Surface Detection by Robot Movements - Report

Marian Dumitrascu

March 31, 2019

Introduction

For this project I choose a Kaggle.com open competition project. This is *CareerCon 2019 - Help Navigate Robots*. Here is the description of the project from Kaggle:

> *In this competition, you'll help robots recognize the floor surface they're standing on using data collected from Inertial Measurement Units (IMU sensors). We've collected IMU sensor data while driving a small mobile robot over different floor surfaces on the university premises. The task is to predict which one of the nine floor types (carpet, tiles, concrete) the robot is on using sensor data such as acceleration and velocity. Succeed and you'll help improve the navigation of robots without assistance across many different surfaces, so they won't fall down on the job.*

The task is challenging, but I believe I can obtain a decent result using techniques and tools learned in this course. Please note that some code chunks are not showing in the PDF for aesthetic reasons. You can find all code behind in the .Rmd version.

Report Structure

1. First I will describe data provided and the output expected
2. then, will perform data analysis, visualization and get insights
3. pre-process and transform data,
4. analyse features and selectmost impartant ones
5. decide the model to use, try KNN, rpart and two flavosrs of randomForest
6. perform the final data prediction and show the results
7. draw some conclusions

Report and Data Location

I keep this project on GitHub at this Url: https://github.com/mariandumitrascu/ph125_9_HelpRobotsNavigate Data loaded by the R scripts is kept on an AWS public S3 bucket, to be easily loaded. This will be available for the duration of the grading.

The Data

Data Description

Input data from Kaggle consists in 4 files:

- *X_trian.csv* and *X_test.csv* - the input data, covering 10 sensor channels and 128 measurements per time series plus three ID columns:
  - *row_id*: The ID for this row.

  - *series_id*: ID number for the measurement series. Foreign key to y_train/sample_submission.
  - *measurement_number*: Measurement number within the series.

  The orientation channels encode the current angles of how the robot is oriented as a quaternion (see Wikipedia). Angular velocity describes the angle and speed of motion, and linear acceleration components describe how the speed is changing at different times. The 10 sensor channels are:

  - *orientation_X*
  - *orientation_Y*
  - *orientation_Z*
  - *orientation_W*
  - *angular_velocity_X*
  - *angular_velocity_Y*
  - *angular_velocity_Z*
  - *linear_acceleration_X*
  - *linear_acceleration_Y*
  - *linear_acceleration_Z*

- *y_train.csv* - the surfaces for training set.

  - *series_id*: ID number for the measurement series.
  - *group_id*: ID number for all of the measurements taken in a recording session. Provided for the training set only, to enable more cross validation strategies.
  - *surface*: labels or classes of the training data. this is the element that need to be predicted

- *sample_submission.csv* - a sample submission file in the correct format.

In this report I will split the training data into two partitions, will fit a model on the first one, and measure it's accuracy on the second. I will use a small part of data to make it run faster. The R script for generating the final results will use all data.

Data Analysis

I will make the following assumptions about observations:

- all observations are made using the same robot
- the interval between the 128 observations for each series is always the same.
- the surface is a plane, no stairs, hills or valleys

From a physicist perspective there are three forces involved: gravitation force, robot propulsion force, and friction force. Gravitation force is constant. Friction force depends on the surface by a coefficient and propulsion is an unknown variable. We need to basically determine the friction coefficient based on a movement pattern. Moving objects will travel longer if the surface has a lower friction than on a surface with higher friction. On the other side, changing direction can be slower on a surface with higher friction such as carpet.

Quick look at the first 5 rows of the training data:

row_id	series_id	measurement_number	orientation_X	orientation_Y	orientation_Z	orientation_W	angular_
0_0	0	0	-0.75853	-0.63435	-0.10488	-0.10597	
0_1	0	1	-0.75853	-0.63434	-0.10490	-0.10600	
0_2	0	2	-0.75853	-0.63435	-0.10492	-0.10597	
0_3	0	3	-0.75852	-0.63436	-0.10495	-0.10597	
0_4	0	4	-0.75852	-0.63435	-0.10495	-0.10596	

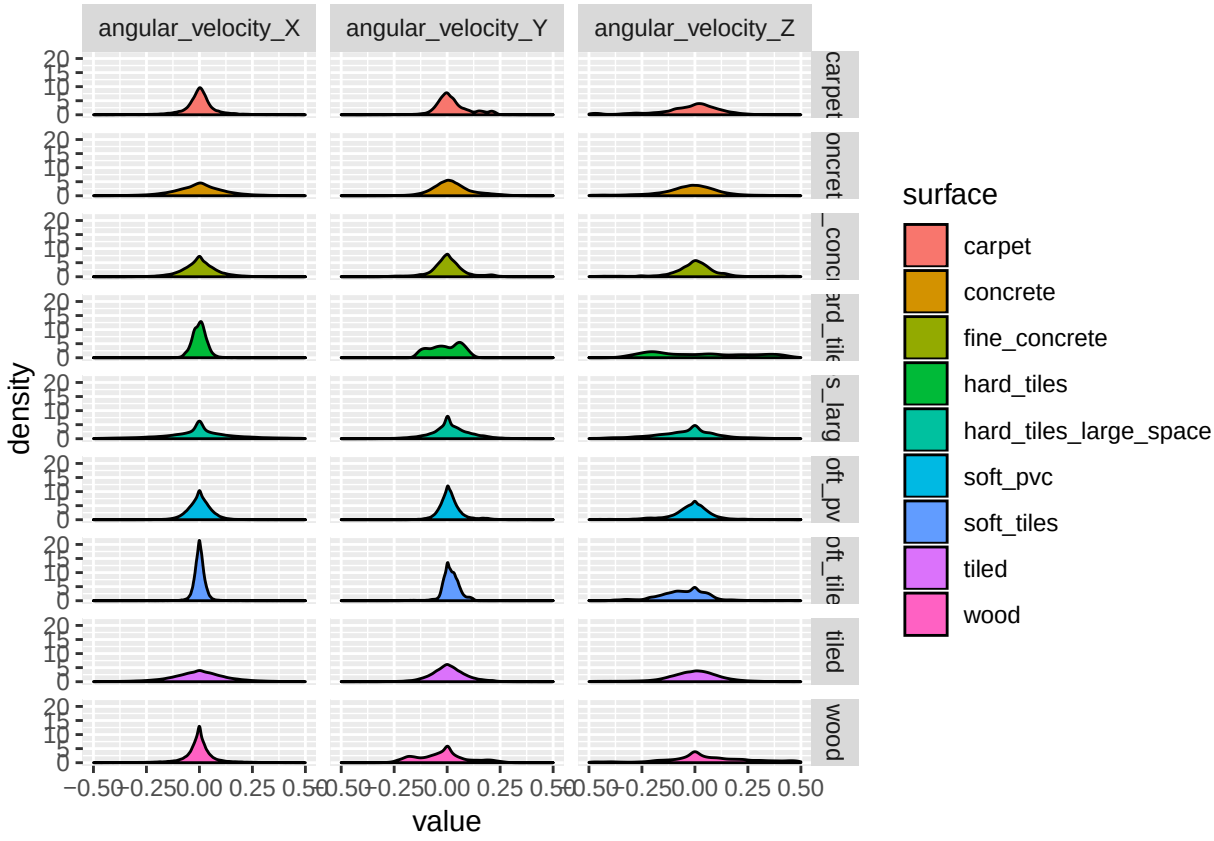
Data Distribution

Here is a distribution of measurements by surface in the training set:

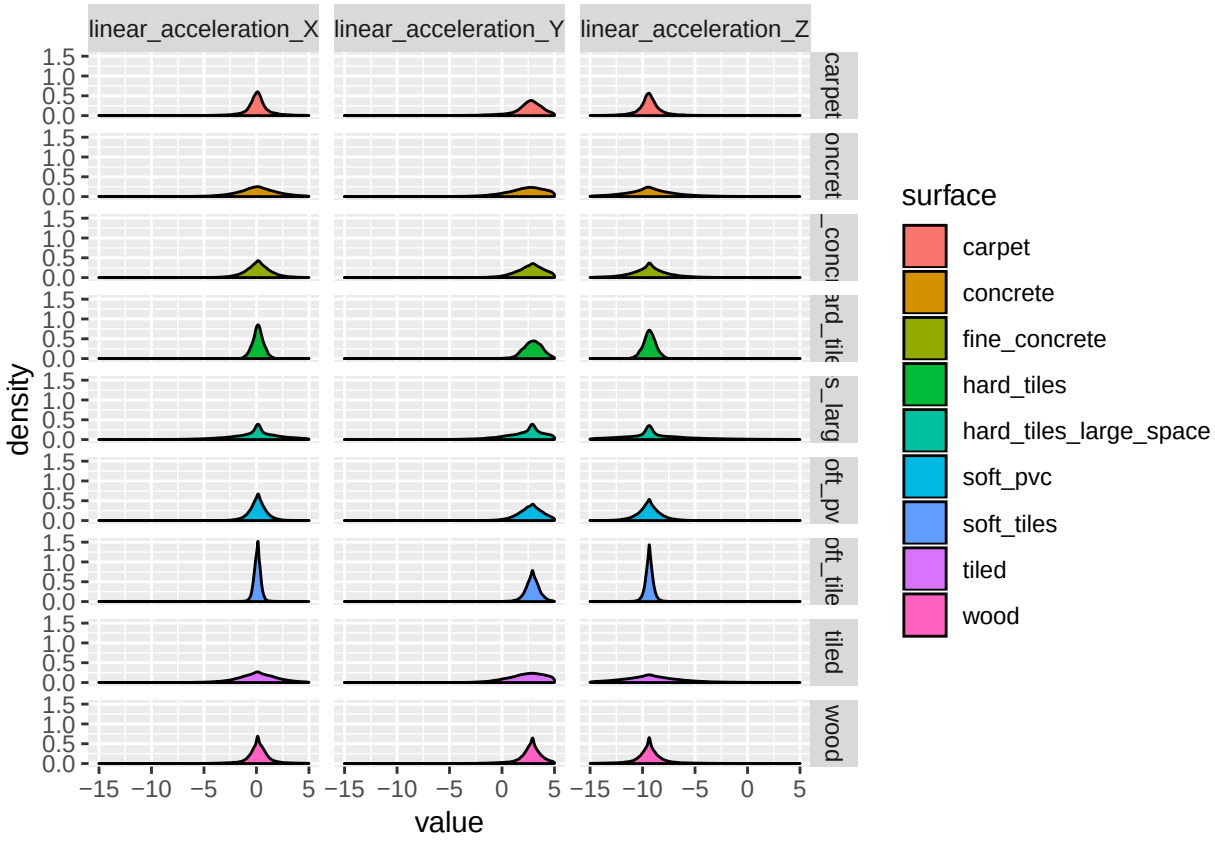
surface	measurements
carpet	189
concrete	779
fine_concrete	363
hard_tiles	21
hard_tiles_large_space	308
soft_pvc	732
soft_tiles	297
tiled	514
wood	607

We can see that we deal with an unbalanced data set. *hard_tiles* have only 21 records while *concrete* 779

Angular velocity data is produced by a magnetostatic sensor, it indicates the angular speed the robot is moving in reference with earth orientation. Here is the distribution of angular velocity by surface:



Linear acceleration data is produced by an inertial sensor. We can approximate this later with linear distance if we consider the unit of time to be 1. Here is the distribution of linear acceleration by surface:



We can observe noticeable differences in distribution of these variables by surface.

Orientation data comes from a gyroscope sensor. To draw the distribution of orientation, we'll convert quaternion values to euler angles which are more intuitive and easier to interpret. Euler angles provide a way to represent the 3D orientation of an object using a combination of three rotations about different axes:

- roll - rotation around x, noted *phi*,
- pitch - rotation around y, noted *theta*,
- yaw - rotation around z, noted *psi*

See the image below (from: http://www.chrobotics.com/library/understanding-euler-angles)

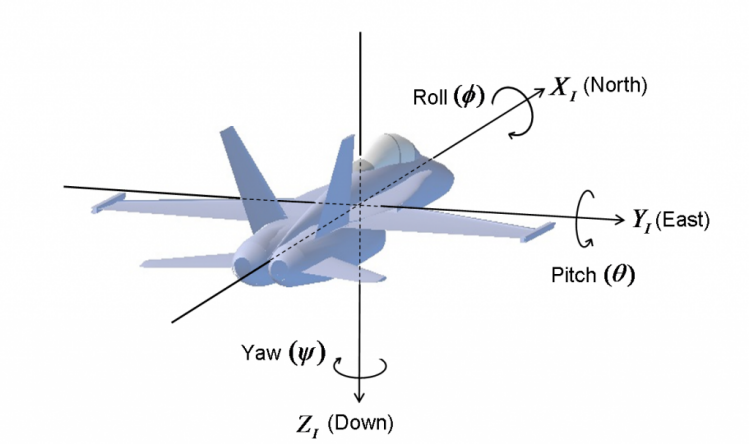


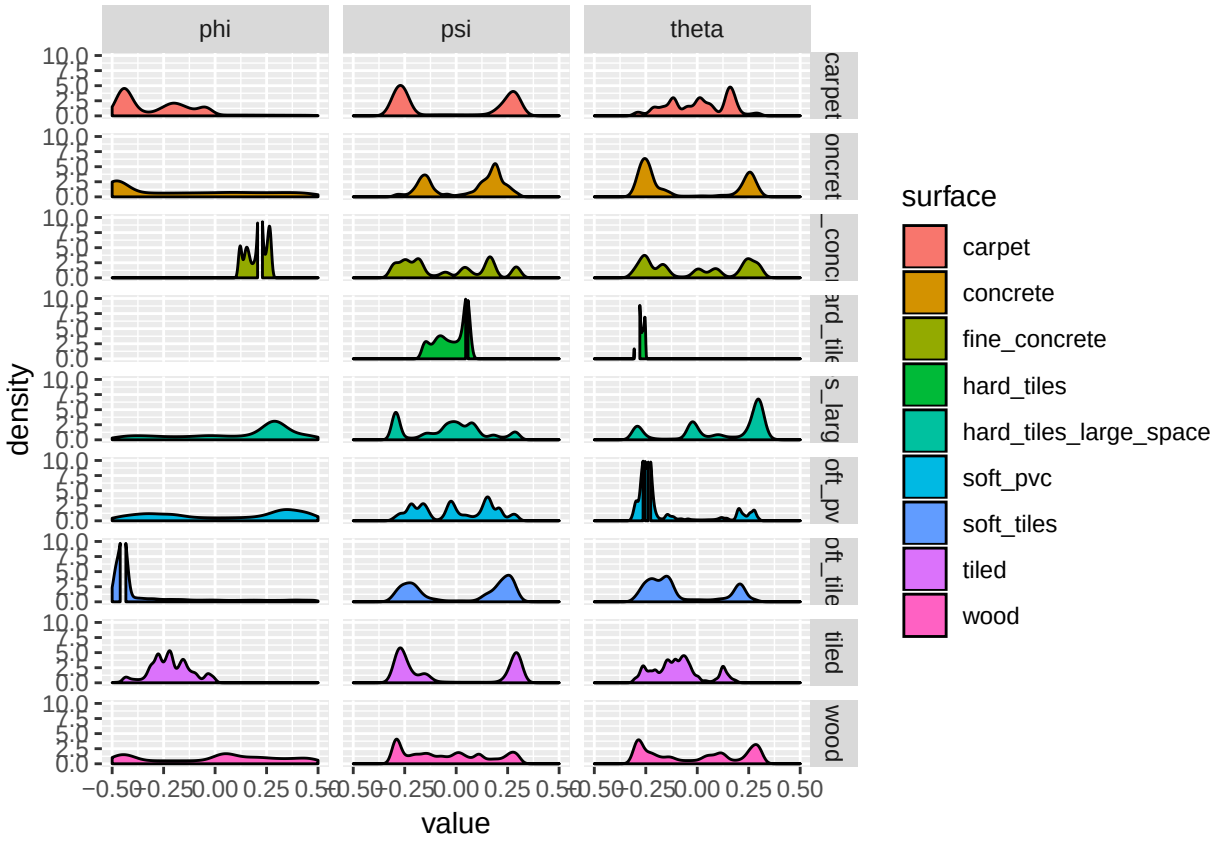
Figure 1: Euler Angles

To convert quaternions to euler angles I used *Q2EA* function from *orientlib* package.(See: https://www.rdocumentation.org/packages/orientlib/versions/0.10.3)

```
# define a function to convert quaternion values to euler angles.
convert_quaternions_to_euler <- function(a_dataset){

    # use Q2EA from RSpincalc to convert quaternions to euler angles
    Q <- a_dataset %>% select(
                    orientation_X,
                    orientation_Y,
                    orientation_Z,
                    orientation_W) %>%
                as.matrix()

    euler_matrix <- Q2EA(Q,
                 EulerOrder='xyz',
                 tol = 10 * .Machine$double.eps,
                 ichk = FALSE,
                 ignoreAllChk = FALSE)

    # add the new columns to the dataset
    a_dataset <- a_dataset %>% mutate(
                    phi = euler_matrix[,1],
                    theta = euler_matrix[,2],
                    psi = euler_matrix[,3])

    # remove quaternion columns
    a_dataset <- a_dataset %>% select(
                -orientation_X,
                -orientation_Y,
                -orientation_Z,
                -orientation_W)

    # return the new dataset
    a_dataset
```

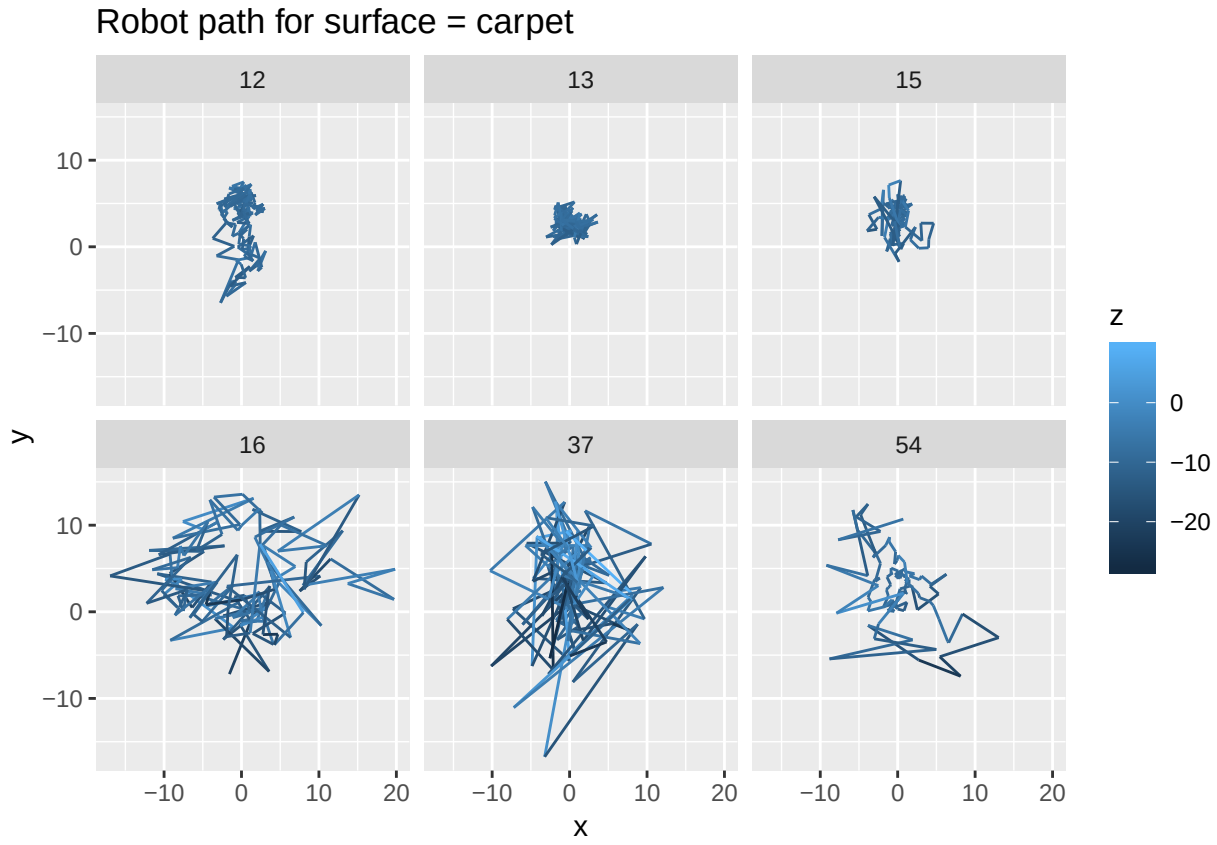
```
}

x_train <- convert_quaternions_to_euler(x_train)
x_test <- convert_quaternions_to_euler(x_test)
```

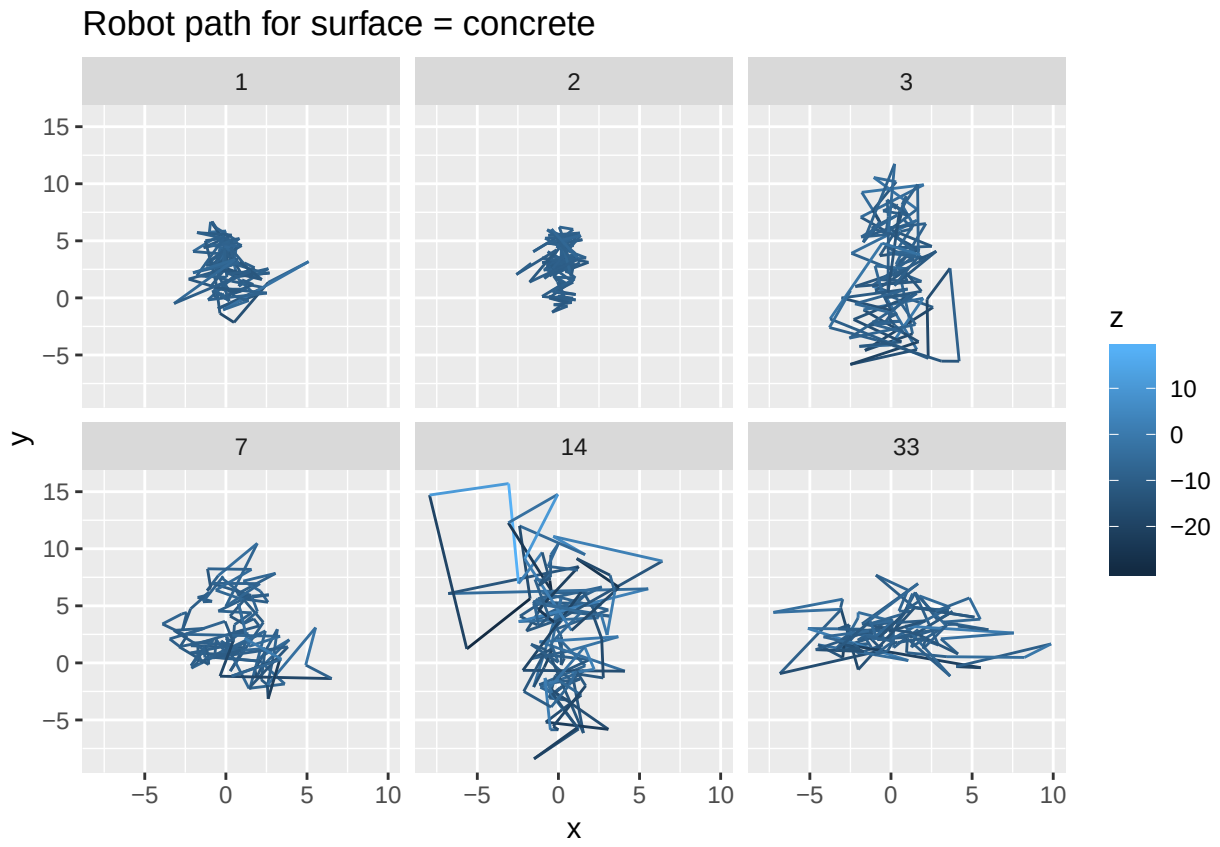
Distribution of orientation angles by surface:



It looks like the most noticeable differences between surfaces can be observed in the orientation distribution.

If we consider unit of time to be 1, we can draw the path of the robot movement. In the following figures I draw the path for several cases, faceted by surface. I expect that the surface would influence the shape of these paths.


Robot path for surface = carpet
12
13
15
16
37
54
y
x
z
10
0
−10
20
−20



I cannot observe obvious differences between paths on carpet vs concrete, but maybe a machine can. An idea would be to convert all paths to images and do an image analysis, but I will stick with a more classical approach for this project. I'll use several models from caret package: *KNN*, *rpart* and *randomForest.*

Data Pre-Processing

An important step in fitting a good model is pre-processing the data. In our case we have series of observation of 128 measurements. In this section I will create aggregations around these measurements. These are: mean, standard deviation, distances of segments from one point to the next. total distance, total angle change both from magnetostatic and gyro sensors. All of these are encapsulated in the following function:

```
# function for pre-processing
# n_of_rows defaults to total number ofseries
# is_train indicates that the data is training, thus will do an extra action
pre_process <- function(a_dataframe, n_of_rows = nrow(a_dataframe)/128 ) {

    # get data grouped by seeries_id and compute some means
    processed_data_df <- a_dataframe %>%
    group_by(series_id) %>%
    summarize(
        phi_mean_all = mean(phi),
        phi_sd_all = sd(phi),
        phi_mean_to_sd_all = mean(phi)/sd(phi),
        theta_mean_all = mean(theta),
        theta_sd_all = sd(theta),
```

```
        theta_mean_to_sd_all = mean(theta)/sd(theta),
        psi_mean_all = mean(psi),
        psi_sd_all = sd(psi),
        psi_mean_to_sd_all = mean(psi)/sd(psi),
        # this is the rectangular area that surounds the path of linear movement
        dist_area = (max(linear_acceleration_X) - min(linear_acceleration_X)) * (max(linear_acceleratio
            (max(linear_acceleration_X) - min(linear_acceleration_X)) * (max(linear_acceleration_Z) - mi
            (max(linear_acceleration_Y) - min(linear_acceleration_Y)) * (max(linear_acceleration_Z) - mi
        # this is the rectangular area that surounds the path of angular movement
        omega_area = (max(angular_velocity_X) - min(angular_velocity_X)) * (max(angular_velocity_Y) - mi
            (max(angular_velocity_X) - min(angular_velocity_X)) * (max(angular_velocity_Z) - min(angulai
            (max(angular_velocity_Y) - min(angular_velocity_Y)) * (max(angular_velocity_Z) - min(angulai
        euler_area = (max(phi) - min(phi)) * (max(theta) - min(theta)) +
            (max(phi) - min(phi)) * (max(psi) - min(psi)) +
            (max(theta) - min(theta)) * (max(psi) - min(psi)),
        dist_mean_x = mean(linear_acceleration_X),
        dist_mean_y = mean(linear_acceleration_Y),
        dist_mean_z = mean(linear_acceleration_Z),
        omega_mean_x = mean(angular_velocity_X),
        omega_mean_y = mean(angular_velocity_Y),
        omega_mean_Z = mean(angular_velocity_Z),
        dist_sd_x = sd(linear_acceleration_X),
        dist_sd_y = sd(linear_acceleration_Y),
        dist_sd_z = sd(linear_acceleration_Z),
        omega_sd_x = sd(angular_velocity_X),
        omega_sd_y = sd(angular_velocity_Y),
        omega_sd_Z = sd(angular_velocity_Z)

        ) %>%
        slice(1:n_of_rows)

    # define an empty data frame with summary metrics that we'll use for each set of 128 observations
    metrics <- c("dist_total","dist_max","dist_min","dist_max_to_min","dist_mean","dist_sd","dist_mean_
                 "omega_total","omega_max","omega_min","omega_max_to_min","omega_mean","omeg
                 "phi_total","phi_max","phi_min","phi_mean","phi_sd","phi_mean_to_sd",
                 "theta_total","theta_max","theta_min","theta_mean","theta_sd","theta_mean_
                 "psi_total","psi_max","psi_min","psi_mean","psi_sd","psi_mean_to_sd",
                 "euler_total","euler_max","euler_min","euler_mean","euler_sd","euler_mean_
    tmp_df <- data.frame(matrix(ncol = length(metrics), nrow = 0) )
    colnames(tmp_df) <- metrics

    # loop over each series and compute aggegations
    # should use apply type of function here, but I use "for" until I master the apply
    for (s_id in processed_data_df$series_id)
    {
        # get current measurement set
        this_chunk_df <- a_dataframe %>% filter(series_id == s_id)

        # select only columns we are interested in
        this_chunk_df <- this_chunk_df %>%
            select(
            phi, theta, psi,
            angular_velocity_X, angular_velocity_Y, angular_velocity_Z,
```

```
    linear_acceleration_X, linear_acceleration_Y, linear_acceleration_Z)
# this is a vector with euclidian distances from one point to the next
dist_v <-   sqrt(diff(this_chunk_df$linear_acceleration_X)^2 +
                     diff(this_chunk_df$linear_acceleration_Y)^2 +
                     diff(this_chunk_df$linear_acceleration_Z)^2)
omega_v <-  sqrt(diff(this_chunk_df$angular_velocity_X)^2 +
                     diff(this_chunk_df$angular_velocity_Y)^2 +
                     diff(this_chunk_df$angular_velocity_Z)^2)
phi_v <- abs(diff(this_chunk_df$phi))
theta_v <- abs(diff(this_chunk_df$theta))
psi_v <- abs(diff(this_chunk_df$psi))

# fill or temp data frame with summary computations for our 128 measurement set
tmp_df <- bind_rows(tmp_df, data_frame(

    # all features starting with "dist_" refers to linear movement
    dist_total = sum(dist_v),
    dist_max = max(dist_v),
    dist_min = min(dist_v),
    dist_max_to_min = max(dist_v)/min(dist_v),
    dist_mean = mean(dist_v),
    dist_sd = sd(dist_v),
    dist_mean_to_sd = mean(dist_v)/sd(dist_v),  # reciprocal coef of variation

    # all features starting with "omega_" refers to angle velocity measurments
    omega_total = sum(omega_v),
    omega_max = max(omega_v),
    omega_min = min(omega_v),
    omega_max_to_min = max(omega_v)/min(omega_v),
    omega_mean = mean(omega_v),
    omega_sd = sd(omega_v),
    omega_mean_to_sd = mean(omega_v)/sd(omega_v), # reciprocal coef of variation

    # phi, theta and psi reffers to roll, pitch and yaw rotations
    phi_total = sum(phi_v),
    phi_max = max(phi_v),
    phi_min = min(phi_v),
    phi_mean = mean(phi_v),
    phi_sd = sd(phi_v),
    phi_mean_to_sd = mean(phi_v)/sd(phi_v),

    theta_total = sum(theta_v),
    theta_max = max(theta_v),
    theta_min = min(theta_v),
    theta_mean = mean(theta_v),
    theta_sd = sd(theta_v),
    theta_mean_to_sd = mean(theta_v)/sd(theta_v),

    psi_total = sum(psi_v),
    psi_max = max(psi_v),
    psi_min = min(psi_v),
    psi_mean = mean(psi_v),
    psi_sd = sd(psi_v),
```

```
            psi_mean_to_sd = mean(psi_v)/sd(psi_v),

            # features starting with "euler_" refers to
            # agragation of all phi, theta and psi rotations
            euler_total = sum(phi_v + theta_v + psi_v),
            euler_max = max(phi_v + theta_v + psi_v),
            euler_min = min(phi_v + theta_v + psi_v),
            euler_mean = mean(phi_v + theta_v + psi_v),
            euler_sd = sd(phi_v + theta_v + psi_v),
            euler_mean_to_sd = mean(phi_v + theta_v + psi_v)/sd(phi_v + theta_v + psi_v)
            ))

    } # end of for over series

    # add the summary computations to the data set of series
    processed_data_df <- bind_cols(processed_data_df, tmp_df)

    # more features
    # I added thesee as an experimentation after observing the dependencies graphs
    # will explain later
    processed_data_df <- processed_data_df %>% mutate(
        # this is an agular momentum
        f1 = log(dist_mean_to_sd*omega_mean_to_sd),

        # this is an angular momentum
        f2 = log(dist_total*omega_total),

        # this is the angle between theta and psi vectors
        f3 = abs(atan(theta_mean_all/psi_mean_all)))

    # return the proceessed data
    processed_data_df
}
```

Pre-process the data and save it to a local file. This is to save time for further analysis by skipping the pre-processing procedure. I'm using *tic() toc()* functions from *tictoc* package to display processing time.

```
# pre-process train and test data sets
tic("process train data")
x_train_processed <- pre_process(x_train)
toc()
```

```
## process train data: 46.08 sec elapsed
```

```
tic("process test data")
x_test_processed <- pre_process(x_test)
toc()
```

```
## process test data: 44.36 sec elapsed
```

```
# rejoin train data with the labels data set
x_train_processed <- x_train_processed %>% left_join(y_train, by = "series_id")

# create a folder "data" if doesnt exist
if (!dir.exists("data")) dir.create("data")

# save processed data, and use these files from now on
write_csv(x_train_processed, "data/x_train_processed.csv")
write_csv(x_test_processed, "data/x_test_processed.csv")

# clean some variables and the environment
rm(x_train, x_test, y_train)
rm(x_train_processed, x_test_processed)
```

Load pre-processed data from hard-disk and use them from here. In this report I will use a smaller data set from the training data, to make things work faster. Full data set will be used in the final script

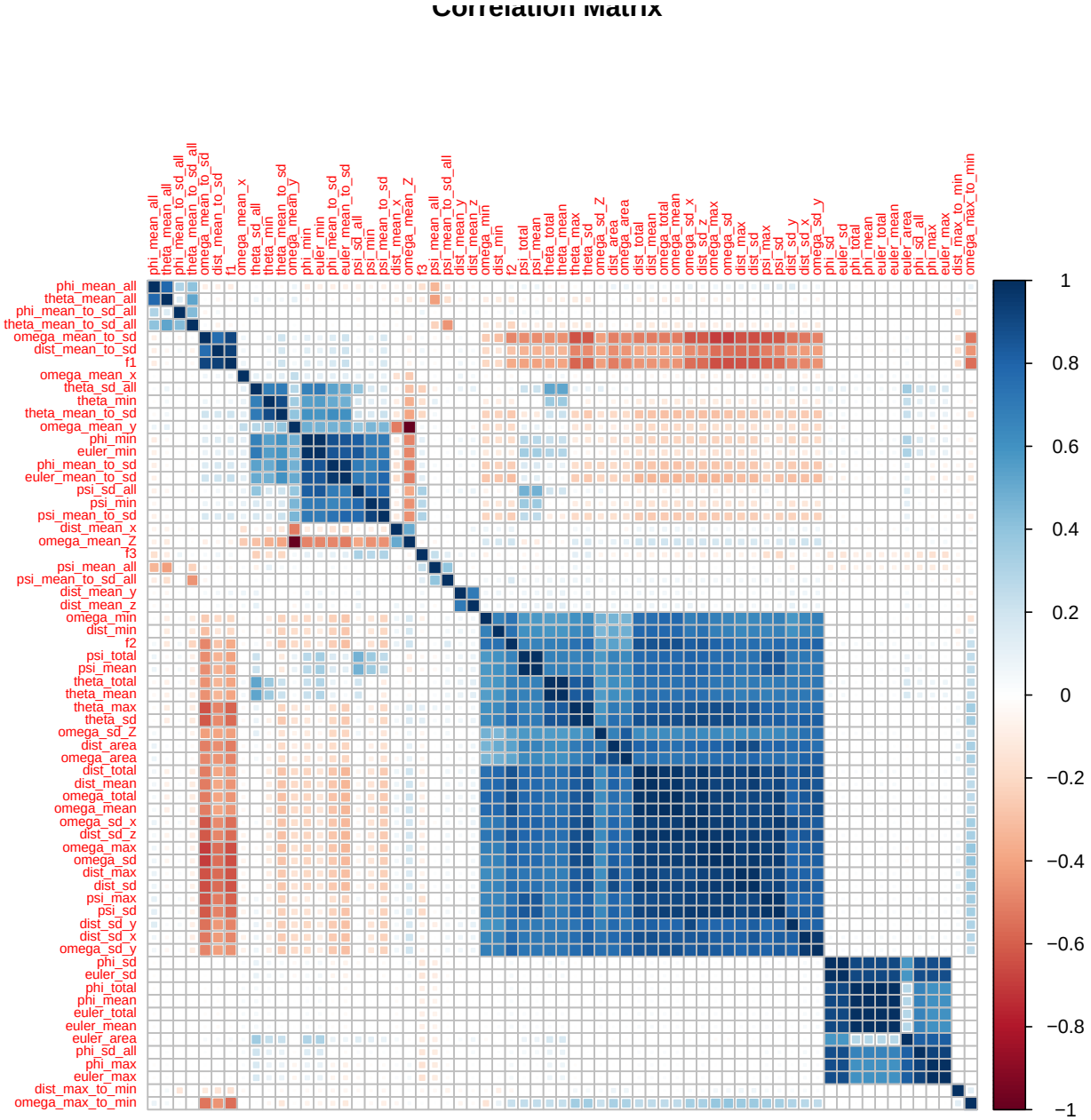
```
x_train_processed_from_file <- read_csv("data/x_train_processed.csv")
x_test_processed_from_file <- read_csv("data/x_test_processed.csv")

# if we load data from a file, convert surface to factor
x_train_processed_from_file <- x_train_processed_from_file %>% mutate(surface = as.factor(surface))

# use a smaller set of datab to save time
x_train_processed_from_file <- x_train_processed_from_file %>%  slice(1:1200)
```

Correlated Predictors

```
# from now we can load pre-processed data from csv files without the need to pre-process again
x_test_processed <- x_test_processed_from_file
x_train_processed <- x_train_processed_from_file
```

We created 65 features, lets trim some of them. First, I will identify those that are strongly correlated. Lets plot the correlation matrix:

```
library(RColorBrewer)
library(corrplot)
x_matrix <- x_train_processed %>% select(-series_id, -group_id, -surface) %>%
    as.matrix()
x_cor <- cor(x_matrix, use = "pairwise.complete")
corrplot(x_cor,method = "square", number.cex = .5, tl.cex = 0.6 ,order = "hclust",  title = "Correlatio
```

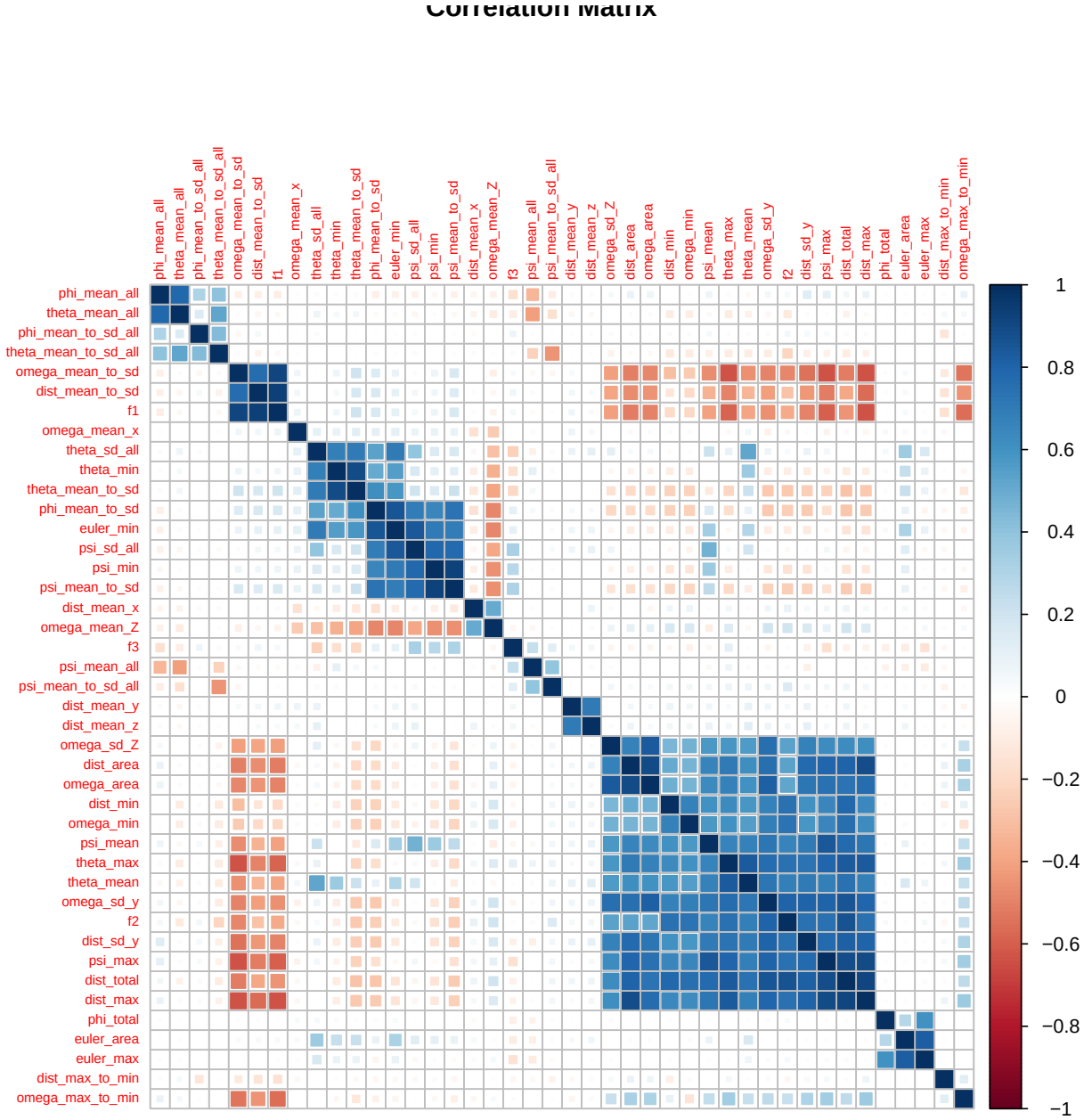



```
rm(x_matrix, x_cor)
```

Using function *findCorrelation* from *caret* we can remove features that are highly correlated

```
# convert both test and train data to matrix in order to analyse feature corelation
x_train_matrix <- x_train_processed %>% select(-surface, -series_id) %>% as.matrix()
x_test_matrix <- x_test_processed %>% select(-series_id) %>% as.matrix()

# find features that are high correlated
```

```r
# find linear dependencies and eliminate them
names_to_remove_test <- findCorrelation(cor(x_test_matrix), cutoff = 0.95, names = TRUE, verbose = FALSE)

# remove correlated features from both train and test sets
x_train_processed <- x_train_processed %>% select(-names_to_remove_test)
x_test_processed <- x_test_processed %>% select(-names_to_remove_test)

# draw again the correlation matrix
x_matrix <- x_train_processed %>% select(-series_id, -group_id, -surface) %>%
    as.matrix()
x_cor <- cor(x_matrix, use = "pairwise.complete")
corrplot(x_cor,method = "square", number.cex = .5, tl.cex = 0.6 ,order = "hclust",  title = "Correlation
```




```
rm(x_train_matrix, x_test_matrix, x_matrix, x_cor)
```

The number of predictors decreased to 42

Another useful technique in pre-processing is centering around means and scaling the predictors:

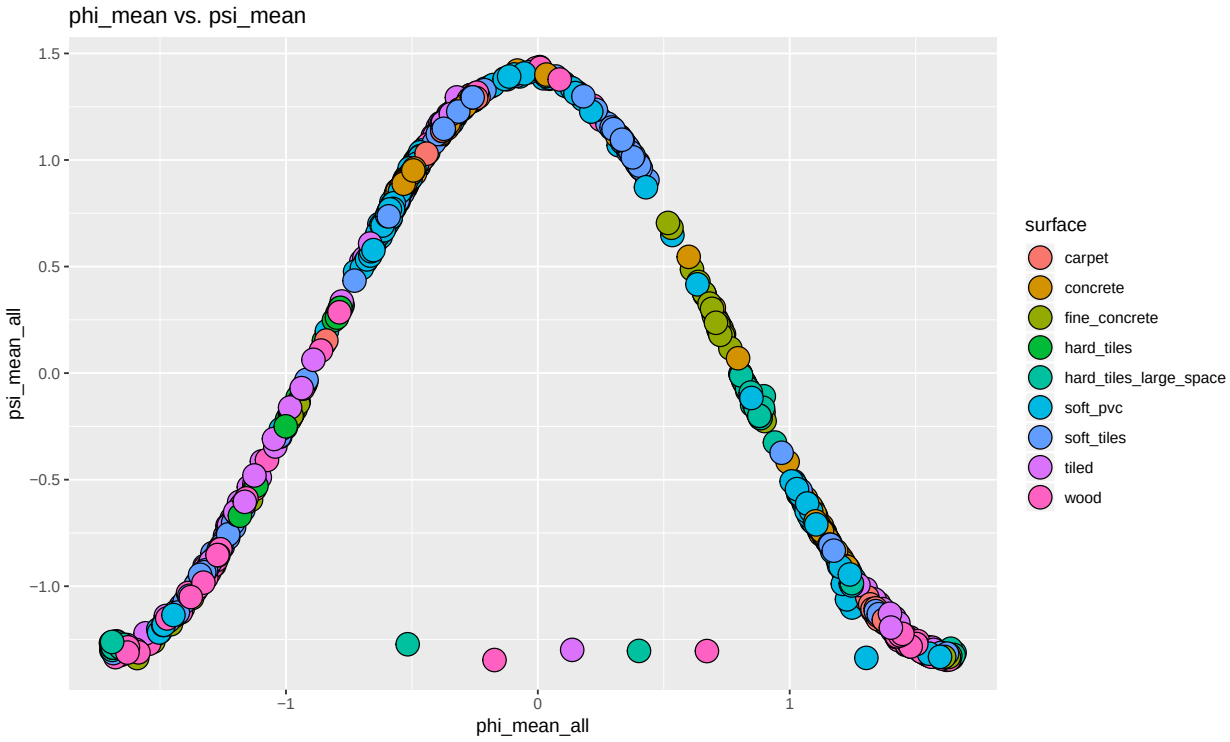
```
# pre-process the data, center and scale the values across all predictors
pre_process <- x_train_processed %>% select(-series_id, -group_id) %>% preProcess(method = c("center",
x_train_processed <- predict(pre_process, x_train_processed)
```

```
x_test_processed <- predict(pre_process, x_test_processed)

rm(pre_process)
```

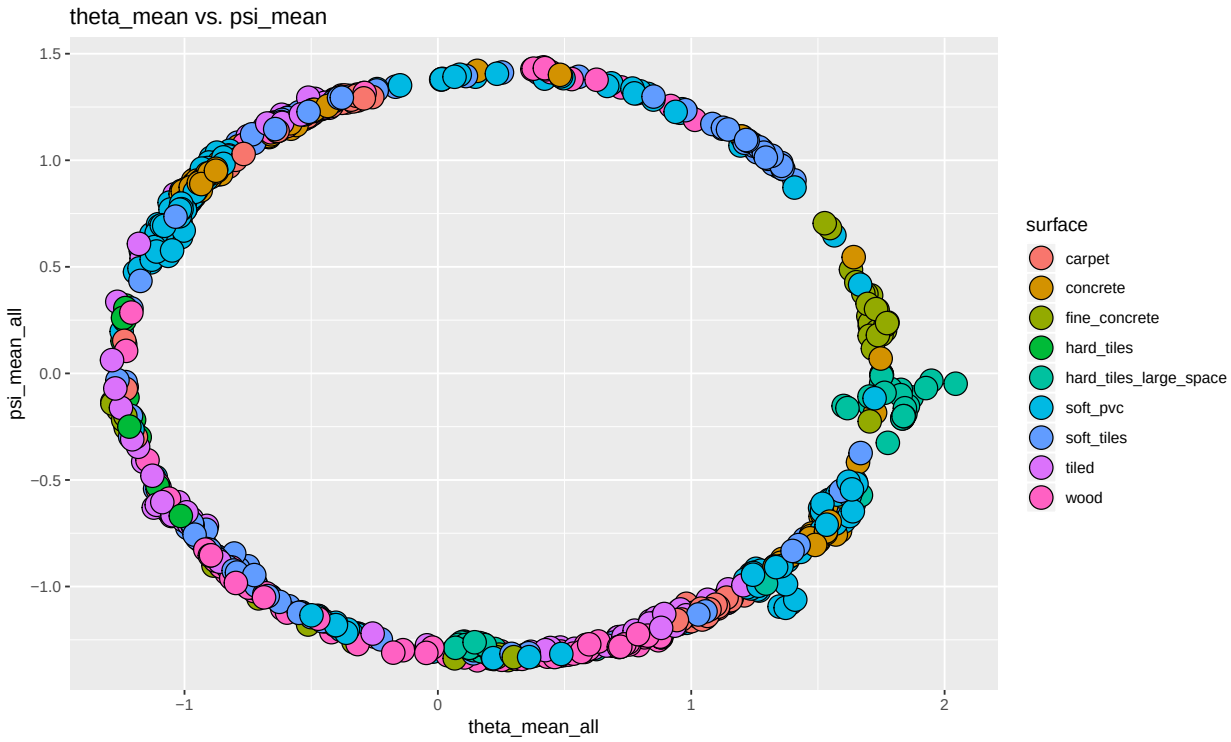
Data Visualization

Before moving forward with more feature selection, I will plot some interesting visualizations of the feature dependencies. These are plots of orientation angles as sums over all 128 points. *phi*, *theta* and *psi* are euler orientation angles.

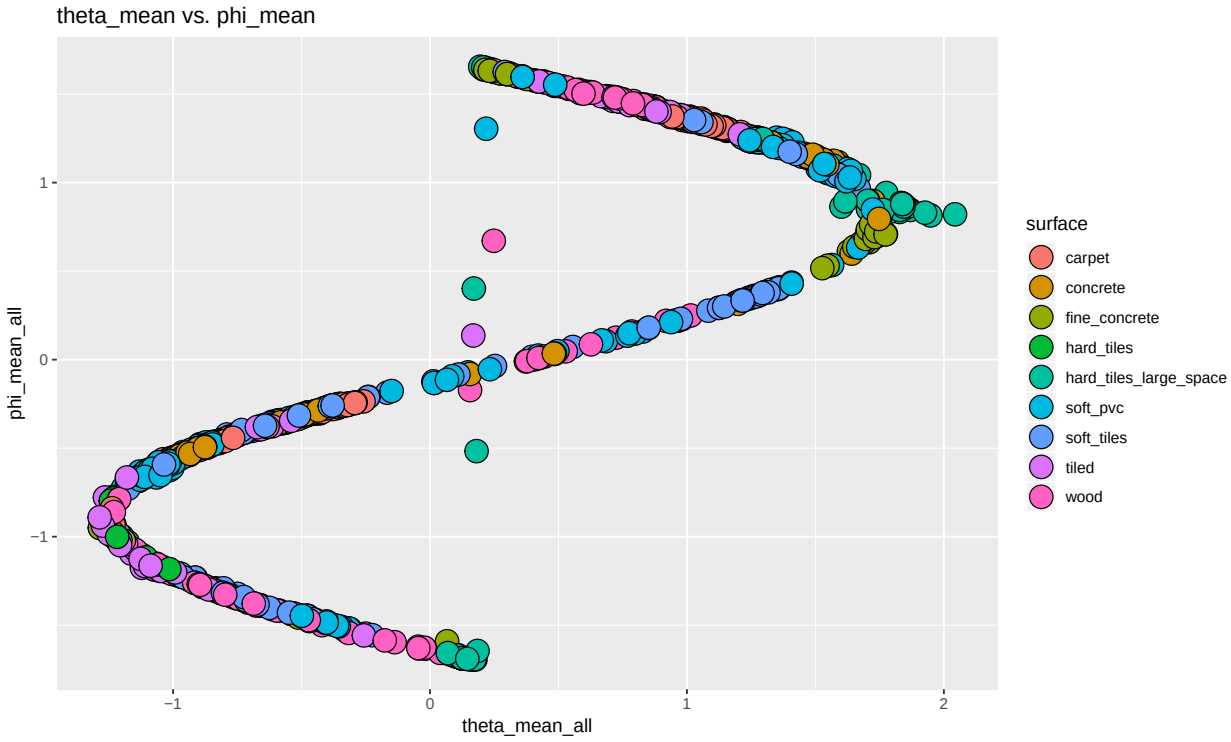
```
x_train_processed %>%  ggplot(aes(phi_mean_all, psi_mean_all, fill = surface)) +
    geom_point(aes(color = surface)) +
    geom_point(cex=6, pch=21) +
    ggtitle("phi_mean vs. psi_mean")
```




```
x_train_processed %>%  ggplot(aes(theta_mean_all, psi_mean_all, fill = surface)) +
    geom_point(aes(color = surface)) +
    geom_point(cex=6, pch=21) +
    ggtitle("theta_mean vs. psi_mean")
```




```
x_train_processed %>%  ggplot(aes(theta_mean_all, phi_mean_all, fill = surface)) +
    geom_point(aes(color = surface)) +
    geom_point(cex=6, pch=21) +
    ggtitle("theta_mean vs. phi_mean")
```

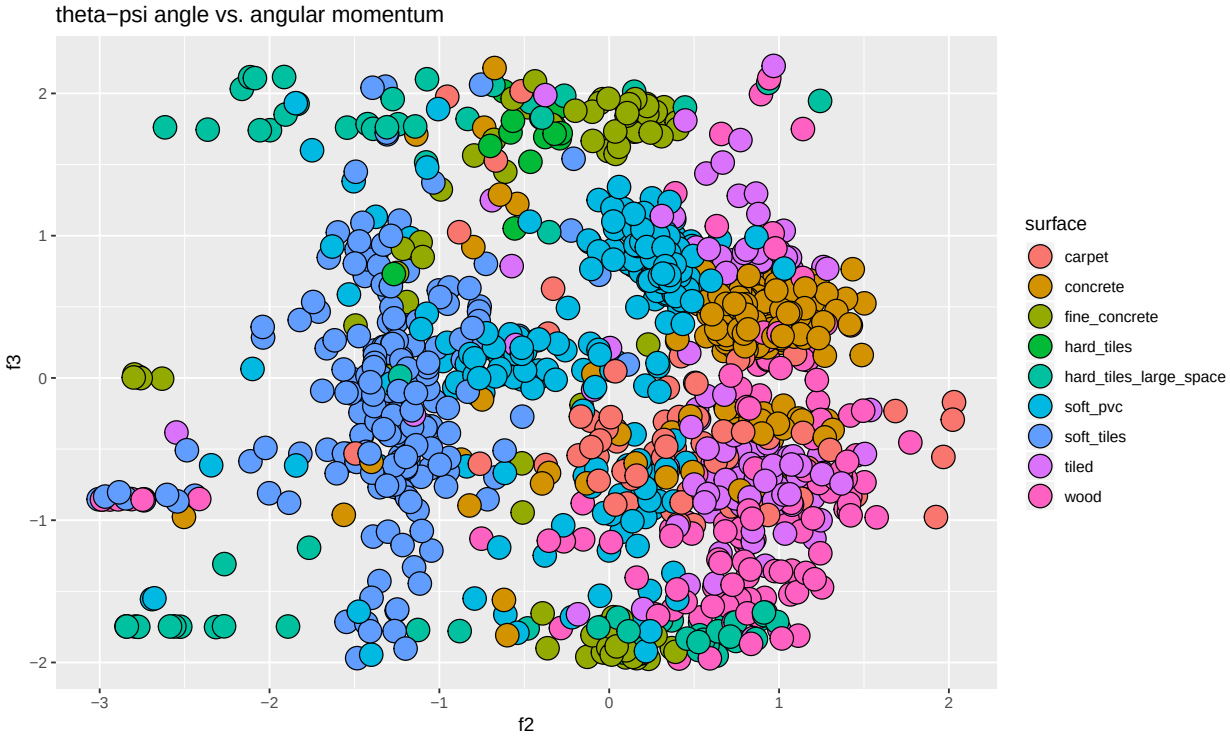


I would speculate that all points that are not on the global pattern are actually outlines, but I did not

explore that hypothesis.

The following is the angle between *theta* - *psi* vectors and the angular momentum:

```
x_train_processed %>%  ggplot(aes(f2, f3, fill = surface)) +
    geom_point(aes(color = surface)) +
    geom_point(cex=6, pch=21) +
    ggtitle("theta-psi angle vs. angular momentum")
```



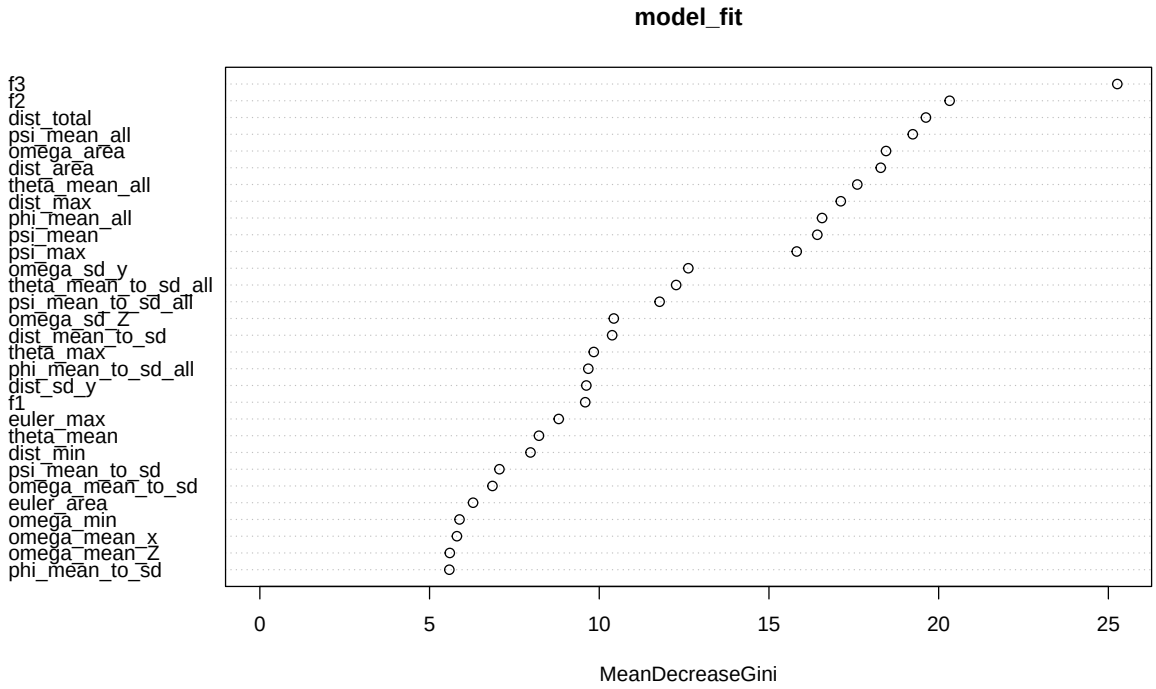
We can see some separations by surfaces that may allow a KNN model to work. In fact if we run KNN for just *surface* ~ *f2* + *f3* we will get an accuracy of 0.56 which is promising.

More Feature Selection

I will continue now removing more features based on their importance in a randomForest model. I will split the training set into two partitions: *x_train_for_train* used for training, and *x_train_for_test* to measure the accuracy. Then will plot variable importance. Will not use here any tuning, since this is not the final model.

```
# partition x_train_processed data for training and testing
test_index <- createDataPartition(y = x_train_processed$surface, times = 1, p = 0.5, list = FALSE)
x_train_for_train <- x_train_processed[-test_index, ]
x_train_for_test <- x_train_processed[test_index, ]
rm(test_index)

model_fit <- randomForest(
    surface ~ .,
    metric = "Accuracy",
    # remove series_id, group_id and select only 500 points
    data = slice(select(x_train_for_train, -series_id, -group_id), 1:500)
```

```
    )

y_hat <- predict(model_fit, select(x_train_for_test, -series_id))
y_test <- x_train_for_test$surface

# # we dont show accuracy and confusion table for now
# conf_matrix <- confusionMatrix(y_hat, x_train_for_test$surface)
# conf_matrix$overall["Accuracy"]
# conf_matrix$table %>% knitr::kable()

importance <- importance(model_fit)
importance <- importance[order(importance[,1], decreasing = TRUE), ]

# plot variable importance
varImpPlot(model_fit)
```




```
# this is the list of variables in order of importance, decreasing
features <- names(importance)
```

We observe following variables does not help prediction too much: *theta_min*, *omega_max_to_min*, *dist_mean_y*, *omega_mean_x*, *dist_mean_x*, *dist_mean_z* So I will remove them:

```
# remove columns do not contribute to classification
x_train_processed <- x_train_processed %>% select(-theta_min, -omega_max_to_min, -dist_mean_y, -omega_m
x_test_processed <- x_test_processed %>% select(-theta_min, -omega_max_to_min, -dist_mean_y, -omega_mea
```

The Model

It is time now to fit some models. I tried many models on the side but I will present in this report only

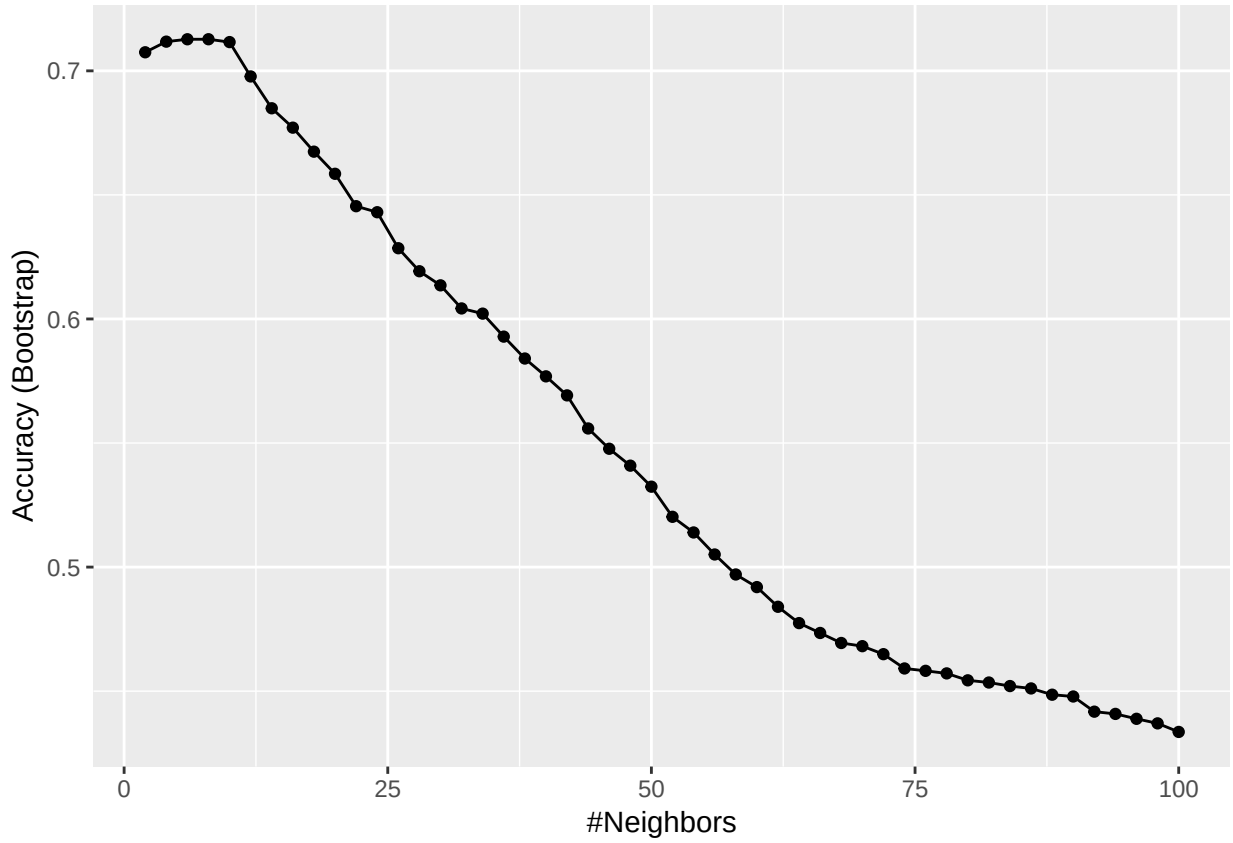
- KNN,
- rpart,
- randomForest,
- randomForest assembly

Please note that the code in this report is using only 1000 records from training, that is further partitioned in train and test with 0.5 ratio I've done this to make it run faster. I am using full data set in the R script and also I will present the final results from Kaggle (i think I said this before).

KNN Model

KNN is getting incrementally slower with the number of features. So for KNN I used only the most important variables as part of the formula.

```
# use most important variables to fit KNN
model_fit <- train(surface ~ f3 + f2 + phi_mean_all + theta_mean_all + psi_mean_all,  method = "knn",
             tuneGrid = data.frame(k = seq(2, 100, 2)),
             data = select(x_train_for_train, -series_id, -group_id))

ggplot(model_fit)
```



```
# get confusion matrix and display it together with the results
y_hat <- predict(model_fit, select(x_train_for_test, -series_id, -group_id), type = "raw")
conf_matrix <- confusionMatrix(y_hat, x_train_for_test$surface)

# display confusion matrix
conf_matrix$table %>% knitr::kable()
```

	carpet	concrete	fine_concrete	hard_tiles	hard_tiles_large_space	soft_pvc	soft_tiles
carpet	20	4	0	0	0	5	0
concrete	5	84	0	0	0	5	0
fine_concrete	0	0	33	0	5	1	1
hard_tiles	1	0	0	7	0	0	0
hard_tiles_large_space	0	2	0	0	23	2	0
soft_pvc	4	4	4	0	2	114	7
soft_tiles	1	1	4	0	6	8	91
tiled	7	5	0	1	0	0	2
wood	0	2	2	0	1	2	1

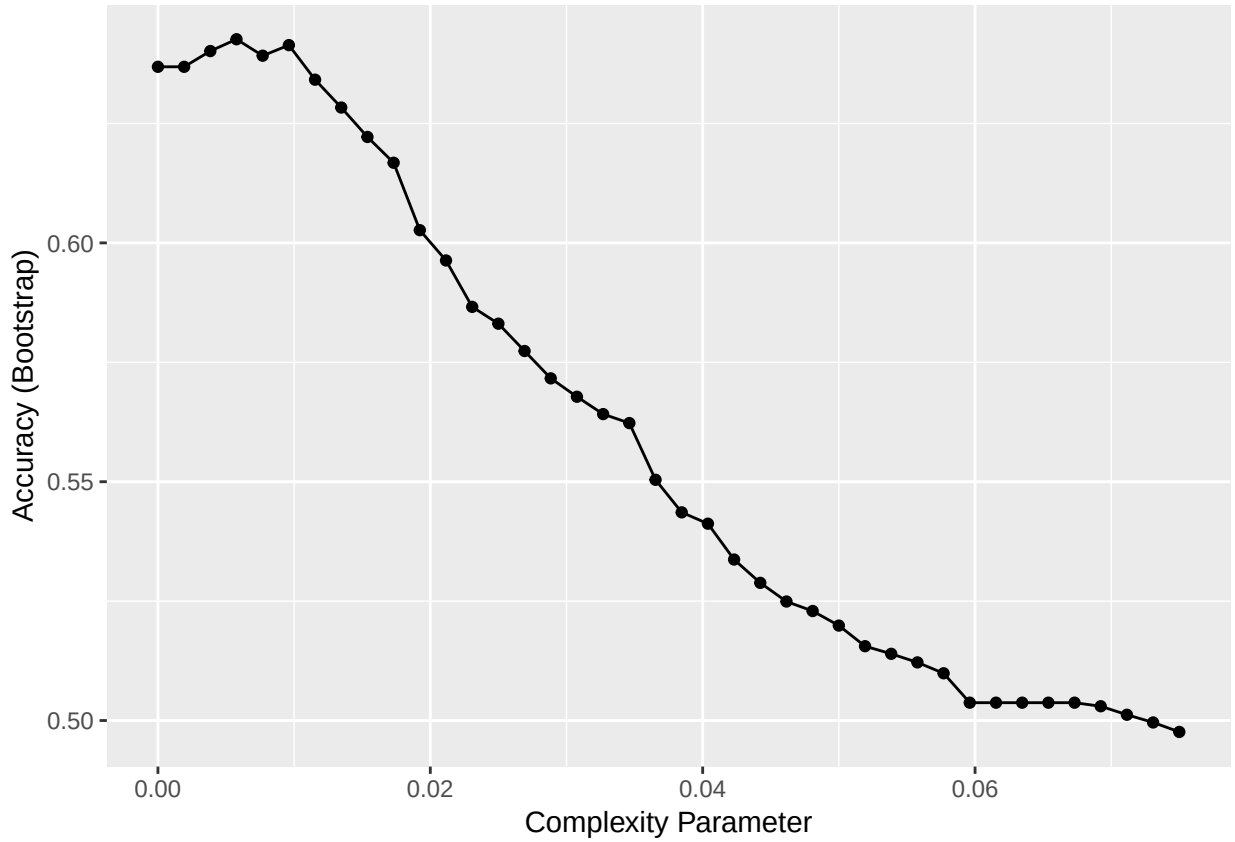
```
# create a data frame to store Accuracy results by model
model_results <- data.frame(Model = "KNN", Accuracy = conf_matrix$overall["Accuracy"])
model_results %>% knitr::kable()
```

	Model	Accuracy
Accuracy	KNN	0.7645108

rpart Model

Another model I tried is rpart (Recursive Partitioning And Regression Tree). I will tune it on the complexity coeficient

```
model_fit <- train(surface ~ .,
                   method = "rpart",
                   tuneGrid = data.frame(cp = seq(0, 0.075, len = 40)),
                   data = select(x_train_for_train, -series_id, -group_id))
ggplot(model_fit)
```




```
# get confusion matrix and display it together with the results
y_hat <- predict(model_fit, select(x_train_for_test, -series_id, -group_id), type = "raw")
conf_matrix <- confusionMatrix(y_hat, x_train_for_test$surface)

# display confusion matrix
conf_matrix$table %>% knitr::kable()
```

	carpet	concrete	fine_concrete	hard_tiles	hard_tiles_large_space	soft_pvc	soft_tiles
carpet	11	4	0	0	1	3	0
concrete	3	83	0	0	0	3	0
fine_concrete	0	0	28	1	0	4	2
hard_tiles	1	2	5	7	1	0	0
hard_tiles_large_space	0	2	0	0	23	3	0
soft_pvc	17	4	5	0	0	116	11
soft_tiles	0	0	5	0	10	8	85
tiled	2	0	0	0	0	0	4
wood	4	7	0	0	2	0	0

```
model_results <- bind_rows(model_results,
                           data_frame(
                             Model = "rpart",
                             Accuracy = conf_matrix$overall["Accuracy"]))
model_results %>% knitr::kable()
```

Model	Accuracy
KNN	0.7645108
rpart	0.7114428

RandomForest Model

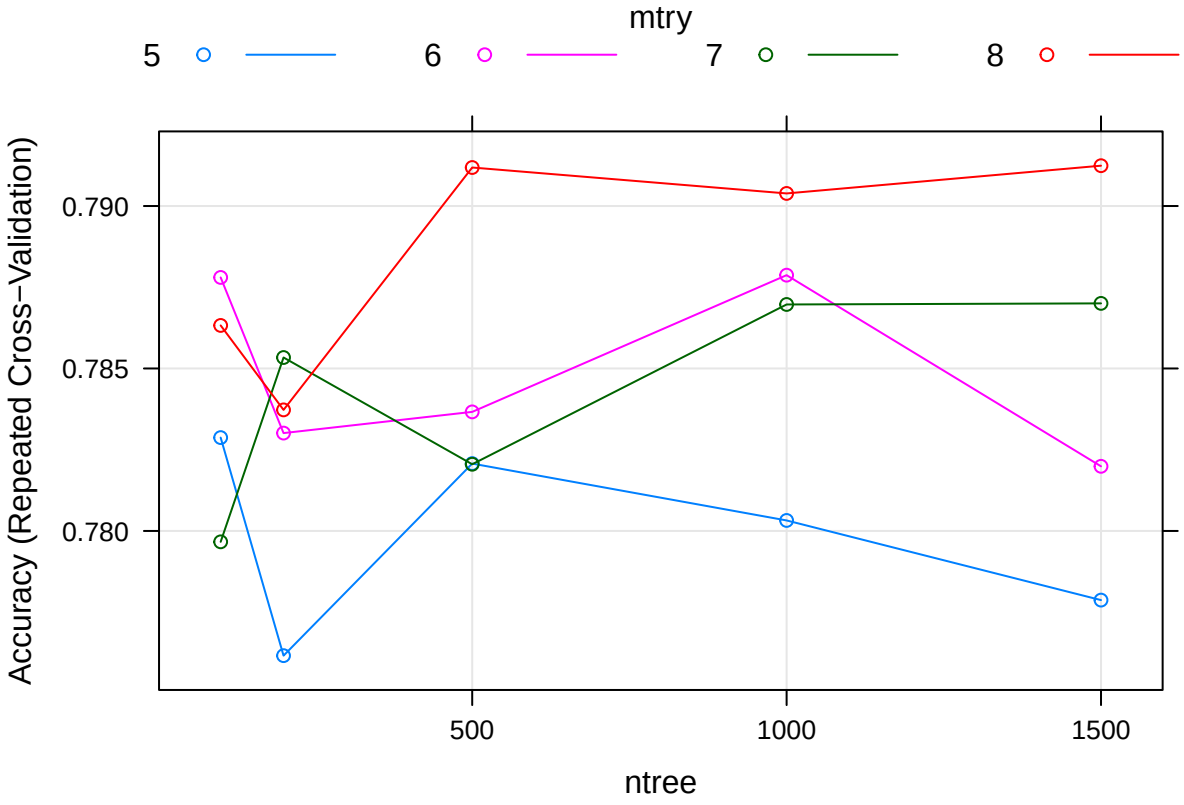
For randomForest model I choose to customize it and try optimization over *mtry* (~max number of ramifications) and *ntree* (max number of forests to grow) parameters. I got the way of customization from following article about tuning randomForests: https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/ This may take several minutes to run

```
# use cross validation 5 folds and repeated twice
control <- trainControl(method="repeatedcv", number=5, repeats=2, search="grid")

metric <- "Accuracy"

# mtry <- sqrt(ncol(x_train_for_train) - 1)

mtry <- 5:8
tunegrid <- expand.grid(.mtry=mtry,.ntree=c(100, 200, 500, 1000, 1500))

# tunegrid <- expand.grid(.mtry=c(1:10))

customRF                <-  list(type = "Classification", library = "randomForest", loop = NULL)
customRF$parameters <-  data.frame(parameter = c("mtry", "ntree"), class = rep("numeric", 2), label = c
customRF$grid           <-  function(x, y, len = NULL, search = "grid") {}
customRF$fit            <-  function(x, y, wts, param, lev, last, weights, classProbs, ...) randomFores
customRF$predict        <-  function(modelFit, newdata, preProc = NULL, submodels = NULL) predict(modell
customRF$prob           <-  function(modelFit, newdata, preProc = NULL, submodels = NULL)   predict(mod
customRF$sort           <-  function(x) x[order(x[,1]),]
customRF$levels         <-  function(x) x$surface


model_fit <- train(surface ~ .,
        data = select(x_train_for_train, -series_id, -group_id),
        method=customRF,
        metric=metric,
        tuneGrid=tunegrid,
        trControl=control
        )
print(model_fit)
```

```
## 597 samples
##  42 predictor
##   9 classes: 'carpet', 'concrete', 'fine_concrete', 'hard_tiles', 'hard_tiles_large_space', 'soft_pv
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 2 times)
## Summary of sample sizes: 480, 475, 479, 478, 476, 475, ...
## Resampling results across tuning parameters:
##
##   mtry  ntree  Accuracy   Kappa
```

```
##    5      100   0.7828726  0.7424711
##    5      200   0.7761617  0.7342551
##    5      500   0.7820729  0.7414457
##    5     1000   0.7803228  0.7391758
##    5     1500   0.7778715  0.7362735
##    6      100   0.7878016  0.7482006
##    6      200   0.7830117  0.7425325
##    6      500   0.7836643  0.7433908
##    6     1000   0.7878719  0.7482775
##    6     1500   0.7819892  0.7415225
##    7      100   0.7796646  0.7384379
##    7      200   0.7853371  0.7453407
##    7      500   0.7820524  0.7414582
##    7     1000   0.7869695  0.7475188
##    7     1500   0.7870038  0.7475757
##    8      100   0.7863243  0.7469293
##    8      200   0.7837252  0.7436429
##    8      500   0.7911847  0.7525742
##    8     1000   0.7903872  0.7517433
##    8     1500   0.7912409  0.7526166
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 8 and ntree = 1500.
```

```
plot(model_fit)
```



```
y_hat <- predict(model_fit, select(x_train_for_test, -series_id))
y_test <- x_train_for_test$surface

# get confusion matrix and display it together with the results
y_hat <- predict(model_fit, x_train_for_test, type = "raw")
conf_matrix <- confusionMatrix(y_hat, x_train_for_test$surface)

# display confusion matrix
conf_matrix$table %>% knitr::kable()
```

	carpet	concrete	fine_concrete	hard_tiles	hard_tiles_large_space	soft_pvc	soft_tiles
carpet	12	1	0	0	1	2	0
concrete	4	94	0	0	0	4	1
fine_concrete	0	2	35	0	2	1	0
hard_tiles	1	0	0	7	0	0	0
hard_tiles_large_space	0	0	0	0	27	2	0
soft_pvc	15	2	3	0	2	123	7
soft_tiles	0	0	5	1	2	4	92
tiled	3	1	0	0	1	1	1
wood	3	2	0	0	2	0	1

```
model_results <- bind_rows(model_results,
                           data_frame(
                             Model = "randomForest",
                             Accuracy = conf_matrix$overall["Accuracy"]))
model_results %>% knitr::kable()
```

Model	Accuracy
KNN	0.7645108
rpart	0.7114428
randomForest	0.8325041

Final model: randomForest one-vs-one

For the final model I choose a *one-vs-one* approach. Where I fit a model for each class. For this I loop over each of surfaces, rename all others to *the_rest* then I fit a randomForest model for each surface. I'm recording the results as probabilities in a table with surfaces as columns, and probability of the class as values. This would be a very unbalanced class. To balance it, I create a 3rd set of data, called *x_train_pool* and move some records of the current class from it into current training data set. I save each model in a folder called *models*. To predict, I'm looping over each surface in the test set and store the result as probabilities. The one with highest probability will determine the class for each point.

I use ROC as the metric to measure model performance. Accuracy wont be the best for this heavy unbalanced data set. This will take significant amount of time even if I tried to use minimum set of tuning options.

Here is the code:

```
# store the train data in a new variable
x_train_processed_ova <- x_train_processed
```

```r
# a prefix to save models on file system
model_prefix <- "model_14_fit_"

# create a subfolder called "models if it doesnt exists"
if (!dir.exists("models")) dir.create("models")

# partition data into:train, test, and balancing pool
# we will use the pool to extract records to balance the dataset
folds <- createFolds(x_train_processed_ova$surface, k = 3, list = TRUE)
x_train_for_train_ova <- x_train_processed_ova[folds$Fold1,]
x_train_for_test_ova <- x_train_processed_ova[folds$Fold2,]
x_train_pool <- x_train_processed_ova[folds$Fold3,]

# get surfaces in a data frame, so we can loop over
surfaces <- x_train_for_train_ova %>% group_by(surface) %>%
    summarize(n = n()) %>%
    mutate(surface = as.character(surface)) %>%
    # filter(surface == "hard_tiles") %>%
    arrange(n)

# idealy, I should use apply function but I'm still working on that
# this can bee also be improved if I would use foreacch packade with %dopar% option for parallelization
# still work in progress
for(current_surface in surfaces$surface)
{
        # convert surface to two values: current surface and "the_rest"
        x_train_for_train_ova_current <- x_train_for_train_ova %>%
            mutate(surface = ifelse(surface == current_surface, current_surface, "the_rest")) %>%
            mutate(surface = as.factor(surface))

        # add records from the pool to balance the recordset
        x_chunk_for_balance <- x_train_pool %>% filter(surface == current_surface)
        x_train_for_train_ova_current <- bind_rows(x_train_for_train_ova_current, x_chunk_for_balance)

        ##############################################################################################
        # custom randomForest
        #
        mtry <- sqrt(ncol(x_train_for_train_ova_current) - 1)
        tunegrid <- expand.grid(.mtry=mtry,.ntree=c( 300,500, 1000))
        control <- trainControl(method="repeatedcv",
                                                number=5,
                                                repeats=2,
                                                search="grid",
                                                classProbs = TRUE,
                                                # we could also use subsampling, but this will
                                                sampling = "up",
                                                summaryFunction = twoClassSummary
                                                )
        customRF                        <-  list(type = "Classification", library = "randomForest", loop
        customRF$parameters <-  data.frame(parameter = c("mtry", "ntree"), class = rep("numeric", 2), la
        customRF$grid           <-  function(x, y, len = NULL, search = "grid") {}
        customRF$fit                <-  function(x, y, wts, param, lev, last, weights, classProbs, ...)
        customRF$predict        <-  function(modelFit, newdata, preProc = NULL, submodels = NULL) predi
```

```
        customRF$prob           <-  function(modelFit, newdata, preProc = NULL, submodels = NULL)   prec
        customRF$sort           <-  function(x) x[order(x[,1]),]
        customRF$levels         <-  function(x) x$surface

        model_fit_current <- train(surface ~ .,
                                                        data = select(x_train_for_train_ova_current
                                                        method=customRF,
                                                        # use ROC for the metric because Accuracy
                                                        # in case of this heavy unballanced data s
                                                        metric="ROC",
                                                        tuneGrid=tunegrid,
                                                        trControl=control)
        ##########################################################################################

        # save the model into /models folder
        model_name <- paste(model_prefix, current_surface, sep = "")
        file <- paste("models/",  model_name, ".rds", sep = "")
        write_rds(model_fit_current, file)
}
```

Now, we'll load the models and perform model prediction and evaluation:

```
# create a data frame the will store probabilities for each model
# we'll use this for voting
# the model with highes prediction will get the vote
results_voting <- data.frame(
    series_id = x_train_for_test_ova$series_id,
    true_surface = x_train_for_test_ova$surface)

for(current_surface in surfaces$surface) {

    # prepare the test dataset: we keep current surface name, and we rename all other surfaces to "the_
    # we have now a binary clasification.
    x_train_for_test_ova_current <- x_train_for_test_ova %>%
            mutate(surface = ifelse(surface == current_surface, current_surface, "the_rest")) %>%
            mutate(surface = as.factor(surface))

    # get the model from a file
    model_name <- paste(model_prefix, current_surface, sep = "")
    model_fit_current <- readRDS(paste("models/", model_name, ".rds", sep = ""))

    # get y_hat_prob
    y_hat_prob <- predict(
                                        model_fit_current,
                                        select(x_train_for_test_ova_current, -series_id),
                                        type = "prob")

    # store the probability of curent model for current surface in a column named by current surface
    results_voting <- results_voting %>% mutate(last_result_prob = y_hat_prob[,current_surface])
    names(results_voting)[ncol(results_voting)] <- current_surface # the column name is current surface

}
```

```
# add an empty column for predicted surfaces
results_voting <- results_voting %>%  mutate(pred_surface = rep("", nrow(results_voting)))

# set the value on predicted surface to the surface that got maximum probability
for (i in 1:nrow(results_voting)) {
        results_voting[i, "pred_surface"] <- names(which.max(select(results_voting[i,], -series_id, -tru
}

results_voting <- results_voting %>% mutate(pred_surface = as.factor(pred_surface))

# show a sample of the voting table
nicetable <- results_voting %>% head(5) %>%  knitr::kable()
```

The voting table looks like this:

series_id	true_surface	hard_tiles	carpet	hard_tiles_large_space	fine_concrete	tiled	wood	concrete	so
1	concrete	0.000	0.3700000	0.004	0.000	0.166	0.166	0.638	
2	concrete	0.000	0.0133333	0.026	0.626	0.032	0.020	0.124	
6	soft_pvc	0.005	0.0933333	0.064	0.496	0.006	0.006	0.004	
7	concrete	0.000	0.5033333	0.000	0.000	0.008	0.128	0.928	
19	soft_tiles	0.379	0.0066667	0.000	0.026	0.008	0.000	0.038	

And here is the confusion matrix:

```
# compute confusion matrix and print it
conf_matrix <- confusionMatrix(results_voting$pred_surface,
                                                    results_voting$true_surface)

# display confusion matrix
conf_matrix$table %>% knitr::kable()
```

	carpet	concrete	fine_concrete	hard_tiles	hard_tiles_large_space	soft_pvc	soft_tiles
carpet	17	1	0	0	0	0	0
concrete	1	56	0	0	0	3	0
fine_concrete	1	0	24	0	1	0	0
hard_tiles	0	0	0	5	0	0	0
hard_tiles_large_space	0	0	0	0	20	0	0
soft_pvc	1	7	0	0	0	81	2
soft_tiles	0	1	5	0	1	6	66
tiled	2	3	0	0	1	1	0
wood	3	0	0	0	1	0	0

Results

Here are the final results:

```
model_results <- bind_rows(model_results,
            data_frame(
```

```
            Model = "randomForest one-vs-one",
            Accuracy = conf_matrix$overall["Accuracy"]))
# model_results %>% knitr::kable()

model_results %>% knitr::kable()
```

Model	Accuracy
KNN	0.7645108
rpart	0.7114428
randomForest	0.8325041
randomForest one-vs-one	0.8500000

Conclusion

In this project I used several models to try to determine the surface a robot is moving based on data from three sensors: inertial, magnetostatic and gyroscopic. This is part of an open competition on Kaggle: *CareerCon 2019 - Help Navigate Robots* I used KNN, rpart and two flavors of randomForest. I also tried on the side several other models with similar results.

I believe the key is in determining the right feature set that would predict accurately the surface. Although the accuracy obtained on the local data set is not bad, when submitting the predictions to Kaggle.com I get a relatively low result. This could mean that somehow the model is overfitted for the train data. Even if I didn't touch the test partition that I kept for evaluation. My guess is that should work on eliminating the outlines observed in Visualization section. They may alter the model in a bad way.

One of the most important outcome of this project is that I learned a few things in addition to what was presented in the course.

I hope this was interesting for you.

Thank you for reading!

Reference

1. Applied Predictive Modeling - Max Kuhn, Kjell Johnson
2. Q2EA: Convert from rotation Quaternions to Euler Angles. Q2EA converts from Quaternions (Q) to Euler Angles (EA) based on D. M. Henderson (1977). Q2EA.Xiao is the algorithm by J. Xiao (2013) for the Princeton Vision Toolkit. https://rdrr.io/cran/RSpincalc/man/Q2EA.html
3. Understanding Quaternions. http://www.chrobotics.com/library/understanding-quaternions
4. Understanding Euler Angles. http://www.chrobotics.com/library/understanding-euler-angles
5. Tune Machine Learning Algorithms in R (random forest case study) by Jason Brownlee. https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/
6. Classification with more than two classes - Introduction to Information Retrieval, Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze,, Cambridge University Press 2008 https://nlp.stanford.edu/IR-book/html/htmledition/classification-with-more-than-two-classes-1.html
7. A Guide To using IMU (Accelerometer and Gyroscope Devices) in Embedded Applications - http://www.starlino.com/imu_guide.html