis so far (sitting, walking, basketball):

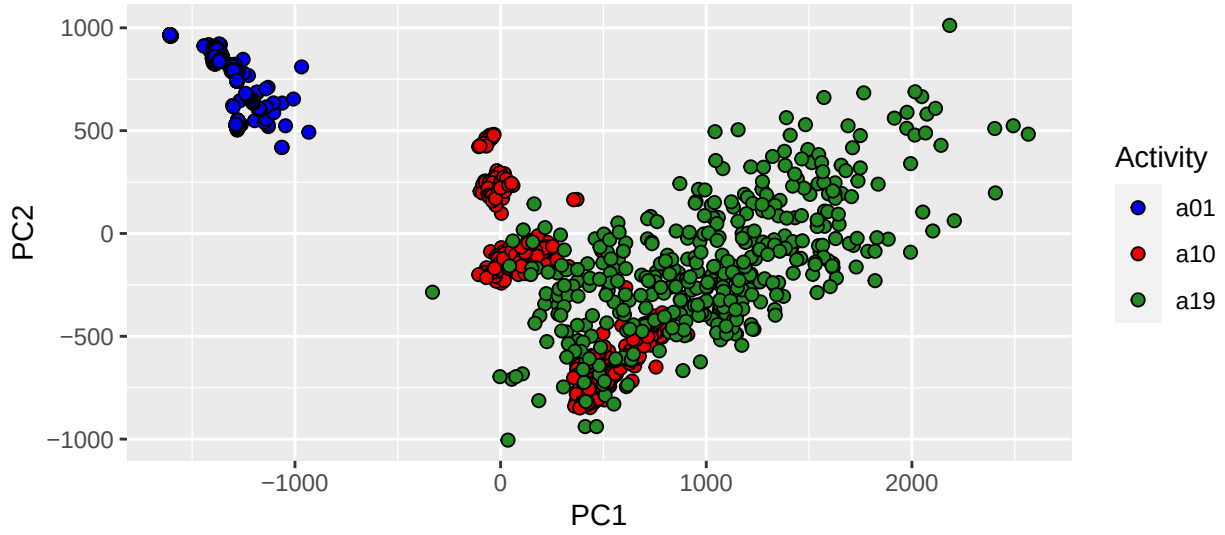

Figure 11: Distribution of a01, a10, a19 in realtion to PC1 and PC2

As we can see a01 (sitting) is clear distinguishable from the other two, but also a10 (walking) and a19 (basketball) have their own specific patterns. Therefore, hope is justified that also with the chosen option for the scale.-parameter we will have a good chance to distinguish the different activities from one another. The next chapters will deliver us the final answer.

0.3.4 Train the models

Before we start a short word about the content of this chapter: In the script I structured the code in a way that there are all actions needed per model in one block - from training the model to measuring the accuracy after applying the algorithm against the test dataset. But to get a clear(er) structure in this report we will not discuss or disclose the achieved accuracy in this chapter but in the upcoming "Analyse and discuss the results" chapter. As a starting point for the model training, I created two new tibbles: pca_train and pca_test. For pca_train I took the scores-matrix (pca$x) and selected the first 41 columns (PC1 to PC41), added a further column named "Activity" with the corresponding activities per row and finally changed this new column to class "factor". For the first 5 rows/columns this new tibble looks like following:

```
##   Activity       PC1      PC2       PC3      PC4
## 1      a01 -1386.851 825.6180 29.737486 232.9921
## 2      a01 -1387.038 823.2160 24.226170 226.7600
## 3      a01 -1392.353 823.8635 15.785827 225.9626
## 4      a01 -1390.527 826.6909 14.020306 222.1135
## 5      a01 -1397.564 829.1750  9.632741 220.6968
```

And as we can see the class of "Activity" is "factor":

```
class(pca_train[,"Activity"])
```

```
## [1] "factor"
```

Next, we generate pca_test. We get this dataset by multiplying the vector of Feature_test column means with the loading/rotation matrix of the pca decomposition of Feature_train[8] - not not to forget to subtract the "near zero variance" features first...:

```
Feature_test <- Feature_test[,-nzv] #so that we have again the same Feature-Set as in train set
test_labels <- rownames(Feature_test)
pca_test <- sweep(Feature_test, 2, colMeans(Feature_test)) %*% loading
```

As we now have pca_test we also change this matrix into a tibble with the exact same steps as explained above.

The last step before we start model training is to decide about the "fitControl" parameters. As I had again to consider the existing compute power, I was thinking about using a k-fold cross validation with k=10 or a repeated k-fold cross validation with k=5 and 2 repetitions. At the end I decided for the later - to be honest, without any smart reasoning behind. In the "Machine Learning" module of this HarvardX course we learned that k=5 is ok and adding a second iteration just sounded reasonable to me...:

```
#k-Fold CrossValidation with 5 folds and 2 repetitions:
fitControl <- trainControl(
  method = "repeatedcv",
  number = 5,
  repeats = 2,
  verboseIter = FALSE) #TRUE in the code script
```

Now we are at the point to choose the models. After a certain back and forth I decided for the following set of models:

- Penalized Multinomial Regression
- k-Nearest Neighbours
- naive Bayes
- Support Vector Machines (SVM)
- Gradient Boosting (linear)
- Gradient Boosting (tree)

Why those? I wanted to use different models as Altun/Barshan/Tuncel used for their study - exceptions are k-Nearest Neighbours and SVM, as I then have something to compare my results with. Another reason was that I only wanted to use models provided by the caret package. Even so I wanted to have a mix from models I knew from the HarvardX courses and models I have never seen before. The ones I have never seen before are Penalized Multinomial Regression, SVM and the two Gradient Boosting models. With the exception of SVM (this one I took in because of the Altun/Barshan/Tuncel study), I have chosen them because of a book I started to read[9]. The book itself is about deep learning but it also includes a chapter about what they call "shallow learning". There "logistic Regression" is named as default choice and a good starting point to set a benchmark for the "rest". But to apply this method in a multiclass challenge means either to apply the "one-vs-rest" scheme or use a regression method which is adapted to multiclass challenges. After checking caret documentation I found the "Penalized Multinomial Regression" as my answer to the question. So, I went for this. And "Gradient Boosting" Methods were praised as the most successful ones in the Kaggle competitions - so it was clear that I wanted them in my set of models...
Concerning the different tuning parameters I also had to find some compromises between the existing compute power and an optimally tuned model. So, what I did is to first let run every model with its default settings with a k-fold Cross Validation with k=10. Then I sorted the achieved results (best to worst) and looked at the influence of each parameter constellation. Then I fixed all parameters which did not change in the top 10

[8] https://rafalab.github.io/dsbook/large-datasets.html
[9] Deep Learning with R

results at this value for my model-trainings. Even so I deleted values which did not seem to have a lot of influence from the list of values the remaining variable parameters shall consider. I cannot demonstrate this process in this report or my script anymore, as this would take far too much time to always load this again. What the set of parameters per model are we will now see directly in the outputs of the training process:

```
## Penalized Multinomial Regression
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
##               'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results across tuning parameters:
##
##   decay  Accuracy   Kappa
##   0e+00  0.9377553  0.9342973
##   1e-04  0.9377553  0.9342973
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was decay = 1e-04.
```

```
## [1] "****************************************"
```

```
## k-Nearest Neighbors
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
##               'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.9882625  0.9876103
##    7  0.9860041  0.9852266
##    9  0.9840686  0.9831835
##   11  0.9818753  0.9808684
##   15  0.9777456  0.9765092
##   20  0.9746510  0.9732427
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 5.
```

```
## [1] "****************************************"
```

```
## Naive Bayes
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
```

```
##              'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9620759  0.959969
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'usekernel' was held constant at a value of TRUE
## Tuning
##  parameter 'adjust' was held constant at a value of 1
## [1] "****************************************"
## Support Vector Machines with Linear Kernel
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
##               'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results across tuning parameters:
##
##   cost  Accuracy   Kappa
##   0.5   0.9821353  0.9811428
##   1.0   0.9832951  0.9823670
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cost = 1.
## [1] "****************************************"
## eXtreme Gradient Boosting
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
##               'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results across tuning parameters:
##
##   lambda  eta  Accuracy   Kappa
##   0.0     0.3  0.9859395  0.9851584
##   0.1     0.4  0.9855526  0.9847499
##
## Tuning parameter 'nrounds' was held constant at a value of 150
```

```
## Tuning
##  parameter 'alpha' was held constant at a value of 0
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were nrounds = 150, lambda = 0, alpha =
##  0 and eta = 0.3.
```

```
## [1] "**************************************"
```

```
## eXtreme Gradient Boosting
##
## 7752 samples
##   41 predictor
##   19 classes: 'a01', 'a02', 'a03', 'a04', 'a05', 'a06', 'a07', 'a08', 'a09', 'a10', 'a11',
##               'a12', 'a13', 'a14', 'a15', 'a16', 'a17', 'a18', 'a19'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 6202, 6203, 6201, 6201, 6201, 6199, ...
## Resampling results across tuning parameters:
##
##   max_depth  colsample_bytree  subsample  nrounds  Accuracy   Kappa
##   2          0.6               0.50       100      0.9878758  0.9872022
##   2          0.6               0.50       150      0.9887136  0.9880866
##   2          0.6               0.75       100      0.9881967  0.9875410
##   2          0.6               0.75       150      0.9892285  0.9886300
##   2          0.8               0.50       100      0.9872301  0.9865206
##   2          0.8               0.50       150      0.9882624  0.9876103
##   2          0.8               0.75       100      0.9875528  0.9868612
##   2          0.8               0.75       150      0.9884559  0.9878146
##   3          0.6               0.50       100      0.9885838  0.9879495
##   3          0.6               0.50       150      0.9890351  0.9884259
##   3          0.6               0.75       100      0.9889068  0.9882905
##   3          0.6               0.75       150      0.9893581  0.9887669
##   3          0.8               0.50       100      0.9887149  0.9880879
##   3          0.8               0.50       150      0.9891017  0.9884962
##   3          0.8               0.75       100      0.9890358  0.9884267
##   3          0.8               0.75       150      0.9891005  0.9884949
##
## Tuning parameter 'eta' was held constant at a value of 0.3
## Tuning
##  parameter 'gamma' was held constant at a value of 0
## Tuning
##  parameter 'min_child_weight' was held constant at a value of 1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were nrounds = 150, max_depth = 3, eta
##  = 0.3, gamma = 0, colsample_bytree = 0.6, min_child_weight = 1 and subsample
##  = 0.75.
```

Now all models are trained and at least the training results look not too bad already. Let us see what results we achieved with the test dataset in the next chapter.

0.4 Analyse and discuss the results

Here the table of the results after applying the trained models against pca_test:

method	Accuracy
eXtreme Gradient Boosting (Tree)	0.9948830
Support Vector Machines	0.9926901
eXtreme Gradient Boosting (linear)	0.9904971
k-Nearest Neighbours	0.9904971
Naive Bayes	0.9736842
Penalized Multinomial Regression	0.9576023

As we can see we have 4 models which outdo the set goal of 99% accuracy. Therefore, I do not see the necessity during this project to go back on some of the decisions we have taken so far. Even so I think the results are good enough to be applied in practice. But to discuss this further let us have a closer look at the predictions of the different models:

```
## [1] "Gradient Boosting (tree)"
```

```
##            Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a04   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  72   1   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a06   0   0   0   0   0  71   0   1   0   0   0   0   0   0   0   0   0   1   0
##        a07   0   0   0   0   0   0  71   1   0   0   0   0   0   0   0   0   0   0   0
##        a08   0   0   0   0   0   0   1  69   0   0   0   0   0   0   0   0   0   0   0
##        a09   0   0   0   0   0   0   0   1  72   0   0   0   0   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0
##        a11   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0
##        a14   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   1   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  71   0   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0
##        a18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  71   0
##        a19   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72
```

```
## [1] "Support Vector Machine"
```

```
##            Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  72   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a04   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a06   0   0   0   0   0  72   0   1   0   0   0   0   0   0   0   0   0   0   0
##        a07   0   0   0   0   0   0  70   4   0   0   0   0   0   0   0   0   0   0   0
##        a08   0   0   0   0   0   0   1  65   0   0   0   0   0   0   0   0   0   0   1
##        a09   0   0   0   0   0   0   0   1  72   0   0   0   0   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0
##        a11   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0
```

```
##        a14   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0
##        a18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0
##        a19   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0  71
```

```
## [1] "Gradient Boosting (linear)"
```

```
##           Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a04   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  71   1   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a06   0   0   0   0   0  71   0   1   0   0   0   0   0   0   0   0   0   0   0
##        a07   0   0   0   0   0   0  71   2   0   0   0   0   0   0   0   0   0   0   0
##        a08   0   0   0   0   1   0   1  69   0   0   0   0   2   0   0   0   0   0   1
##        a09   0   0   0   0   0   0   0   0  72   0   0   0   1   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0
##        a11   0   0   0   0   0   0   0   0   0   0  71   0   0   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   1   0  69   0   0   0   0   0   0
##        a14   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   1   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  71   0   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0
##        a18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   1
##        a19   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  70
```

```
## [1] "k-Nearest Neighbour"
```

```
##           Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  72   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a04   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  71   0   0   1   1   0   0   0   0   0   0   0   0   0   0
##        a06   0   0   0   0   0  72   0   1   0   0   0   0   0   0   0   0   0   0   0
##        a07   0   0   0   0   0   0  72   4   0   0   0   0   0   0   0   0   0   0   0
##        a08   0   0   0   0   1   0   0  63   0   0   0   0   0   0   0   0   0   0   1
##        a09   0   0   0   0   0   0   0   1  71   0   0   0   0   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0
##        a11   0   0   0   0   0   0   0   0   0   0  72   0   1   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   0   0  71   0   0   0   0   0   0
##        a14   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0
##        a18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0
##        a19   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0  71
```

```
## [1] "naive Bayes"
```

```
##           Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  72   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a04   0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  68   0   0   0   3   0   0   0   1   0   0   0   0   0
##        a06   0   0   0   0   0  71   0   2   0   0   0   0   0   0   0   0   0   0
##        a07   0   0   0   0   0   0  71   5   0   0   0   0   0   0   0   0   0   0
##        a08   0   0   0   0   1   1   1  63   0   0   1   0   5   0   0   0   0   0
##        a09   0   0   0   0   1   0   0   0  68   2   0   0   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   1  68   2   0   0   0   0   0   0   0
##        a11   0   0   0   0   1   0   0   0   0   0  68   0   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  72   0   1   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   0   0  66   0   0   0   0   0
##        a14   0   0   0   0   0   0   0   0   0   0   0   0   0  71   0   0   0   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  70   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1  72   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0
##        a18   0   0   0   0   1   0   0   0   0   1   1   0   0   0   1   0   0  72
##        a19   0   0   0   0   0   0   0   1   0   1   0   0   0   0   0   0   0   0  72
```

```
## [1] "Penalized Multinomial Regression"
```

```
##           Reference
## Prediction a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19
##        a01  72   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
##        a02   0  65   0   3   0   0   6   2   0   0   0   1   0   0   0   0   0   0   0
##        a03   0   0  72   0   0   0   0   0   0   0   0   6   0   0   0   0   0   0   0
##        a04   0   0   0  69   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a05   0   0   0   0  66   0   0   0   0   0   1   0   0   0   0   0   0   0   0
##        a06   0   0   0   0   3  71   0   0   0   0   0   0   0   0   0   0   0   0   0
##        a07   0   5   0   0   0   0  63   5   0   0   0   0   0   0   0   0   0   0   0
##        a08   0   2   0   0   2   1   3  62   0   0   0   1   3   0   0   0   0   0   1
##        a09   0   0   0   0   1   0   0   1  69   1   0   0   0   0   0   0   0   0   0
##        a10   0   0   0   0   0   0   0   0   3  71   1   0   0   0   0   0   0   0   0
##        a11   0   0   0   0   0   0   0   0   0   0  70   0   0   0   0   0   0   0   0
##        a12   0   0   0   0   0   0   0   0   0   0   0  61   0   0   0   0   0   0   0
##        a13   0   0   0   0   0   0   0   0   0   0   0   0  69   0   0   0   0   0   0
##        a14   0   0   0   0   0   0   0   0   0   0   0   2   0  72   0   0   0   1   0
##        a15   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0
##        a16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0
##        a17   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0
##        a18   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0  71   0
##        a19   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0  71
```

Looking at these matrices we see that the prediction of a08 (moving around in an elevator) was the hardest, it got especially often confused with a07 (standing in an elevator still). Interestingly to predict a07 seems to be easier. This is probably since on a small space like an elevator it was hard to permanently be in motion for 5 minutes (recording time). Therefore, chances are high, that there are some samples labelled with a08 which have in fact more in common with a07, as for a (split) second the subject stood still. It also shows that "Penalized Multinomial Regression" and "naive Bayes" are probably not the best algorithms for this specific challenge. We also observed this during the training with naive Bayes as it generates quite a lot of warnings with the message "XX: In FUN(X[[i]], ...) : Numerical 0 probability for all classes with observation xx".

According to a post of "topepo" in a chat forum on GitHub[10] this 0 probability issue comes because naive Bayes is not the best model for challenges with a lot of predictors. Even though we still have a not too bad accuracy of over 97% a set of 41 predictors seems already to "scratch" at these limits... Next let us have a look at the overall statistics of the 4 top performing models:

```
## [1] "Gradient Boosting (tree)"
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull AccuracyPValue
##     0.99488304     0.99459877     0.98948569     0.99794032     0.05263158     0.00000000
##  McnemarPValue
##            NaN
```

```
## [1] "Support Vector Machine"
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull AccuracyPValue
##     0.99269006     0.99228395     0.98659797     0.99648920     0.05263158     0.00000000
##  McnemarPValue
##            NaN
```

```
## [1] "Gradient Boosting (linear)"
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull AccuracyPValue
##     0.99049708     0.98996914     0.98380453     0.99493067     0.05263158     0.00000000
##  McnemarPValue
##            NaN
```

```
## [1] "k-Nearest Neighbor"
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull AccuracyPValue
##     0.99049708     0.98996914     0.98380453     0.99493067     0.05263158     0.00000000
##  McnemarPValue
##            NaN
```

Again, the good results get confirmed, no (negative) surprises with Kappa-values, nor with the confidence interval or the p-values. We could also have a look at the sensitivity and specificity rates now. But as we have seen in the prediction/reference-matrices there were only few wrong predictions and only a08 will have a bit a lower sensitivity (it is at 0.9583333) because of some "false negative" values. For this reason, we skip this step.
But there was one thing which bothered me about the results: "k-Nearest Neighbour" and "Gradient Boosting (linear)" deliverd the exact same result. This obviously because they had the same amount of wrong predictions:

```
#Same amount of wrong predictions for knn and Gradient Boosting (linear):
sum(CM_gb$table[upper.tri(CM_gb$table)],CM_gb$table[lower.tri(CM_gb$table)])
```

```
## [1] 13
```

```
sum(CM_gb$table[upper.tri(CM_knn$table)],CM_knn$table[lower.tri(CM_knn$table)])
```

```
## [1] 13
```

First I thought I made a mistake with the variables or so. But when we look at the two prediction/reference-matrices again we see that they look different, so this was "just" a coincident. As the top 4 models are so close together chances are quite high that the same amount of wrong predictions may occur, even though based on different "ways to get there". Now I want to come back to my statement "good enough to be applied in practice": We see that the predictions are really very accurate - even more when we would exclude this very specific use case of "walking in an elevator". Even so all other parameters which could give us hints concerning some potential issues are not striking at all. So yes, I think with these results we could really carry on and try to apply them in some real-life use cases. To close this chapter let us have a quick look to

[10] https://github.com/topepo/caret/issues/339

the results of the Altun/Barshan/Tuncel study:

Method	Correct differentiation rate (%) ± one standard deviation		
	RRSS	*P*-fold	L1O
BDM	99.1 ± 0.12	99.2 ± 0.02	75.8
RBA	81.0 ± 1.52	84.5 ± 0.44	53.6
LSM	89.4 ± 0.75	89.6 ± 0.10	85.3
k-NN (*k*=7)	98.2 ± 0.12	98.7 ± 0.07	86.9
DTW_I	82.6 ± 1.36	83.2 ± 0.26	80.4
DTW_D	98.5 ± 0.18	98.5 ± 0.08	85.2
SVM	98.6 ± 0.12	98.8 ± 0.03	87.6
ANN	86.9 ± 3.31	96.2 ± 0.19	74.3

Figure 12: Table of results of Altun/Barshan/Tulcen study

If we compare the results with the p-fold column we see that the results are comparable - at least when we look at their five best performing models. Interesting is that SVM was always in the top 2 for me, even though I played with the different options (and as I remember was always above 99%) - this seemed not to be the case in the their study. Nevertheless, I will not do a further deep dive on this as I do not see "under the hood" of their work and therefore do not know what exactly they have done.

0.5 Personal Conclusion

As this project achieved its goals I am quite happy with the results: The data loading part was a bit of a challenge but the final solution is quite efficient, the pre-processing part was very interesting and even opened up a whole new world to me with topics like Discrete Fourier Transform and complex numbers..., and last but not least the final accuracy scores were on eyes height with the results of the Altun/Barshan/Tulcen study which reassured me that during this whole journey no crucial mistakes did happen. Nevertheless, there are always aspects to improve or dig deeper into, so for me potential future work could be:

- Personal interests: As I have not learned a programming language before this course, I am a newbie in coding. So, I assume an experienced coder would have solved several topics more efficient then me, therefore it would certainly be interesting to have a look at some parts of my code with such an expert. Same applies for some mathematical parts as I did not find an answer on all the questions which popped up.
- Explore further optimisations: If I not had to deal with the compute (and RAM) limits of my laptop I would once like to work with the PCA version where I set scale.=TRUE to see how far I get with this option. Even so I am fascinated about deep learning but have no practical experience with it yet. To see how such an approach would perform on this type of problem would certainly be interesting too.

0.6 Main References

The main references I used are listed below. But these are just the sources I consulted multiple times throughout the project. Beside those I also googled several things and therefore read multiple websites, chat forums, Wikipedia pages and so on to solve the different challenges I faced.

- K. Altun, B. Barshan, and O. Tuncel, "Comparative study on classifying human activities with miniature interial and magnetic sensors", Pattern Recognition, 43(10):3605-3620, October 2010
- Francois Chollet, J.J. Allaire, "Deep Learning with R", 2018, ISBN 9781617295546
- Steven M. Holland, "PRINCIPAL COMPONENTS ANALYSIS (PCA)", 3rd of December 2019, (https://strata.uga.edu/software/pdf/pcaTutorial.pdf)
- Rafael A. Irizarry, "Introduction to Data Science, Data Analysis and Prediction Algorithms with R", last update 2020-05-10, (https://rafalab.github.io/dsbook/large-datasets.html)
- Max Kuhn, "The caret Package", last update 2019-03-27, (http://topepo.github.io/caret)