

Parkinson's disease analysis

Rana M. Khalil

2024-06-06

Contents

Preface	1
1 Preprocessing	2
1.1 Filtering	2
1.2 Segmentation	3
2 Feature engineering	18
2.1 Define time-domain and frequency-domain features	19
2.2 Build features table	25
2.3 Feature reduction	36
3 Train/ Test split	43
3.1 PD and parkinsonism participants	43
3.2 Mild PD and parkinsonism participants	46
3.3 Moderate PD and parkinsonism participants	48
3.4 Severe PD and parkinsonism participants	50
4 Machine learning model	54
4.1 PD and parkinsonism participants	55
4.2 Mild PD and parkinsonism participants	55
4.3 Moderate PD and parkinsonism participants	55
4.4 Severe PD and parkinsonism participants	56
5 Performance evaluation	57
5.1 PD and parkinsonism participants	57
5.2 Mild PD and parkinsonism participants	59
5.3 Moderate PD and parkinsonism participants	60
5.4 Severe PD and parkinsonism participants	61

Preface

This notebook provides the code of the analysis described in the manuscript “*Machine learning analysis of wearable sensor data from mobility testing distinguishes Parkinson’s disease from other forms of parkinsonism*”. The work involves using machine learning and movement data to distinguish idiopathic PD from non-PD parkinsonism. Wearable sensor data were collected from a cohort of 260 individuals diagnosed with PD and 18 participants who were diagnosed with other forms of parkinsonism. Each participant performed five motor tasks, including a 32-foot walk involving walking back and forth four times with 180 degree turns between segments, standing with eyes open, standing with eyes closed, two trials of the Timed Up & Go test (TUG), and two trials of the cognitive TUG (cogTUG). Besides sensor-derived features, various non-sensor features including demographics and clinical evaluation scores were added to the feature set. Prior to constructing the classifiers, we employed a forward feature selection approach to reduce concerns of overfitting by reducing the number of features. Then, we randomly divided our data into three groups using stratified sampling. Two groups were used for training and the remaining group for testing. This three-fold cross-validation process was repeated five times with different seeds and the final predicted class for each participant was determined by majority vote from the predicted classes across the five repeats. For each training set, we constructed a random forest model following the random under-sampling of the PD group to match the size of the non-PD parkinsonism group.

Chapter 1

Preprocessing

```
library(stats)
library(ggplot2)
library(reshape2)
library(signal)
library(gridExtra)
library(pracma)
library(forecast)
library(psych)
library(lme4)

sampling_rate = 100
T = 1/sampling_rate # Sampling period
tried_pos = FALSE
titles = c("ax", "ay", "az", "gx", "gy", "gz")
```

1.1 Filtering

```
# Define Butterworth filter
butter_filter <- function(signal, order, cutoff) {
  bf <- butter(order, cutoff/sampling_rate, type = "low")
  cleaned <- filtfilt(bf, signal)

  return(cleaned)
}

# Filter 6 channel of a subject
```

```

filter <- function(sub) {
  filtered_sub = data.frame(matrix(nrow = nrow(sub), ncol = length(sub)))
  # Loop over channels
  for (j in 1:6) {
    ch_sub = as.vector(scale(sub[, j]))
    # filter with 4th order Butterworth filter
    ch_sub = as.vector(butter_filter(ch_sub, order = 4, cutoff = 20))
    filtered_sub[, j] = ch_sub
  }
  return(filtered_sub)
}

```

1.2 Segmentation

```

# Trim constant parts of a subject from all channels
trim <- function(sub, t, a_const_thr, g_const_thr) {
  min_start_const_len = 10000
  min_end_const_len = 10000
  for (i in 1:6) {
    if (i < 4)
      const_thr = a_const_thr else const_thr = g_const_thr

    ch_sub = sub[, i]
    df_acc <- data.frame(t, ch_sub)
    names(df_acc) <- c("t", titles[i])
    # calculate slope between successive points
    slope = c(0, diff(df_acc[, 2])/diff(df_acc[, 1]))
    labels = rep(TRUE, length(t))
    # label parts with slope as not constant
    labels[abs(slope) > const_thr] = FALSE
    constant <- data.frame(matrix(ncol = 2, nrow = length(t)))
    colnames(constant) = c("slope", "label")
    constant$slope = slope
    constant$label = labels

    lengths = rle(constant$label)
    true_lengths = lengths$lengths * lengths$values
    # find constant parts at the beginning and end of a
    # signal
    start_const_length = true_lengths[1]
    end_const_length = true_lengths[length(true_lengths)]
  }
}

```

```

    if (start_const_length < min_start_const_len)
      min_start_const_len = start_const_length
    if (end_const_length < min_end_const_len)
      min_end_const_len = end_const_length
  }
  return(c(start = min_start_const_len, end = min_end_const_len))
}

```

1.2.1 Timed Up & Go (TUG) segmentation

```

# Segment sit to stand for a subject
find_sit_to_stand <- function(sub, t) {
  ch_indx = 3 # az channel
  ch_sub = sub[, ch_indx]
  # apply trapezoidal integration
  ch_sub_pos = as.vector(cumtrapz(t, ch_sub))
  # find peaks
  max_peaks = pracma::findpeaks(ch_sub_pos)
  max_peak = t(max_peaks[2, ])
  return(c(1, max_peak[1, 4]))
}

# Segment stand to sit for a subject
find_stand_to_sit <- function(sub, t) {
  ch_indx = 3 # az channel
  ch_sub = sub[, ch_indx]
  # apply trapezoidal integration
  ch_sub_pos = as.vector(cumtrapz(t, ch_sub))
  # find valleys
  min_peak = -pracma::findpeaks(-ch_sub_pos)
  return(c(-min_peak[nrow(min_peak) - 1, 3], length(t)))
}

# Segment two turns for a subject
find_turns <- function(sub, t, smoothing_order) {
  ch_indx = 4 # gx channels
  ch_sub = sub[, ch_indx]
  # apply trapezoidal integration
  ch_sub_pos = as.vector(cumtrapz(t, ch_sub))
  # smooth the integrated line to avoid unnecessary parts

```

```

# with confusing slopes
ch_sub_pos_filter = as.vector(ma(ch_sub_pos, order = smoothing_order))
ch_sub_pos_filter[is.na(ch_sub_pos_filter)] = 0
df_acc_pos_filter <- data.frame(t, ch_sub_pos_filter)
names(df_acc_pos_filter) <- c("t", titles[ch_indx])
# calculate slope between successive points
slope = c(0, diff(df_acc_pos_filter[, 2])/diff(df_acc_pos_filter[,
  1]))
labels = rep(NA, length(t))
# find segments with slopes
labels[slope > 0.45] = TRUE
labels[slope < -0.45] = FALSE
constant <- data.frame(matrix(ncol = 2, nrow = length(t)))
colnames(constant) = c("slope", "label")
constant$slope = slope
constant$label = labels
lengths = rle(constant$label)
all_pos_slope_lengths = lengths$lengths * lengths$values
all_pos_slope_lengths = all_pos_slope_lengths[!is.na(all_pos_slope_lengths)]
all_neg_slope_lengths = lengths$lengths * !lengths$values
all_neg_slope_lengths = all_neg_slope_lengths[!is.na(all_neg_slope_lengths)]

min_turn_length = sampling_rate
two_pos_turns = NA

# check if two turns are in opposite directions
if ((length(all_pos_slope_lengths[all_pos_slope_lengths >
  min_turn_length]) == 1) && (length(all_neg_slope_lengths[all_neg_slope_lengths >
  min_turn_length]) == 1)) {
  # turns are two largest segments with positive and
  # negative slopes
  turn1_slope_length = max(all_pos_slope_lengths)
  turn2_slope_length = max(all_neg_slope_lengths)
  max_pos_slope_length = turn1_slope_length
  max_neg_slope_length = turn2_slope_length
} else {
  # turns are in same directions
  max_pos_slope_length = max(all_pos_slope_lengths)
  second_max_pos_slope_length = max(all_pos_slope_lengths[all_pos_slope_lengths !=
    max_pos_slope_length])
  max_neg_slope_length = max(all_neg_slope_lengths)
  second_max_neg_slope_length = max(all_neg_slope_lengths[all_neg_slope_lengths !=

```

```

max_neg_slope_length])

# check if two turns have negative slopes
if (max_neg_slope_length > min_turn_length && second_max_neg_slope_length >
    min_turn_length) {
  tried_pos <- FALSE
  two_pos_turns = FALSE
  # turns are the largest two segments with
  # negative slopes
  turn1_slope_length = max_neg_slope_length
  if (length(grep(turn1_slope_length, all_neg_slope_lengths)) ==
      1) {
    turn2_slope_length = second_max_neg_slope_length
  } else {
    turn2_slope_length = max_neg_slope_length
  }
} else {
  # two turns have positive slopes
  tried_pos <- TRUE
  two_pos_turns = TRUE
  # turns are the largest two segments with
  # positive slopes
  turn1_slope_length = max_pos_slope_length
  if (length(grep(turn1_slope_length, all_pos_slope_lengths)) ==
      1) {
    turn2_slope_length = second_max_pos_slope_length
  } else {
    turn2_slope_length = max_pos_slope_length
  }
}
}

# find the start and end of the two turns
if (is.na(two_pos_turns)) {
  before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
    lengths$lengths)[1] - 1)])
} else if (two_pos_turns == TRUE) {
  before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
    lengths$lengths * lengths$values)[1] - 1)])
} else {
  before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
    lengths$lengths * !lengths$values)[1] - 1)])
}

```



```

}
if (turn2_slope_length != max_pos_slope_length) {
  if (length(grep(turn2_slope_length, lengths$lengths)) ==
      1) {
    turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
  } else {
    if (is.na(two_pos_turns)) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
      before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
        1)])
      if (before_turn1_slope_lengths == before_turn2_slope_lengths) {
        turn2_indx = grep(turn2_slope_length, lengths$lengths)[2]
      }
    } else if (two_pos_turns == TRUE) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        lengths$values)[1]
    } else {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        !lengths$values)[1]
    }
  }
}
before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
  1)])
} else {
  if (length(grep(turn2_slope_length, lengths$lengths)) ==
      1)
    turn2_indx = grep(turn2_slope_length, lengths$lengths)[1] else {
    if (is.na(two_pos_turns)) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
      before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
        1)])
      if (before_turn1_slope_lengths == before_turn2_slope_lengths) {
        turn2_indx = grep(turn2_slope_length, lengths$lengths)[2]
      }
    } else if (two_pos_turns == TRUE) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        lengths$values)[1]
    } else {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        !lengths$values)[1]
    }
  }
}
}

```

```

        before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
            1)])
    }
    return(c(before_turn1_slope_lengths + 1, before_turn1_slope_lengths +
        turn1_slope_length, before_turn2_slope_lengths + 1, before_turn2_slope_lengths +
        turn2_slope_length))
}

# Segment two turns for a subject (second try)
find_turns_swap <- function(sub, t, smoothing_order) {
    ch_indx = 4 # gx channel
    ch_sub = sub[, ch_indx]
    # apply trapezoidal integration
    ch_sub_pos = as.vector(cumtrapz(t, ch_sub))
    # smooth the integrated line to avoid unnecessary parts
    # with confusing slopes
    ch_sub_pos_filter = as.vector(ma(ch_sub_pos, order = smoothing_order))
    ch_sub_pos_filter[is.na(ch_sub_pos_filter)] = 0
    df_acc_pos_filter <- data.frame(t, ch_sub_pos_filter)
    names(df_acc_pos_filter) <- c("t", titles[ch_indx])
    # calculate slope between successive points
    slope = c(0, diff(df_acc_pos_filter[, 2])/diff(df_acc_pos_filter[,
        1]))
    labels = rep(NA, length(t))
    # find segments with slopes
    labels[slope > 0.45] = TRUE
    labels[slope < -0.45] = FALSE
    constant <- data.frame(matrix(ncol = 2, nrow = length(t)))
    colnames(constant) = c("slope", "label")
    constant$slope = slope
    constant$label = labels
    lengths = rle(constant$label)
    all_pos_slope_lengths = lengths$lengths * lengths$values
    all_pos_slope_lengths = all_pos_slope_lengths[!is.na(all_pos_slope_lengths)]
    all_neg_slope_lengths = lengths$lengths * !lengths$values
    all_neg_slope_lengths = all_neg_slope_lengths[!is.na(all_neg_slope_lengths)]

    min_turn_length = sampling_rate
    two_pos_turns = NA

    max_pos_slope_length = max(all_pos_slope_lengths)
    second_max_pos_slope_length = max(all_pos_slope_lengths[all_pos_slope_lengths !=

```

```

    max_pos_slope_length])
max_neg_slope_length = max(all_neg_slope_lengths)
second_max_neg_slope_length = max(all_neg_slope_lengths[all_neg_slope_lengths !=
    max_neg_slope_length])

# if finding two turns with positive slopes failed then
# find two turns with negative slopes
if (tried_pos == TRUE) {
    two_pos_turns = FALSE
    # turns are the largest two segments with negative
    # slopes
    turn1_slope_length = max_neg_slope_length
    if (length(grep(turn1_slope_length, all_neg_slope_lengths)) ==
        1) {
        turn2_slope_length = second_max_neg_slope_length
    } else {
        turn2_slope_length = max_neg_slope_length
    }
} else {
    # finding two turns with negative slopes failed
    # then find two turns with positive slopes
    two_pos_turns = TRUE
    # turns are the largest two segments with positive
    # slopes
    turn1_slope_length = max_pos_slope_length
    if (length(grep(turn1_slope_length, all_pos_slope_lengths)) ==
        1) {
        turn2_slope_length = second_max_pos_slope_length
    } else {
        turn2_slope_length = max_pos_slope_length
    }
}

# Find the start and end of the two turns
if (is.na(two_pos_turns)) {
    before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
        lengths$lengths)[1] - 1)])
} else if (two_pos_turns == TRUE) {
    before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
        lengths$lengths * lengths$values)[1] - 1)])
} else {
    before_turn1_slope_lengths = sum(lengths$lengths[1:(grep(turn1_slope_length,
        lengths$lengths * !lengths$values)[1] - 1)])
}

```

```

}
if (turn2_slope_length != max_pos_slope_length) {
  if (length(grep(turn2_slope_length, lengths$lengths)) ==
      1) {
    turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
  } else {
    if (is.na(two_pos_turns)) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
      before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
        1)])
      if (before_turn1_slope_lengths == before_turn2_slope_lengths) {
        turn2_indx = grep(turn2_slope_length, lengths$lengths)[2]
      }
    } else if (two_pos_turns == TRUE) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        lengths$values)[1]
    } else {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        !lengths$values)[1]
    }
  }
  before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
    1)])
} else {
  if (length(grep(turn2_slope_length, lengths$lengths)) ==
      1) {
    turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
  } else {
    if (is.na(two_pos_turns)) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths)[1]
      before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
        1)])
      if (before_turn1_slope_lengths == before_turn2_slope_lengths) {
        turn2_indx = grep(turn2_slope_length, lengths$lengths)[2]
      }
    } else if (two_pos_turns == TRUE) {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        lengths$values)[1]
    } else {
      turn2_indx = grep(turn2_slope_length, lengths$lengths *
        !lengths$values)[1]
    }
  }
}

```

```

    }
    before_turn2_slope_lengths = sum(lengths$lengths[1:(turn2_indx -
      1)])
  }
  return(c(before_turn1_slope_lengths + 1, before_turn1_slope_lengths +
    turn1_slope_length, before_turn2_slope_lengths + 1, before_turn2_slope_lengths +
    turn2_slope_length))
}

# Write segmented parts to files
write_segments_to_files <- function(TUG_file, smoothing_order) {
  # read file with 6 channels of a subject
  sub = read.table(TUG_file)
  L = nrow(sub)
  t = (0:(L - 1)) * T
  # trim constant parts
  trim_sub_lengths = trim(sub, t, a_const_thr = 5, g_const_thr = 450)
  start_const_length = trim_sub_lengths[1]
  end_const_length = trim_sub_lengths[2]
  sub = sub[((start_const_length + 1):(length(t) - end_const_length)),
    ]
  t = t[((start_const_length + 1):(length(t) - end_const_length))]
  # filter channels
  sub = filter(sub)
  # start segmentation
  is_walking = rep(TRUE, length(t))
  # get sit to stand part
  range_sit_to_stand = find_sit_to_stand(sub, t)
  is_walking[range_sit_to_stand[1]:range_sit_to_stand[2]] = FALSE
  # get stand to sit part
  range_stand_to_sit = find_stand_to_sit(sub, t)
  is_walking[range_stand_to_sit[1]:range_stand_to_sit[2]] = FALSE
  # get turning part
  range_turns = find_turns(sub, t, smoothing_order)
  is_walking[range_turns[1]:range_turns[2]] = FALSE
  is_walking[range_turns[3]:range_turns[4]] = FALSE
  # get walking part (segments not labeled as
  # sit-to-stand, stand-to-sit, and turns)
  lengths = rle(is_walking)
  walking_lengths = lengths$lengths * lengths$values
  not_walking_lengths = lengths$lengths * !lengths$values
  range_walks = c(not_walking_lengths[1] + 1, not_walking_lengths[1] +

```

```

walking_lengths[2] + 1, sum(lengths$lengths[1:3]) + 1,
sum(lengths$lengths[1:3]) + walking_lengths[4] + 1)

# if error with segmentation is found, try finding
# other turns
if ((is.na(range_walks[3]) || is.na(range_walks[4]))) {
  is_walking = rep(TRUE, length(t))
  is_walking[range_sit_to_stand[1]:range_sit_to_stand[2]] = FALSE
  is_walking[range_stand_to_sit[1]:range_stand_to_sit[2]] = FALSE
  # reextract turning part
  range_turns = find_turns_swap(sub, t, smoothing_order)
  is_walking[range_turns[1]:range_turns[2]] = FALSE
  is_walking[range_turns[3]:range_turns[4]] = FALSE
  # reextract walking part
  lengths = rle(is_walking)
  walking_lengths = lengths$lengths * lengths$values
  not_walking_lengths = lengths$lengths * !lengths$values
  range_walks = c(not_walking_lengths[1] + 1, not_walking_lengths[1] +
    walking_lengths[2] + 1, sum(lengths$lengths[1:3]) +
    1, sum(lengths$lengths[1:3]) + walking_lengths[4] +
    1)
  # if error with segmentation is still found, try
  # finding turns in another direction
  if ((is.na(range_walks[3]) || is.na(range_walks[4]))) {
    is_walking = rep(TRUE, length(t))
    is_walking[range_sit_to_stand[1]:range_sit_to_stand[2]] = FALSE
    is_walking[range_stand_to_sit[1]:range_stand_to_sit[2]] = FALSE
    # reextract turning parts
    tried_pos <- !tried_pos
    range_turns = find_turns_swap(sub, t, smoothing_order)
    t_turn1 = t[range_turns[1]:range_turns[2]]
    t_turn2 = t[range_turns[3]:range_turns[4]]
    is_walking[range_turns[1]:range_turns[2]] = FALSE
    is_walking[range_turns[3]:range_turns[4]] = FALSE
    # reextract walking parts
    lengths = rle(is_walking)
    walking_lengths = lengths$lengths * lengths$values
    not_walking_lengths = lengths$lengths * !lengths$values
    range_walks = c(not_walking_lengths[1] + 1, not_walking_lengths[1] +
      walking_lengths[2] + 1, sum(lengths$lengths[1:3]) +
      1, sum(lengths$lengths[1:3]) + walking_lengths[4] +
      1)
  }
}

```

```

    }
}

min_turn_sit_length = sampling_rate + 400
turn2_end = max(range_turns[2], range_turns[4])

# check if 2nd turn is far from stand to sit, then
# search for other turns
if (((range_stand_to_sit[1] - turn2_end) > min_turn_sit_length) ||
    ((turn2_end - range_stand_to_sit[1]) > 650)) {
  is_walking = rep(TRUE, length(t))
  is_walking[range_sit_to_stand[1]:range_sit_to_stand[2]] = FALSE
  is_walking[range_stand_to_sit[1]:range_stand_to_sit[2]] = FALSE
  # reextract turning parts
  range_turns = find_turns_swap(sub, t, smoothing_order)
  t_turn1 = t[range_turns[1]:range_turns[2]]
  t_turn2 = t[range_turns[3]:range_turns[4]]
  is_walking[range_turns[1]:range_turns[2]] = FALSE
  is_walking[range_turns[3]:range_turns[4]] = FALSE
  # reextract walking parts
  lengths = rle(is_walking)
  walking_lengths = lengths$lengths * lengths$values
  not_walking_lengths = lengths$lengths * !lengths$values
  range_walks = c(not_walking_lengths[1] + 1, not_walking_lengths[1] +
    walking_lengths[2] + 1, sum(lengths$lengths[1:3]) +
    1, sum(lengths$lengths[1:3]) + walking_lengths[4] +
    1)
  turn2_end = max(range_turns[2], range_turns[4])

  # if 2nd turn is still far from stand to sit,
  # search for other turns
  if (((range_stand_to_sit[1] - turn2_end) > min_turn_sit_length) ||
      ((turn2_end - range_stand_to_sit[1]) > 650)) {
    is_walking = rep(TRUE, length(t))
    is_walking[range_sit_to_stand[1]:range_sit_to_stand[2]] = FALSE
    is_walking[range_stand_to_sit[1]:range_stand_to_sit[2]] = FALSE
    # reextract turning parts
    tried_pos <- !tried_pos
    range_turns = find_turns_swap(sub, t, smoothing_order)
    t_turn1 = t[range_turns[1]:range_turns[2]]
    t_turn2 = t[range_turns[3]:range_turns[4]]
    is_walking[range_turns[1]:range_turns[2]] = FALSE

```

```

        is_walking[range_turns[3]:range_turns[4]] = FALSE
        # reextract walking parts
        lengths = rle(is_walking)
        walking_lengths = lengths$lengths * lengths$values
        not_walking_lengths = lengths$lengths * !lengths$values
        range_walks = c(not_walking_lengths[1] + 1, not_walking_lengths[1] +
            walking_lengths[2] + 1, sum(lengths$lengths[1:3]) +
            1, sum(lengths$lengths[1:3]) + walking_lengths[4] +
            1)
    }
}
# swap turns labeling if 1st turn starts after 2nd turn
if (range_turns[1] > range_turns[3]) {
    holder = c(range_turns[1], range_turns[2])
    range_turns[1] = range_turns[3]
    range_turns[2] = range_turns[4]
    range_turns[3] = holder[1]
    range_turns[4] = holder[2]
    t_turn1 = t[range_turns[1]:range_turns[2]]
    t_turn2 = t[range_turns[3]:range_turns[4]]
}
# divide channels into corresponding segments
sub_sit_to_stand = sub[range_sit_to_stand[1]:range_sit_to_stand[2],
    ]
sub_stand_to_sit = sub[range_stand_to_sit[1]:range_stand_to_sit[2],
    ]
sub_turn1 = sub[range_turns[1]:range_turns[2], ]
sub_turn2 = sub[range_turns[3]:range_turns[4], ]
sub_walk1 = sub[range_walks[1]:range_walks[2], ]
sub_walk2 = sub[range_walks[3]:range_walks[4], ]
# write segmented components to files
name_ext = "sTs_1.txt"
write.table(sub_sit_to_stand, file = name_ext, sep = "\t",
    col.names = FALSE, row.names = FALSE)
name_ext = "sTs_2.txt"
write.table(sub_stand_to_sit, file = name_ext, sep = "\t",
    col.names = FALSE, row.names = FALSE)
name_ext = "turn_1.txt"
write.table(sub_turn1, file = name_ext, sep = "\t", col.names = FALSE,
    row.names = FALSE)
name_ext = "turn_2.txt"
write.table(sub_turn2, file = name_ext, sep = "\t", col.names = FALSE,

```



```

    row.names = FALSE)
name_ext = "walk_1.txt"
write.table(sub_walk1, file = name_ext, sep = "\t", col.names = FALSE,
    row.names = FALSE)
name_ext = "walk_2.txt"
write.table(sub_walk2, file = name_ext, sep = "\t", col.names = FALSE,
    row.names = FALSE)
}

```

1.2.2 32-feet walk segmentation

```

# Segment and write segmented parts to files
write_segments_to_files <- function(walk_file, smoothing_order) {
  # read task file with 6 channels
  sub = read.table(walk_file)
  L = nrow(sub)
  t = (0:(L - 1)) * T
  # trim constant parts
  trim_sub_lengths = trim(sub, t, a_const_thr = 5, g_const_thr = 450)
  start_const_length = trim_sub_lengths[1]
  end_const_length = trim_sub_lengths[2]
  sub = sub[((start_const_length + 1):(length(t) - end_const_length)),
    ]
  t = t[((start_const_length + 1):(length(t) - end_const_length))]
  # filter channels
  sub = filter(sub)
  # find three turns
  j = 4 # gx channel
  ch_sub = sub[, j]
  # apply trapezoidal integration
  ch_sub_pos = as.vector(cumtrapz(t, ch_sub))
  # smooth the integrated line to avoid unnecessary parts
  # with confusing slopes
  ch_sub_pos_filter = as.vector(ma(ch_sub_pos, order = smoothing_order))
  ch_sub_pos_filter[is.na(ch_sub_pos_filter)] = 0
  df_acc <- data.frame(t, ch_sub)
  names(df_acc) <- c("t", titles[j])
  df_acc_pos_filter <- data.frame(t, ch_sub_pos_filter)
  names(df_acc_pos_filter) <- c("t", titles[j])
  # calculate slope between successive points
  slope = c(0, diff(df_acc_pos_filter[, 2])/diff(df_acc_pos_filter[,

```

```

1]))
labels = rep(NA, length(t))
# find segments with slopes
labels[abs(slope) > 0.45] = TRUE
constant <- data.frame(matrix(ncol = 2, nrow = length(t)))
colnames(constant) = c("slope", "label")
constant$slope = slope
constant$label = labels
lengths = rle(constant$label)
all_slope_lengths = lengths$lengths * lengths$values
all_slope_lengths = all_slope_lengths[!is.na(all_slope_lengths)]
# turns are the largest three segments with slopes
max_three_slopes = sort(all_slope_lengths, decreasing = TRUE)[1:3]
max_three_slopes_indx = which(lengths$lengths %in% max_three_slopes)
max_three_slopes = lengths$lengths[max_three_slopes_indx]
# walks are the parts not labeled as turns
turn1_start = sum(lengths$lengths[1:(grep(max_three_slopes[1],
lengths$lengths) - 1)]) + 1
turn2_start = sum(lengths$lengths[1:(grep(max_three_slopes[2],
lengths$lengths) - 1)]) + 1
turn3_start = sum(lengths$lengths[1:(grep(max_three_slopes[3],
lengths$lengths) - 1)]) + 1
walk1_start = 1
walk1_end = turn1_start - 1
turn1_end = turn1_start + max_three_slopes[1] - 1
walk2_start = turn1_end + 1
walk2_end = turn2_start - 1
turn2_end = turn2_start + max_three_slopes[2] - 1
walk3_start = turn2_end + 1
walk3_end = turn3_start - 1
turn3_end = turn3_start + max_three_slopes[3] - 1
walk4_start = turn3_end + 1
walk4_end = length(ch_sub)
# divide channels into its corresponding segments
sub_turn1 = sub[turn1_start:turn1_end, ]
sub_turn2 = sub[turn2_start:turn2_end, ]
sub_turn3 = sub[turn3_start:turn3_end, ]
sub_walk1 = sub[walk1_start:walk1_end, ]
sub_walk2 = sub[walk2_start:walk2_end, ]
sub_walk3 = sub[walk3_start:walk3_end, ]
sub_walk4 = sub[walk4_start:walk4_end, ]
# write segmented components to files

```

```
name_ext = "turn_1.txt"
write.table(sub_turn1, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "turn_2.txt"
write.table(sub_turn2, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "turn_3.txt"
write.table(sub_turn3, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "_walk_1.txt"
write.table(sub_walk1, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "walk_2.txt"
write.table(sub_walk2, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "walk_3.txt"
write.table(sub_walk3, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
name_ext = "walk_4.txt"
write.table(sub_walk4, file = name_ext, sep = "\t", col.names = FALSE,
            row.names = FALSE)
}
```

Chapter 2

Feature engineering

```
library(lomb)
library(infotheo)
library(moments)
library(signal)
library(pracma)
library(seewave)
library(e1071)
library(Metrics)
library(nonlinearTseries)
library(musclesyneRgies)
library(fractaldim)
library(Rdimtools)
library(readxl)
library(psych)
library(randomForest)
library(stringr)

sampling_rate = 100
max_freq = sampling_rate/2
num_pairs = 6
fft_peaks = 10 # fft or lsp
frequency_features = 24
time_features = 47
cross_features = 4
fft_features = fft_peaks * 2 # for fft and lsp
num_channels = 6
features_per_channel = fft_features + frequency_features + time_features
```

```

features_per_task = num_channels * features_per_channel + num_pairs *
    cross_features
num_sub_tasks_TUGs = 6
num_sub_tasks_walk = 7
num_cases = 300

```

2.1 Define time-domain and frequency-domain features

```

# Get top 10 FFT peaks
fourier_peaks <- function(signal) {
  sig_fft <- fft(signal)
  L = length(signal)
  amplitude <- abs(sig_fft/L)[1:(L/2 + 1)]
  frequencies <- sampling_rate * (0:(L/2))/L

  sorted <- sort.int(amplitude, decreasing = TRUE, index.return = TRUE)
  top <- sorted$ix[1:fft_peaks] # indexes of the largest n components
  return(frequencies[top]) # convert indexes to frequencies
}

# Get top 10 LSP peaks
lsp_peaks <- function(signal) {
  X.k <- lsp(signal, plot = FALSE)
  power = X.k$power
  frequencies = X.k$scanned * 100

  sorted <- sort.int(power, decreasing = TRUE, index.return = TRUE)
  top <- sorted$ix[1:fft_peaks] # indexes of the largest n components
  return(frequencies[top]) # convert indexes to frequencies
}

# Transform to frequency domain
to_fft <- function(signal) {
  sig_fft <- fft(signal)
  L = length(signal)
  sig_fft = abs(sig_fft/L)[1:(L/2 + 1)]
  f = sampling_rate * (0:(L/2))/L
  return(cbind(f, sig_fft))
}

# Calculate frequency domain features
frequency_measures <- function(signal) {
  # return frequencies and power amplitudes
  signal_to_freq = to_fft(signal)

```

```

freq = signal_to_freq[, 1]
ampl = signal_to_freq[, 2]
# mean frequency
MNF = sum(freq * ampl)/sum(ampl)
# median frequency
median_ampl = median(ampl)
MDF = freq[which.min(abs(ampl - median_ampl))]
# maximum to minimum drop in power density
N = 13
dpr_filter = rep(1/N, N)
mean_psd = na.omit(stats::filter(ampl, dpr_filter, sides = 2))
DPR = max(mean_psd)/min(mean_psd)
# signal to noise ratio
N = round(length(freq)/5)
index_f_upper = (length(freq) - N + 1):length(freq)
N = length(index_f_upper)
noise_power = sum(ampl[index_f_upper])/N * length(ampl)
total_power = sum(ampl)
SNR = total_power/noise_power
# power spectrum deformation
M0 = sum(ampl)
M1 = sum(ampl * freq)
M2 = sum(ampl * freq^2)
PSD = sqrt(M2/M0)/(M1/M0)
# freeze index
j_half = which.min(abs(freq - 0.5))
j_three = which.min(abs(freq - 3))
j_eight = which.min(abs(freq - 8))
FI = sum(ampl[j_three:j_eight])/sum(ampl[j_half:j_three])
# entropy
prob = ampl/sum(ampl)
ENT = -sum(prob * log(prob))
# total power
TTP = sum(ampl)
# mean power
MNP = sum(ampl)/length(ampl)
# peak frequency
PKF = freq[which.max(ampl)]
# peak Power
PKP = max(ampl)
# frequency Ratio
non_zero_ampl = (ampl >= 0.001)

```

```

last_non_zero_indx = length(ampl) - match(TRUE, rev(non_zero_ampl)) +
  1
max_freq = freq[last_non_zero_indx]
min_freq = freq[2]
FR = max_freq/min_freq
# power spectrum ratio
max_indx = which.max(ampl)
n = ifelse(max_indx > 10, 10, max_indx - 1)
n = ifelse(n > 1, n, 0)
PSR = sum(ampl[max_indx - n:max_indx + n])/sum(ampl)
# spectral moments
SM1 = sum(ampl * freq)
SM2 = sum(ampl * freq^2)
SM3 = sum(ampl * freq^3)
# variance
VR = var(ampl)
# standard deviation
SD = sd(ampl)
# skewness
SS = skewness(ampl)
# kurtosis
SK = kurtosis(ampl)
# spectral bandwidth
SBW = sum((freq - MNF)^2 * ampl)/sum(ampl)
# spectral roll-off
non_zero_ampl = ampl[ampl > 0]
SR = 0.95 * sum(non_zero_ampl)
# variance of central frequency
SM0 = sum(ampl)
VCF = (SM2/SM0) - (SM1/SM0)^2
# mean spectral energy
MSE = mean((abs(ampl))^2)
return(c(MNF, MDF, DPR, SNR, PSD, FI, ENT, TTP, MNP, PKF,
  PKP, FR, PSR, SM1, SM2, SM3, VR, SD, SS, SK, SBW, SR,
  VCF, MSE))
}

# Calculate time domain features
time_measures <- function(signal) {
  signal = scale(signal, center = TRUE, scale = FALSE)
  N = length(signal)
  # mean
  MN = mean(signal)

```

```

# variance
VR = var(signal)
# standard deviation
SD = sd(signal)
# skewness
SS = skewness(signal)
# kurtosis
SK = kurtosis(signal)
# integrated absolute value
IAV = sum(abs(signal))
# mean absolute value
MAV = mean(abs(signal))
# simple square interval
SSI = sum((abs(signal))^2)
# root mean square
RMS = sqrt(mean(signal^2))
# V-order 2 and 3
V2 = (mean(signal^2))^(1/2)
V3 = (mean((abs(signal))^3))^(1/3)
# waveform length
WL = sum(abs(diff(signal)))
# average amplitude change
AAC = mean(abs(diff(signal)))
# difference absolute standard deviation value
DASDV = sqrt((sum((diff(signal))^2))/(N - 1))
# maximum fractal length
MFL = log(sqrt(sum((diff(signal))^2)))
# zero crossing
ZC = sum(diff(sign(signal)) != 0)
# rate of change
RC = sum(abs(diff(signal)) >= 0)
# slope sign change
SSC = sum((diff(signal)[1:(N - 2)] * diff(signal[2:N]) *
-1) >= 0)
# data range
DR = max(signal) - min(signal)
# entropy
ENT = entropy(entropy::discretize(signal, numBins = 10))
# log detector
LOG = exp(mean(log(abs(signal))))
# mean absolute deviation
MAD = mean(abs(signal - MN))

```



```

# percentiles
Qs = quantile(signal, names = FALSE)
Q1 = Qs[2]
Q2 = Qs[3]
Q3 = Qs[4]
# inter-quartile range
IQR = Q3 - Q1
# CV
CV = (Q3 - Q1)/Q2
# median
MED = median(signal)
# mode
MOD <- Mode(signal)
# Teager-Kaiser energy operator
TKEO = mean(TKEO(signal, f = sampling_rate, plot = F)[, 2],
            na.rm = T)
# auto-regressive coefficients
ar_model = ar(signal, aic = FALSE, order.max = 4)
ar_coeff = ar_model$ar
AR_1 = ar_coeff[1]
AR_2 = ar_coeff[2]
AR_3 = ar_coeff[3]
AR_4 = ar_coeff[4]
# box-counting dimension
box_dim = est.boxcount(signal)$estdim
# detrended fluctuation Analysis
dfa_model = dfa(time.series = signal, do.plot = FALSE)
DFA = estimate(dfa_model, do.plot = FALSE)
# Higuchi's fractal dimension
HFD = HFD(signal)$Higuchi
# Katz's fractal dimension
T = 1/sampling_rate
t = (0:(length(signal) - 1)) * T
L = sum(sqrt(diff(t)^2 + diff(signal)^2))
a = mean(sqrt(diff(t)^2 + diff(signal)^2))
d = max(sapply(signal, function(x) sqrt((x - signal[1])^2 +
T^2)))
KATZ = log(L/a)/log(d/a)
# modified mean absolute value type 1
N = length(signal)
w_i = rep(0.5, N)
w_i[(0.25 * N):(0.75 * N)] = 1

```

```

MAV1 = mean(abs(w_i * signal))
# modified mean absolute value type 2
N = length(signal)
w_i = vector()
for (i in 1:N) {
  if (i >= 0.25 * N && i <= 0.75 * N) {
    w = 1
  } else if (i < 0.25 * N) {
    w = 4 * i/N
  } else {
    w = 4 * (i - N)/N
  }
  w_i = c(w_i, w)
}
MAV2 = mean(abs(w_i * signal))
# mean absolute value slope
channel_sub1 = signal[1:(N/2)]
channel_sub2 = signal[(N/2 + 1):N]
MAV_sub1 = mean(abs(channel_sub1))
MAV_sub2 = mean(abs(channel_sub2))
MAVS = MAV_sub2 - MAV_sub1
# mean binarized values
threshold = 0.5
f = rep(0, length(signal))
f[abs(signal) >= threshold] = 1
MBV = sum(f)/N
# absolute temporal moment
TM4 = mean(signal^4)
# variation fractal dimension
VFD = fd.estim.variation(signal)$fd
# maximum
MAX = max(signal)
# minimum
MIN = min(signal)
# geometric mean
GMN = exp(mean(log(signal[signal > 0])))
# harmonic mean
HMN = harmonic.mean(signal)
# median absolute deviation
MDAD = median(abs(signal - MED))

return(c(MN, VR, SD, SS, SK, IAV, MAV, SSI, RMS, V2, V3,

```

```

    WL, AAC, DASDV, MFL, ZC, RC, SSC, DR, ENT, LOG, MAD,
    Q1, Q3, IQR, CV, MED, MOD, TKEO, AR_1, AR_2, AR_3, AR_4,
    DFA, HFD, KATZ, MAV1, MAV2, MAVS, MBV, TM4, VFD, MAX,
    MIN, GMN, HMN, MDAD))
}
# Calculate cross time domain features
cross_measures <- function(signal1, signal2) {
  dis1 = entropy::discretize(signal1, numBins = 5)
  dis2 = entropy::discretize(signal2, numBins = 5)
  # cross entropy
  prob1 = dis1/sum(dis1)
  prob2 = dis2/sum(dis2)
  ENT = -sum(prob1 * log(prob2))
  # cross correlation function
  cc = ccf(signal1, signal2, plot = FALSE)
  acf = cc$acf
  CCP = acf[which.max(abs(acf))]
  CCL = cc$lag[which.max(abs(acf))]
  # Mutual information
  MI = mutinformation(dis1, dis2)
  return(c(ENT, CCP, CCL, MI))
}
# Get cross measures between each pair of x,y,z channels
get_cross_measures <- function(xyz) {
  cross_measures_vec = cross_measures(xyz[, 1], xyz[, 2])
  cross_measures_vec = c(cross_measures_vec, cross_measures(xyz[,
    1], xyz[, 3]))
  cross_measures_vec = c(cross_measures_vec, cross_measures(xyz[,
    2], xyz[, 3]))

  return(cross_measures_vec)
}

```

2.2 Build features table

```

# Extract time domain and frequency domain features for
# each subject and a complex task
calculate_features_complex <- function(segmented_folder) {
  if (segmented_folder == "walk_seg") {
    num_sub_tasks = num_sub_tasks_walk
  } else {

```

```

    # TUG
    num_sub_tasks = num_sub_tasks_TUGs
}
# create empty data frame
feature_mat = data.frame(matrix(nrow = 0, ncol = features_per_task *
    num_sub_tasks + 1))
indx = 0
# loop over all cases
subj_start = 1
subj_end = num_cases
start_id = 0

# loop over subjects
for (i in subj_start:subj_end) {
    current_path = "./data/"
    sub_folder = paste0("s", i)
    current_path = paste0(current_path, sub_folder, "/",
        segmented_folder)
    files <- list.files(path = current_path, pattern = "*.txt",
        full.names = TRUE, recursive = FALSE)
    if (length(files) > 0) {
        indx = indx + 1
        current_row = i + start_id
        # loop over components of a task and calculate
        # features
        for (f in 1:num_sub_tasks) {
            file_name = files[f]
            channels <- read.table(file_name, header = FALSE) # load file
            names(channels) <- c("ax", "ay", "az", "gx",
                "gy", "gz")
            # loop over channels
            for (j in 1:num_channels) {
                channel = channels[, j]
                # get FFT and LSP features
                fft_top = fourier_peaks(channel)
                lsp_top = lsp_peaks(channel)
                fft_top = as.vector(t(fft_top))
                lsp_top = as.vector(t(lsp_top))
                sig_fft_lsp = c(fft_top, lsp_top)
                current_row = c(current_row, sig_fft_lsp)
                # get frequency domain features
                freq_measures = frequency_measures(channel)
            }
        }
        feature_mat[current_row, ] = c(1, sig_fft_lsp, freq_measures)
    }
}

```

```

        current_row = c(current_row, freq_measures)
        # get time domain features
        t_measures = time_measures(channel)
        current_row = c(current_row, t_measures)
    }
    # get cross time domain features
    channel_acc = channels[, 1:3]
    channel_gyro = channels[, 4:6]
    current_row = c(current_row, get_cross_measures(channel_acc))
    current_row = c(current_row, get_cross_measures(channel_gyro))
}
# add subject features to the data frame
feature_mat = rbind(feature_mat, current_row)
}
}

return(feature_mat)
}

# Define column names for complex tasks
get_col_names_complex <- function(task_name) {
    col_names = c("PDGP")
    freq_names = c("fMNF", "fMDF", "fDPR", "fSN", "fOHM", "fFI",
        "fENT", "fTTP", "fMNP", "fPKF", "fPKP", "fFR", "fPSR",
        "fSM1", "fSM2", "fSM3", "fVR", "fSD", "fSS", "fSK", "fSBW",
        "fSR", "fVCF", "fMSE")
    time_names = c("tMN", "tVR", "tSD", "tSS", "tSK", "tIAV",
        "tMAV", "tSSI", "tRMS", "tV2", "tV3", "tWL", "tAAC",
        "tDASDV", "tMFL", "tZC", "tRC", "tSSC", "tDR", "tENT",
        "tLOG", "tMAD", "tQ1", "tQ3", "tIQR", "tCV", "tMED",
        "tMOD", "tTKEO", "tAR_1", "tAR_2", "tAR_3", "tAR_4",
        "tDFA", "tHFD", "tKATZ", "tMAV1", "tMAV2", "tMAVS", "tMBV",
        "tTM4", "tVFD", "tMAX", "tMIN", "tGMN", "tHMN", "tMDAD")
    channel_names = c("ax", "ay", "az", "gx", "gy", "gz")
    cross_acc = c("axy", "axz", "ayz")
    cross_gyro = c("gxy", "gxz", "gyz")
    cross_names = c("ENT", "CCP", "CCL", "MI")

    if (task_name == "walk") {
        num_sub_tasks = num_sub_tasks_walk
        tasks_names = c("_turn1", "_turn2", "_turn3", "_walk1",
            "_walk2", "_walk3", "_walk4")
    }
}

```

```

} else {
  num_sub_tasks = num_sub_tasks_TUGs
  tasks_names = c("_sTs1", "_sTs2", "_turn1", "_turn2",
    "_walk1", "_walk2")
}
# loop over task components
for (t in 1:num_sub_tasks) {
  task_name = tasks_names[t]
  for (c in 1:num_channels) {
    ch_name = channel_names[c]
    for (j in 1:fft_peaks) {
      peak_name = paste0("_fft", j)
      col_name = paste0(ch_name, peak_name)
      col_name = paste0(col_name, task_name)
      col_names <- c(col_names, col_name)
    }
    for (j in 1:fft_peaks) {
      peak_name = paste0("_lsp", j)
      col_name = paste0(ch_name, peak_name)
      col_name = paste0(col_name, task_name)
      col_names <- c(col_names, col_name)
    }
    for (j in 1:frequency_features) {
      col_name = paste0(ch_name, freq_names[j])
      col_name = paste0(col_name, task_name)
      col_names <- c(col_names, col_name)
    }
    for (j in 1:time_features) {
      col_name = paste0(ch_name, time_names[j])
      col_name = paste0(col_name, task_name)
      col_names <- c(col_names, col_name)
    }
  }
}
for (j in 1:(num_pairs/2)) {
  for (k in 1:cross_features) {
    col_name = paste0(cross_acc[j], "_")
    col_name = paste0(col_name, cross_names[k])
    col_name = paste0(col_name, task_name)
    col_names <- c(col_names, col_name)
  }
}
for (j in 1:(num_pairs/2)) {

```

```

        for (k in 1:cross_features) {
            col_name = paste0(cross_gyro[j], "_")
            col_name = paste0(col_name, cross_names[k])
            col_name = paste0(col_name, task_name)
            col_names <- c(col_names, col_name)
        }
    }
}

return(col_names)
}

# Extract time domain and frequency domain features for
# each subject and a simple task
calculate_features <- function(task_name) {
    # create empty data frame
    feature_mat = data.frame(matrix(nrow = 0, ncol = features_per_task +
        1))
    indx = 0

    subj_start = 1
    subj_end = 300
    start_id = 0

    # loop over subjects
    for (i in subj_start:subj_end) {
        current_path = "./data/"
        sub_folder = paste0("s", i)
        current_path = paste0(current_path, sub_folder)
        files <- list.files(path = current_path, pattern = "*.txt",
            full.names = TRUE, recursive = FALSE)
        found = FALSE
        file_name = ""
        if (length(files) > 0) {
            if (task_name == "EyesClosed" && (length(grep("Eyes closed",
                files)) > 0 || length(grep("EyesClosed", files)) >
                0)) {
                found = TRUE
                if (length(grep("Eyes closed", files)) > 0) {
                    file_name = files[grep("Eyes closed", files)]
                } else {
                    file_name = files[grep("EyesClosed", files)]
                }
            }
        }
    }
}

```

```

    }
  } else if (task_name == "EyesOpen" && (length(grep("Eyes open",
    files)) > 0 || length(grep("EyesOpen", files)) >
    0)) {
    found = TRUE
    if (length(grep("Eyes open", files)) > 0) {
      file_name = files[grep("Eyes open", files)]
    } else {
      file_name = files[grep("EyesOpen", files)]
    }
  } else if (task_name == "TUG1" && length(grep("_TUG_1",
    files)) > 0) {
    found = TRUE
    file_name = files[grep("_TUG_1", files)]
  }
}
if (found) {
  indx = indx + 1
  print(file_name)
  # read 6 channels data
  channels = read.table(file_name, header = FALSE) # load file
  names(channels) <- c("ax", "ay", "az", "gx", "gy",
    "gz")
  current_row = i + start_id
  # loop over channels
  for (j in 1:num_channels) {
    channel = channels[, j]
    if (sum(is.na(channel)) == nrow(channels)) {
      current_row = c(current_row, rep(NA, features_per_channel))
    } else {
      # get FFT and LSP features
      fft_top = fourier_peaks(channel)
      lsp_top = lsp_peaks(channel)
      fft_top = as.vector(t(fft_top))
      lsp_top = as.vector(t(lsp_top))
      sig_fft_lsp = c(fft_top, lsp_top)
      current_row = c(current_row, sig_fft_lsp)
      # get frequency domain features
      freq_measures = frequency_measures(channel)
      current_row = c(current_row, freq_measures)
      # get time domain features
      t_measures = time_measures(channel)
    }
  }
}

```



```

        current_row = c(current_row, t_measures)
    }
}
# get cross time domain features
channel_acc = channels[, 1:3]
channel_gyro = channels[, 4:6]
current_row = c(current_row, get_cross_measures(channel_acc))
current_row = c(current_row, get_cross_measures(channel_gyro))
# add subject features to the data frame
feature_mat = rbind(feature_mat, current_row)
}
}

return(feature_mat)
}

# Define column names for simple tasks
get_col_names <- function() {
    col_names = c("PDGP")
    freq_names = c("fMNF", "fMDF", "fDPR", "fSN", "fOHM", "fFI",
        "fENT", "fTTP", "fMNP", "fPKF", "fPKP", "fFR", "fPSR",
        "fSM1", "fSM2", "fSM3", "fVR", "fSD", "fSS", "fSK", "fSBW",
        "fSR", "fVCF", "fMSE")
    time_names = c("tMN", "tVR", "tSD", "tSS", "tSK", "tIAV",
        "tMAV", "tSSI", "tRMS", "tV2", "tV3", "tWL", "tAAC",
        "tDASDV", "tMFL", "tZC", "tRC", "tSSC", "tDR", "tENT",
        "tLOG", "tMAD", "tQ1", "tQ3", "tIQR", "tCV", "tMED",
        "tMOD", "tTKEO", "tAR_1", "tAR_2", "tAR_3", "tAR_4",
        "tDFA", "tHFD", "tKATZ", "tMAV1", "tMAV2", "tMAVS", "tMBV",
        "tTM4", "tVFD", "tMAX", "tMIN", "tGMN", "tHMN", "tMDAD")
    cross_names = c("ENT", "CCP", "CCL", "MI")
    channel_names = c("ax", "ay", "az", "gx", "gy", "gz")
    cross_acc = c("axy", "axz", "ayz")
    cross_gyro = c("gxy", "gxz", "gyz")

    for (c in 1:num_channels) {
        ch_name = channel_names[c]
        for (j in 1:fft_peaks) {
            peak_name = paste0("_fft", j)
            col_name = paste0(ch_name, peak_name)
            col_names <- c(col_names, col_name)
        }
    }
}

```

```

    for (j in 1:fft_peaks) {
      peak_name = paste0("_lsp", j)
      col_name = paste0(ch_name, peak_name)
      col_names <- c(col_names, col_name)
    }
    for (j in 1:frequency_features) {
      col_name = paste0(ch_name, freq_names[j])
      col_names <- c(col_names, col_name)
    }
    for (j in 1:time_features) {
      col_name = paste0(ch_name, time_names[j])
      col_names <- c(col_names, col_name)
    }
  }
  for (j in 1:(num_pairs/2)) {
    for (k in 1:cross_features) {
      col_name = paste0(cross_acc[j], "_")
      col_name = paste0(col_name, cross_names[k])
      col_names <- c(col_names, col_name)
    }
  }
  for (j in 1:(num_pairs/2)) {
    for (k in 1:cross_features) {
      col_name = paste0(cross_gyro[j], "_")
      col_name = paste0(col_name, cross_names[k])
      col_names <- c(col_names, col_name)
    }
  }
  return(col_names)
}

```

```

# Build features tables for simple tasks
names_tasks = c("EyesClosed", "EyesOpen")
tasks_feature_mat = list()
for (i in 1:length(names_tasks)) {
  task = names_tasks[i]
  feature_mat = calculate_features(task)
  colnames(feature_mat) = get_col_names()
  tasks_feature_mat[[i]] = feature_mat
}
# Build features tables for complex tasks
task_names = c("TUG1", "TUG2", "CogTUG1", "CogTUG2")

```

```

seg_folders = paste0(task_names, "_seg")
for (i in 1:length(seg_folders)) {
  task = seg_folders[i]
  feature_mat = calculate_features_complex(task)
  colnames(feature_mat) = get_col_names_complex(task_names[i])
  tasks_feature_mat[[i + 2]] = feature_mat
}
feature_mat = calculate_features_complex("walk_seg")
colnames(feature_mat) = get_col_names_complex("walk")
tasks_feature_mat[[7]] = feature_mat
# save features table
save(tasks_feature_mat, file = "./rdata/sensor_features.RData")

load("./rdata/sensor_features.RData")

# Add demographics and rating scales variables to data
# table
add_demographics_cc <- function(data_table) {
  demo_table = read_excel("./data/dynaportbase2020Nov_no PID_updated_imputed.xlsx")
  demo_table$disease_dur = as.numeric(format(as.Date(demo_table$DOV,
    format = "%m/%d/%Y"), "%Y")) - as.numeric(demo_table$dxs_dxyr)
  demo_table = subset(demo_table, select = -c(DOV, rdate, edu,
    datediff, dxs_dxyr, dxs_sxyr, deltatdate, absdiff, rdate1_moca,
    `MoCA date rvc`, Medication))
  demo_table = demo_table[, !str_detect(colnames(demo_table),
    "adf_OLD")]
  demo_table = demo_table[, !str_detect(colnames(demo_table),
    "kmoca")]
  demo_table[, str_detect(colnames(demo_table), "updrs") &
    !str_detect(colnames(demo_table), "updrs_piii") & !str_detect(colnames(demo_table),
    "updrs_se") & !str_detect(colnames(demo_table), "updrs_t")] = data.frame(apply(demo_table[,
    str_detect(colnames(demo_table), "updrs") & !str_detect(colnames(demo_table),
    "updrs_piii") & !str_detect(colnames(demo_table),
    "updrs_se") & !str_detect(colnames(demo_table), "updrs_t")],
    2, as.factor), stringsAsFactors = TRUE)
  demo_table[, str_detect(colnames(demo_table), "moca") & !str_detect(colnames(demo_table),
    "moca_t")] = data.frame(apply(demo_table[, str_detect(colnames(demo_table),
    "moca") & !str_detect(colnames(demo_table), "moca_t")],
    2, as.factor), stringsAsFactors = TRUE)
  demo_table[, str_detect(colnames(demo_table), "oars") & !str_detect(colnames(demo_table),
    "oars_a") & !str_detect(colnames(demo_table), "oars_i") &
    !str_detect(colnames(demo_table), "oars_t")] = data.frame(apply(demo_table[,

```

```

    str_detect(colnames(demo_table), "oars") & !str_detect(colnames(demo_table),
      "oars_a") & !str_detect(colnames(demo_table), "oars_i") &
      !str_detect(colnames(demo_table), "oars_t")], 2,
    as.factor), stringsAsFactors = TRUE)
demo_table[, str_detect(colnames(demo_table), "adf")] = data.frame(apply(demo_table[,
  str_detect(colnames(demo_table), "adf")], 2, as.factor),
  stringsAsFactors = TRUE)
demo_table$HYstg = as.factor(demo_table$HYstg)
demo_table$Race = as.factor(demo_table$Race)
demo_table$maxgender = as.factor(demo_table$maxgender)

missing_PDGP = setdiff(data_table$PDGP, demo_table$PDGP)

extended_mat = demo_table
for (id in missing_PDGP) {
  insert_indx = grep(paste0("\\b", id, "\\b"), extended_mat$PDGP)
  extended_mat = rbind(extended_mat[1:insert_indx, ], c(id,
    rep(NA, ncol(demo_table) - 1)), extended_mat[-(1:insert_indx),
    ])
}

extended_mat = extended_mat[extended_mat$PDGP %in% data_table$PDGP,
  ]
extended_mat = subset(extended_mat, select = -PDGP)
colnames(extended_mat) = paste0(colnames(extended_mat), ".demo")
vec_data = cbind(data_table, extended_mat)
return(vec_data)
}

# Exclude subjects with missing CogTUGs, then merge data
# frames of all tasks
PDGP_CogTUGs = intersect(tasks_feature_mat[[5]]$PDGP, tasks_feature_mat[[6]]$PDGP)
sensor_df = data.frame(matrix(nrow = length(PDGP_CogTUGs), ncol = 0))
sensor_df$PDGP = PDGP_CogTUGs
tasks_order = c(7, 1, 2, 3, 4, 5, 6)
for (k in 1:length(tasks_feature_mat)) {
  i = tasks_order[k]
  current_mat = tasks_feature_mat[[i]]
  missing_PDGP = setdiff(PDGP_CogTUGs, current_mat$PDGP)
  # set missing tasks with NA
  extended_mat = current_mat
  for (id in missing_PDGP) {

```

```

insert_indx = grep(paste0("\\b", id - 1, "\\b"), extended_mat$PDGP)
extended_mat = rbind(extended_mat[1:insert_indx, ], c(id,
  rep(NA, ncol(current_mat) - 1)), extended_mat[-(1:insert_indx),
  ])
}
extended_mat = extended_mat[extended_mat$PDGP %in% PDGP_CogTUGs,
  ]
extended_mat = subset(extended_mat, select = -PDGP)
colnames(extended_mat) = paste0(colnames(extended_mat), "_t",
  k)
sensor_df = cbind(sensor_df, extended_mat)
}

# Extend feature table with mean and difference columns of
# TUG and CogTUG
tug1 = sensor_df[, grep("_t4", colnames(sensor_df))]
tug2 = sensor_df[, grep("_t5", colnames(sensor_df))]
mean_tug = (tug1 + tug2)/2
colnames(mean_tug) = paste0(colnames(mean_tug), "t5")
diff_tug = tug1 - tug2
colnames(diff_tug) = paste0(colnames(diff_tug), "t5_diff")

cogtug1 = sensor_df[, grep("_t6", colnames(sensor_df))]
cogtug2 = sensor_df[, grep("_t7", colnames(sensor_df))]
mean_cogtug = (cogtug1 + cogtug2)/2
colnames(mean_cogtug) = paste0(colnames(mean_cogtug), "t7")
diff_cogtug = cogtug1 - cogtug2
colnames(diff_cogtug) = paste0(colnames(diff_cogtug), "t7_diff")

sensor_df = cbind(sensor_df, mean_tug, diff_tug, mean_cogtug,
  diff_cogtug)

# Add demographics and rating scales variables
sensor_df = add_demographics_cc(sensor_df)

# save merged features table
save(sensor_df, file = "./rdata/sensor_features_all_tasks.RData")

```

2.3 Feature reduction

```
# AIC calculations
crossEntropy = function(y_true, y_pred, eps = .Machine$double.xmin) {
  y_pred = y_pred/apply(y_pred, 1, sum)
  y_pred = pmax(pmin(y_pred, 1 - eps), eps)
  y_true = as.numeric(y_true == levels(y_true)[-1])
  H = -mean(y_true * log(y_pred) + (1 - y_true) * log(1 - y_pred))
  return(H)
}

clfAIC = function(y_true, y_pred, k, eps = .Machine$double.xmin) {
  H = crossEntropy(y_true, y_pred, eps)
  AIC = 2 * length(y_true) * H + 2 * k
  return(AIC)
}

# Find elbow point
elbowRule = function(x, y = NULL, norm = F, nonegative = T) {
  # account for extreme input cases of having constant
# values or no values
  if (length(unique(x)) == 1 || length(x) == 0) {
    return(NA)
  }
  n = length(x)
  if (is.null(y)) {
    x = sort(x)
    y = 1:n
  } else {
    stopifnot(n == length(y))
    y = y[order(x)]
    x = sort(x)
  }
  if (nonegative) {
    n = sum(x >= 0)
    y = y[x >= 0]
    x = x[x >= 0]
  }
  if (norm) {
    x0 = (x - min(x))/(max(x) - min(x))
    y0 = (y - min(y))/(max(y) - min(y))
  } else {
```

```

    x0 = x
    y0 = y
  }
  m = (y0[n] - y0[1])/(x0[n] - x0[1]) # calculate slope
  b = y0[1] - m * x0[1] # calculate intercept
  dist_norm = sapply(1:n, function(i) {
    abs(-1 * m * x0[i] + y0[i] - b)
  })
  return(x[which.max(dist_norm)])
}

# Apply forward selection and return RF with selected
# features
forward_select <- function(predictor_vars, response_var) {
  n = nrow(predictor_vars)
  K = ncol(predictor_vars)
  # downsample to the smaller class size
  count = t(as.data.frame(table(response_var))[, 2])
  min_size = min(count)
  num_classes = length(unique(response_var))
  sampsize = rep(min_size, num_classes)
  # build initial RF
  rf = randomForest(x = predictor_vars, y = response_var, ntree = 1e+05,
    importance = TRUE, sampsize = sampsize, proximity = TRUE)
  y_pred = rf$votes
  aic = clfAIC(y_true = response_var, y_pred = y_pred, k = K)
  min_aic = aic
  # get initial variable importance
  var_imp = getVarImp(rf)
  var_imp_ = var_imp[, 1]
  best_features = rownames(var_imp)
  best_rf = rf
  # find elbow point
  threshold = which(var_imp_ == elbowRule(var_imp_))[1]
  # train multiple RF using subsets of important
  # variables before the elbow point
  res = for (i in 1:threshold) {
    predictor_vars_ = predictor_vars[, colnames(predictor_vars) %in%
      rownames(var_imp)[1:i], drop = FALSE]
    rf = randomForest(x = predictor_vars_, y = response_var,
      ntree = 1000, importance = TRUE, proximity = TRUE)
    y_pred = rf$votes
  }
}

```

```

    aic = clfAIC(y_true = response_var, y_pred = y_pred,
                k = ncol(predictor_vars_))
    # find model with min AIC
    if (aic < min_aic) {
      min_aic = aic
      var_imp = getVarImp(rf)
      best_features = rownames(var_imp)
      best_rf = rf
    }
  }
  return(best_rf)
}

```

2.3.1 PD and parkinsonism participants

```

load("./rdata/sensor_features_all_tasks.RData")
parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
                    235, 241, 259, 261, 263, 265, 268, 297, 298)
excluded_PD = c(246, 252, 293)

## All tasks model
sensor_df_ = sensor_df[!sensor_df$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(sensor_df_))
labels[sensor_df_$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)
predictor_vars = subset(sensor_df_, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/PD_PDism.RData")

## TUG-only model
sensor_df_ = sensor_df[!sensor_df$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(sensor_df_))
labels[sensor_df_$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)

predictor_vars = sensor_df_
predictor_vars = predictor_vars[, str_detect(colnames(predictor_vars),
  "PDGP") | str_detect(colnames(predictor_vars), "_t4") | str_detect(colnames(predictor_vars),
  "_t5") | str_detect(colnames(predictor_vars), "_t4t5") |

```



```

    str_detect(colnames(predictor_vars), "_t4t5_diff") | str_detect(colnames(predictor_vars),
      ".demo")]]
predictor_vars = subset(predictor_vars, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/PD_PDism_tug.RData")

# Split PD patient according to HY stage
data = sensor_df

case_demos = read_excel("./data/dynaportbase2020Nov_no PID_updated_imputed.xlsx")
PD_demos = case_demos[case_demos$PDGP %in% data$PDGP, c("PDGP",
  "HYstg")]
PD_demos$HYstg = as.factor(PD_demos$HYstg)

missing_PDGP = setdiff(data$PDGP, PD_demos$PDGP)
extended_PD_demos = PD_demos
for (id in missing_PDGP) {
  insert_indx = grep(paste0("\\b", id - 1, "\\b"), extended_PD_demos$PDGP)
  extended_PD_demos = rbind(extended_PD_demos[1:insert_indx,
    ], c(id, rep(NA, ncol(PD_demos) - 1)), extended_PD_demos[-(1:insert_indx),
    ])
}
PD_demos = extended_PD_demos
PD_demos = na.roughfix(PD_demos)

PD_data = data
PD_early = PD_data[PD_demos$HYstg == 1 | PD_demos$HYstg == 1.5 |
  PD_demos$HYstg == 2, ]
PD_mild = PD_data[PD_demos$HYstg == 2.5 | PD_demos$HYstg == 3,
  ]
PD_severe = PD_data[PD_demos$HYstg == 4, ]

data_early = PD_early
data_mild = PD_mild
data_severe = PD_severe

save(data_early, file = "./rdata/HY_early_sensor_features_all_tasks.RData")
save(data_mild, file = "./rdata/HY_mild_sensor_features_all_tasks.RData")
save(data_severe, file = "./rdata/HY_severe_sensor_features_all_tasks.RData")

```

2.3.2 Mild PD and parkinsonism participants

```
load("./rdata/HY_early_sensor_features_all_tasks.RData")

## All tasks model
data_early = data_early[!data_early$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(data_early))
labels[data_early$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)
predictor_vars = subset(data_early, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_early.RData")

## TUG-only model
data_early = data_early[!data_early$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(data_early))
labels[data_early$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)

predictor_vars = data_early
predictor_vars = predictor_vars[, str_detect(colnames(predictor_vars),
  "PDGP") | str_detect(colnames(predictor_vars), "_t4") | str_detect(colnames(predictor_vars),
  "_t5") | str_detect(colnames(predictor_vars), "_t4t5") |
  str_detect(colnames(predictor_vars), "_t4t5_diff") | str_detect(colnames(predictor_vars),
  ".demo")]
predictor_vars = subset(predictor_vars, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_early_tug.RData")
```

2.3.3 Moderate PD and parkinsonism participants

```
load("./rdata/HY_mild_sensor_features_all_tasks.RData")

## All tasks model
data_mild = data_mild[!data_mild$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(data_mild))
labels[data_mild$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)
```

```

predictor_vars = subset(data_mild, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_mild.RData")

## TUG-only model
data_mild = data_mild[!data_mild$PDGP %in% excluded_PD, ]
labels = rep("PD", nrow(data_mild))
labels[data_mild$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)

predictor_vars = data_mild
predictor_vars = predictor_vars[, str_detect(colnames(predictor_vars),
  "PDGP") | str_detect(colnames(predictor_vars), "_t4") | str_detect(colnames(predictor_vars),
  "_t5") | str_detect(colnames(predictor_vars), "_t4t5") |
  str_detect(colnames(predictor_vars), "_t4t5_diff") | str_detect(colnames(predictor_vars),
  ".demo")]
predictor_vars = subset(predictor_vars, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_mild_tug.RData")

```

2.3.4 Severe PD and parkinsonism participants

```

load("./rdata/HY_severe_sensor_features_all_tasks.RData")

## All tasks model
data_severe = data_severe[!data_severe$PDGP %in% excluded_PD,
  ]
labels = rep("PD", nrow(data_severe))
labels[data_severe$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)
predictor_vars = subset(data_severe, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_severe.RData")

## TUG-only model

```

```
data_severe = data_severe[!data_severe$PDGP %in% excluded_PD,
  ]
labels = rep("PD", nrow(data_severe))
labels[data_severe$PDGP %in% parkinsonism_ids] = "PDism"
response_var = as.factor(labels)

predictor_vars = data_severe
predictor_vars = predictor_vars[, str_detect(colnames(predictor_vars),
  "PDGP") | str_detect(colnames(predictor_vars), "_t4") | str_detect(colnames(predictor_vars),
  "_t5") | str_detect(colnames(predictor_vars), "_t4t5") |
  str_detect(colnames(predictor_vars), "_t4t5_diff") | str_detect(colnames(predictor_vars),
  ".demo")]
predictor_vars = subset(predictor_vars, select = -PDGP)

# Save RF with features corresponding to min AIC
best_rf = forward_select(predictor_vars, response_var)
save(best_rf, file = "./rdata/HY_PDism_severe_tug.RData")
```

Chapter 3

Train/ Test split

```
library(UBL)
library(DMwR)
library(stringr)
library(dplyr)
library(randomForest)
library(caret)
```

3.1 PD and parkinsonism participants

```
set.seed(NULL)
load("./rdata/PD_PDism_tug.RData")
load("./rdata/sensor_features_all_tasks.RData")

data = sensor_df
colnames(data) = str_replace_all(colnames(data), "_turn", "_Turn")
colnames(data) = str_replace_all(colnames(data), "_t", ".t")

parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
  235, 241, 259, 261, 263, 265, 268, 297, 298)
excluded_PD = c(246, 252, 293)

data = data[data$PDGP < 600, ]
data = data[!data$PDGP %in% excluded_PD, ]

labels = rep(1, nrow(data))
```

```

labels[data$PDGP %in% parkinsonism_ids] = 2
imp_data = data.frame(randomForest::importance(rf)[, 1, drop = FALSE])
data = data[, colnames(data) %in% c(rownames(imp_data), "PDGP")]
data$response = as.factor(labels)

predictor_vars = subset(data, select = -c(PDGP, response))
response_var = data$response
PDGP = data$PDGP

# Convert predictors to data frame
predictor_vars = data.frame(predictor_vars, stringsAsFactors = TRUE,
                             check.names = FALSE)

# Convert non-integer and non-numeric variables to factor
# variables
factor_cols = sapply(predictor_vars, function(x) class(x) !=
                     "integer" & class(x) != "numeric")
predictor_vars[, factor_cols] = data.frame(apply(predictor_vars[,
    factor_cols, drop = FALSE], 2, as.factor), stringsAsFactors = TRUE,
    check.names = FALSE)

# Remove variables with constant values (including all NAs)
predictor_vars = predictor_vars %>%
    select(where(~n_distinct(.) > 1))

# Impute missing values
predictor_vars = na.roughfix(predictor_vars)

# Remove variables having infinite values
predictor_vars = predictor_vars[, unlist(lapply(predictor_vars,
    function(x) if (class(x) != "factor") is.finite(sum(x)) else TRUE)))]

# Select rows with no NAs in the response variable
predictor_vars = predictor_vars[!is.na(response_var), ]
response_var = response_var[!is.na(response_var)]
PDGP = PDGP[!is.na(response_var)]

# Normalize
norm = preProcess(predictor_vars)
predictor_vars = predict(norm, predictor_vars)
data = cbind(PDGP, predictor_vars, response_var)

```

```

create_groups <- function(num_folds) {
  groups = list()
  PD_gp_size = ceiling(nrow(PD_data)/num_folds)
  PDism_gp_size = ceiling(nrow(PDism_data)/num_folds)

  for (i in 1:(num_folds - 1)) {
    PD_gp = sample(1:nrow(PD_data), size = PD_gp_size, replace = FALSE)
    PD_gp = PD_data[PD_gp, ]
    PD_data = PD_data[!rownames(PD_data) %in% rownames(PD_gp),
      ]

    PDism_gp = sample(1:nrow(PDism_data), size = PDism_gp_size,
      replace = FALSE)
    PDism_gp = PDism_data[PDism_gp, ]
    PDism_data = PDism_data[!rownames(PDism_data) %in% rownames(PDism_gp),
      ]

    group = rbind(PD_gp, PDism_gp)
    groups[[i]] = group
  }
  group = rbind(PD_data, PDism_data)
  groups[[num_folds]] = group
  return(groups)
}

all_splits = list()
PD_data = data[data$response == 1, ]
PDism_data = data[data$response == 2, ]
num_folds = 3

for (i in 1:5) {
  groups = create_groups(num_folds = num_folds)
  all_splits[[i]] = groups
}

data = rbind(PD_data, PDism_data)
data <- data[order(data$PDGP), ]
save(data, all_splits, file = "./rdata/var_reduct_PD_PDism_splits_tug.RData")

```

3.2 Mild PD and parkinsonism participants

```

set.seed(NULL)
load("./rdata/HY_PDism_early_tug.RData")
load("./rdata/HY_early_sensor_features_all_tasks.RData")

data = data_early
colnames(data) = str_replace_all(colnames(data), "_turn", "_Turn")
colnames(data) = str_replace_all(colnames(data), "_t", ".t")

parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
  235, 241, 259, 261, 263, 265, 268, 297, 298)
excluded_PD = c(246, 252, 293)

data = data[data$PDGP < 600, ]
data = data[!data$PDGP %in% excluded_PD, ]

labels = rep(1, nrow(data))
labels[data$PDGP %in% parkinsonism_ids] = 2
imp_data = data.frame(randomForest::importance(rf)[, 1, drop = FALSE])
data = data[, colnames(data) %in% c(rownames(imp_data), "PDGP")]
data$response = as.factor(labels)

predictor_vars = subset(data, select = -c(PDGP, response))
response_var = data$response
PDGP = data$PDGP

# Convert predictors to data frame
predictor_vars = data.frame(predictor_vars, stringsAsFactors = TRUE,
  check.names = FALSE)

# Convert non-integer and non-numeric variables to factor
# variables
factor_cols = sapply(predictor_vars, function(x) class(x) !=
  "integer" & class(x) != "numeric")
predictor_vars[, factor_cols] = data.frame(apply(predictor_vars[,
  factor_cols, drop = FALSE], 2, as.factor), stringsAsFactors = TRUE,
  check.names = FALSE)

# Remove variables with constant values (including all NAs)

```



```

predictor_vars = predictor_vars %>%
  select(where(~n_distinct(.) > 1))

# Impute missing values
predictor_vars = na.roughfix(predictor_vars)

# Remove variables having infinite values
predictor_vars = predictor_vars[, unlist(lapply(predictor_vars,
  function(x) if (class(x) != "factor") is.finite(sum(x)) else TRUE)))]

# Select rows with no NAs in the response variable
predictor_vars = predictor_vars[!is.na(response_var), ]
response_var = response_var[!is.na(response_var)]
PDGP = PDGP[!is.na(response_var)]

# Normalize
norm = preProcess(predictor_vars)
predictor_vars = predict(norm, predictor_vars)
data = cbind(PDGP, predictor_vars, response_var)

create_groups <- function(num_folds) {
  groups = list()
  PD_gp_size = ceiling(nrow(PD_data)/num_folds)
  PDism_gp_size = ceiling(nrow(PDism_data)/num_folds)

  for (i in 1:(num_folds - 1)) {
    PD_gp = sample(1:nrow(PD_data), size = PD_gp_size, replace = FALSE)
    PD_gp = PD_data[PD_gp, ]
    PD_data = PD_data[!rownames(PD_data) %in% rownames(PD_gp),
      ]

    PDism_gp = sample(1:nrow(PDism_data), size = PDism_gp_size,
      replace = FALSE)
    PDism_gp = PDism_data[PDism_gp, ]
    PDism_data = PDism_data[!rownames(PDism_data) %in% rownames(PDism_gp),
      ]

    group = rbind(PD_gp, PDism_gp)
    groups[[i]] = group
  }
  group = rbind(PD_data, PDism_data)
  groups[[num_folds]] = group
}

```

```

    return(groups)
}

all_splits = list()
PD_data = data[data$response == 1, ]
PDism_data = data[data$response == 2, ]
num_folds = 3

for (i in 1:5) {
  groups = create_groups(num_folds = num_folds)
  all_splits[[i]] = groups
}

data = rbind(PD_data, PDism_data)
data <- data[order(data$PDGP), ]
save(data, all_splits, file = "./rdata/var_reduct_HY_early_PDism_splits_tug.RData")

```

3.3 Moderate PD and parkinsonism participants

```

set.seed(NULL)
load("./rdata/HY_PDism_mild_tug.RData")
load("./rdata/HY_mild_sensor_features_all_tasks.RData")

data = data_mild
colnames(data) = str_replace_all(colnames(data), "_turn", "_Turn")
colnames(data) = str_replace_all(colnames(data), "_t", ".t")

parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
  235, 241, 259, 261, 263, 265, 268, 297, 298)
excluded_PD = c(246, 252, 293)

data = data[data$PDGP < 600, ]
data = data[!data$PDGP %in% excluded_PD, ]

labels = rep(1, nrow(data))
labels[data$PDGP %in% parkinsonism_ids] = 2
imp_data = data.frame(randomForest::importance(rf)[, 1, drop = FALSE])
data = data[, colnames(data) %in% c(rownames(imp_data), "PDGP")]
data$response = as.factor(labels)

```

```

predictor_vars = subset(data, select = -c(PDGP, response))
response_var = data$response
PDGP = data$PDGP

# Convert predictors to data frame
predictor_vars = data.frame(predictor_vars, stringsAsFactors = TRUE,
                             check.names = FALSE)

# Convert non-integer and non-numeric variables to factor
# variables
factor_cols = sapply(predictor_vars, function(x) class(x) !=
                      "integer" & class(x) != "numeric")
predictor_vars[, factor_cols] = data.frame(apply(predictor_vars[,
                                                  factor_cols, drop = FALSE], 2, as.factor), stringsAsFactors = TRUE,
                                                  check.names = FALSE)

# Remove variables with constant values (including all NAs)
predictor_vars = predictor_vars %>%
  select(where(~n_distinct(.) > 1))

# Impute missing values
predictor_vars = na.roughfix(predictor_vars)

# Remove variables having infinite values
predictor_vars = predictor_vars[, unlist(lapply(predictor_vars,
                                                function(x) if (class(x) != "factor") is.finite(sum(x)) else TRUE)))]

# Select rows with no NAs in the response variable
predictor_vars = predictor_vars[!is.na(response_var), ]
response_var = response_var[!is.na(response_var)]
PDGP = PDGP[!is.na(response_var)]

# Normalize
norm = preProcess(predictor_vars)
predictor_vars = predict(norm, predictor_vars)
data = cbind(PDGP, predictor_vars, response_var)

create_groups <- function(num_folds) {
  groups = list()
  PD_gp_size = ceiling(nrow(PD_data)/num_folds)
  PDism_gp_size = ceiling(nrow(PDism_data)/num_folds)

```

```

for (i in 1:(num_folds - 1)) {
  PD_gp = sample(1:nrow(PD_data), size = PD_gp_size, replace = FALSE)
  PD_gp = PD_data[PD_gp, ]
  PD_data = PD_data[!rownames(PD_data) %in% rownames(PD_gp),
    ]

  PDism_gp = sample(1:nrow(PDism_data), size = PDism_gp_size,
    replace = FALSE)
  PDism_gp = PDism_data[PDism_gp, ]
  PDism_data = PDism_data[!rownames(PDism_data) %in% rownames(PDism_gp),
    ]

  group = rbind(PD_gp, PDism_gp)
  groups[[i]] = group
}
group = rbind(PD_data, PDism_data)
groups[[num_folds]] = group
return(groups)
}

all_splits = list()
PD_data = data[data$response == 1, ]
PDism_data = data[data$response == 2, ]
num_folds = 3

for (i in 1:5) {
  groups = create_groups(num_folds = num_folds)
  all_splits[[i]] = groups
}

data = rbind(PD_data, PDism_data)
data <- data[order(data$PDGP), ]
save(data, all_splits, file = "./rdata/var_reduct_HY_mild_PDism_splits_tug.RData")

```

3.4 Severe PD and parkinsonism participants

```

set.seed(NULL)
load("./rdata/HY_PDism_severe_tug.RData")
load("./rdata/HY_severe_sensor_features_all_tasks.RData")

```

```

data = data_severe
colnames(data) = str_replace_all(colnames(data), "_turn", "_Turn")
colnames(data) = str_replace_all(colnames(data), "_t", ".t")

parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
  235, 241, 259, 261, 263, 265, 268, 297, 298)
excluded_PD = c(246, 252, 293)

data = data[data$PDGP < 600, ]
data = data[!data$PDGP %in% excluded_PD, ]

labels = rep(1, nrow(data))
labels[data$PDGP %in% parkinsonism_ids] = 2
imp_data = data.frame(randomForest::importance(rf)[, 1, drop = FALSE])
data = data[, colnames(data) %in% c(rownames(imp_data), "PDGP")]
data$response = as.factor(labels)

predictor_vars = subset(data, select = -c(PDGP, response))
response_var = data$response
PDGP = data$PDGP

# Convert predictors to data frame
predictor_vars = data.frame(predictor_vars, stringsAsFactors = TRUE,
  check.names = FALSE)

# Convert non-integer and non-numeric variables to factor
# variables
factor_cols = sapply(predictor_vars, function(x) class(x) !=
  "integer" & class(x) != "numeric")
predictor_vars[, factor_cols] = data.frame(apply(predictor_vars[,
  factor_cols, drop = FALSE], 2, as.factor), stringsAsFactors = TRUE,
  check.names = FALSE)

# Remove variables with constant values (including all NAs)
predictor_vars = predictor_vars %>%
  select(where(~n_distinct(.) > 1))

# Impute missing values
predictor_vars = na.roughfix(predictor_vars)

```

```

# Remove variables having infinite values
predictor_vars = predictor_vars[, unlist(lapply(predictor_vars,
  function(x) if (class(x) != "factor") is.finite(sum(x)) else TRUE)))]

# Select rows with no NAs in the response variable
predictor_vars = predictor_vars[!is.na(response_var), ]
response_var = response_var[!is.na(response_var)]
PDGP = PDGP[!is.na(response_var)]

# Normalize
norm = preProcess(predictor_vars)
predictor_vars = predict(norm, predictor_vars)
data = cbind(PDGP, predictor_vars, response_var)

create_groups <- function(num_folds) {
  groups = list()
  PD_gp_size = ceiling(nrow(PD_data)/num_folds)
  PDism_gp_size = ceiling(nrow(PDism_data)/num_folds)

  for (i in 1:(num_folds - 1)) {
    PD_gp = sample(1:nrow(PD_data), size = PD_gp_size, replace = FALSE)
    PD_gp = PD_data[PD_gp, ]
    PD_data = PD_data[!rownames(PD_data) %in% rownames(PD_gp),
      ]

    PDism_gp = sample(1:nrow(PDism_data), size = PDism_gp_size,
      replace = FALSE)
    PDism_gp = PDism_data[PDism_gp, ]
    PDism_data = PDism_data[!rownames(PDism_data) %in% rownames(PDism_gp),
      ]

    group = rbind(PD_gp, PDism_gp)
    groups[[i]] = group
  }
  group = rbind(PD_data, PDism_data)
  groups[[num_folds]] = group
  return(groups)
}

all_splits = list()
PD_data = data[data$response == 1, ]
PDism_data = data[data$response == 2, ]

```

```
num_folds = 3

for (i in 1:5) {
  groups = create_groups(num_folds = num_folds)
  all_splits[[i]] = groups
}

data = rbind(PD_data, PDism_data)
data <- data[order(data$PDGP), ]
save(data, all_splits, file = "./rdata/var_reduct_HY_severe_PDism_splits_tug.RData")
```

Chapter 4

Machine learning model

```
library(randomForest)

build_models <- function(data, all_splits, setup_name) {
  num_folds = 3
  seq = 1:num_folds

  for (split_num in 1:5) {
    split = all_splits[[split_num]]
    split_preds = data.frame(matrix(nrow = 0, ncol = 4))
    colnames(split_preds) = c("PD", "PDism", "PDGP", "response")

    for (i in 1:num_folds) {
      test = split[[i]]
      train_indx = seq[seq != i]
      train = data.frame(matrix(nrow = 0, ncol = length(data)))
      colnames(train) = colnames(data)
      for (indx in train_indx) {
        train = rbind(train, split[[indx]])
      }
      train_PDGP = train$PDGP
      test_PDGP = test$PDGP
      train = subset(train, select = -PDGP)
      test = subset(test, select = -PDGP)

      predictor_vars = subset(train, select = -response_var)
      response_var = train$response_var
      count = t(as.data.frame(table(response_var))[, 2])
    }
  }
}
```



```

min_size = min(count)
num_classes = length(unique(response_var))
sampsize = rep(min_size, num_classes)

rf = randomForest(x = predictor_vars, y = response_var,
  ntree = 1e+05, importance = TRUE, sampsize = sampsize,
  proximity = TRUE)
save(rf, file = paste0("./models/PD_PDism_RF_split",
  split_num, "_iter", i, "_tug.RData"))

preds = as.data.frame(predict(rf, test, type = "prob"))
preds$PDGP = test_PDGP
preds$response = factor(ifelse(preds[, 1] > preds[,
  2], "PD", "PDism"))
split_preds = rbind(split_preds, preds)
}
write.csv(split_preds, file = paste0("models/split",
  split_num, "_PD_PDism_RF_predictions_tug.csv"), row.names = FALSE)
}
}

```

4.1 PD and parkinsonism participants

```

load("./rdata/var_reduct_PD_PDism_splits_tug.RData")
build_models(data, all_splits, "PD_PDism")

```

4.2 Mild PD and parkinsonism participants

```

load("./rdata/var_reduct_HY_early_PDism_splits_tug.RData")
build_models(data, all_splits, "HY_early_PDism")

```

4.3 Moderate PD and parkinsonism participants

```

load("./rdata/var_reduct_HY_mild_PDism_splits_tug.RData")
build_models(data, all_splits, "HY_mild_PDism")

```

4.4 Severe PD and parkinsonism participants

```
load("./rdata/var_reduct_HY_severe_PDism_splits_tug.RData")  
build_models(data, all_splits, "HY_severe_PDism")
```

Chapter 5

Performance evaluation

```
library(stringr)
library(DescTools)
library(pROC)
library(MLmetrics)
library(caret)
library(pROC)
library(kableExtra)

balanced_accuracy <- function(y_pred, y_true) {
  return(mean(c(sensitivity(y_pred, y_true), specificity(y_pred,
    y_true))), na.rm = TRUE))
}
```

5.1 PD and parkinsonism participants

```
load("./rdata/var_reduct_PD_PDism_splits_tug.RData")
data$response_var = factor(ifelse(data$response_var == 1, "PD",
  "PDism"))

sl_preds = read.csv("./models/split1_PD_PDism_RF_predictions_3folds_tug.csv")
sl_preds <- sl_preds[order(sl_preds$PDGP), ]
PDGP = sl_preds$PDGP
sl_preds = subset(sl_preds, select = -PDGP)

num_splits = 5
all_sl_preds = sl_preds
```

```

colnames(all_sl_preds) = paste0(colnames(sl_preds), "_split1")
for (i in 2:num_splits) {
  sl_preds = read.csv(paste0("./models/split", i, "_PD_PDism_RF_predictions_3folds_tug.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "response")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1, function(x) {
  uniqx <- unique(na.omit(x))
  uniqx[which.max(tabulate(match(x, uniqx)))]
})

conf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))
knitr::kable(conf$table)

```

	PD	PDism
PD	230	4
PDism	30	14

```

cat("Balanced accuracy = ", round(balanced_accuracy(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))) * 100, 2), "%\n")

```

```
## Balanced accuracy = 83.12 %
```

```

cat("F1_score = ", F1_Score(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism"))), "\n")

```

```
## F1_score = 0.9311741
```

```

auc = auc(as.numeric(data$response_var), as.numeric(factor(all_sl_pred_cls$predict_mode)))
print(auc)

```

```
## Area under the curve: 0.8312
```

5.2 Mild PD and parkinsonism participants

```
load("./rdata/var_reduct_HY_early_PDism_splits_tug.RData")
data$response_var = factor(ifelse(data$response_var == 1, "PD",
  "PDism"))

sl_preds = read.csv("./models/split1_HY_early_PDism_RF_predictions_3folds_tug.csv")
sl_preds <- sl_preds[order(sl_preds$PDGP), ]
PDGP = sl_preds$PDGP
sl_preds = subset(sl_preds, select = -PDGP)

num_splits = 5
all_sl_preds = sl_preds
colnames(all_sl_preds) = paste0(colnames(sl_preds), "_split1")
for (i in 2:num_splits) {
  sl_preds = read.csv(paste0("./models/split", i, "_HY_early_PDism_RF_predictions_3folds_tug.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "response")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1, function(x) {
  uniqx <- unique(na.omit(x))
  uniqx[which.max(tabulate(match(x, uniqx)))]
})

conf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))
knitr::kable(conf$table)
```

	PD	PDism
PD	169	0
PDism	15	6

```
cat("Balanced accuracy = ", round(balanced_accuracy(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))) * 100, 2), "%\n")
```

```
## Balanced accuracy = 95.92 %
```

```
cat("F1_score = ", F1_Score(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism"))), "\n")

## F1_score = 0.9575071

auc = auc(as.numeric(data$response_var), as.numeric(factor(all_sl_pred_cls$predict_mode)))
print(auc)
```

```
## Area under the curve: 0.9592
```

5.3 Moderate PD and parkinsonism participants

```
load("./rdata/var_reduct_HY_mild_PDism_splits_tug.RData")
data$response_var = factor(ifelse(data$response_var == 1, "PD",
  "PDism"))

sl_preds = read.csv("./models/split1_HY_mild_PDism_RF_predictions_3folds_tug.csv")
sl_preds <- sl_preds[order(sl_preds$PDGP), ]
PDGP = sl_preds$PDGP
sl_preds = subset(sl_preds, select = -PDGP)

num_splits = 5
all_sl_preds = sl_preds
colnames(all_sl_preds) = paste0(colnames(sl_preds), "_split1")
for (i in 2:num_splits) {
  sl_preds = read.csv(paste0("./models/split", i, "_HY_mild_PDism_RF_predictions_3folds_tug.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "response")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1, function(x) {
  uniqx <- unique(na.omit(x))
  uniqx[which.max(tabulate(match(x, uniqx)))]
})

conf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
```

```
"PDism"))))
knitr::kable(conf$table)
```

	PD	PDism
PD	56	1
PDism	3	7

```
cat("Balanced accuracy = ", round(balanced_accuracy(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))) * 100, 2), "%\n")
```

```
## Balanced accuracy = 91.21 %
```

```
cat("F1_score = ", F1_Score(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism"))), "\n")
```

```
## F1_score = 0.9655172
```

```
auc = auc(as.numeric(data$response_var), as.numeric(factor(all_sl_pred_cls$predict_mode)))
print(auc)
```

```
## Area under the curve: 0.9121
```

5.4 Severe PD and parkinsonism participants

```
load("./rdata/var_reduct_HY_severe_PDism_splits_tug.RData")
data$response_var = factor(ifelse(data$response_var == 1, "PD",
  "PDism"))
```

```
sl_preds = read.csv("./models/split1_HY_severe_PDism_RF_predictions_3folds_tug.csv")
sl_preds <- sl_preds[order(sl_preds$PDGP), ]
PDGP = sl_preds$PDGP
sl_preds = subset(sl_preds, select = -PDGP)
```

```
num_splits = 5
all_sl_preds = sl_preds
colnames(all_sl_preds) = paste0(colnames(sl_preds), "_split1")
for (i in 2:num_splits) {
  sl_preds = read.csv(paste0("./models/split", i, "_HY_severe_PDism_RF_predictions_3folds_tug.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
```

```

    all_sl_preds = cbind(all_sl_preds, sl_preds)
  }
  all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
    "response")]
  all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1, function(x) {
    uniqx <- unique(na.omit(x))
    uniqx[which.max(tabulate(match(x, uniqx)))]
  })

  conf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
    levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
    "PDism")))
  knitr::kable(conf$table)

```

	PD	PDism
PD	17	0
PDism	0	4

```

cat("Balanced accuracy = ", round(balanced_accuracy(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism")))) * 100, 2), "%\n")

```

```
## Balanced accuracy = 100 %
```

```

cat("F1_score = ", F1_Score(factor(all_sl_pred_cls$predict_mode,
  levels = c("PD", "PDism")), factor(data$response_var, levels = c("PD",
  "PDism"))), "\n")

```

```
## F1_score = 1
```

```

auc = auc(as.numeric(data$response_var), as.numeric(factor(all_sl_pred_cls$predict_mode)))
print(auc)

```

```
## Area under the curve: 1
```