HarvardX Data Science program capstone project

Human Activity and Postural Transisions (HAPT) recognition

Vladimir Pedchenko

11/24/2021

Contents

1 Introduction

Human activities recognition is the necessary basis for the development of many applications such as:

- Health monitoring;
- Sport trackers;
- Context notifications/reminders, based on activity;
- Personal exercise advisers;
- VR/AR applications.

In the past, to track human activities was not possible without expensive hardware which had to be mounted on the body and connected by wires to data processing computer. Fortunately, it is not a problem anymore: most of the people always have smartphone or/and smartwatch with them. These devises have set of sensors, such as accelerometer and gyroscope which able to provide an information about device movements and position. These signals can be preprocessed and be used as input for machine learning algorithms to recognize an activity of the person who has the device in hands or pocket.

Goal of the project is to build a model, which can recognize human activities based on preprocessed smartphone sensors data.

2 Dataset description

In order to build, train and test the model HAPT dataset was used. It can be found at: http://archive.ics.uci.edu/ml/datasets/Smartphone-Based+Recognition+of+Human+Activities+and+Postural+Transitions

This dataset was build by the experiments were carried out with a group of 30 volunteers within an age bracket of 19-48 years. They performed a protocol of activities composed of six basic activities: three static postures (standing, sitting, lying) and three dynamic activities (walking, walking downstairs and walking upstairs). The experiment also included postural transitions that occurred between the static postures. These are: stand-to-sit, sit-to-stand, sit-to-lie, lie-to-sit, stand-to-lie, and lie-to-stand. All the participants were wearing a smartphone (Samsung Galaxy S II) on the waist during the experiment execution. Experimenters captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz using the embedded accelerometer and gyroscope of the device. The experiments were video-recorded to label the data manually. The obtained dataset was randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of 561 features was obtained by calculating variables from the time and frequency domain.

The dataset archive includes the following files:

- 'README.txt'
- 'RawData/acc_expXX_userYY.txt': The raw triaxial acceleration signal for the experiment number XX and associated to the user number YY. Every row is one acceleration sample (three axis) captured at a frequency of 50Hz.
- 'RawData/gyro_expXX_userYY.txt': The raw triaxial angular speed signal for the experiment number XX and associated to the user number YY. Every row is one angular velocity sample (three axis) captured at a frequency of 50Hz.
- 'RawData/labels.txt': include all the activity labels available for the dataset (1 per row).
  - Column 1: experiment number ID,
  - Column 2: user number ID,
  - Column 3: activity number ID
  - Column 4: Label start point (in number of signal log samples (recorded at 50Hz))
  - Column 5: Label end point (in number of signal log samples)
- 'features_info.txt': Shows information about the variables used on the feature vector.

- 'features.txt': List of all features.
- 'activity_labels.txt': Links the activity ID with their activity name.
- 'Train/X_train.txt': Training set.
- 'Train/y_train.txt': Training labels.
- 'Test/X_test.txt': Test set.
- 'Test/y_test.txt': Test labels.
- 'Train/subject_id_train.txt': Each row identifies the subject who performed the activity for each window sample. Its range is from 1 to 30.
- 'Test/subject_id_test.txt': Each row identifies the subject who performed the activity for each window sample. Its range is from 1 to 30.

Because there are already preprocessed data in Training and Testing sets, raw data will not be used in this project.

Archive is downloaded and read by the following code:

```r
# If 'Data' folder is not exist, create it
if (!dir.exists("./data")) {
    dir.create("./data")
}

# Download file if it is not downloaded yet
if (!file.exists("./data//HAPT Data Set.zip")) {
    download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/00341/HAPT%20Data%20Set.zip
        "./data//HAPT Data Set.zip")
}

# Read data from all files in the archive and assign to variables

# First read features labels, to be able to name columns in the features
# dataframe
features <- read.csv(unzip("./data//HAPT Data Set.zip", "features.txt"), header = FALSE)
features <- as.vector(features[, 1])

# Replace '-' to '_' from the features names, because it replaces by '.' when
# apply to column names
features <- str_replace_all(features, "-", "_")
# And remove spaces
features <- str_replace_all(features, " ", "")

# Read activity labels to replace activity number by activity label in the
# outcome vector
activity_labels <- read.csv(unzip("./data//HAPT Data Set.zip", "activity_labels.txt"),
    sep = " ", header = FALSE)
colnames(activity_labels) <- c("Activity_number", "Activity")
activity_labels <- activity_labels %>%
    select("Activity_number", "Activity")

# Unzip and read training data
x_train <- read.csv(unzip("./data//HAPT Data Set.zip", "Train/X_train.txt"), sep = " ",
    header = FALSE, col.names = features)
y_train <- read.csv(unzip("./data//HAPT Data Set.zip", "Train/y_train.txt"), sep = " ",
    header = FALSE, col.names = "Activity_number")

# Merge outcome vector and activity labels to have labels instead of numbers
y_train <- y_train %>%
    left_join(activity_labels, by = "Activity_number") %>%
    select(-"Activity_number")


# Unzip and read testing data (feature names vector is the same that for
# training data)
x_test <- read.csv(unzip("./data//HAPT Data Set.zip", "Test/X_test.txt"), sep = " ",
    header = FALSE, col.names = features)
y_test <- read.csv(unzip("./data//HAPT Data Set.zip", "Test/y_test.txt"), sep = " ",
    header = FALSE, col.names = "Activity_number")

# Merge outcome vector and activity labels to have labels instead of numbers
y_test <- y_test %>%
    left_join(activity_labels, by = "Activity_number") %>%
    select(-"Activity_number")

# Delete unzipped files and folders
unlink(c("Test", "Train", "activity_labels.txt", "features.txt"), recursive = T)

# Change Activity column type to factor
y_train <- y_train %>%
    mutate(Activity = factor(Activity))
y_test <- y_test %>%
    mutate(Activity = factor(Activity))


# df_validation = x_test + y_test will be considered as unknown data and will
# not be used till the end of the project as final validation.  Data analysis,
# model training and selection will be performed entirely on the train dataset
# (df = x_train + y_train)

# combine x_train and y_train to one dataframe for EDA

df <- cbind(x_train, y_train)
df_validation <- cbind(x_test, y_test)

rm(x_train, y_train, x_test, y_test)
```

It reads Training and Testing sets from files, apply feature names from features.txt to columns names. Then, activities encoded as numbers are converted to meaningful labels. Finally, train features and outcome data are combined to *df* dataset and test features and outcome data are combined to *df_validation* dataset.

df is the dataset which will be used for analysis and machine learning. *df_validation* is the validation hold-out dataset, which will not be used for any purposes until final model validation at the very end of the project.

Additional notes from Readme.txt:

- Features are normalized and bounded within [-1,1].

- Each feature vector is a row on the 'X' and 'y' files.
- The units used for the accelerations (total and body) are 'g's (gravity of earth -> 9.80665 m/seg2).
- The gyroscope units are rad/seg.
- A video of the experiment including an example of the 6 recorded activities with one of the participants can be seen in the following link: http://www.youtube.com/watch?v=XOEN9W05_4A

2.1 Features description

Dataset archive contains features_info.txt file, which provides important information about features, how raw data from sensors was preprocessed. In case of using different raw data, they can be preprocessed the same way.

The features selected for this database come from the accelerometer and gyroscope 3-axial raw signals *tAcc-XYZ* and *tGyro-XYZ*. These time domain signals (prefix '*t*' to denote time) were captured at a constant rate of 50 Hz. Then they were filtered using a median filter and a 3rd order low pass Butterworth filter with a corner frequency of 20 Hz to remove noise. Similarly, the acceleration signal was then separated into body and gravity acceleration signals (*tBodyAcc-XYZ* and *tGravityAcc-XYZ*) using another low pass Butterworth filter with a corner frequency of 0.3 Hz.

Subsequently, the body linear acceleration and angular velocity were derived in time to obtain Jerk signals (*tBodyAccJerk-XYZ* and *tBodyGyroJerk-XYZ*). Also the magnitude of these three-dimensional signals were calculated using the Euclidean norm (*tBodyAccMag*, *tGravityAccMag*, *tBodyAccJerkMag*, *tBodyGyroMag*, *tBodyGyroJerkMag*).

Finally a Fast Fourier Transform (FFT) was applied to some of these signals producing *fBodyAcc-XYZ*, *fBodyAccJerk-XYZ*, *fBodyGyro-XYZ*, *fBodyAccJerkMag*, *fBodyGyroMag*, *fBodyGyroJerkMag*. (Note the '*f*' to indicate frequency domain signals).

These signals were used to estimate variables of the feature vector for each pattern:

'*-XYZ*' is used to denote 3-axial signals in the X, Y and Z directions.

- *tBodyAcc-XYZ*
- *tGravityAcc-XYZ*
- *tBodyAccJerk-XYZ*
- *tBodyGyro-XYZ*
- *tBodyGyroJerk-XYZ*
- *tBodyAccMag*
- *tGravityAccMag*
- *tBodyAccJerkMag*
- *tBodyGyroMag*
- *tBodyGyroJerkMag*
- *fBodyAcc-XYZ*
- *fBodyAccJerk-XYZ*
- *fBodyGyro-XYZ*

- *fBodyAccMag*
- *fBodyAccJerkMag*
- *fBodyGyroMag*
- *fBodyGyroJerkMag*

The set of variables that were estimated from these signals are:

- mean(): Mean value
- std(): Standard deviation
- mad(): Median absolute deviation
- max(): Largest value in array
- min(): Smallest value in array
- sma(): Signal magnitude area
- energy(): Energy measure. Sum of the squares divided by the number of values.
- iqr(): Interquartile range
- entropy(): Signal entropy
- arCoeff(): Autorregresion coefficients with Burg order equal to 4
- correlation(): correlation coefficient between two signals
- maxInds(): index of the frequency component with largest magnitude
- meanFreq(): Weighted average of the frequency components to obtain a mean frequency
- skewness(): skewness of the frequency domain signal
- kurtosis(): kurtosis of the frequency domain signal
- bandsEnergy(): Energy of a frequency interval within the 64 bins of the FFT of each window.
- angle(): Angle between to vectors.

Additional vectors obtained by averaging the signals in a signal window sample. These are used on the angle() variable:

- *gravityMean*
- *tBodyAccMean*
- *tBodyAccJerkMean*
- *tBodyGyroMean*
- *tBodyGyroJerkMean*

3 Exploratory Data Analysis

Goal of Exploratory Data Analysis is to get information of the dataset, distributions of features and outcomes, their types. Possibility for feature reduction will be considered here as well. Interpretability of the final model is not a priority for this task, because features are mathematically transformed sensors signals, which will be interpreted only by computer/smartphone.

First, let's get some general information about dataset.

```r
# dataset dimensions
dim(df)
```

```
## [1] 7767  562
```

```r
# NAs in dataset
df %>%
    is.na() %>%
    sum()
```

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

The dataset contains 7767 rows and 562 columns. There are 0 missing values in it. List of features names is too long to print it out completely, therefore we will look on head and tail of the dataset with just first three features and outcome column:

Table 1: First 10 rows of the dataset with three features and outcome

	tBodyAcc_Mean_1	tBodyAcc_Mean_2	tBodyAcc_Mean_3	Activity
1	0.0435797	-0.0059702	-0.0350543	STANDING
2	0.0394800	-0.0021313	-0.0290674	STANDING
3	0.0399778	-0.0051527	-0.0226507	STANDING
4	0.0397846	-0.0118088	-0.0289158	STANDING
5	0.0387581	-0.0022885	-0.0238629	STANDING
6	0.0389880	0.0041089	-0.0173403	STANDING
7	0.0398975	-0.0053243	-0.0204565	STANDING
8	0.0390823	-0.0160471	-0.0302413	STANDING
9	0.0390262	-0.0074100	-0.0273007	STANDING
10	0.0403539	0.0042448	-0.0179322	STANDING

Table 2: Last 10 rows of the dataset with three features and outcome

	tBodyAcc_Mean_1	tBodyAcc_Mean_2	tBodyAcc_Mean_3	Activity
7758	0.0385597	-0.0927131	0.0135742	WALKING_UPSTAIRS
7759	0.0458565	-0.0254181	-0.0417472	WALKING_UPSTAIRS
7760	0.0164071	-0.0218810	-0.0572198	WALKING_UPSTAIRS
7761	0.0110252	0.0767841	-0.0909770	WALKING_UPSTAIRS
7762	0.0231663	0.0130153	-0.0448918	WALKING_UPSTAIRS
7763	0.0480484	-0.0424452	-0.0658843	WALKING_UPSTAIRS
7764	0.0376386	0.0064304	-0.0443447	WALKING_UPSTAIRS

	tBodyAcc_Mean_1	tBodyAcc_Mean_2	tBodyAcc_Mean_3	Activity
7765	0.0374509	-0.0027244	0.0210094	WALKING_UPSTAIRS
7766	0.0440110	-0.0045358	-0.0512422	WALKING_UPSTAIRS
7767	0.0689538	0.0018103	-0.0803234	WALKING_UPSTAIRS

Types of all features should be numeric. To check it we will look on all classes are in the dataframe and then count how many of each class:

```
unique(lapply(df, class))
```

```
## [[1]]
## [1] "numeric"
##
## [[2]]
## [1] "factor"
```

```
sum(lapply(df, class) == "numeric")
```

```
## [1] 561
```

```
sum(lapply(df, class) == "factor")
```

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

```
class(df$Activity)
```

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

Only one variable has class "factor", this is the outcome variable "Activity". All other variables are numeric.

Distribution of outcomes in the dataset:

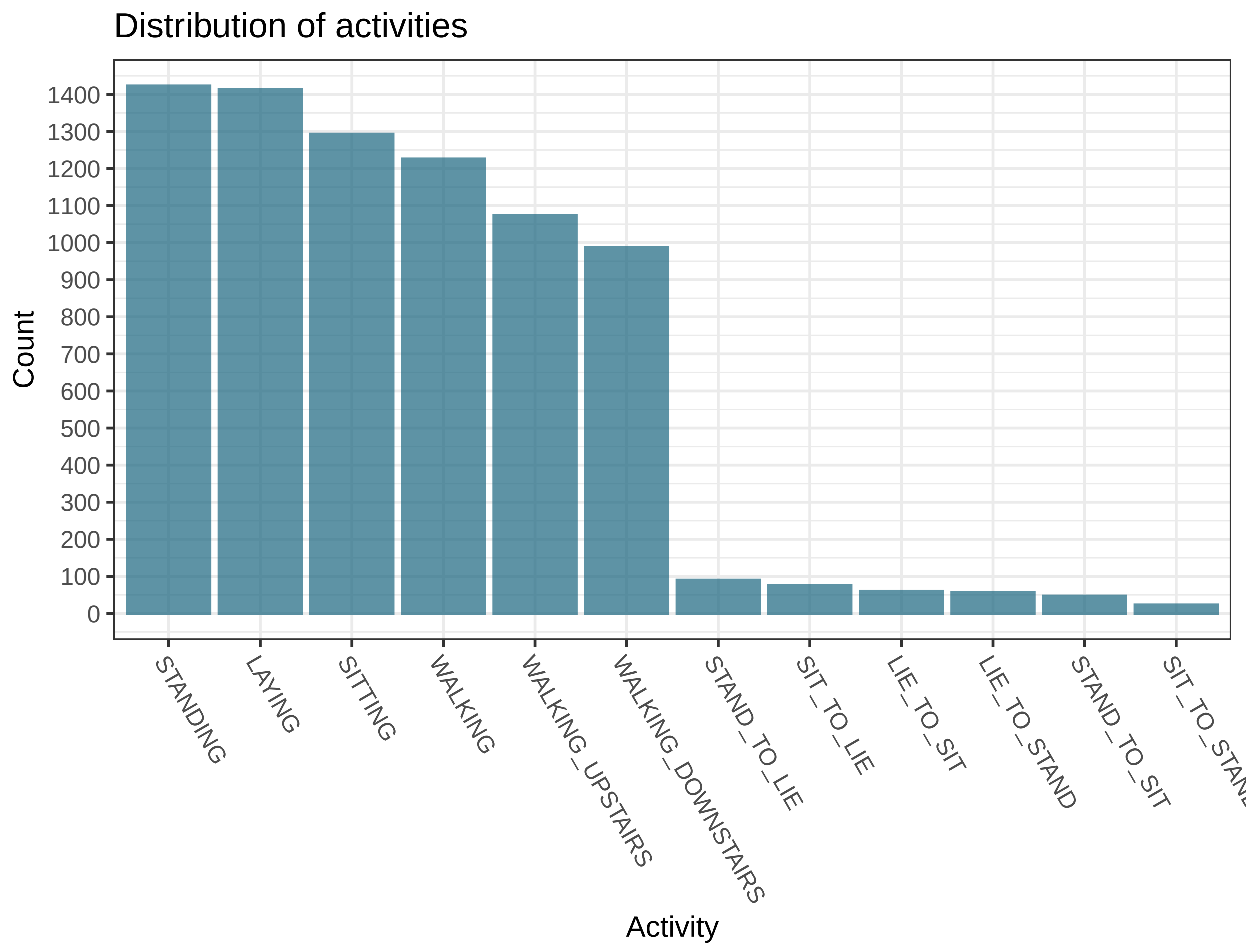


Figure 1: Distribution of activities

As we can see, outcomes are not equally represented in the dataset. The biggest amount of measures are classified as STANDING and there are 1423 of them. From other side, the least appearing activity SIT__TO__STAND have only 23 representations in the dataset.

The dataset is highly unbalanced, which must be taken in account during model building and evaluation. All postural transitions are minority classes, compare to other activities. It is expected that real distribution is the same: static activities takes more time compare to posture changes, which are short, momentary events.

3.1 Features analysis

The dataset contains 561 features. They are are normalized and bounded within [-1,1]. Visualization of features statistics:

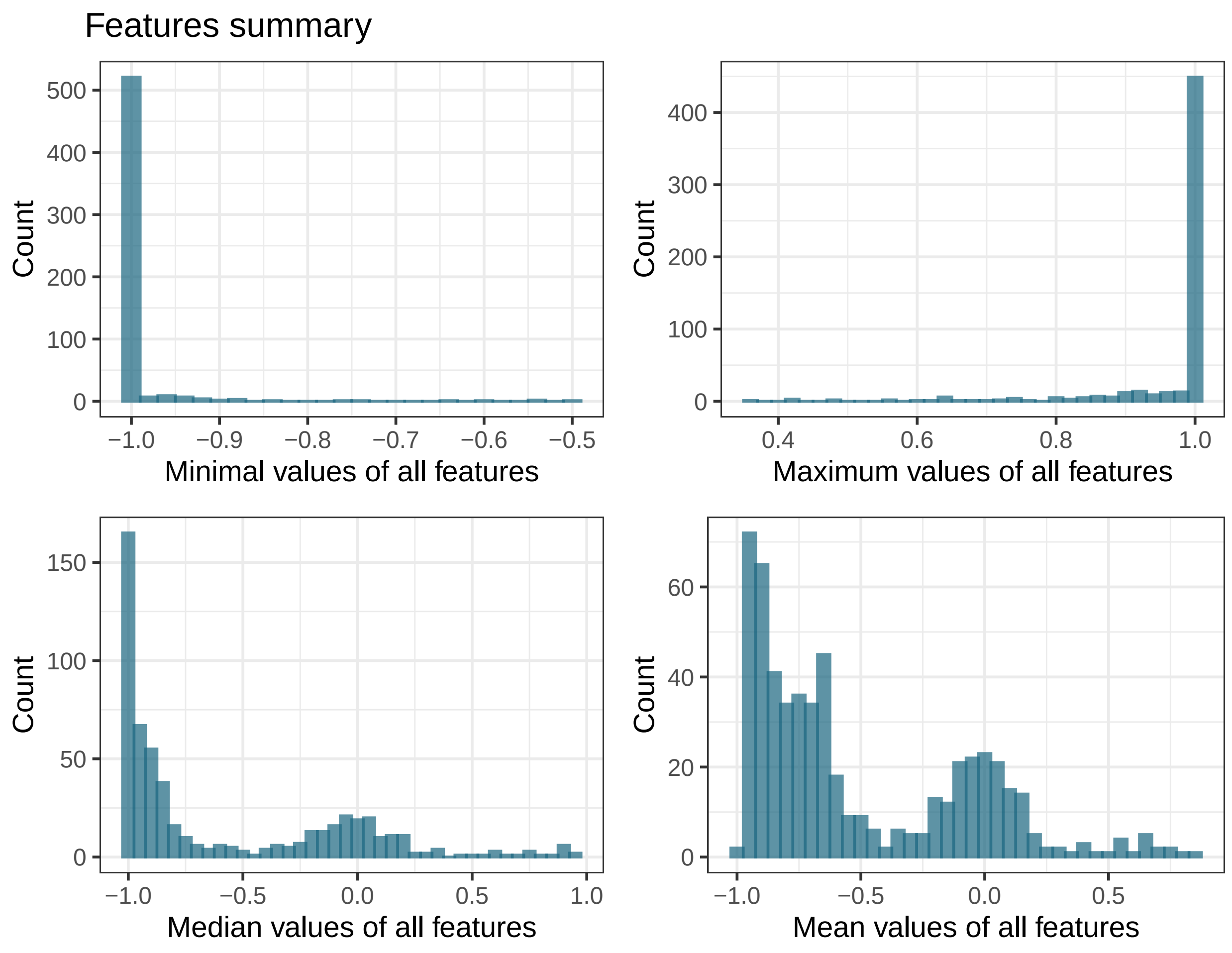


Figure 2: Features summary

Figure 2 confirms that vast majority of the features have minimal value as -1 and maximum value as 1. Also, we can see, that many features have very low median value (-1 - -0.9), which means that they have low variances. But looking to mean values trend, we see, that peak is shifted towards center a bit, which means that they have some variance. Remember Figure 1, which tells us, that some classes are vast minority, we can't call these features "not important" and drop them. It can be, that these features are important for minority classes, such as posture changes, detection. Let us look on five features with minimum mean value:

Table 3: Five features with lowest mean value

fBodyAccJerk__BandsEnergyOld__8
fBodyGyro__BandsEnergyOld__19
fBodyGyro__BandsEnergyOld__22
fBodyGyro__BandsEnergyOld__8
fBodyGyro__BandsEnergyOld__25

These features are energy measure of frequency domain signals (see chapter 2.1). Visualization of their distributions for different outcome classes are shown on figure 3:

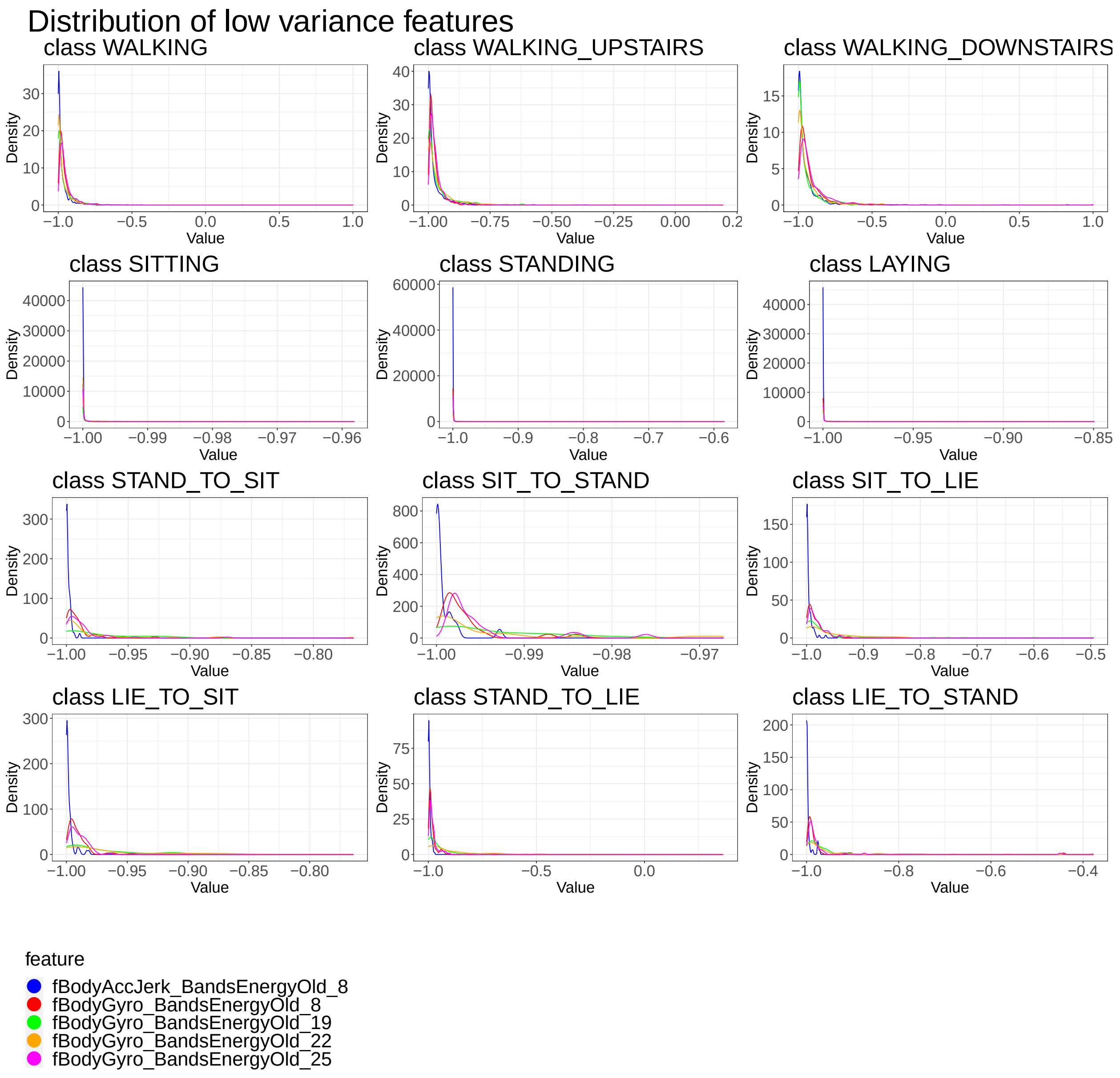


Figure 3: Distribution of low variance features

It is clearly seen that some of these features define posture transitions and specifically the minor class SIT_TO_STAND.

Unfortunately, it is not possible to effectively visualize all 561 features, therefore Principal Component Analysis will be performed.

3.2 Principal Component Analisys

Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest. PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

In order to equalize importance of rare classes with common classes, Synthetic Minority Oversampling Technique (SMOTE) will be performed before applying PCA. SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line. Following code is used to balance the dataset with SMOTE:

```r
# Apply SMOTE before PCA
df_smote <- SmoteClassif(Activity ~ ., dat = df)
```

Let's plot outcome distribution after SMOTE is applied:

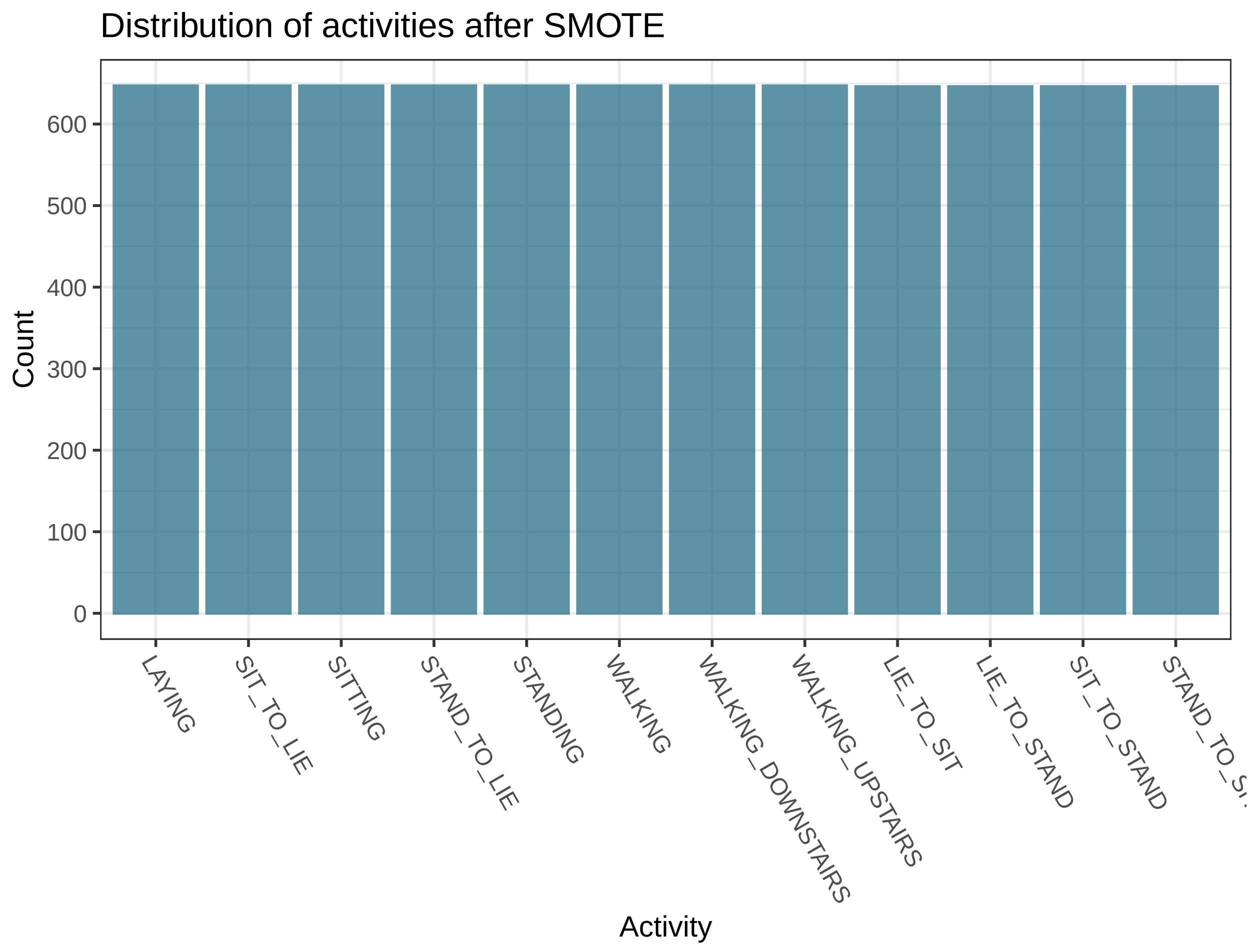


Figure 4: Distribution of activities after SMOTE

Now we can calculate principal components using function *prcomp* with argument *scale = TRUE*, which scales the variables to have unit variance before the analysis takes place:

```r
pca <- prcomp(df_smote[-ncol(df_smote)], scale. = TRUE)
```

Visualizing the variance explained by each component helps us identifying visually, how many principal components are needed to explain the data variation:

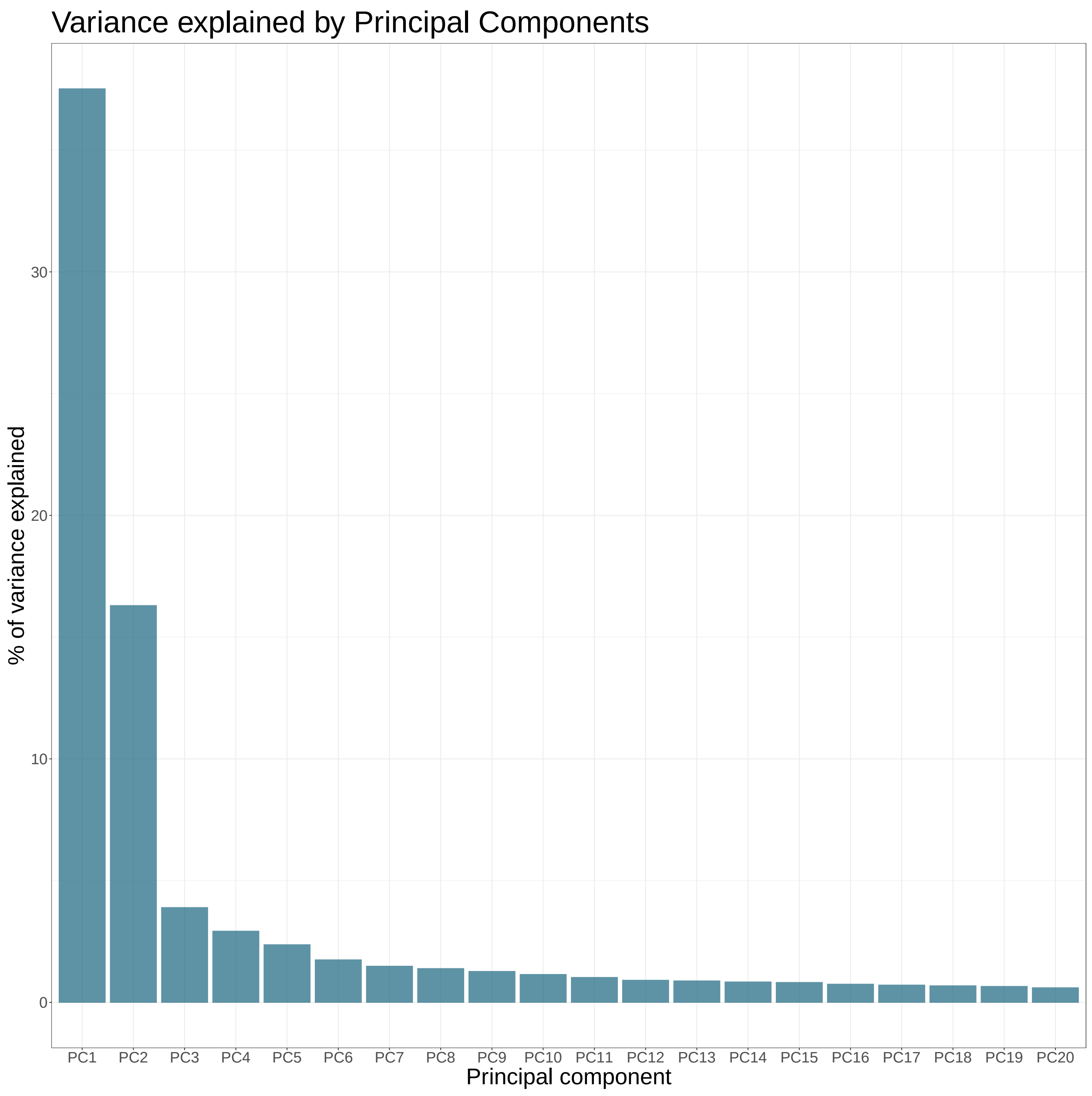


Figure 5: Variance explained by each of principal components

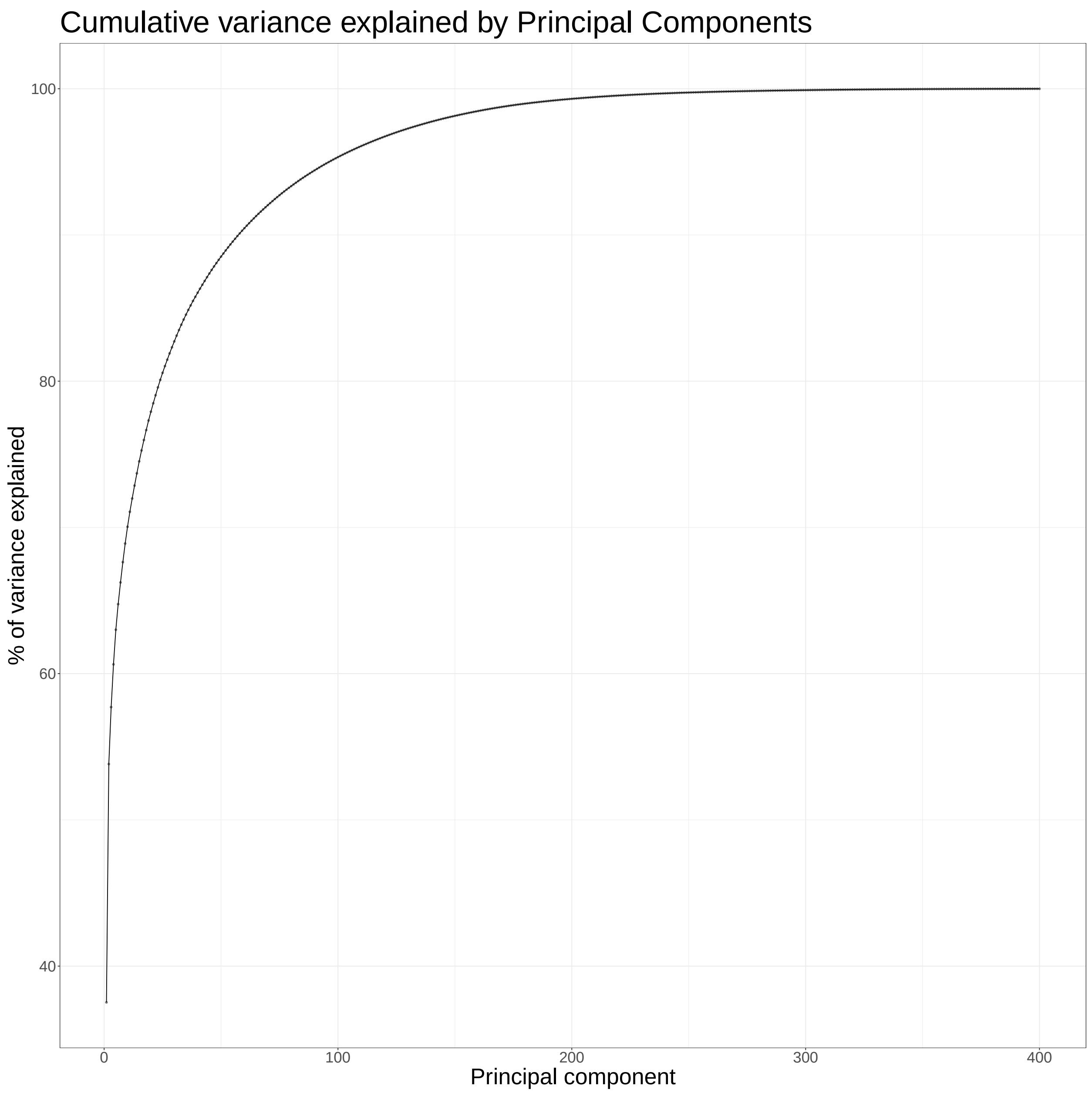


Figure 6: Cumulative variance explained by each of principal components

As we can see on Figure 5, about half of variances can be explained by two principal components. But looking on the total variance explained by principal components (Figure 6), we can see, that to explain at least 95% of variance, we need about 100 principal components. Conclusion is that our data space is indeed very multidimensional.

Let's check which classes are easily distinguished by few first principal components:

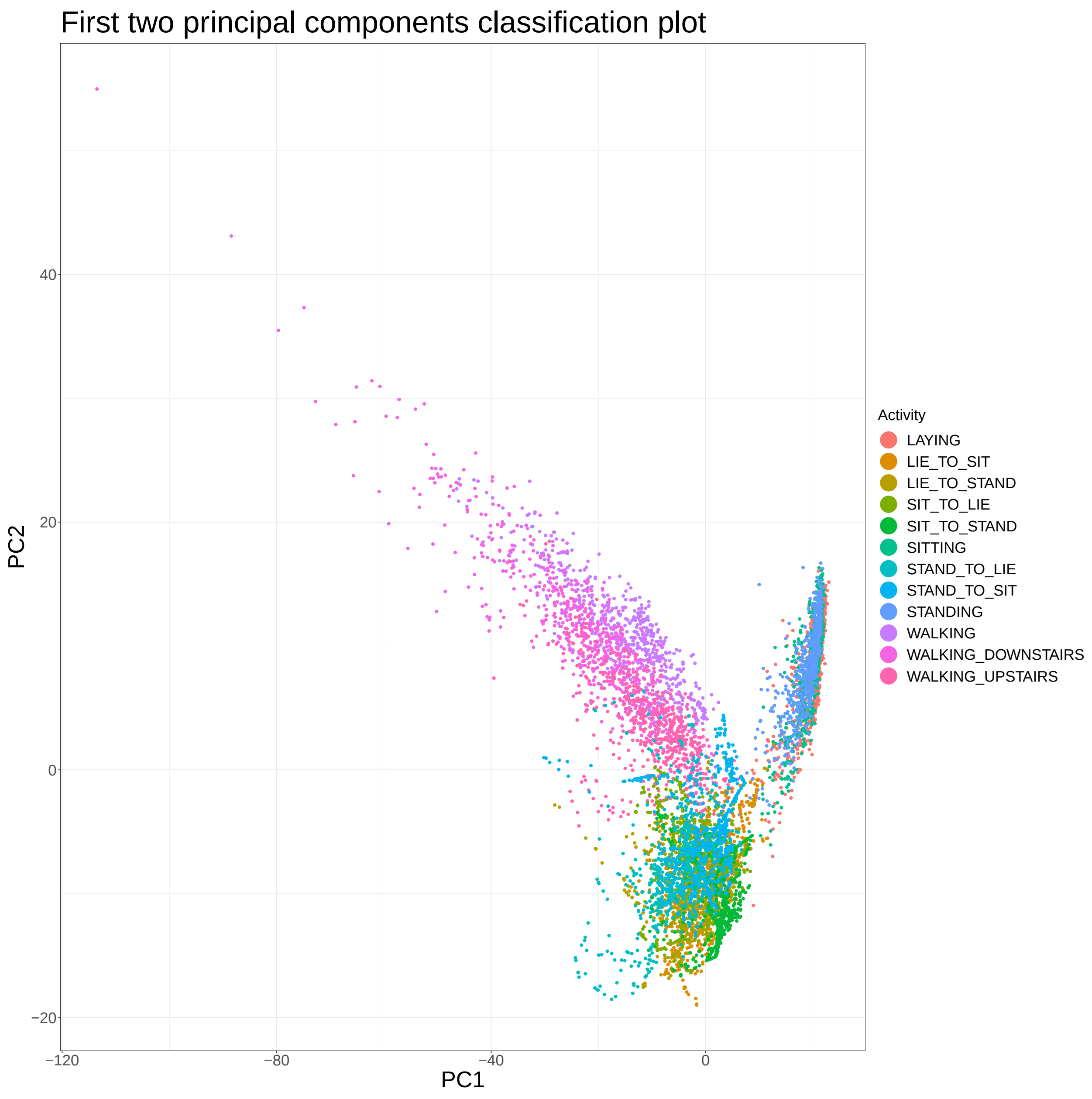


Figure 7: First two principal components classification plot

We can see, that some of the classes are very easily can be distinct one from another, but some of them occupy the same area when projecting on the first two principal components. To see, how classes can be easily distinguished from others, let us combine them into three groups:

```
# for better visualization we will introduce some combined classes to
# distinguish activities
static <- c("LAYING", "SITTING", "STANDING")
moving <- c("WALKING", "WALKING_DOWNSTAIRS", "WALKING_UPSTAIRS")
postural_trans <- activity_labels$Activity[!activity_labels$Activity %in% c(static,
    moving)]
```

Now, we can see, how different groups are distinguished:

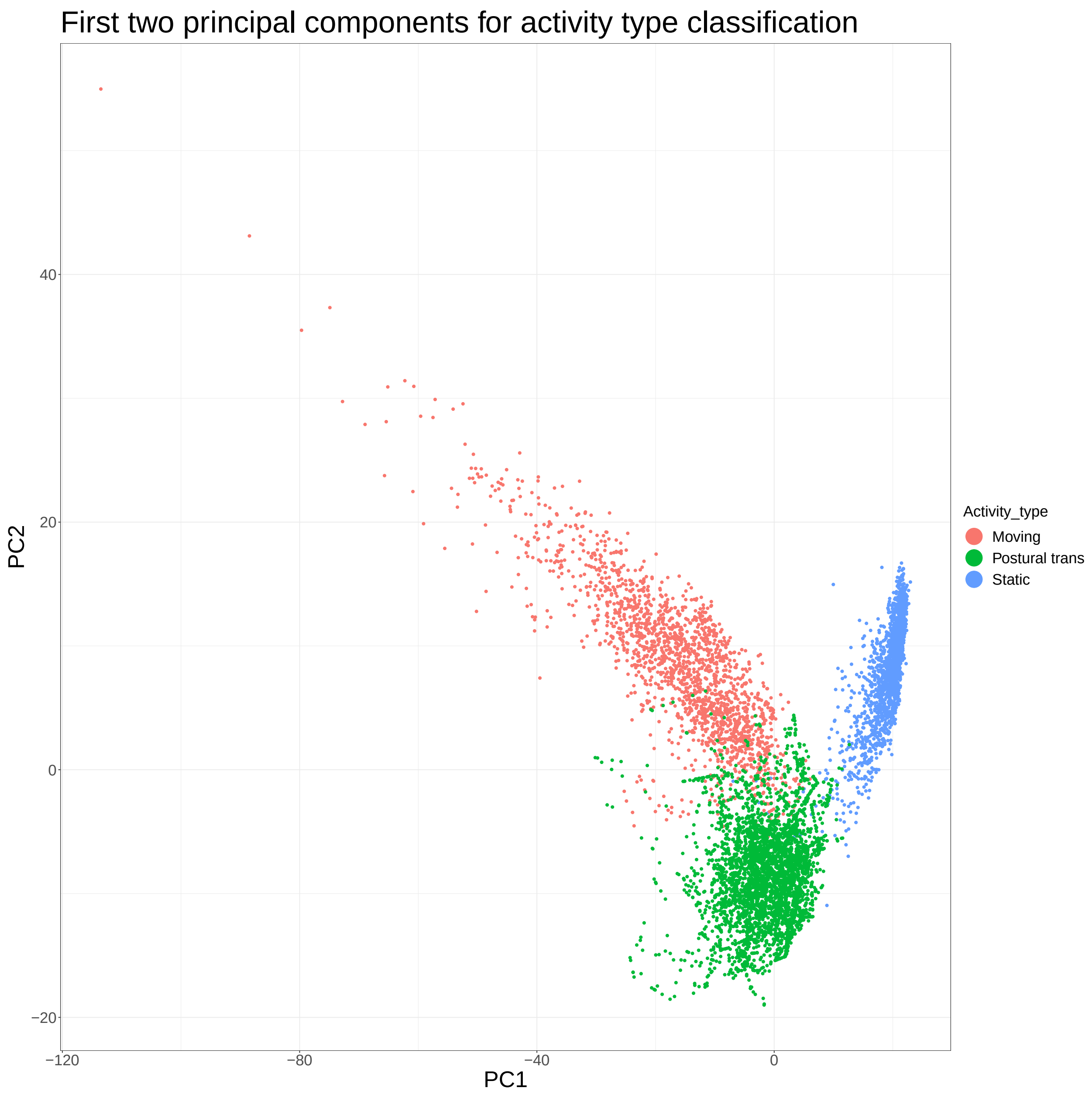


Figure 8: First two principal components for activity type classification

Looking on the Figure 8, we can conclude that type of activity is well explained by two first principal components. Here we have to deal with extremely multidimensional predictors space, therefore for visualization of classification inside the groups four plots for each group will be plotted.

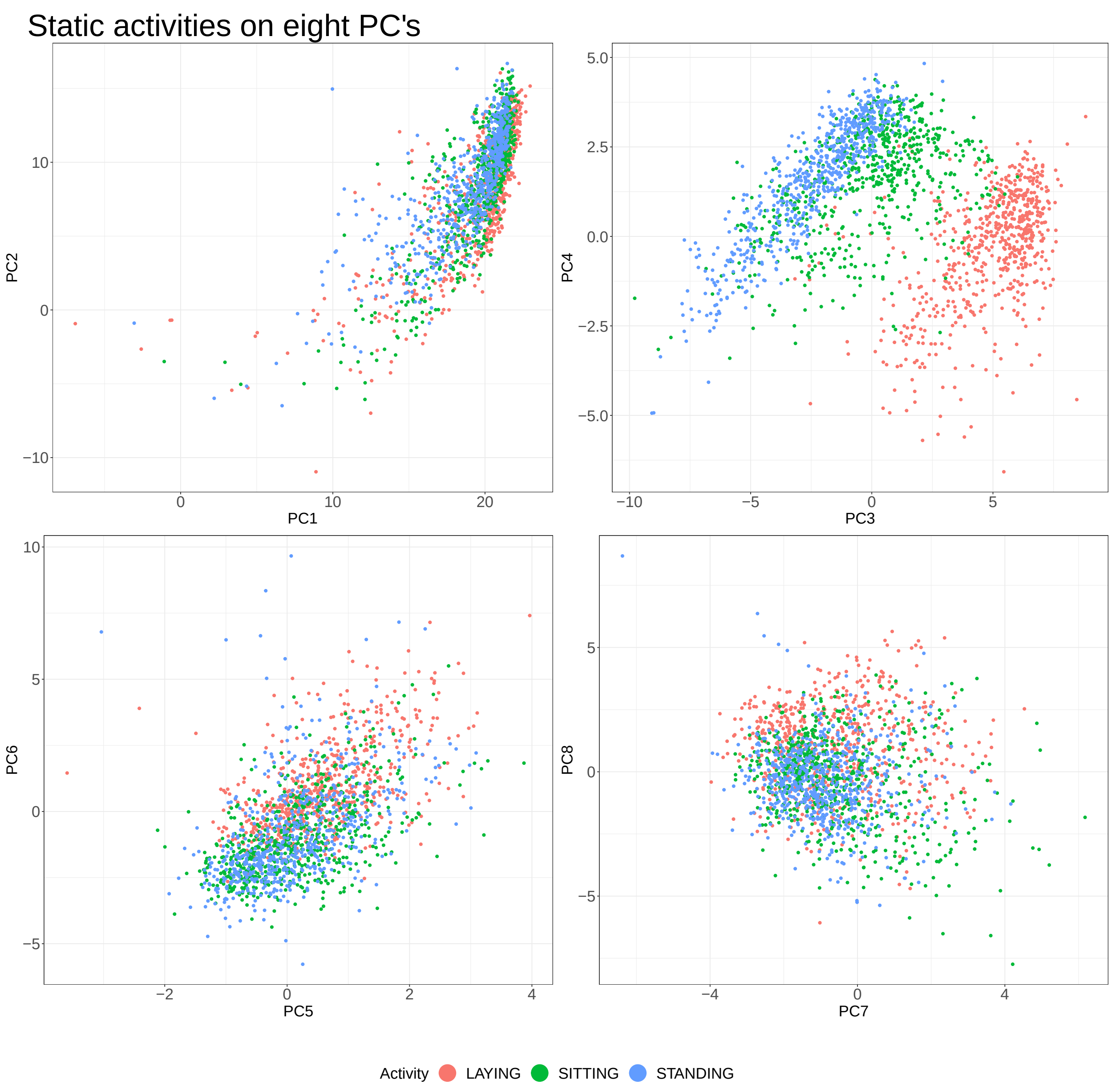


Figure 9: Static activities on eight PC's

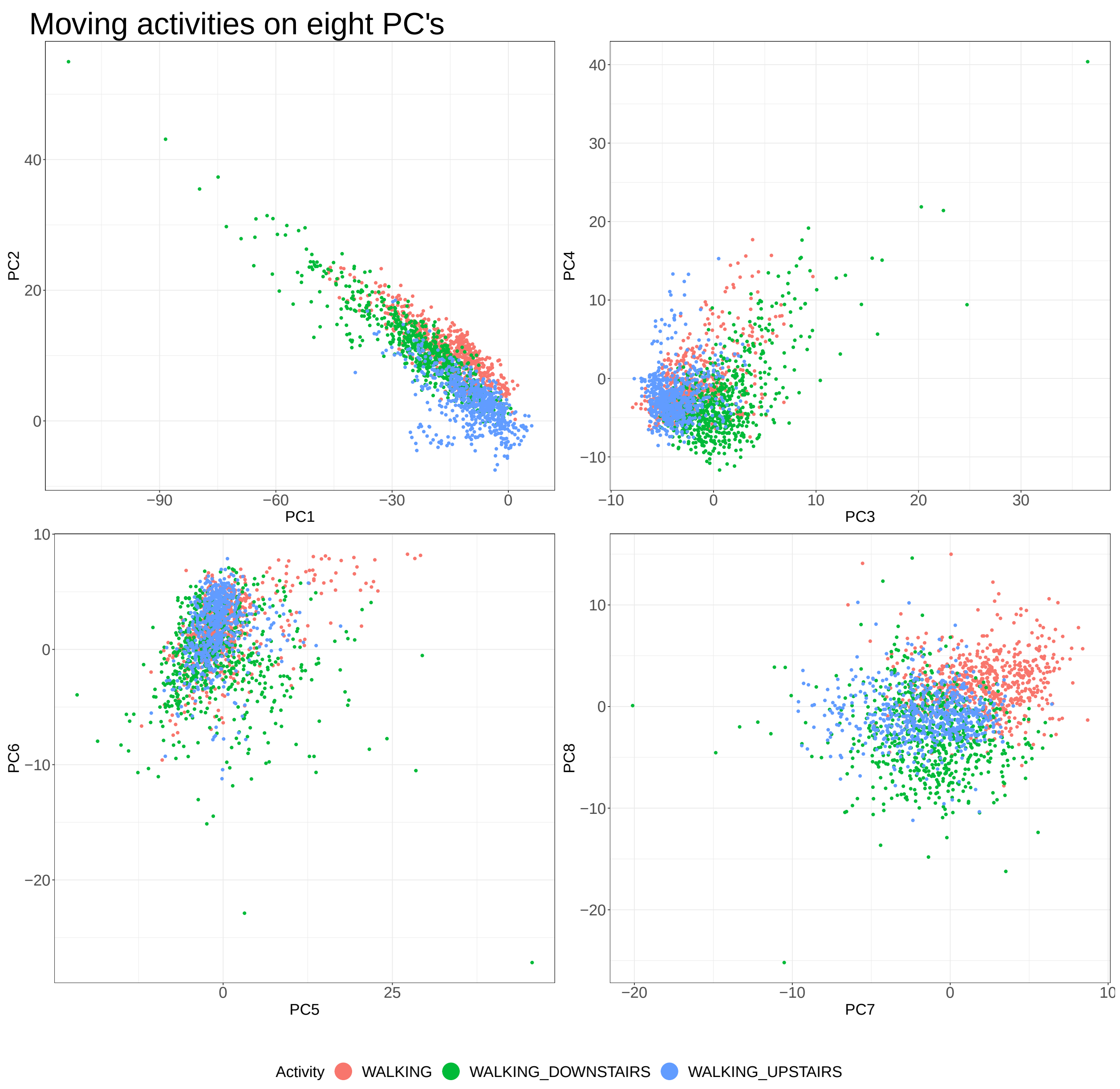


Figure 10: Moving activities on eight PC's

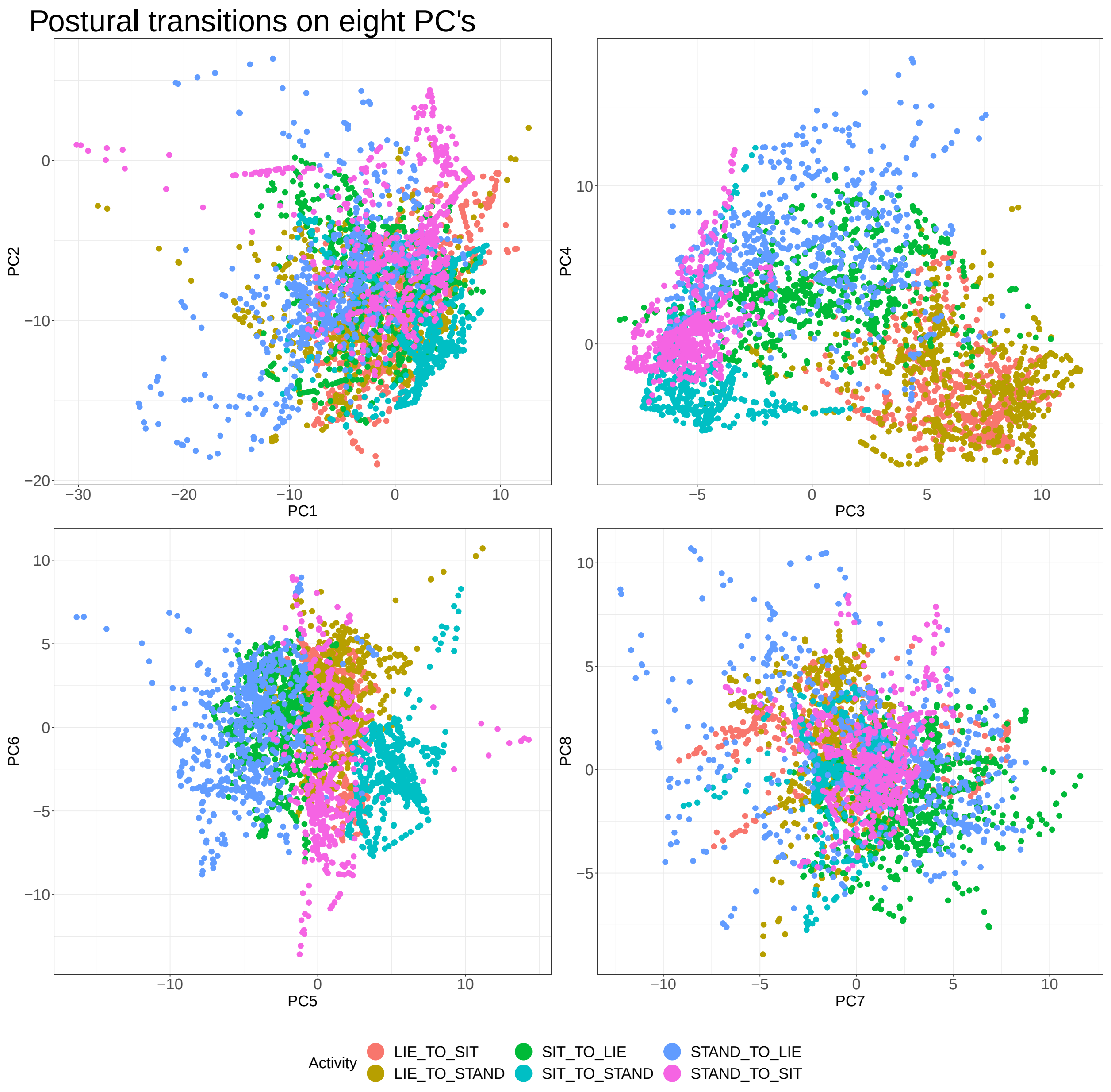


Figure 11: Postural transitions on eight PC's

Classes inside "static" and "moving" groups can mostly be distinguished from each other using only 2-8 dimensional principal components space. Postural transitions difference is not clearly seen in 8 dimensions. Plot in higher principal components dimensions:

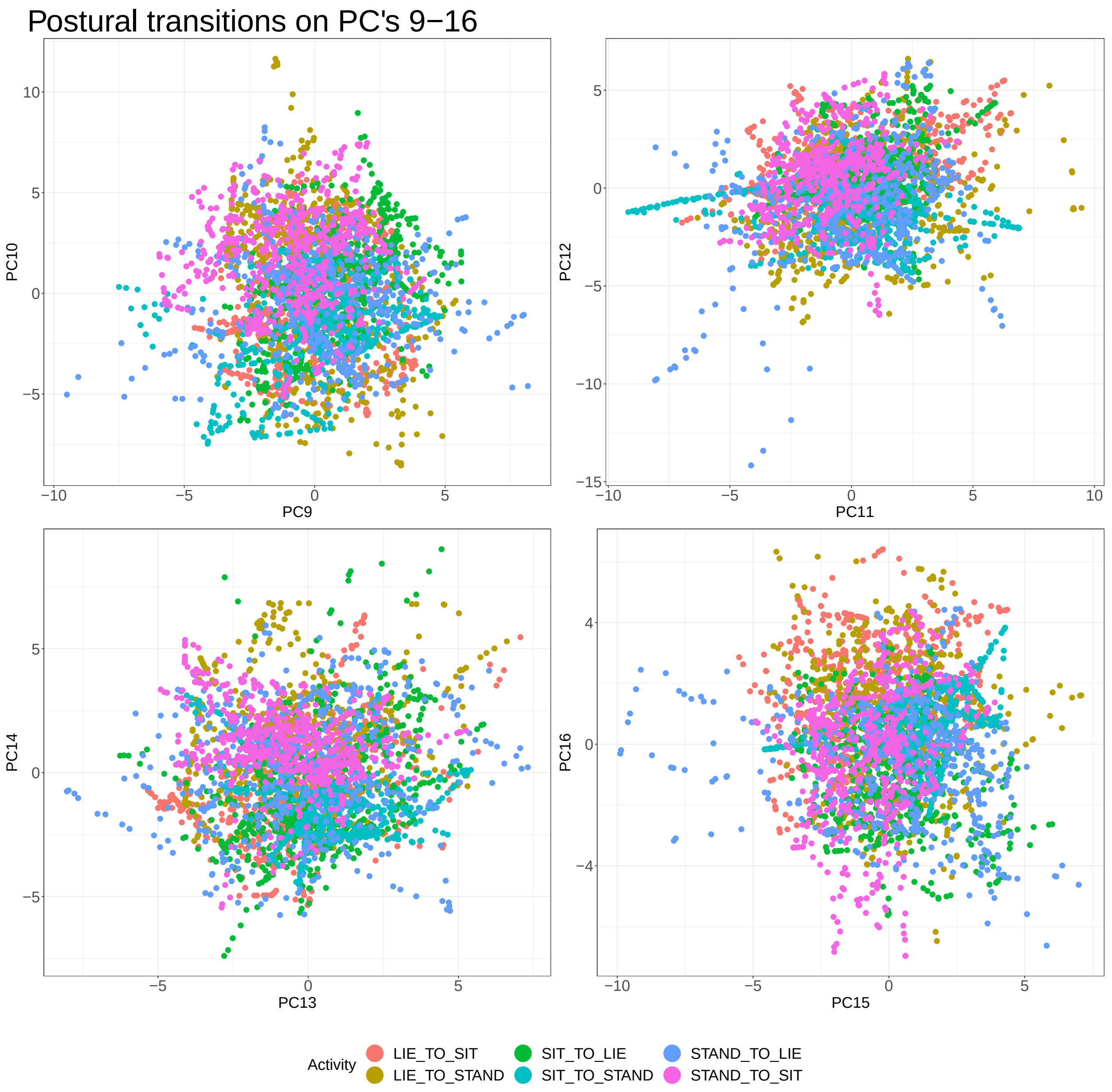


Figure 12: Postural transitions on PC's 9-16

There is a probability that some of classes can be distinguished in higher dimensions.

3.3 Summary

- We are dealing with very high-dimensional data;
- It is multiclass problem with unbalanced data;
- Some classes are intersects with another by their nature, e.g. "STANDING" and "SITTING" (see Figure 9). Some uncertainty in these classes distinguishing is expected;
- All features are already scaled and have numeric class;
- There are no missing values in the dataset.

4 Model building

For multiclass problem 7 different machine learning (ML) algorithms will be tested. Initially, default parameters gridsearch and 5-fold cross validation will be used. Predictors for ML will be transformed by PCA and only 100 of them will be used. From Figure 6 we can see, that it should explain about 95% of outcome variability and will reduce training and cross-validation time. To transform original dataset the following code is used:

```
# apply the same pca to non SMOTE df keep 100 features from df_pca
df_pca_or <- as.data.frame(predict(pca, newdata = df[1:ncol(df) - 1]))
df_pca_or <- df_pca_or[1:100] %>%
    cbind(df[ncol(df)])
```

All models will be trained using *trainControl* function from the *caret* package. Metric for best parameter selection will be balanced accuracy, because our dataset is not balanced and we do not have any “important” or “not important” classes in the dataset.

```
control <- trainControl(method = "cv", number = 5, classProbs = TRUE, summaryFunction = multiClassSumma
    savePredictions = "final", search = "grid", verbose = PRINT_DEBUG)


metric <- "Mean_Balanced_Accuracy"
```

Models performances will be compared after all seven models are trained.

4.1 k-nearest neighbors algorithm

The k-nearest neighbors algorithm (k-NN) is a non-parametric classification method. It is used for classification and regression. In both cases, the input consists of the k closest training examples in a data set. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.

k-NN implementation from *kknn* library will be used with default parameter (k) search grid. 5-fold cross-validation will be used to find optimal k.

The following code will be used to train k-NN model (it will load trained model from file if *RETRAIN* bit set to False):

```
file_name <- "./models//kknn.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_kknn <- train(Activity ~ ., data = df_pca_or, method = "kknn", metric = metric,
        trControl = control)
    time_end <- unclass(Sys.time())
    kknn_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_kknn, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/kknn.rds",
            file_name)
    }
    # read from file
    fit_kknn <- readRDS(file_name)
}
```

4.2 Penalized Discriminant Analysis

Fisher's linear discriminant analysis (LDA) is a popular data-analytic tool for studying the relationship between a set of predictors and a categorical response. It uses to to find a linear combination of features that characterizes or separates two or more classes of objects or events. Penalized Discriminant Analysis (PDA) is a penalized version of LDA. PDA shrinks the coefficients by penalizing the regression model with a penalty term called L2-norm, which is the sum of the squared coefficients. PDA algorithm from *mda* library will be used.

PDA will be trained the same way as k-NN.

```
file_name <- "./models//pda.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_pda <- train(Activity ~ ., data = df_pca_or, method = "pda", metric = metric,
        trControl = control)
    time_end <- unclass(Sys.time())
    pda_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_pda, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/pda.rds",
            file_name)
    }
    # read from file
    fit_pda <- readRDS(file_name)
}
```

4.3 Multinomial logistic regression

Multinomial logistic regression (MNL) is a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes. That is, it is a model that is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables. *nnet* library contains MNL implementation:

```
file_name <- "./models//multinom.rds"
if (RETRAIN) {
```

```r
    time_start <- unclass(Sys.time())
    fit_multinom <- train(Activity ~ ., data = df_pca_or, method = "multinom", metric = metric,
        trControl = control, MaxNWts = 15000)
    time_end <- unclass(Sys.time())
    multinom_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_multinom, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/multinom.rds",
            file_name)
    }
    # read from file
    fit_multinom <- readRDS(file_name)
}
```

4.4 Stochastic Gradient Boosting

Gradient boosting constructs additive regression models by sequentially fitting a simple parameterized function (base learner) to current “pseudo”-residuals by least squares at each iteration. The pseudo-residuals are the gradient of the loss functional being minimized, with respect to the model values at each training data point evaluated at the current step.Accuracy and execution speed of gradient boosting can be substantially improved by incorporating randomization into the procedure. Specifically, at each iteration a subsample of the training data is drawn at random (without replacement) from the full training data set. This randomly selected subsample is then used in place of the full sample to fit the base learner and compute the model update for the current iteration. This randomized approach also increases robustness against overcapacity of the base learner. Stochastic Gradient Boosting algorith from library *gbm* will be used:

```r
file_name <- "./models//gbm.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_gbm <- train(Activity ~ ., data = df_pca_or, method = "gbm", metric = metric,
        trControl = control, bag.fraction = 1, nTrain = 3000)
    time_end <- unclass(Sys.time())
    gbm_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_gbm, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/gbm.rds",
            file_name)
    }
    # read from file
    fit_gbm <- readRDS(file_name)
}
```

4.5 eXtreme Gradient Boosting

Extreme Gradient Boosting (XGBoost) is a specific implementation of the Gradient Boosting method which uses more accurate approximations to find the best tree model. It employs a number tricks that make it exceptionally successful, particularly with structured data. The most important are: - Computing second-order gradients, i.e. second partial derivatives of the loss function (similar to Newton's method), which provides more information about the direction of gradients and how to get to the minimum of our loss function. While regular gradient boosting uses the loss function of our base model (e.g. decision tree) as a proxy for minimizing the error of the overall model, XGBoost uses the 2nd order derivative as an approximation. - Advanced regularization (L1 & L2), which improves model generalization. Library *xgboost* will be used for the model:

```
file_name <- "./models//xgbTree.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_xgbTree <- train(Activity ~ ., data = df_pca_or, method = "xgbTree", metric = metric,
        trControl = control)
    time_end <- unclass(Sys.time())
    xgbTree_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_xgbTree, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/xgbTree.rds",
            file_name)
    }
    # read from file
    fit_xgbTree <- readRDS(file_name)
}
```

4.6 Parallel Random Forest

Random forest is a method for building models by combining decision trees or decision trees generated from bootstrap samples and random features. A common problem that often occurs when implementing random forest is long processing time because it uses a lot of data and build many tree models to form random trees because it uses single processor. Library *randomForest* contains implementation of random forest method with parallel computing which can improve computational time.

```
file_name <- "./models//parRF.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_parRF <- train(Activity ~ ., data = df_pca_or, method = "parRF", metric = metric,
        trControl = control)
    time_end <- unclass(Sys.time())
    parRF_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_parRF, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/parRF.rds",
            file_name)
    }
    # read from file
    fit_parRF <- readRDS(file_name)
}
```

4.7 Neural Network

Finally, we will try Artificial Neural Network (ANN) implementation. A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. Neural networks are able to find non-linear dependencies in multidimensional data. Especially they are effective for images or sound recognition. Because the data in the current project is multidimensional, ANN may have some advantage. Library *nnet* will be used to train the model:

```
file_name <- "./models//nnet.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_nnet <- train(Activity ~ ., data = df_pca_or, method = "nnet", metric = metric,
        trControl = control, MaxNWts = 15000)
    time_end <- unclass(Sys.time())
    nnet_time <- time_end - time_start
    # If 'models' folder is not exist, create it
    if (!dir.exists("./models")) {
        dir.create("./models")
    }
    # save fits
    saveRDS(fit_nnet, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/nnet.rds",
            file_name)
    }
    # read from file
    fit_nnet <- readRDS(file_name)
}
```

4.8 Model comparison and selection

To select a model which will be improved later on, we will look on models metrics and confusion matrices. First, make a table with model name, computational time (5-fold cross validation and default gridSearch for parameters) and different metrics. Models with highest Mean Balanced Accuracy from gridSearch will be compared:

Table 4: Different models train results

Name	Mean_Balanced_Accuracy	Kappa	Accuracy	Time
kknn	0.8526	0.8896	0.9070	183.430
pda	0.9139	0.9075	0.9218	8.991
multinom	0.9139	0.9270	0.9383	148.718
gbm	0.8260	0.8386	0.8638	353.974
xgbTree	0.9066	0.9321	0.9427	2873.427
parRF	0.8742	0.8922	0.9090	599.350
nnet	0.8260	0.8720	0.8919	146.196

We can see, that pda, multinom and xgbTree show the best balanced accuracy. Compare by kappa value, multinom and xgbTree are the leaders. But taking in account amount of tuning parameters and computational time, multinom looks the best one. To check models performance for specific classes, confusion matrices will be plotted for each model:

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

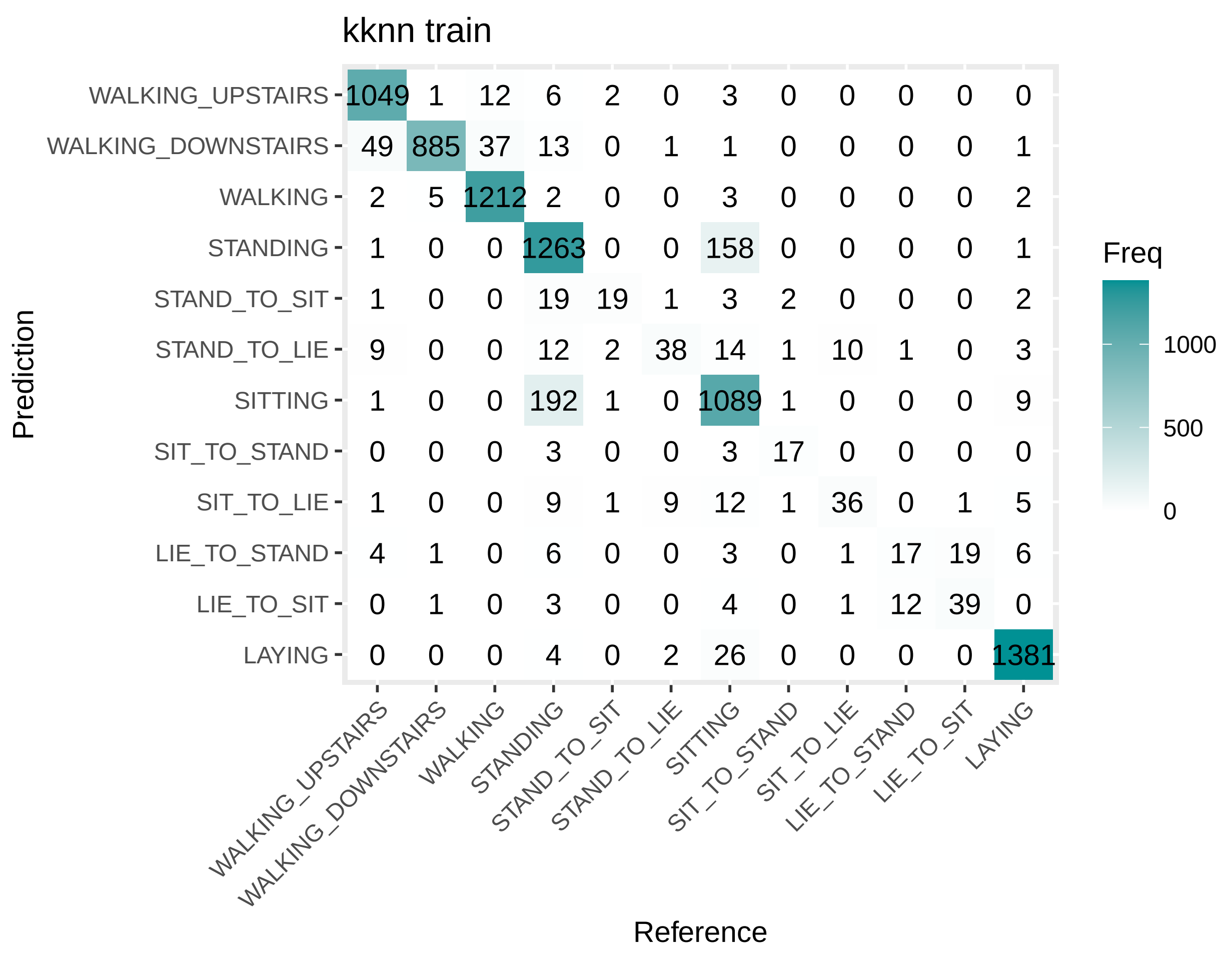
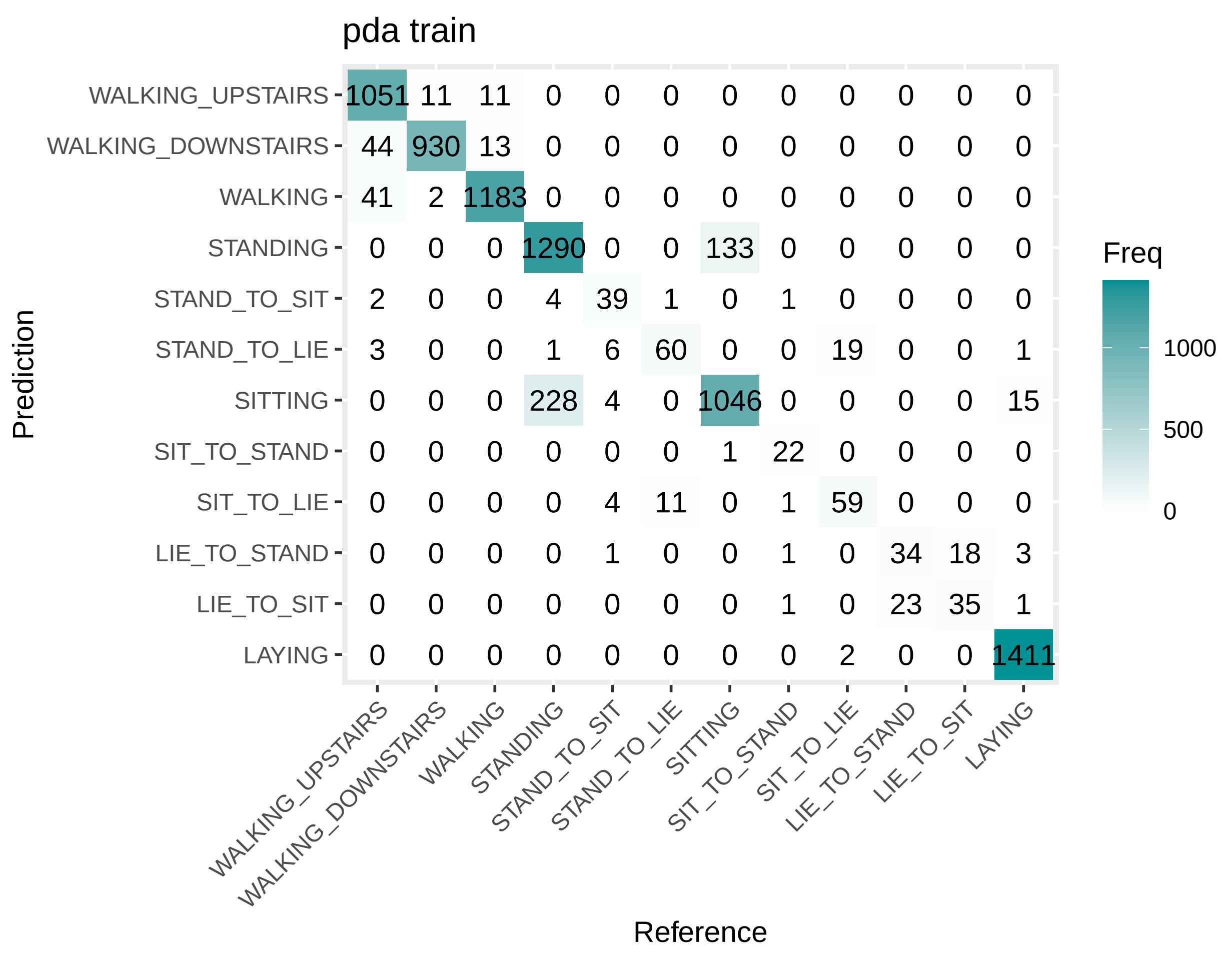


Figure 13: Confusion matrices

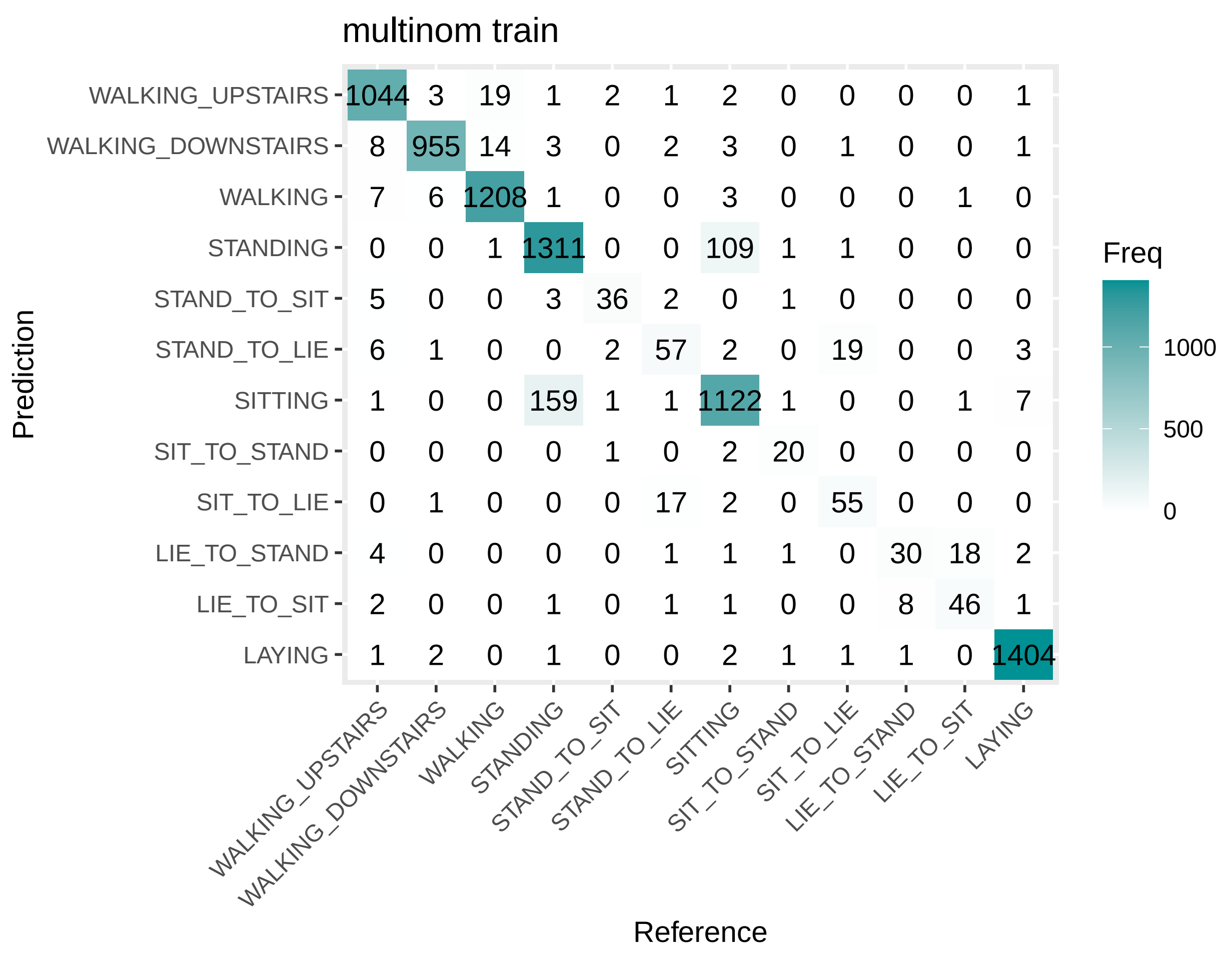
```
##
## [[2]]
```



Prediction \ Reference	WALKING_UPSTAIRS	WALKING_DOWNSTAIRS	WALKING	STANDING	STAND_TO_SIT	STAND_TO_LIE	SITTING	SIT_TO_STAND	SIT_TO_LIE	LIE_TO_STAND	LIE_TO_SIT	LAYING
WALKING_UPSTAIRS	1051	11	11	0	0	0	0	0	0	0	0	0
WALKING_DOWNSTAIRS	44	930	13	0	0	0	0	0	0	0	0	0
WALKING	41	2	1183	0	0	0	0	0	0	0	0	0
STANDING	0	0	0	1290	0	0	133	0	0	0	0	0
STAND_TO_SIT	2	0	0	4	39	1	0	1	0	0	0	0
STAND_TO_LIE	3	0	0	1	6	60	0	0	19	0	0	1
SITTING	0	0	0	228	4	0	1046	0	0	0	0	15
SIT_TO_STAND	0	0	0	0	0	0	1	22	0	0	0	0
SIT_TO_LIE	0	0	0	0	4	11	0	1	59	0	0	0
LIE_TO_STAND	0	0	0	0	1	0	0	1	0	34	18	3
LIE_TO_SIT	0	0	0	0	0	0	0	1	0	23	35	1
LAYING	0	0	0	0	0	0	0	0	2	0	0	1411

Figure 14: Confusion matrices

```
##
## [[3]]
```



	WALKING_UPSTAIRS	WALKING_DOWNSTAIRS	WALKING	STANDING	STAND_TO_SIT	STAND_TO_LIE	SITTING	SIT_TO_STAND	SIT_TO_LIE	LIE_TO_STAND	LIE_TO_SIT	LAYING
WALKING_UPSTAIRS	1044	3	19	1	2	1	2	0	0	0	0	1
WALKING_DOWNSTAIRS	8	955	14	3	0	2	3	0	1	0	0	1
WALKING	7	6	1208	1	0	0	3	0	0	0	1	0
STANDING	0	0	1	1311	0	0	109	1	1	0	0	0
STAND_TO_SIT	5	0	0	3	36	2	0	1	0	0	0	0
STAND_TO_LIE	6	1	0	0	2	57	2	0	19	0	0	3
SITTING	1	0	0	159	1	1	1122	1	0	0	1	7
SIT_TO_STAND	0	0	0	0	1	0	2	20	0	0	0	0
SIT_TO_LIE	0	1	0	0	0	17	2	0	55	0	0	0
LIE_TO_STAND	4	0	0	0	0	1	1	1	0	30	18	2
LIE_TO_SIT	2	0	0	1	0	1	1	0	0	8	46	1
LAYING	1	2	0	1	0	0	2	1	1	1	0	1404

Figure 15: Confusion matrices

```
##
## [[4]]
```

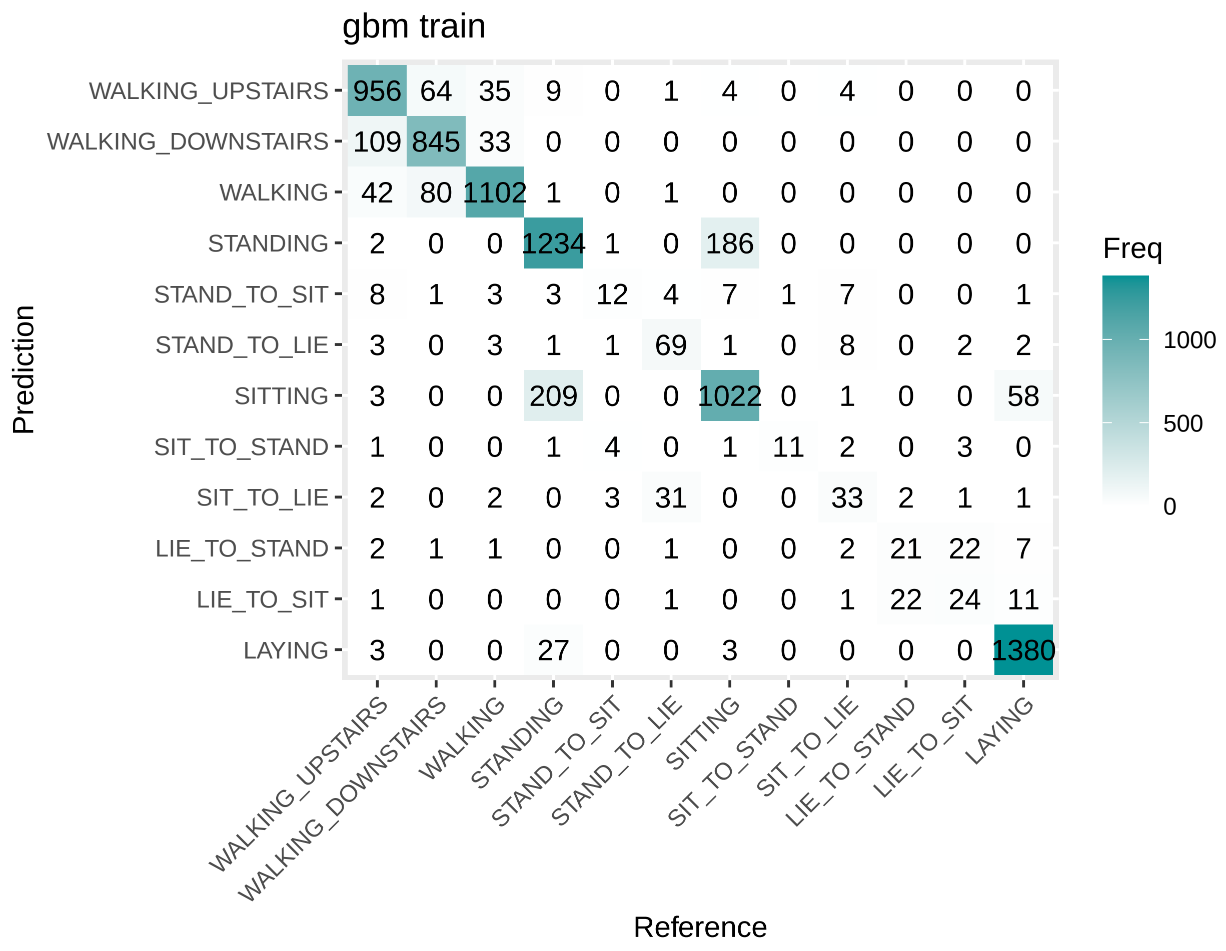
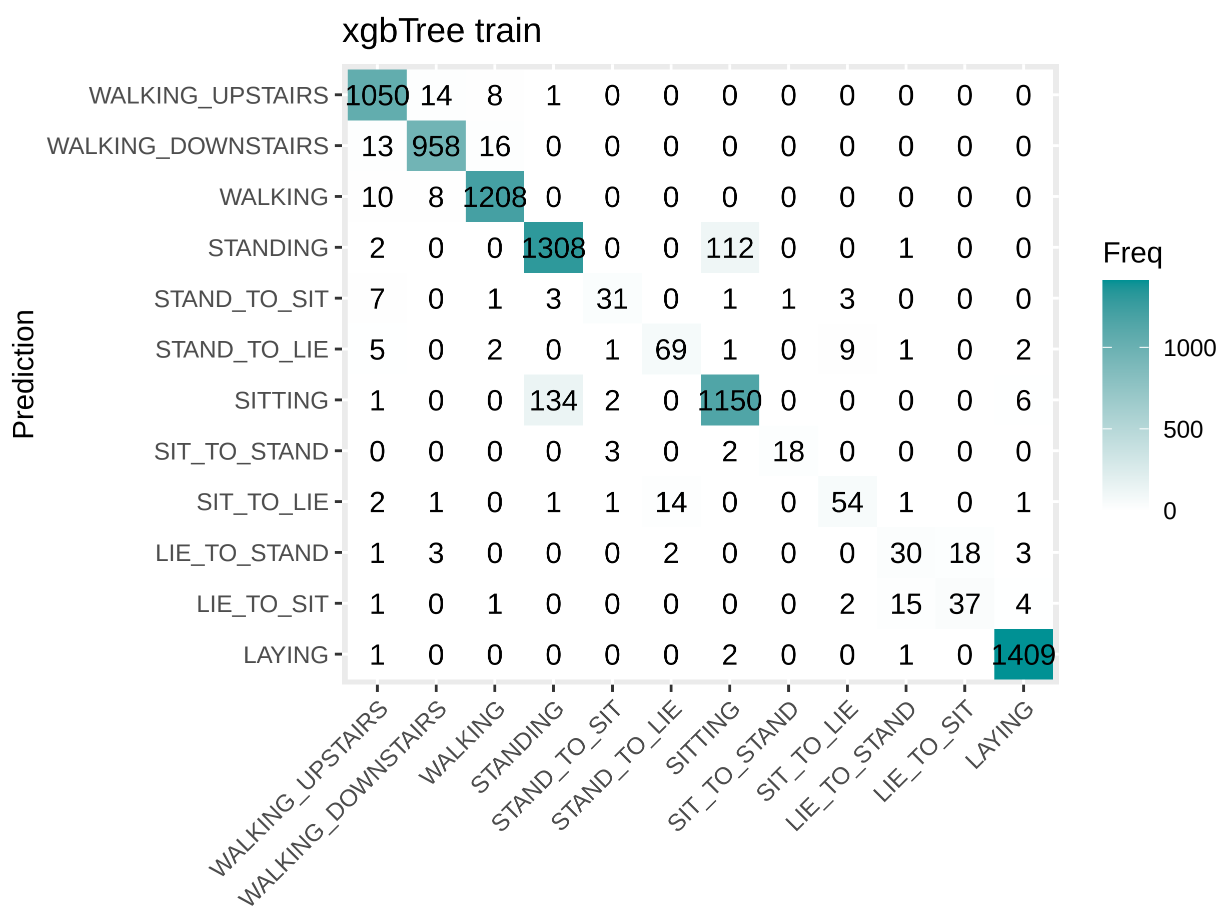


Figure 16: Confusion matrices

```
##
## [[5]]
```



Prediction \ Reference	WALKING_UPSTAIRS	WALKING_DOWNSTAIRS	WALKING	STANDING	STAND_TO_SIT	STAND_TO_LIE	SITTING	SIT_TO_STAND	SIT_TO_LIE	LIE_TO_STAND	LIE_TO_SIT	LAYING
WALKING_UPSTAIRS	1050	14	8	1	0	0	0	0	0	0	0	0
WALKING_DOWNSTAIRS	13	958	16	0	0	0	0	0	0	0	0	0
WALKING	10	8	1208	0	0	0	0	0	0	0	0	0
STANDING	2	0	0	1308	0	0	112	0	0	1	0	0
STAND_TO_SIT	7	0	1	3	31	0	1	1	3	0	0	0
STAND_TO_LIE	5	0	2	0	1	69	1	0	9	1	0	2
SITTING	1	0	0	134	2	0	1150	0	0	0	0	6
SIT_TO_STAND	0	0	0	0	3	0	2	18	0	0	0	0
SIT_TO_LIE	2	1	0	1	1	14	0	0	54	1	0	1
LIE_TO_STAND	1	3	0	0	0	2	0	0	0	30	18	3
LIE_TO_SIT	1	0	1	0	0	0	0	0	2	15	37	4
LAYING	1	0	0	0	0	0	2	0	0	1	0	1409

Figure 17: Confusion matrices

```
##
## [[6]]
```

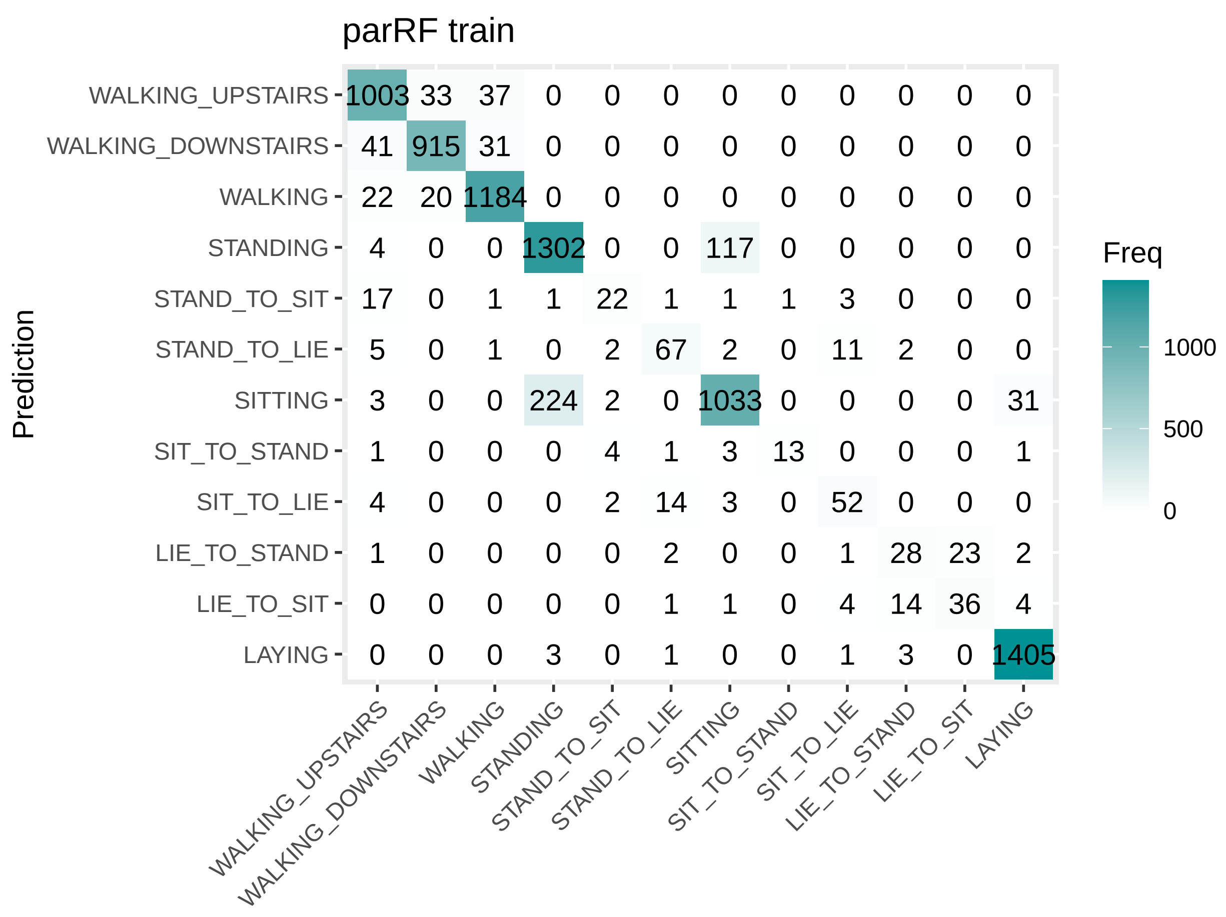
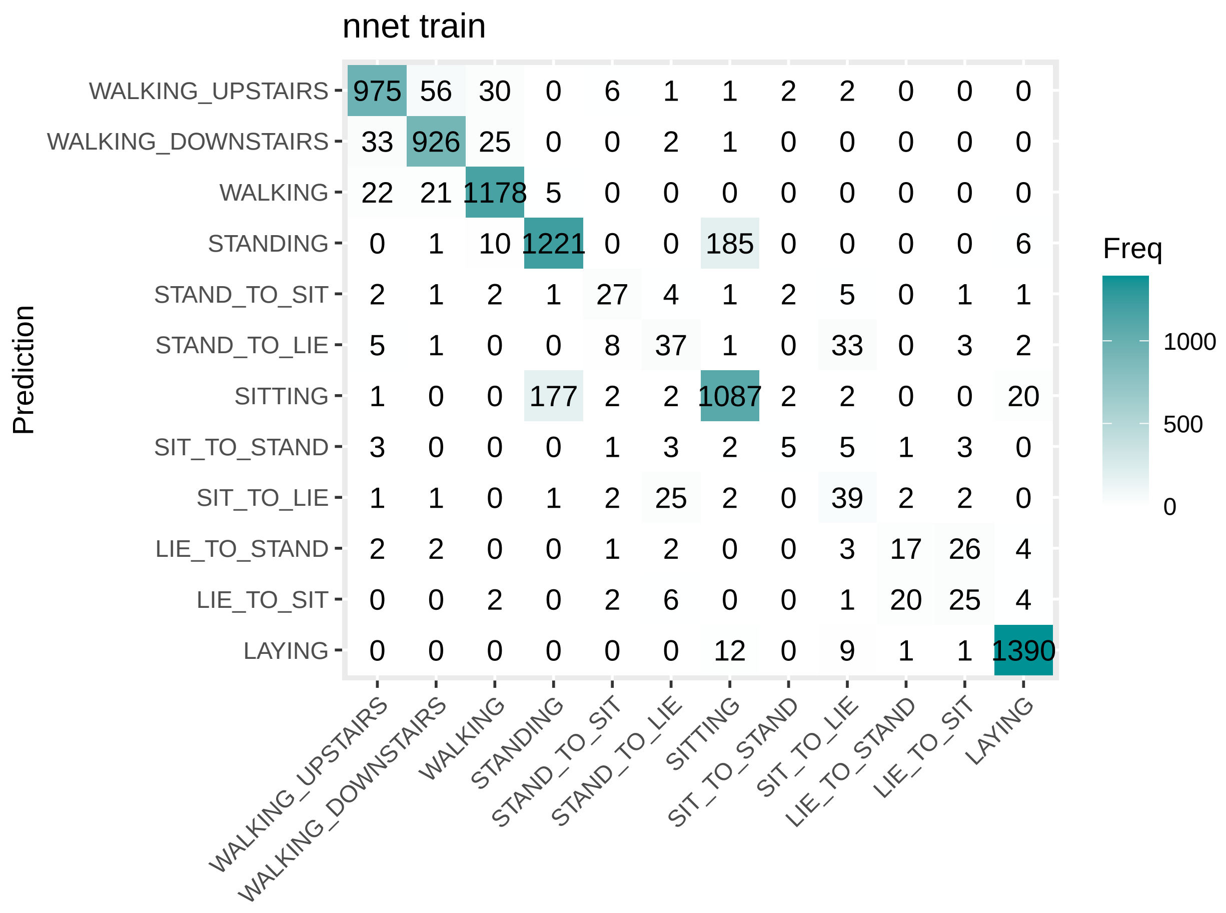


Figure 18: Confusion matrices

```
##
## [[7]]
```



Prediction \ Reference	WALKING_UPSTAIRS	WALKING_DOWNSTAIRS	WALKING	STANDING	STAND_TO_SIT	STAND_TO_LIE	SITTING	SIT_TO_STAND	SIT_TO_LIE	LIE_TO_STAND	LIE_TO_SIT	LAYING
WALKING_UPSTAIRS	975	56	30	0	6	1	1	2	2	0	0	0
WALKING_DOWNSTAIRS	33	926	25	0	0	2	1	0	0	0	0	0
WALKING	22	21	1178	5	0	0	0	0	0	0	0	0
STANDING	0	1	10	1221	0	0	185	0	0	0	0	6
STAND_TO_SIT	2	1	2	1	27	4	1	2	5	0	1	1
STAND_TO_LIE	5	1	0	0	8	37	1	0	33	0	3	2
SITTING	1	0	0	177	2	2	1087	2	2	0	0	20
SIT_TO_STAND	3	0	0	0	1	3	2	5	5	1	3	0
SIT_TO_LIE	1	1	0	1	2	25	2	0	39	2	2	0
LIE_TO_STAND	2	2	0	0	1	2	0	0	3	17	26	4
LIE_TO_SIT	0	0	2	0	2	6	0	0	1	20	25	4
LAYING	0	0	0	0	0	0	12	0	9	1	1	1390

Figure 19: Confusion matrices

Looking on Figure 13 we see, that all models have difficulties to distinguish STANDING and SITTING classes, which was expected based on EDA. From confusion matrices we can conclude, that xgbTree and multinom are showing the least wrongly predicted classes. Taking into account results from Table 4, the Multinomial logistic regression will be used for further improvement. ## Multinomial logistic regression tuning Multinom has one tunable parameter: Weight Decay. It helps to avoid overfitting by penalize complexity of the model. Let's see, how mean balanced accuracy was changed during default decay gridSearch:

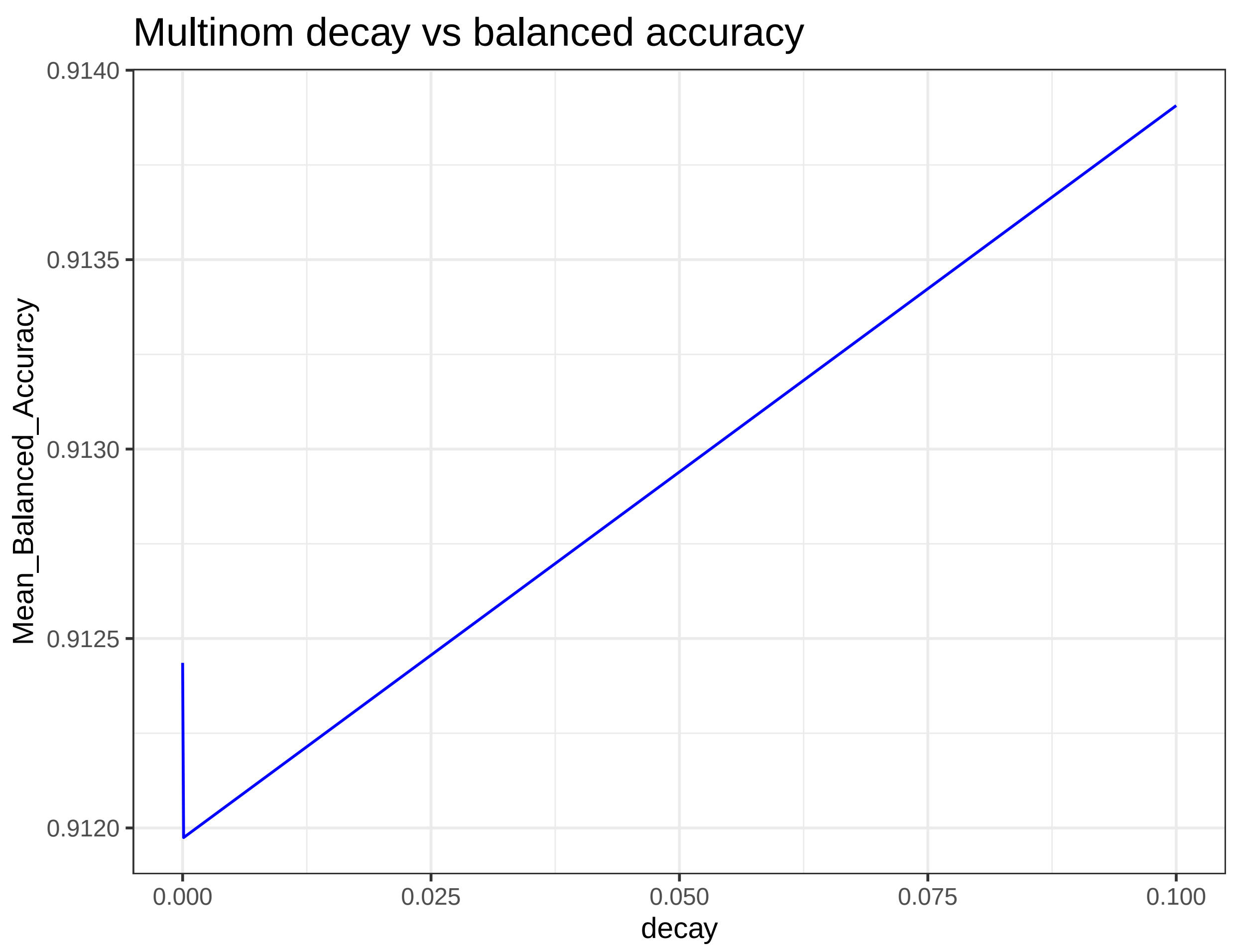


Figure 20: Multinom decay vs balanced accuracy

As we see, balanced accuracy increases, when decay is getting larger. During EDA, we saw that activity clusters are intersecting with each other in low dimensions. To avoid overfitting, larger value of decay can be needed. We will try the following gridSearch:

```
multinom_grid <- expand.grid(decay = c(0.1, 0.5, 1, 3, 5, 8, 10, 13, 15))
```

Also, we will change training cross-validation method. It will be bootstrap, because now we will train only one model, and computational time has less priority compare to well-done multiple cross-validation:

```
control <- trainControl(method = "boot", classProbs = TRUE, summaryFunction = multiClassSummary,
    savePredictions = "final", search = "grid", verbose = PRINT_DEBUG)
```

Train model to find the best decay:

```
file_name <- "./models//multinom_expand_grid.rds"
if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_multinom_grid <- train(Activity ~ ., data = df_pca_or, method = "multinom",
        metric = metric, trControl = control, MaxNWts = 15000, tuneGrid = multinom_grid)
    time_end <- unclass(Sys.time())

    multinom_time_grid <- time_end - time_start
    saveRDS(fit_multinom_grid, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/multinom_expand_grid
            file_name)
    }
    # read from file
    fit_multinom_grid <- readRDS(file_name)
}
```

Plot balanced accuracy versus Weight decay:

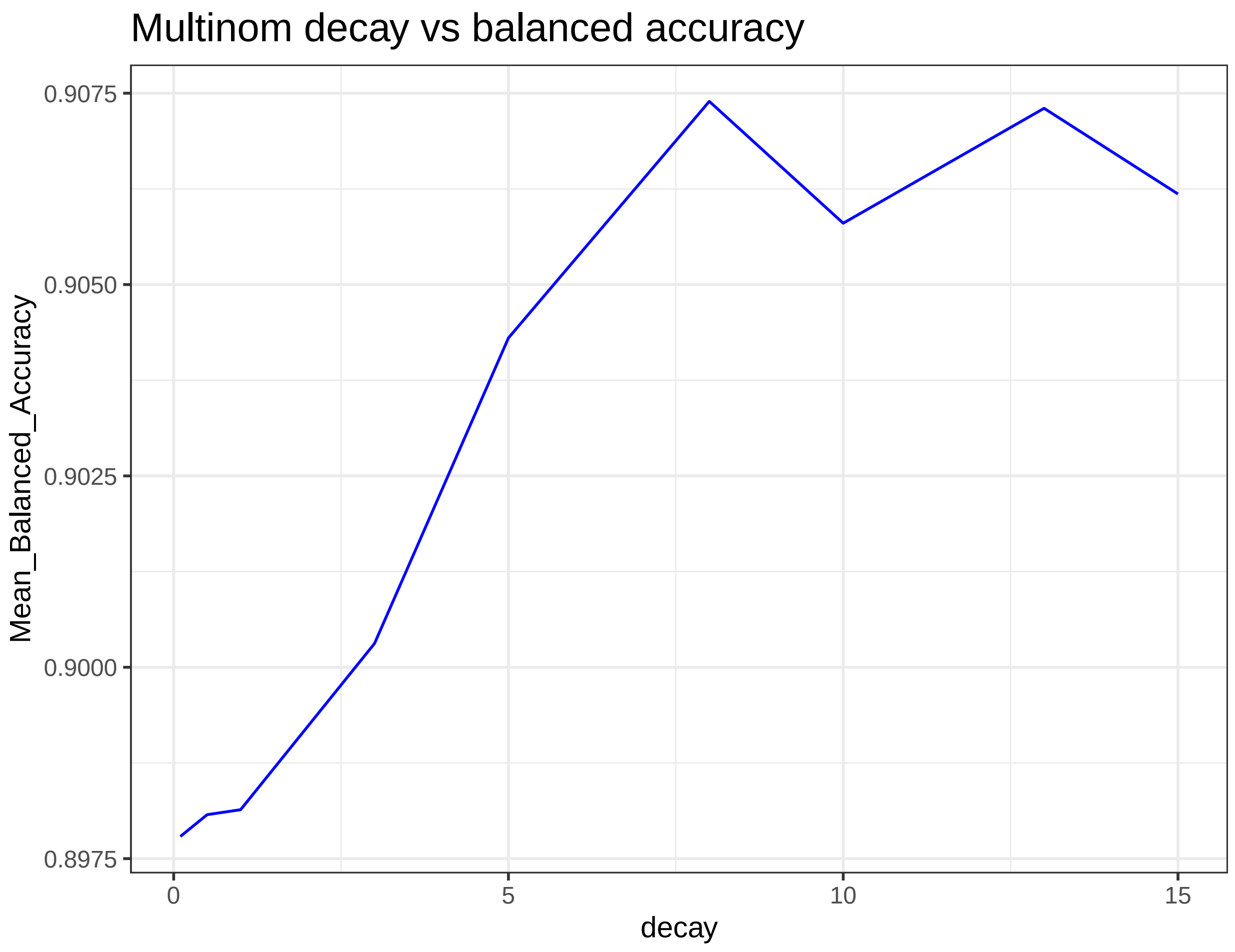


Figure 21: Multinom decay vs balanced accuracy

The best performance is achieved with decay value 8. Let's train model with only this decay and increase maximum iterations number from default value 100 to 500:

```
multinomGrid_fin <- expand.grid(decay = decay)
control <- trainControl(method = "boot", classProbs = TRUE, summaryFunction = multiClassSummary,
    savePredictions = "final", search = "grid", verbose = PRINT_DEBUG)

file_name <- "./models//multinom_final_pca_100.rds"

if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_multinom_fin_100 <- train(Activity ~ ., data = df_pca_or, method = "multinom",
        metric = metric, trControl = control, MaxNWts = 15000, maxit = 500, tuneGrid = multinomGrid_fin
    time_end <- unclass(Sys.time())

    multinom_time_fin_100 <- time_end - time_start
    saveRDS(fit_multinom_fin_100, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/multinom_final_pca_1
            file_name)
    }
    # read from file
    fit_multinom_fin_100 <- readRDS(file_name)
}
```

Add result to the comparison table:

Table 5: Different models train results

Name	Mean_Balanced_Accuracy	Kappa	Accuracy	Time
kknn	0.8526	0.8896	0.9070	183.430
pda	0.9139	0.9075	0.9218	8.991
multinom	0.9139	0.9270	0.9383	148.718
gbm	0.8260	0.8386	0.8638	353.974
xgbTree	0.9066	0.9321	0.9427	2873.427
parRF	0.8742	0.8922	0.9090	599.350
nnet	0.8260	0.8720	0.8919	146.196
multinom_optimal_PCA100	0.9244	0.9427	0.9516	1169.337

All metrics are improved, let's also look on the confusion matrix:

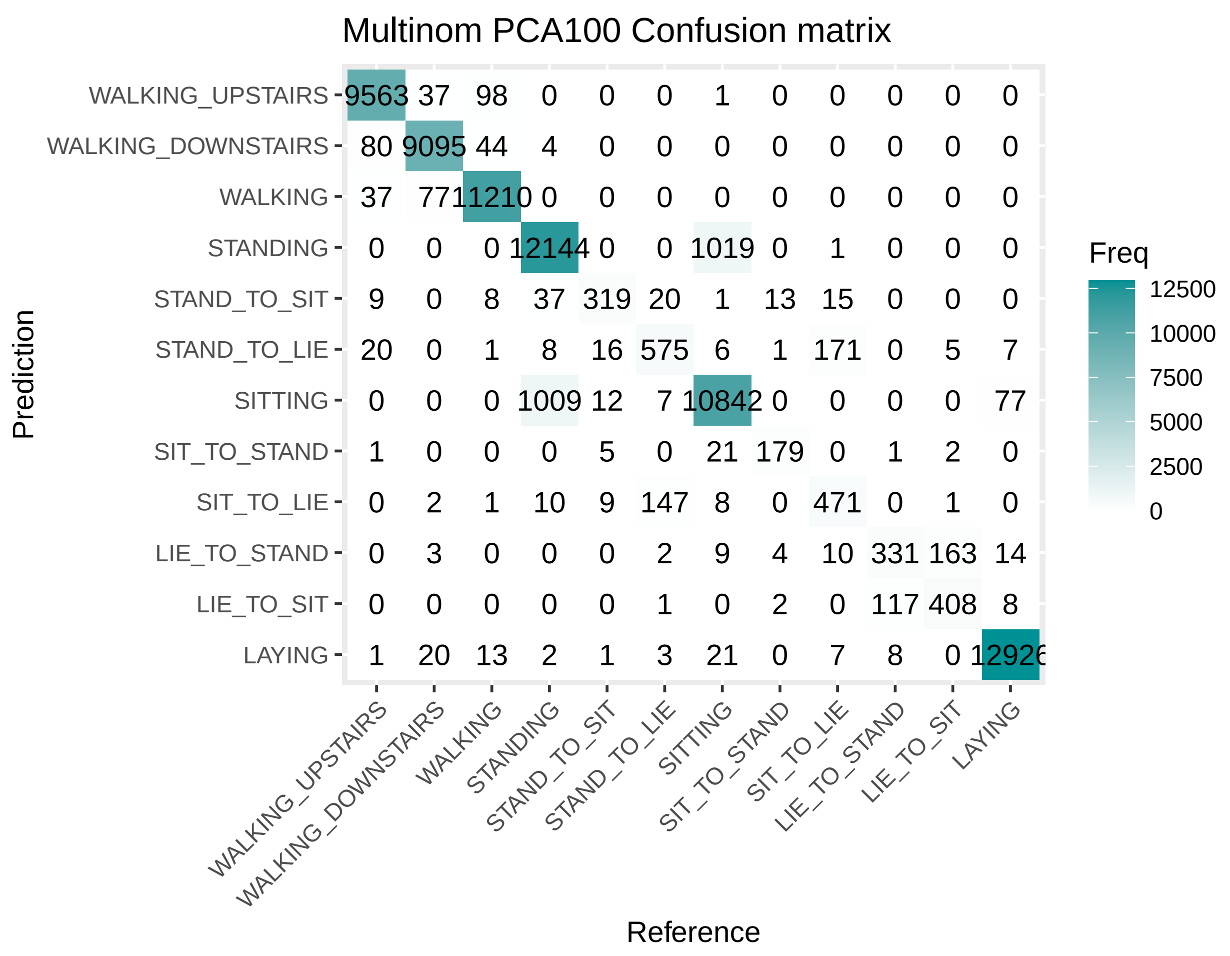


Figure 22: Multinom PCA100 Confusion matrix

Looking on Figure , we see, that except expected difficulties between SITTING, STANDING and LAYING, LIE_TO_SIT and LIE_TO_STAND, and other classes inside one activity type, there is some uncertainties between STAND_TO_LIE and WALKING_UPSTAIRS. Remember, that 100 PC's explain only 95% of outcomes variability, when 200 PC's should be able to explain almost 100%. Let's train our model using 200 PC's:

```r
# keep 200 features from df_pca
df_pca_or <- as.data.frame(predict(pca, newdata = df[1:561]))
df_pca_or <- df_pca_or[1:200] %>%
    cbind(df[562])

file_name <- "./models//multinom_final_pca_200.rds"

if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_multinom_fin_200 <- train(Activity ~ ., data = df_pca_or, method = "multinom",
        metric = metric, trControl = control, MaxNWts = 15000, maxit = 500, tuneGrid = multinomGrid_fin
    time_end <- unclass(Sys.time())

    multinom_time_fin_200 <- time_end - time_start
    saveRDS(fit_multinom_fin_200, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/multinom_final_pca_20
            file_name)
    }
    # read from file
    fit_multinom_fin_200 <- readRDS(file_name)
}
```

Add result to the comparison table:

Table 6: Different models train results

Name	Mean_Balanced_Accuracy	Kappa	Accuracy	Time
kknn	0.8526	0.8896	0.9070	183.430
pda	0.9139	0.9075	0.9218	8.991
multinom	0.9139	0.9270	0.9383	148.718
gbm	0.8260	0.8386	0.8638	353.974
xgbTree	0.9066	0.9321	0.9427	2873.427
parRF	0.8742	0.8922	0.9090	599.350
nnet	0.8260	0.8720	0.8919	146.196
multinom_optimal_PCA100	0.9244	0.9427	0.9516	1169.337
multinom_optimal_PCA200	0.9297	0.9597	0.9660	3002.971

Plot confusion matrix:

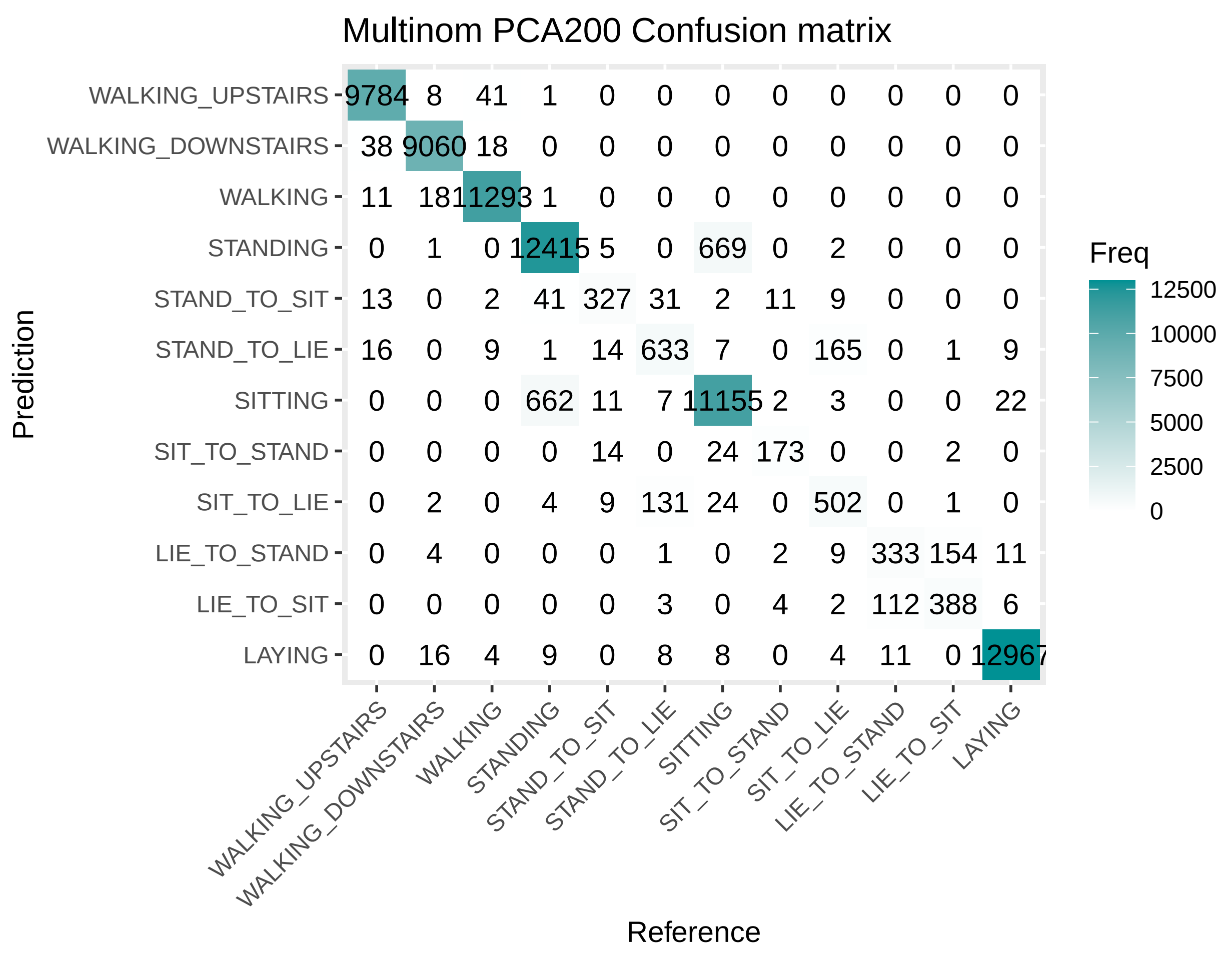


Figure 23: Multinom PCA200 Confusion matrix

All metrics are slightly improved. Of course, computational time is also increased, as numbers of dimensions is getting higher. Looking on confusion matrix, we can see some improvement in class distinguishing.

Finally, let's try to fit all features from dataset without PCA transformation, to check if some important information is hidden in higher dimensions:

```
file_name <- "./models//multinom_final_orig.rds"

if (RETRAIN) {
    time_start <- unclass(Sys.time())
    fit_multinom_orig <- train(Activity ~ ., data = df, method = "multinom", metric = metric,
        trControl = control, MaxNWts = 15000, maxit = 500, tuneGrid = multinomGrid_fin)
    time_end <- unclass(Sys.time())

    multinom_time_orig <- time_end - time_start
    saveRDS(fit_multinom_orig, file_name)
} else {
    # if file is not found, download it.
    if (!file.exists(file_name)) {
        download.file("https://github.com/ParalogyX/HAPT/releases/download/trained/multinom_final_orig.
            file_name)
    }
    # read from file
    fit_multinom_orig <- readRDS(file_name)
}
```

Add result to the comparison table:

Table 7: Different models train results

Name	Mean_Balanced_Accuracy	Kappa	Accuracy	Time
kknn	0.8526	0.8896	0.9070	183.430
pda	0.9139	0.9075	0.9218	8.991
multinom	0.9139	0.9270	0.9383	148.718
gbm	0.8260	0.8386	0.8638	353.974
xgbTree	0.9066	0.9321	0.9427	2873.427
parRF	0.8742	0.8922	0.9090	599.350
nnet	0.8260	0.8720	0.8919	146.196
multinom_optimal_PCA100	0.9244	0.9427	0.9516	1169.337
multinom_optimal_PCA200	0.9297	0.9597	0.9660	3002.971
multinom_optimal_orig	0.9178	0.9550	0.9620	12214.006

and plot confusion matrix:

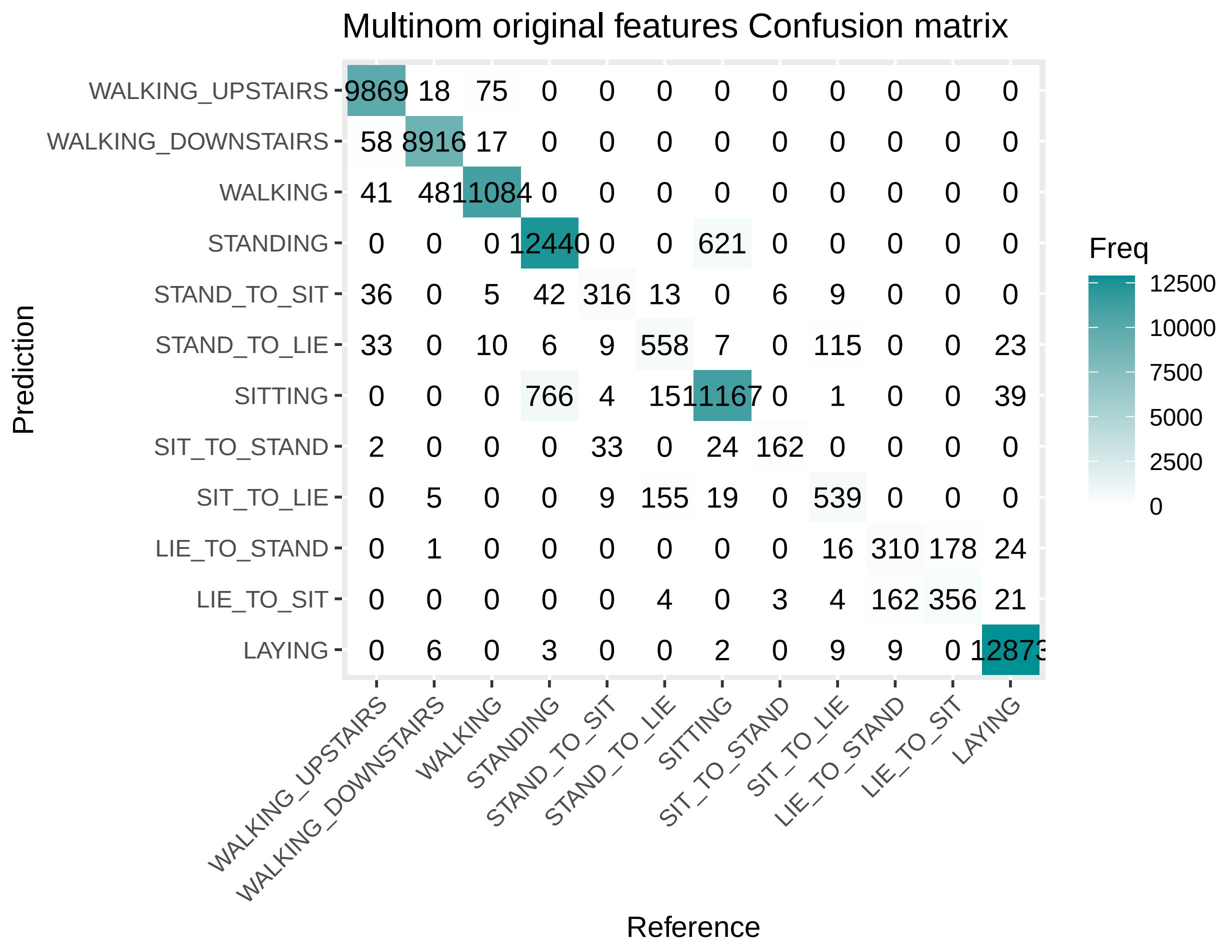


Figure 24: Multinom original features Confusion matrix

Despite very long computational time, adding all features from the original dataset did not improve the metrics. Confusion matrix also doesn't show less errors in the neighboring activity classes. Therefore Multinominal logistic regression on 200 Principal component is chosen as final model.

4.9 Final model function

The following function will be used to make predictions:

```r
multinom_final_predict <- function(features_space) {
    val_pca <- as.data.frame(predict(pca, newdata = features_space))
    predict(fit_multinom_fin_200, val_pca[1:200])
}
```

4.10 Summary

In this chapter we tried seven different models to apply to the dataset. Multinomial logistic regression was selected for tuning. Different features spaces such as PCA-transformed with 100 PC's, PCA-tranformed with 200 PC's and original 561 features were tested. Finally, the model which use Principal Component Analisys first and Multinomial logistic regression with weight decay = 8 after was selected.

5 Validation

To see how our model will perform on real life data, the validation dataset was hold-out from the beginning of the project. *df_validation* was not used for EDA, PCA transformation, models comparison etc. These are completely unseen by our model data. First, predictions from validation dataset are computed:

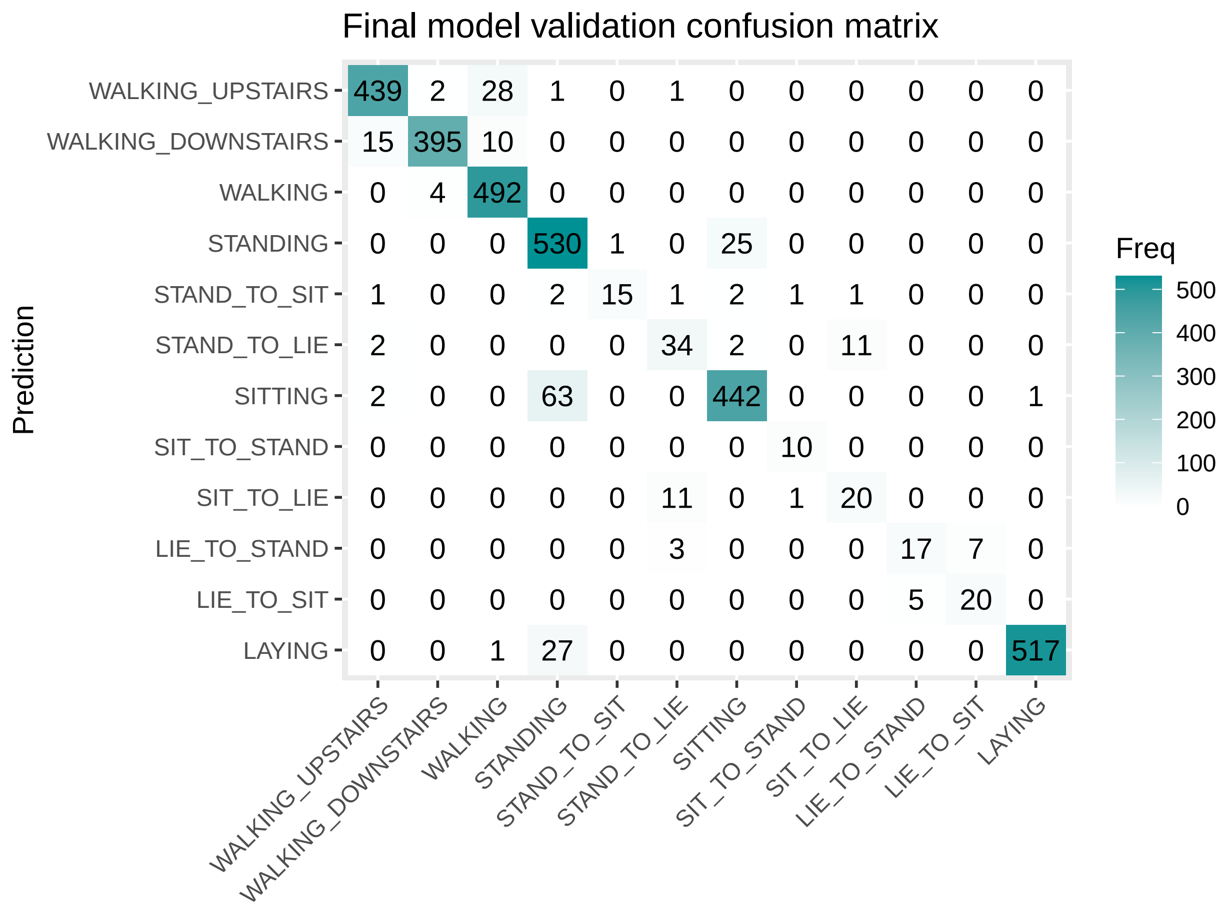
```r
truth <- df_validation$Activity
pred <- multinom_final_predict(df_validation[1:561])
```

Now, we can check final model metrics on the validation dataset and plot confusion matrix:

```r
xtab <- table(pred, truth)

cm <- confusionMatrix(xtab)


result_final_val <- data.frame("multinom_final_validation", mean(cm$byClass[, "Balanced Accuracy"]),
    cm$overall["Kappa"], cm$overall["Accuracy"], "Not applicable")

colnames(result_final_val) <- c("Name", "Mean_Balanced_Accuracy", "Kappa", "Accuracy",
    "Time")

result_final_val[2:4] <- lapply(result_final_val[2:4], as.numeric)
result_final_val[, 2:4] <- round(result_final_val[, 2:4], digits = 4)
result_df <- rbind(result_df, result_final_val)
knitr::kable(result_df, caption = "Different models train and final validation results",
    row.names = FALSE, tidy = TRUE)
```

Table 8: Different models train and final validation results

Name	Mean_Balanced_Accuracy	Kappa	Accuracy	Time
kknn	0.8526	0.8896	0.9070	183.43
pda	0.9139	0.9075	0.9218	8.991
multinom	0.9139	0.9270	0.9383	148.718
gbm	0.8260	0.8386	0.8638	353.974
xgbTree	0.9066	0.9321	0.9427	2873.427
parRF	0.8742	0.8922	0.9090	599.35
nnet	0.8260	0.8720	0.8919	146.196
multinom_optimal_PCA100	0.9244	0.9427	0.9516	1169.337
multinom_optimal_PCA200	0.9297	0.9597	0.9660	3002.971
multinom_optimal_orig	0.9178	0.9550	0.9620	12214.006
multinom_final_validation	0.9147	0.9138	0.9269	Not applicable



	WALKING_UPSTAIRS	WALKING_DOWNSTAIRS	WALKING	STANDING	STAND_TO_SIT	STAND_TO_LIE	SITTING	SIT_TO_STAND	SIT_TO_LIE	LIE_TO_STAND	LIE_TO_SIT	LAYING
WALKING_UPSTAIRS	439	2	28	1	0	1	0	0	0	0	0	0
WALKING_DOWNSTAIRS	15	395	10	0	0	0	0	0	0	0	0	0
WALKING	0	4	492	0	0	0	0	0	0	0	0	0
STANDING	0	0	0	530	1	0	25	0	0	0	0	0
STAND_TO_SIT	1	0	0	2	15	1	2	1	1	0	0	0
STAND_TO_LIE	2	0	0	0	0	34	2	0	11	0	0	0
SITTING	2	0	0	63	0	0	442	0	0	0	0	1
SIT_TO_STAND	0	0	0	0	0	0	0	10	0	0	0	0
SIT_TO_LIE	0	0	0	0	0	11	0	1	20	0	0	0
LIE_TO_STAND	0	0	0	0	0	3	0	0	0	17	7	0
LIE_TO_SIT	0	0	0	0	0	0	0	0	0	5	20	0
LAYING	0	0	1	27	0	0	0	0	0	0	0	517

Figure 25: Final model validation confusion matrix

Looking on validation metrics and confusion matrix, we can conclude, that model is not overfitted and, generally, have good performance. But distinguishing between some classes (especially static positions) is not completely reliable.

6 Conclusion

In the capstone project I examine HAPT dataset and build a multyclass classification model which able to predict different activities and postural transitions with average accuracy 91.5%. Some classes are distinguished from each other with very high reliability while some of them can be mixed with each other. During EDA it was concluded, that it is a nature of the data: smartphone sensors generate almost identical information when person is standing still, sitting or laying. Although, some potential improvements can be done: - Consider more complicated algorithms with more tunable parameters, e.g. neural networks, which can find subtitle, non-linear difference in high dimensions; - Convert multiclass classification task to number of binary classification tasks (1-vs-all or each-vs-each) and then use different models with different parameters for each of binary classification; - collect more data using more modern devices (remember, that original dataset is from 2012): smartphones and smartwatches should have better sensors since then; - use data with timestamps, and then previous activity can be used as one of predictors for current activity.

7 Literature

1. Jorge-L. Reyes-Ortiz, Luca Oneto, Albert Samà, Xavier Parra, Davide Anguita. Transition-Aware Human Activity Recognition Using Smartphones. 2015
2. Rafael A. Irizarry, Introduction to Data Science
3. Bex T., Comprehensive Guide to Multiclass Classification Metrics
4. Max Kuhn, The caret Package
5. Jolliffe, I. T. (2002). Principal Component Analysis
6. N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer (2002)
7. N.S. Altman. An Introduction to Kernel and Nearest Neighbor Nonparametric Regression (1991)
8. Trevor Hastie, Andreas Buja, Robert Tibshirani. Penalized Discriminant Analysis (1995)
9. Johan de Rooi. Penalized Estimation in High-Dimensional Data Analysis (2013)
10. Menard, Scott W. Applied logistic regression analysis (2002)
11. Jerome H.Friedman. Stochastic Gradient Boosting (1999)
12. Tianqi Chen, Tong He. xgboost: eXtreme Gradient Boosting (2020)
13. N Azizah et al 2019 J. Phys.: Conf. Ser. 1280 022028. Implementation of random forest algorithm with parallel computing in R
14. Margherita Grandini, Enrico Bagli, Giorgio Visani. Metrics for multi-class classification. (2020)
15. Xavier Robin, Natacha Turck, Alexandre Hainard, Natalia Tiberti, Frédérique Lisacek, Jean-Charles Sanchez and Markus Müller (2011). pROC: an open-source package for R and S+ to analyze and compare ROC curves. BMC Bioinformatics, 12, p. 77. DOI: 10.1186/1471-2105-12-77