

Parkinson's disease analysis

Rana M. Khalil

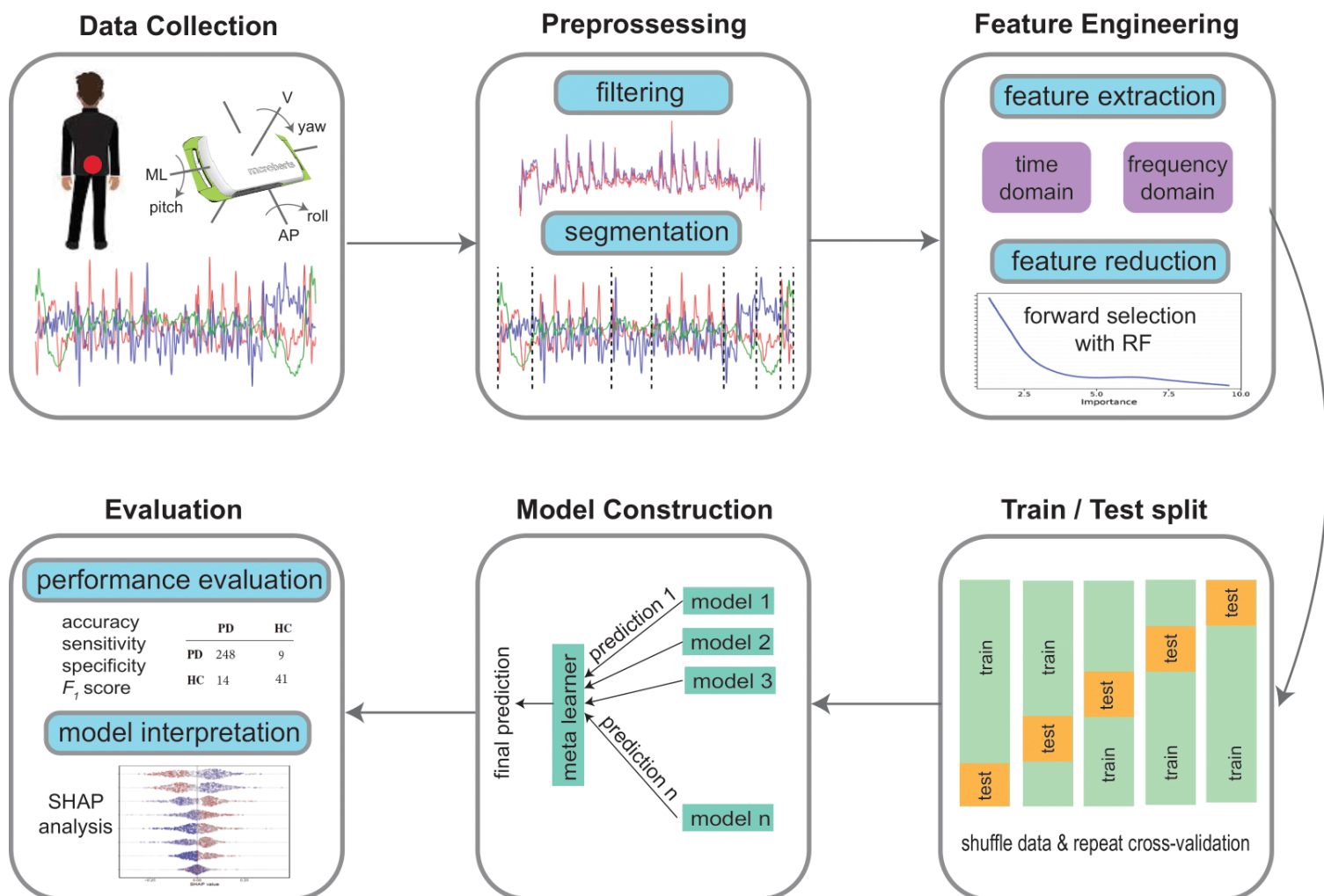
2023-12-21

Contents

Preface	1
1 Preprocessing	3
1.1 Filtering	3
1.2 Segmentation	4
2 Feature engineering	19
2.1 Define time-domain and frequency-domain features	20
2.2 Build features table	26
2.3 Feature reduction	36
3 Train/ Test split	41
3.1 PD participants and controls	42
3.2 Mild PD participants and controls	43
3.3 Moderate PD participants and controls	44
3.4 Severe PD participants and controls	45
4 Model construction	46
4.1 PD participants and controls	50
4.2 Mild PD participants and controls	50
4.3 Moderate PD participants and controls	50
4.4 Severe PD participants and controls	50
5 Evaluation	51
5.1 Performance evaluation	51
5.2 Model interpretation	57

Preface

This notebook provides the code used to build the machine learning pipeline described in the manuscript “*Characterization of high-yield mobility features to identify Parkinson’s disease with a wearable sensor*”. The process starts by filtering and segmenting the composite tasks. Then, a large set of time-domain and frequency-domain features was calculated. This set was reduced using forward selection based on feature importance calculated with random forests. After that, the data was shuffled multiple times, and a cross-validation framework was applied in each iteration to generate training and testing sets. Then, a super learner model was built using a wide array of base models and the final prediction was calculated by assigning a weight to the prediction of each base learner. Finally, the performance of the model was evaluated using the accuracy, sensitivity, specificity, and F1 score measures. The model interpretation was performed using SHAP analysis to understand the contribution of features to the final model prediction.



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)

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
```

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))]
  # Mutulal 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
folder_names = c("case", "control")
# loop over all cases and controls
for (folder_name in folder_names) {
  if (folder_name == "case") {
    subj_start = 1
    subj_end = num_cases
    start_id = 0
  } else {
    subj_start = 1
    subj_end = num_controls
    start_id = 600
  }
  # loop over subjects
  for (i in subj_start:subj_end) {
    current_path = paste0("../data/", folder_name, "/")
    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,
            apply_fft)
        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
    folder_names = c("case", "control")
    # loop over all cases and controls
    for (folder_name in folder_names) {
        if (folder_name == "case") {
            subj_start = 1
            subj_end = 300
            start_id = 0
        } else {
            subj_start = 1
            subj_end = 50
            start_id = 600
        }
        # loop over subjects
        for (i in subj_start:subj_end) {
            current_path = paste0("../data/", folder_name, "/")
            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,
            apply_fft)
        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")
parkinsonism_ids = c(38, 168, 194, 199, 207, 212, 214, 223, 224,
  235, 241, 252, 259, 261, 263, 265, 268, 297, 298)

# Add demographics and rating scales variables to data
# table
add_demographics_cc <- function(data_table) {
  demo_table_cases = read_excel("../data/case/dynaportbase2020Nov_no PID_updated_imputed.xlsx")
  demo_table_controls = read_excel("../data/control/Control_demographics.xlsx")
  common_vars = c("PDGP", "maxgender", "Race", "Height", "age")
  common_vars = c(common_vars, colnames(demo_table_cases)[str_detect(colnames(demo_table_cases),
    "oars")])
  common_vars = c(common_vars, colnames(demo_table_cases)[str_detect(colnames(demo_table_cases),
    "adf") & !str_detect(colnames(demo_table_cases), "adf_OLD")])
  vec_data = data_table
  vec_response_all_cases = demo_table_cases[, common_vars]
  vec_response_all_controls = demo_table_controls[, common_vars]
  vec_cc = data.frame(matrix(nrow = nrow(vec_response_all_cases) +

```



```

    nrow(vec_response_all_controls), ncol = length(common_vars)))
colnames(vec_cc) = common_vars

for (variable in common_vars) {
  vec_cc[, variable] = rbind(demo_table_cases[, variable],
    demo_table_controls[, variable])
}

vec_cc[, 6:20] = as.data.frame(lapply(vec_cc[, 6:20], as.factor))
vec_cc[, 24:38] = as.data.frame(lapply(vec_cc[, 24:38], as.factor))

# set missing variables with NA
missing_PDGP = setdiff(vec_data$PDGP, vec_cc$PDGP)
extended_mat = vec_cc
for (id in missing_PDGP) {
  insert_indx = grep(paste0("\\b", id - 1, "\\b"), extended_mat$PDGP)
  extended_mat = rbind(extended_mat[1:insert_indx, ], c(id,
    rep(NA, ncol(vec_cc) - 1)), extended_mat[-(1:insert_indx),
    ])
}
extended_mat = extended_mat[extended_mat$PDGP %in% vec_data$PDGP,
  ]
extended_mat = subset(extended_mat, select = -PDGP)
colnames(extended_mat) = paste0(colnames(extended_mat), ".demo")
vec_data = cbind(vec_data, extended_mat)
vec_data$maxgender.demo = as.factor(vec_data$maxgender.demo)
vec_data$Race.demo = as.factor(vec_data$Race.demo)
vec_data = droplevels(vec_data)
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$PDGP = PDGP_CogTUGs
for (i in 1:length(tasks_feature_mat)) {
  current_mat = tasks_feature_mat[[i]]
  missing_PDGP = setdiff(PDGP_CogTUGs, current_mat$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",
  i)
sensor_df = cbind(sensor_df, extended_mat)
}

# Extend feature table with mean columns of TUG and CogTUG
tug1 = sensor_df[, grep("_t3", colnames(sensor_df))]
tug2 = sensor_df[, grep("_t4", colnames(sensor_df))]
mean_tug = (tug1 + tug2)/2
colnames(mean_tug) = paste0(colnames(mean_tug), "t4")
cogtug1 = sensor_df[, grep("_t5", colnames(sensor_df))]
cogtug2 = sensor_df[, grep("_t6", colnames(sensor_df))]
mean_cogtug = (cogtug1 + cogtug2)/2
colnames(mean_cogtug) = paste0(colnames(mean_cogtug), "t6")
sensor_df = cbind(sensor_df, mean_tug, mean_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)
  y_pred = rf$votes
  aic = clfAIC(y_true = response_var, y_pred = y_pred, k = K)
  min_aic = aic
  # get initial variable importance
  var_imp = getVarImp(rf)
  var_imp_ = var_imp[, 1]
  best_features = rownames(var_imp)
  best_rf = rf
  # find elbow point
  threshold = which(var_imp_ == elbowRule(var_imp_))[1]
  # train multiple RF using subsets of important
  # variables before the elbow point
  res = for (i in 1:threshold) {
    predictor_vars_ = predictor_vars[, colnames(predictor_vars) %in%
      rownames(var_imp)[1:i], drop = FALSE]
    rf = randomForest(x = predictor_vars_, y = response_var,
      ntree = 1000, importance = TRUE)
    y_pred = rf$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 participants and controls

```
load("./rdata/sensor_features_all_tasks.RData")
sensor_df_ = sensor_df[!sensor_df$PDGP %in% parkinsonism_ids,
]
labels = rep("PD", nrow(sensor_df_))
labels[sensor_df_$PDGP > 600] = "control"
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_control_seg.RData")

# Split PD patient according to HY stage
case_demos = read_excel("../data/case/dynaportbase2020Nov_no PID_updated_imputed.xlsx")
PD_demos = case_demos[case_demos$PDGP %in% sensor_df_$PDGP, c("PDGP",
  "HYstg")]
PD_demos$HYstg = as.factor(PD_demos$HYstg)
missing_PDGP = setdiff(sensor_df_$PDGP[sensor_df_$PDGP < 600],
  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 = sensor_df_[sensor_df_$PDGP < 600, ]
HC_data = sensor_df_[sensor_df_$PDGP > 600, ]
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, ]
sensor_df_early = rbind(PD_early, HC_data)
sensor_df_mild = rbind(PD_mild, HC_data)
sensor_df_severe = rbind(PD_severe, HC_data)
```

2.3.2 Mild PD participants and controls

```
labels = rep("PD", nrow(sensor_df_early))
labels[sensor_df_early$PDGP > 600] = "control"
response_var = as.factor(labels)
predictor_vars = subset(sensor_df_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_control_early.RData")
```

2.3.3 Moderate PD participants and controls

```
labels = rep("PD", nrow(sensor_df_mild))
labels[(sensor_df_mild)$PDGP > 600] = "control"
response_var = as.factor(labels)
predictor_vars = subset((sensor_df_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_control_mild.RData")
```

2.3.4 Severe PD participants and controls

```
labels = rep("PD", nrow(sensor_df_severe))
labels[(sensor_df_severe)$PDGP > 600] = "control"
response_var = as.factor(labels)
predictor_vars = subset((sensor_df_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_control_severe.RData")
```

Chapter 3

Train/ Test split

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

# Create five split stratified by group size
create_groups <- function(group1, group2) {
  groups = list()
  group1_size = floor(nrow(group1)/5)
  group2_size = floor(nrow(group2)/5)

  for (i in 1:4) {
    gp1 = sample(1:nrow(group1), size = group1_size, replace = FALSE)
    gp1 = group1[gp1, ]
    group1 = group1[!rownames(group1) %in% rownames(gp1),
      ]

    gp2 = sample(1:nrow(group2), size = group2_size, replace = FALSE)
    gp2 = group2[gp2, ]
    group2 = group2[!rownames(group2) %in% rownames(gp2),
      ]

    group = rbind(gp1, gp2)
    groups[[i]] = group
  }
  group = rbind(group1, group2)
  groups[[5]] = group
}
```

```

    return(groups)
}

```

3.1 PD participants and controls

```

load("./rdata/PD_control_seg.RData")
data = sensor_df
labels = rep(1, nrow(data))
labels[data$PDGP > 600] = 0
data = data[, colnames(data) %in% c(rownames(rf[["importance"]]),
  "PDGP")]
data$response = as.factor(labels)

PD_data = data[data$response == 1, ]
HC_data = data[data$response == 0, ]
all_splits = list()
# Repeat with five different shuffles
for (i in 1:5) {
  groups = create_groups(PD_data, HC_data)
  all_splits[[i]] = groups
}
save(data, all_splits, file = "./rdata/var_reduct_PD_control_splits.RData")

# Write to csv files
seq = 1:5
for (split_num in 1:5) {
  split = all_splits[[split_num]]
  for (i in 1:5) {
    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]])
    }
    write.csv(train, paste0("./files/PD_HC/train_test_files/train_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
    write.csv(test, paste0("./files/PD_HC/train_test_files/test_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
  }
}

```


3.2 Mild PD participants and controls

```

load("./rdata/HY_control_early.RData")
data = sensor_df_early
labels = rep(1, nrow(data))
labels[data$PDGP > 600] = 0
data = data[, colnames(data) %in% c(rownames(rf[["importance"]]),
  "PDGP")]
data$response = as.factor(labels)

PD_data = data[data$response == 1, ]
HC_data = data[data$response == 0, ]
all_splits = list()
# Repeat with five different shuffles
for (i in 1:5) {
  groups = create_groups(PD_data, HC_data)
  all_splits[[i]] = groups
}
save(data, all_splits, file = "./rdata/var_reduct_HY_early_HC_splits.RData")

# Write to csv files
seq = 1:5
for (split_num in 1:5) {
  split = all_splits[[split_num]]
  for (i in 1:5) {
    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]])
    }
    write.csv(train, paste0("./files/HY_early_HC/train_test_files/train_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
    write.csv(test, paste0("./files/HY_early_HC/train_test_files/test_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
  }
}

```

3.3 Moderate PD participants and controls

```

load("./rdata/HY_control_mild.RData")
data = sensor_df_mild
labels = rep(1, nrow(data))
labels[data$PDGP > 600] = 0
data = data[, colnames(data) %in% c(rownames(rf[["importance"]]),
  "PDGP")]
data$response = as.factor(labels)

PD_data = data[data$response == 1, ]
HC_data = data[data$response == 0, ]
all_splits = list()
# Repeat with five different shuffles
for (i in 1:5) {
  groups = create_groups(PD_data, HC_data)
  all_splits[[i]] = groups
}
save(data, all_splits, file = "./rdata/var_reduct_HY_mild_HC_splits.RData")

# Write to csv files
seq = 1:5
for (split_num in 1:5) {
  split = all_splits[[split_num]]
  for (i in 1:5) {
    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]])
    }
    write.csv(train, paste0("./files/HY_mild_HC/train_test_files/train_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
    write.csv(test, paste0("./files/HY_mild_HC/train_test_files/test_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
  }
}

```

3.4 Severe PD participants and controls

```

load("./rdata/HY_control_severe.RData")
data = sensor_df_severe
labels = rep(1, nrow(data))
labels[data$PDGP > 600] = 0
data = data[, colnames(data) %in% c(rownames(rf[["importance"]]),
  "PDGP")]
data$response = as.factor(labels)

PD_data = data[data$response == 1, ]
HC_data = data[data$response == 0, ]
all_splits = list()
# Repeat with five different shuffles
for (i in 1:5) {
  groups = create_groups(PD_data, HC_data)
  all_splits[[i]] = groups
}
save(data, all_splits, file = "./rdata/var_reduct_HY_severe_HC_splits.RData")

# Write to csv files
seq = 1:5
for (split_num in 1:5) {
  split = all_splits[[split_num]]
  for (i in 1:5) {
    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]])
    }
    write.csv(train, paste0("./files/HY_severe_HC/train_test_files/train_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
    write.csv(test, paste0("./files/HY_severe_HC/train_test_files/test_split",
      split_num, "_iter", i, ".csv"), row.names = FALSE)
  }
}

```

Chapter 4

Model construction

```
library(h2o)

nfold = 5
# GLM grid parameters
glm_params <- list(lambda = seq(from = 0, to = 1, by = 0.1))
# DRF grid parameters
rf_params <- list(ntrees = 10000, max_depth = c(5, 10, 15, 20),
  nbins = c(30, 100, 300), nbins_cats = c(64, 256, 1024), sample_rate = c(0.7,
    0.8, 0.9, 1))
# GBM grid parameters
gbm_params <- list(ntrees = 10000, max_depth = c(5, 10, 15, 20),
  learn_rate = c(0.01, 0.1), sample_rate = c(0.7, 0.8, 0.9,
    1), col_sample_rate = c(0.7, 0.8, 0.9, 1), nbins = c(30,
    100, 300), nbins_cats = c(64, 256, 1024))
# XGboost grid parameters
xgboost_params <- list(max_depth = c(5, 10, 15, 20), ntrees = 10000,
  sample_rate = c(0.7, 0.8, 0.9, 1))
# Deep learning grid parameters
deeplearning_params <- list(activation = c("Rectifier", "Maxout",
  "Tanh", "RectifierWithDropout", "MaxoutWithDropout", "TanhWithDropout"),
  hidden = list(c(50, 50, 50, 50), c(200, 200), c(200, 200,
    200), c(200, 200, 200, 200)), epochs = c(50, 100, 200),
  adaptive_rate = c(TRUE, FALSE), rate = c(0, 0.1, 0.005, 0.001),
  input_dropout_ratio = c(0, 0.1, 0.2))

# Train a super learner model on a
# split of data
```

```

train_SL <- function(train_data, split_num,
  iter_num) {
  models_path = paste0("./models/", classes,
    "/split", split_num, "/iter", iter_num)
  train = as.h2o(train_data)
  y = "response"
  x = setdiff(names(train), y)
  train[y] <- as.factor(train[y])
  res <- as.data.frame(train_data$response)
  samp_factors <- as.vector(mean(table(res))/table(res))
  # train XGboost base models
  xgboost <- h2o.grid(algorithm = "xgboost",
    x = x, y = y, training_frame = train,
    nfolds = nfolds, keep_cross_validation_predictions = TRUE,
    hyper_params = xgboost_params, stopping_rounds = 3,
    search_criteria = list(strategy = "RandomDiscrete",
      max_models = 100), keep_cross_validation_models = FALSE,
    fold_assignment = "Modulo", parallelism = 0)
  # train GLM base models
  glm <- h2o.grid(algorithm = "glm", x = x,
    y = y, training_frame = train, nfolds = nfolds,
    keep_cross_validation_predictions = TRUE,
    hyper_params = glm_params, stopping_rounds = 3,
    balance_classes = TRUE, class_sampling_factors = samp_factors,
    search_criteria = list(strategy = "RandomDiscrete",
      max_models = 100), keep_cross_validation_models = FALSE,
    fold_assignment = "Modulo", parallelism = 0)
  # train DFR base models
  rf <- h2o.grid(algorithm = "randomForest",
    x = x, y = y, training_frame = train,
    nfolds = nfolds, keep_cross_validation_predictions = TRUE,
    hyper_params = rf_params, stopping_rounds = 3,
    balance_classes = TRUE, class_sampling_factors = samp_factors,
    search_criteria = list(strategy = "RandomDiscrete",
      max_models = 100), keep_cross_validation_models = FALSE,
    fold_assignment = "Modulo", parallelism = 0)
  # train GBM base models
  gbm <- h2o.grid(algorithm = "gbm", x = x,
    y = y, training_frame = train, nfolds = nfolds,
    keep_cross_validation_predictions = TRUE,
    hyper_params = gbm_params, stopping_rounds = 3,
    balance_classes = TRUE, class_sampling_factors = samp_factors,

```

```

    search_criteria = list(strategy = "RandomDiscrete",
        max_models = 100), keep_cross_validation_models = FALSE,
    fold_assignment = "Modulo", parallelism = 0)
# train Deep learning base models
deeplearning <- h2o.grid(algorithm = "deeplearning",
    x = x, y = y, training_frame = train,
    nfolds = nfolds, keep_cross_validation_predictions = TRUE,
    hyper_params = deeplearning_params,
    stopping_rounds = 3, balance_classes = TRUE,
    class_sampling_factors = samp_factors,
    search_criteria = list(strategy = "RandomDiscrete",
        max_models = 100), keep_cross_validation_models = FALSE,
    fold_assignment = "Modulo", parallelism = 0)
# list base models
base_models <- as.list(c(unlist(glm@model_ids),
    unlist(rf@model_ids), unlist(gbm@model_ids),
    unlist(xgboost@model_ids), unlist(deeplearning@model_ids)))
# get prediction of each base mode
for (model_id in base_models) {
    res <- cbind(res, as.data.frame(h2o.getFrame(paste0("cv_holdout_prediction_",
        model_id))$predict, col.names = model_id))
    h2o.saveModel(h2o.getModel(model_id),
        path = models_path, force = TRUE)
}
# build SL model
sl <- h2o.stackedEnsemble(x = x, y = y,
    model_id = "superlearner", training_frame = train,
    base_models = base_models, metalearner_algorithm = "glm",
    metalearner_nfolds = nfolds, keep_levelone_frame = TRUE,
    metalearner_params = list(standardize = TRUE,
        keep_cross_validation_predictions = TRUE))
# save metalearner predictions
model_id = sl@model[["metalearner"]][["name"]]
res <- cbind(res, as.data.frame(h2o.getFrame(paste0("cv_holdout_prediction_",
    model_id))$predict, col.names = model_id))
# save SL model
h2o.saveModel(sl, path = models_path,
    force = TRUE)
h2o.saveModel(h2o.getModel(sl@model[["metalearner"]][["name"]]),
    path = models_path, force = TRUE)
# save all models predictions
write.csv(res, file = paste0(models_path,

```

```

    "/cv_holdout_predictions.csv"), row.names = FALSE)
}

# Build a SL model for all shuffles and splits of a data
build_SL <- function(data, all_splits, setup_name) {
  seq = 1:5
  # loop over all shuffles
  for (split_num in 1:5) {
    split = all_splits[[split_num]]
    split_preds = data.frame(matrix(nrow = 0, ncol = 4))
    colnames(split_preds) = c("predict", "p0", "p1", "PDGP")
    # loop over splits of a shuffle
    for (i in 1:5) {
      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)
      # train the SL model
      train_SL(train, split_num = split_num, iter_num = i)
      # predict on test data
      sl = h2o.loadModel(paste0("./models/", setup_name,
                                "/split", split_num, "/iter", i, "/superlearner"))
      preds = as.data.frame(h2o.predict(sl, as.h2o(test)))
      preds$PDGP = test_PDGP
      # combine splits predictions
      split_preds = rbind(split_preds, preds)
    }
    # save predictions of the shuffled data
    write.csv(split_preds, file = paste0("./models/", setup_name,
                                          "/split", split_num, "/sl_predictions.csv"), row.names = FALSE)
  }
}

```

4.1 PD participants and controls

```
load("./rdata/var_reduct_PD_control_splits.RData")  
build_SL(data, all_splits, "PD_HC")
```

4.2 Mild PD participants and controls

```
load("./rdata/var_reduct_HY_early_HC_splits.RData")  
build_SL(data, all_splits, "HY_early_HC")
```

4.3 Moderate PD participants and controls

```
load("./rdata/var_reduct_HY_mild_HC_splits.RData")  
build_SL(data, all_splits, "HY_mild_HC")
```

4.4 Severe PD participants and controls

```
load("./rdata/var_reduct_HY_severe_HC_splits.RData")  
build_SL(data, all_splits, "HY_severe_HC")
```


Chapter 5

Evaluation

```
library(stringr)
library(MLmetrics)
library(h2o)
library(reshape2)
library(ggplot2)
library(readxl)
library(stringr)
library(ggforce)
library(cowplot)
library(plotly)
library(ggrepel)
library(gridExtra)
library(scales)
```

5.1 Performance evaluation

5.1.1 PD participants and controls

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

sl_preds = read.csv("./files/PD_HC/split1/sl_predictions.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("./files/PD_HC/split", i,
    "/sl_predictions.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "predict")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1,
  function(x) {
    uniqx <- unique(na.omit(x))
    uniqx[which.max(tabulate(match(x, uniqx)))]
  })

cf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c(1, 0)), factor(data$response, levels = c(1,
  0)), positive = "1")
sens = Sensitivity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
spec = Specificity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
f1_score = F1_Score(factor(data$response, levels = c(1,
  0)), factor(all_sl_pred_cls$predict_mode, levels = c(1,
  0)))

print(cf$table)

##           Reference
## Prediction    1    0
##           1 248    9
##           0  14   41

cat("Accuracy = ", round(cf$overall[["Accuracy"]], 4) *
  100, "%\n")

## Accuracy = 92.63 %

```

```
cat("Sensitivity = ", round(sens, 3), "\n")

## Sensitivity = 0.947

cat("Specificity = ", round(spec, 3), "\n")

## Specificity = 0.82

cat("F1 score = ", round(f1_score, 3), "\n")

## F1 score = 0.956
```

5.1.2 Mild PD participants and controls

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

sl_preds = read.csv("./files/HY_early_HC/split1/sl_predictions.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("./files/HY_early_HC/split",
    i, "/sl_predictions.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "predict")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1,
  function(x) {
    uniqx <- unique(na.omit(x))
    uniqx[which.max(tabulate(match(x, uniqx)))]
  })

cf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c(1, 0)), factor(data$response, levels = c(1,
```

```

    0)), positive = "1")
sens = Sensitivity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
spec = Specificity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
f1_score = F1_Score(factor(data$response, levels = c(1,
  0)), factor(all_sl_pred_cls$predict_mode, levels = c(1,
  0)))

print(cf$table)

```

```

##           Reference
## Prediction    1    0
##           1 174  14
##           0   11  36

```

```

cat("Accuracy = ", round(cf$overall[["Accuracy"]], 4) *
  100, "%\n")

```

```

## Accuracy = 89.36 %

```

```

cat("Sensitivity = ", round(sens, 3), "\n")

```

```

## Sensitivity = 0.941

```

```

cat("Specificity = ", round(spec, 3), "\n")

```

```

## Specificity = 0.72

```

```

cat("F1 score = ", round(f1_score, 3), "\n")

```

```

## F1 score = 0.933

```

5.1.3 Moderate PD participants and controls

```

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

sl_preds = read.csv("./files/HY_mild_HC/split1/sl_predictions.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("./files/HY_mild_HC/split",
    i, "/sl_predictions.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "predict")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1,
  function(x) {
    uniqx <- unique(na.omit(x))
    uniqx[which.max(tabulate(match(x, uniqx)))]
  })

cf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c(1, 0)), factor(data$response, levels = c(1,
  0)), positive = "1")
sens = Sensitivity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
spec = Specificity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
f1_score = F1_Score(factor(data$response, levels = c(1,
  0)), factor(all_sl_pred_cls$predict_mode, levels = c(1,
  0)))

print(cf$table)

```

```

##           Reference
## Prediction  1  0
##           1 55  6
##           0  5 44

```

```

cat("Accuracy = ", round(cf$overall[["Accuracy"]], 4) *
  100, "%\n")

```

```

## Accuracy = 90 %

```

```

cat("Sensitivity = ", round(sens, 3), "\n")

```

```

## Sensitivity = 0.917

```

```
cat("Specificity = ", round(spec, 3), "\n")

## Specificity = 0.88
cat("F1 score = ", round(f1_score, 3), "\n")

## F1 score = 0.909
```

5.1.4 Severe PD participants and controls

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

sl_preds = read.csv("./files/HY_severe_HC/split1/sl_predictions.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("./files/HY_severe_HC/split",
    i, "/sl_predictions.csv"))
  sl_preds <- sl_preds[order(sl_preds$PDGP), ]
  sl_preds = subset(sl_preds, select = -PDGP)
  colnames(sl_preds) = paste0(colnames(sl_preds), "_split",
    i)
  all_sl_preds = cbind(all_sl_preds, sl_preds)
}
all_sl_pred_cls = all_sl_preds[, str_detect(colnames(all_sl_preds),
  "predict")]
all_sl_pred_cls$predict_mode = apply(all_sl_pred_cls, 1,
  function(x) {
    uniqx <- unique(na.omit(x))
    uniqx[which.max(tabulate(match(x, uniqx)))]
  })

cf = caret::confusionMatrix(data = factor(all_sl_pred_cls$predict_mode,
  levels = c(1, 0)), factor(data$response, levels = c(1,
  0)), positive = "1")
sens = Sensitivity(factor(data$response, levels = c(1, 0)),
  factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
spec = Specificity(factor(data$response, levels = c(1, 0)),
```

```

    factor(all_sl_pred_cls$predict_mode, levels = c(1, 0)))
f1_score = F1_Score(factor(data$response, levels = c(1,
  0)), factor(all_sl_pred_cls$predict_mode, levels = c(1,
  0)))

print(cf$table)

```

```

##           Reference
## Prediction  1  0
##           1 15  0
##           0  2 50

```

```

cat("Accuracy = ", round(cf$overall[["Accuracy"]], 4) *
    100, "%\n")

```

```

## Accuracy = 97.01 %

```

```

cat("Sensitivity = ", round(sens, 3), "\n")

```

```

## Sensitivity = 0.882

```

```

cat("Specificity = ", round(spec, 3), "\n")

```

```

## Specificity = 1

```

```

cat("F1 score = ", round(f1_score, 3), "\n")

```

```

## F1 score = 0.938

```

5.2 Model interpretation

```

# Calculate and save feature importance of a data using top
# model of each iteration
save_var_imp <- function(data, all_splits, top_models, setup_name) {
  h2o.init(nthreads = -1)
  h2o_var_imp = data.frame(matrix(nrow = 0, ncol = 2))
  colnames(h2o_var_imp) = c("Variable", "avg_score")
  seq = 1:5
  # loop over shuffles and splits
  for (i in 0:24) {
    split_num = floor(i/5) + 1
    iter_num = i%%5 + 1
    top_model = top_models[i + 1]
    split = all_splits[[split_num]]

```

```

train_indx = seq[seq != iter_num]
train = data.frame(matrix(nrow = 0, ncol = length(data)))
colnames(train) = colnames(data)
for (indx in train_indx) {
  train = rbind(train, split[[indx]])
}
train = subset(train, select = -PDGP)
# load top model
model = h2o.loadModel(paste0("./models/", setup_name,
  "/split", split_num, "/iter", iter_num, "/", top_model))
# calculate feature importance
imp = h2o.permutation_importance(model, as.h2o(train),
  n_repeats = 10)
imp$avg_score = rowMeans(imp[, 2:11])
imp = subset(imp, select = c(Variable, avg_score))
h2o_var_imp = rbind(h2o_var_imp, imp)
}
# average importance from shuffles and splits
variables = unique(h2o_var_imp$Variable)
var_imp = data.frame(matrix(nrow = length(variables), ncol = 2))
colnames(var_imp) = c("variable", "relative_importance")
scores = c()
for (variable in variables) {
  scores = c(scores, mean(h2o_var_imp[h2o_var_imp$Variable ==
    variable, ]$avg_score))
}
var_imp$variable = variables
var_imp$relative_importance = scores
write.csv(var_imp, paste0("./files/", setup_name, "/top_imp_scores.csv"))
h2o.removeAll()
}

```

```

import h2o
import pandas as pd
import numpy as np
import shap
import math

# Calculate and save feature SHAP values of a data using top model of each iteration
class H2OProbWrapper:
    def __init__(self, h2o_model, feature_names):
        self.h2o_model = h2o_model

```



```

        self.feature_names = feature_names

    def predict_binary_prob(self, X):
        if isinstance(X, pd.Series):
            X = X.values.reshape(1,-1)
        self.dataframe = pd.DataFrame(X, columns=self.feature_names)
        self.dataframe_hex = h2o.H2OFrame(self.dataframe)
        self.predictions = self.h2o_model.predict(self.dataframe_hex).as_data_frame().values
        return self.predictions.astype('float64')[:, -1]

def save_var_shap(data_columns, top_models, setup_name):
    h2o.init(nthreads=-1)
    shap_res = []
    for i in range(25):
        split_num = math.floor(i/5)+1
        iter_num = i%5+1
        top_model = top_models[i]
        # load train and test data
        train = pd.read_csv('./files/'+setup_name+'/train_test_files/train_split' + \
                            str(split_num) + '_iter' + str(iter_num) + '.csv', index_col=0)
        X_train = train.loc[:, train.columns != 'PDGP']
        X_train['response'] = X_train['response'].astype('category')
        test = pd.read_csv('./files/'+setup_name+'/train_test_files/test_split' + \
                            str(split_num) + '_iter' + str(iter_num) + '.csv', index_col=0)
        X_test = test.loc[:, test.columns != 'PDGP']
        X_test['response'] = X_test['response'].astype('category')
        # load top model
        model = h2o.load_model('./models/'+setup_name+'/split'+ str(split_num) + \
                               '/iter' + str(iter_num) + '/' + model_names[i])
        # calculate SHAP values
        h2o_wrapper = H2OProbWrapper(model, data_columns)
        explainer = shap.KernelExplainer(h2o_wrapper.predict_binary_prob, X_train, \
                                         nsamples=100)
        shap_values = explainer.shap_values(X_test)
        shap_res.append(shap_values)
    shap_res = pd.DataFrame(np.vstack(shap_res), columns=X_train.columns)
    shap_res.to_csv('./files/'+setup_name+'/top_shap_values.csv', index=False)
    h2o.remove_all()

```

5.2.1 PD participants and controls

```
load("./rdata/var_reduct_PD_control_splits.RData")
top_models = c("GBM_model_1683058139810_399641", "GBM_model_1683058139810_464011",
  "GBM_model_1683058139810_526922", "GBM_model_1683058139810_586894",
  "GBM_model_1683058139810_645861", "GBM_model_1683058139810_997263",
  "GLM_model_1683058139810_1043946", "GBM_model_1683058139810_1118072",
  "GBM_model_1683058139810_1177862", "GBM_model_1683058139810_1238334",
  "GBM_model_1683058139810_1581860", "DeepLearning_model_1683058139810_1673920",
  "GBM_model_1683058139810_1692858", "DRF_model_1683058139810_1738496",
  "DeepLearning_model_1683058139810_1848898", "GBM_model_1683058139810_5261909",
  "GBM_model_1683058139810_5369141", "GLM_model_1683058139810_5465253",
  "DRF_model_1683058139810_5567905", "DeepLearning_model_1683058139810_5724832",
  "GBM_model_1683058139810_5792244", "GBM_model_1683058139810_5904724",
  "GBM_model_1683297158524_88077", "GBM_model_1683297158524_205274",
  "DRF_model_1683297158524_303765")
save_var_imp(data, all_splits, top_models, "PD_HC")

top_models=["GBM_model_1683058139810_399641","GBM_model_1683058139810_464011","\
"GBM_model_1683058139810_526922","GBM_model_1683058139810_586894","\
"GBM_model_1683058139810_645861","GBM_model_1683058139810_997263","\
"GLM_model_1683058139810_1043946","GBM_model_1683058139810_1118072","\
"GBM_model_1683058139810_1177862","GBM_model_1683058139810_1238334","\
"GBM_model_1683058139810_1581860","DeepLearning_model_1683058139810_1673920","\
"GBM_model_1683058139810_1692858","DRF_model_1683058139810_1738496","\
"DeepLearning_model_1683058139810_1848898","GBM_model_1683058139810_5261909","\
"GBM_model_1683058139810_5369141","GLM_model_1683058139810_5465253","\
"DRF_model_1683058139810_5567905","DeepLearning_model_1683058139810_5724832","\
"GBM_model_1683058139810_5792244","GBM_model_1683058139810_5904724","\
"GBM_model_1683297158524_88077","GBM_model_1683297158524_205274","\
"DRF_model_1683297158524_303765"]
data_columns = ["gyfSR_walk1.t6","axtVFD_Turn1.t7","aytMYOP_Turn2.t7","gytMIN_walk1.t7","\
"aytSSC_sTs2.t6t7","az_fft1_sTs2.t6t7","aztSSC_sTs2.t6t7","gxtMAVS_sTs2.t6t7","gztWL.t3","\
"response"]
save_var_shap(data_columns, top_models, "PD_HC")

imp_data = read.csv("./files/PD_HC/top_imp_scores.csv")
imp_data = subset(imp_data, select = -X)
imp_data <- imp_data[order(imp_data$relative_importance,
  decreasing = TRUE), ]

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

```

rownames(imp_data) = imp_data$variable
top_features_names = c("Mean binarized values of 2nd turn",
  "Variation fractal dimension of 1st turn", "Slope sign change of stand-to-sit",
  "Waveform length", "Slope sign change of stand-to-sit",
  "Mean absolute value slope of stand-to-sit", "Frequency at 1st peak of DFT of stand-to-sit",
  "DFT spectral roll-off of 1st walk", "Minimum value of 1st walk")
tasks_channels = c("CogTUG2 (ay)", "CogTUG2 (ax)", "Mean CogTUG (az)",
  "Stand eyes open (gz)", "Mean CogTUG (ay)", "Mean CogTUG (gx)",
  "Mean CogTUG (az)", "CogTUG1 (gy)", "CogTUG2 (gy)")
task_pos = c(1.15, 2.15, 3.13, 4.3, 5.15, 6, 7.15, 8.15,
  9.17)

imp_data$Subset <- factor(sapply(rownames(imp_data), function(x) strsplit(x,
  split = "\\.").[[1]][2]))
p_imp = ggplot(imp_data, aes(x = reorder(variable, relative_importance),
  y = relative_importance)) + geom_bar(stat = "identity",
  width = 0.5, fill = "#A4A2A6") + coord_flip(ylim = c(0.0025,
  max(imp_data$relative_importance) + 0.005), xlim = c(1,
  9), clip = "off") + theme_bw() + theme(panel.grid.major = element_blank(),
  panel.grid.minor = element_blank(), panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),
  panel.border = element_rect(color = "black"), plot.margin = ggplot2::margin(0,
  65, 0, 150, "pt"), axis.title.x = element_text(color = "black",
  size = 15, family = "sans"), axis.title.y = element_text(color = "black",
  size = 15, family = "sans"), axis.text.x = element_text(color = "black",
  size = 12, family = "sans"), axis.text.y = element_text(color = "black",
  size = 12, family = "sans"), legend.title = element_blank(),
  legend.position = "top", legend.margin = ggplot2::margin(b = -10)) +
  scale_y_continuous(breaks = seq(0.01, 0.05, 0.01)) +
  scale_x_discrete(labels = lapply(rev(top_features_names),
  str_wrap, width = 22)) + xlab("") + ylab("Relative importance") +
  ggtitle("") + annotate("text", label = rev(tasks_channels),
  x = task_pos, y = -0.105, fontface = "bold", family = "sans",
  hjust = 0, size = 4) + annotate("text", label = "Task (channel)",
  x = -0.2, y = -0.09, fontface = "bold", family = "sans",
  size = 4) #+

test_data = as.data.frame(matrix(nrow = 0, ncol = length(data)))
colnames(test_data) = colnames(data)
for (split_num in 1:5) {

```

```

    for (iter_num in 1:5) {
      test = read.csv(paste0("./files/PD_HC/train_test_files/test_split",
        split_num, "_iter", iter_num, ".csv"))
      test_data = rbind(test_data, test)
    }
  }
shap_data = read.csv("./files/PD_HC/top_shap_values.csv")
shap_data = subset(shap_data, select = -c(response))
data_ = subset(test_data, select = -c(PDGP, response))
data_ <- apply(data_, 2, function(x) rank(x))
data_ <- apply(data_, 2, function(x) rescale(x, to = c(0,
  100)))

var_diff <- apply(shap_data, 2, function(x) max(x) - min(x))
var_diff <- var_diff[order(-var_diff)]
df <- data.frame(variable = melt(shap_data)$variable, shap = melt(shap_data)$value,
  variable_value = melt(data_)$value)

top_vars <- factor(names(var_diff))
imp_scores = c()
for (variable in top_vars) {
  score = imp_data$relative_importance[imp_data$variable ==
    variable]
  imp_scores = c(imp_scores, score)
}

df <- df[df$variable %in% top_vars, ]
df$variable <- factor(df$variable, levels = top_vars[order(imp_scores)],
  labels = top_vars[order(imp_scores)])

p_shap = ggplot(df, aes(x = shap, y = variable)) + geom_hline(yintercept = top_vars,
  linetype = "dotted", color = "grey80") + geom_vline(xintercept = 0,
  color = "grey80", size = 1) + geom_point(aes(fill = variable_value),
  color = "#6C6C82", shape = 21, alpha = 0.5, size = 2,
  position = "auto", stroke = 0.1) + scale_fill_gradient2(low = "#2B26CB",
  mid = "white", high = "red", midpoint = 50, breaks = c(0,
  100), labels = c("Low", "High")) + theme_bw() +
  theme(plot.margin = ggplot2::margin(18, 0, 0, -60, "pt"),
  panel.grid.major = element_blank(), panel.grid.minor = element_blank(),
  panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),

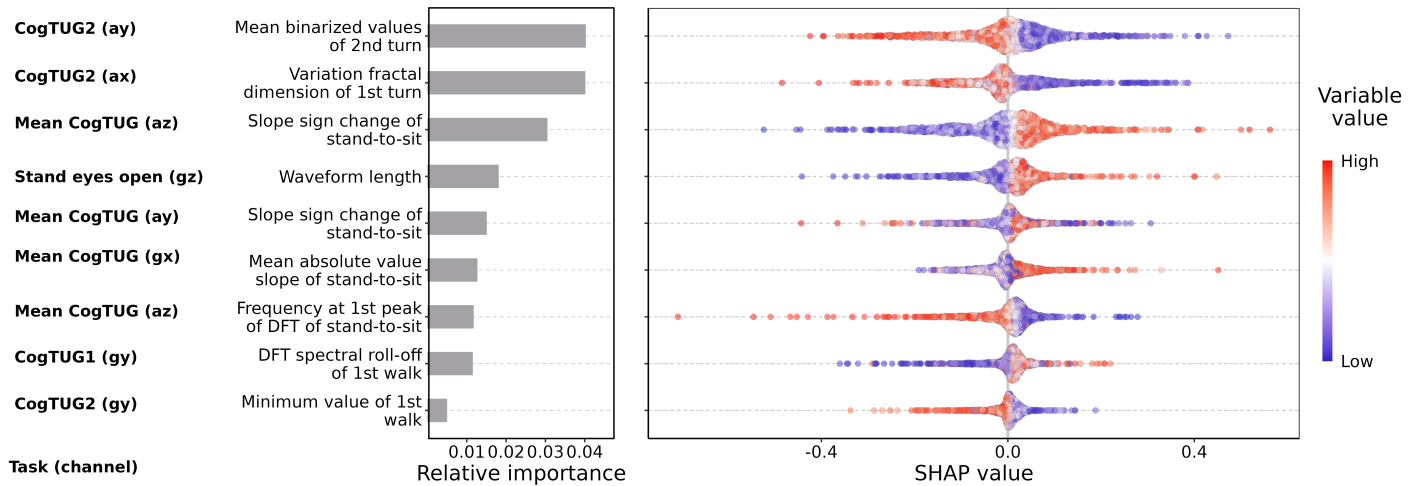
```

```

panel.border = element_rect(color = "black"), axis.line = element_line(color = "black"),
axis.title.x = element_text(colour = "black", size = 15,
  family = "sans"), axis.title.y = element_text(colour = "black",
  size = 15, family = "sans"), axis.text.x = element_text(colour = "black",
  size = 12, family = "sans"), axis.text.y = element_blank(),
legend.title = element_text(size = 15, family = "sans"),
legend.text = element_text(colour = "black", size = 12,
  family = "sans"), legend.title.align = 0.5) +
guides(fill = guide_colourbar("Variable\nvalue\n", ticks = FALSE,
  barheight = 10, barwidth = 0.5), color = "none") +
xlab("SHAP value") + ylab("")

plot_all = arrangeGrob(p_imp, p_shap, ncol = 2)
ggsave(plot_all, file = "figures/PD_HC_imp_plot_bar.png",
  dpi = 500, width = 14, height = 5)

```



5.2.2 Mild PD participants and controls

```

load("./rdata/var_reduct_HY_early_HC_splits.RData")
top_models = c("GBM_model_1684292205921_1340782", "DRF_model_1684292205921_1431707",
  "GBM_model_1684292205921_1572491", "GBM_model_1684292205921_1687878",
  "GBM_model_1684292205921_1810644", "GBM_model_1684334187433_82010",
  "GBM_model_1684334187433_210422", "GBM_model_1684334187433_340011",
  "DRF_model_1684334187433_455088", "GBM_model_1684334187433_592881",
  "GBM_model_1684334187433_717214", "GBM_model_1684334187433_844597",
  "DeepLearning_model_1684334187433_999941", "GBM_model_1684334187433_1060851",
  "GBM_model_1684334187433_1138483", "DeepLearning_model_1684334187433_1242659",

```

```

"GBM_model_1684334187433_1299542", "XGBoost_model_1684334187433_1320794",
"GBM_model_1684334187433_1485635", "GBM_model_1684334187433_1591419",
"DRF_model_1684334187433_1694185", "GBM_model_1684334187433_1832852",
"DeepLearning_model_1684334187433_1918047", "DeepLearning_model_1684334187433_1973941",
"GBM_model_1684334187433_1995264")
save_var_imp(data, all_splits, top_models, "HY_early_HC")

top_models=["GBM_model_1684292205921_1340782", "DRF_model_1684292205921_1431707", \
"GBM_model_1684292205921_1572491", "GBM_model_1684292205921_1687878", \
"GBM_model_1684292205921_1810644", "GBM_model_1684334187433_82010", \
"GBM_model_1684334187433_210422", "GBM_model_1684334187433_340011", \
"DRF_model_1684334187433_455088", "GBM_model_1684334187433_592881", \
"GBM_model_1684334187433_717214", "GBM_model_1684334187433_844597", \
"DeepLearning_model_1684334187433_999941", "GBM_model_1684334187433_1060851", \
"GBM_model_1684334187433_1138483", "DeepLearning_model_1684334187433_1242659", \
"GBM_model_1684334187433_1299542", "XGBoost_model_1684334187433_1320794", \
"GBM_model_1684334187433_1485635", "GBM_model_1684334187433_1591419", \
"DRF_model_1684334187433_1694185", "GBM_model_1684334187433_1832852", \
"DeepLearning_model_1684334187433_1918047", "DeepLearning_model_1684334187433_1973941", \
"GBM_model_1684334187433_1995264"]
data_columns = ["gyfTTP_walk1.t6", "axtVFD_Turn1.t7", "aytMYOP_Turn2.t7", "axtSSC_sTs2.t6t7", \
"aytSSC_sTs2.t6t7", "az_fft1_sTs2.t6t7", "aztSSC_sTs2.t6t7", "gxfSS_sTs2.t6t7", \
"gxtWAMP_sTs2.t6t7", "gxtSSC_sTs2.t6t7", "gytSSC_sTs2.t6t7", "gztSSC_sTs2.t6t7", \
"az_fft3_Turn2.t6t7", "gxfFR.t2", "response"]
save_var_shap(data_columns, top_models, "HY_early_HC")

imp_data = read.csv("../files/HY_early_HC/top_imp_scores.csv")
imp_data = subset(imp_data, select = -X)
imp_data <- imp_data[order(imp_data$relative_importance,
decreasing = TRUE), ]
load("../rdata/var_reduct_HY_early_HC_splits.RData")

rownames(imp_data) = imp_data$variable
top_features_names = c("Variation fractal dimension of 1st turn",
"Mean binarized values of 2nd turn", "Total power of 1st walk",
"DFT frequency ratio", "Frequency at 3rd peak of DFT of 2nd turn",
"Frequency at 1st peak of DFT of stand-to-sit", "Slope sign change of stand-to-sit",
"Skewness of DFT power of stand-to-sit", "Slope sign change of stand-to-sit",
"Slope sign change of stand-to-sit", "Slope sign change of stand-to-sit",
"Rate of change of stand-to-sit", "Slope sign change of stand-to-sit",
"Slope sign change of stand-to-sit")
tasks_channels = c("CogTUG2 (ax)", "CogTUG2 (ay)", "CogTUG1 (gy)",

```

```

    "Stand eyes closed (gx)", "Mean CogTUG (az)", "Mean CogTUG (az)",
    "Mean CogTUG (gz)", "Mean CogTUG (gx)", "Mean CogTUG (ay)",
    "Mean CogTUG (az)", "Mean CogTUG (gy)", "Mean CogTUG (gx)",
    "Mean CogTUG (gx)", "Mean CogTUG (ax)")
task_pos = c(1.15, 2.2, 3.15, 4.2, 5.17, 6.15, 7.15, 8.18,
            9.21, 10.18, 11.05, 12, 13.18, 14.22)

imp_data$Subset <- factor(sapply(rownames(imp_data), function(x) strsplit(x,
split = "\\.").[[1]][2])))
p_imp = ggplot(imp_data, aes(x = reorder(variable, relative_importance),
y = relative_importance)) + geom_bar(stat = "identity",
width = 0.5, fill = "#A4A2A6") + coord_flip(ylim = c(0.003,
max(imp_data$relative_importance) + 0.01), xlim = c(1,
14), clip = "off") + theme_bw() + theme(panel.grid.major = element_blank(),
panel.grid.minor = element_blank(), panel.grid.major.y = element_line(color = "grey80",
size = 0.3, linetype = 2), strip.background = element_blank(),
panel.border = element_rect(color = "black"), legend.text = element_text(color = "black",
size = 10, family = "sans"), plot.margin = ggplot2::margin(0,
65, 0, 150, "pt"), axis.title.x = element_text(color = "black",
size = 15, family = "sans"), axis.title.y = element_text(color = "black",
size = 15, family = "sans"), axis.text.x = element_text(color = "black",
size = 12, family = "sans"), axis.text.y = element_text(color = "black",
size = 12, family = "sans"), legend.title = element_blank(),
legend.position = "top", legend.margin = ggplot2::margin(b = -10)) +
scale_y_continuous(breaks = seq(0.01, 0.06, 0.02)) +
scale_x_discrete(labels = lapply(rev(top_features_names),
str_wrap, width = 25)) + xlab("") + ylab("Relative importance") +
ggtitle("") + annotate("text", label = rev(tasks_channels),
x = task_pos, y = -0.182, fontface = "bold", family = "sans",
hjust = 0, size = 4) + annotate("text", label = "Task (channel)",
x = 0, y = -0.155, fontface = "bold", family = "sans",
size = 4) #+

test_data = as.data.frame(matrix(nrow = 0, ncol = length(data)))
colnames(test_data) = colnames(data)
for (split_num in 1:5) {
  for (iter_num in 1:5) {
    test = read.csv(paste0("./files/HY_early_HC/train_test_files/test_split",
split_num, "_iter", iter_num, ".csv"))
    test_data = rbind(test_data, test)
  }
}

```



```

    }
  }
  shap_data = read.csv("./files/HY_early_HC/top_shap_values.csv")
  shap_data = subset(shap_data, select = -c(response))
  data_ = subset(test_data, select = -c(PDGP, response))
  data_ <- apply(data_, 2, function(x) rank(x))
  data_ <- apply(data_, 2, function(x) rescale(x, to = c(0,
    100)))

  var_diff <- apply(shap_data, 2, function(x) max(x) - min(x))
  var_diff <- var_diff[order(-var_diff)]
  df <- data.frame(variable = melt(shap_data)$variable, shap = melt(shap_data)$value,
    variable_value = melt(data_)$value)

  top_vars <- factor(names(var_diff))
  imp_scores = c()
  for (variable in top_vars) {
    score = imp_data$relative_importance[imp_data$variable ==
      variable]
    imp_scores = c(imp_scores, score)
  }

  df <- df[df$variable %in% top_vars, ]
  df$variable <- factor(df$variable, levels = top_vars[order(imp_scores)],
    labels = top_vars[order(imp_scores)])

  p_shap = ggplot(df, aes(x = shap, y = variable)) + geom_hline(yintercept = top_vars,
    linetype = "dotted", color = "grey80") + geom_vline(xintercept = 0,
    color = "grey80", size = 1) + geom_point(aes(fill = variable_value),
    color = "#6C6C82", shape = 21, alpha = 0.5, size = 2,
    position = "auto", stroke = 0.1) + scale_fill_gradient2(low = "#2B26CB",
    mid = "white", high = "red", midpoint = 50, breaks = c(0,
    100), labels = c("Low", "High")) + theme_bw() +
  theme(plot.margin = ggplot2::margin(18, 0, 0, -60, "pt"),
    panel.grid.major = element_blank(), panel.grid.minor = element_blank(),
    panel.grid.major.y = element_line(color = "grey80",
    size = 0.3, linetype = 2), strip.background = element_blank(),
    panel.border = element_rect(color = "black"), axis.line = element_line(color = "black"),
    axis.title.x = element_text(colour = "black", size = 15,
    family = "sans"), axis.title.y = element_text(colour = "black",
    size = 15, family = "sans"), axis.text.x = element_text(colour = "black",

```



```

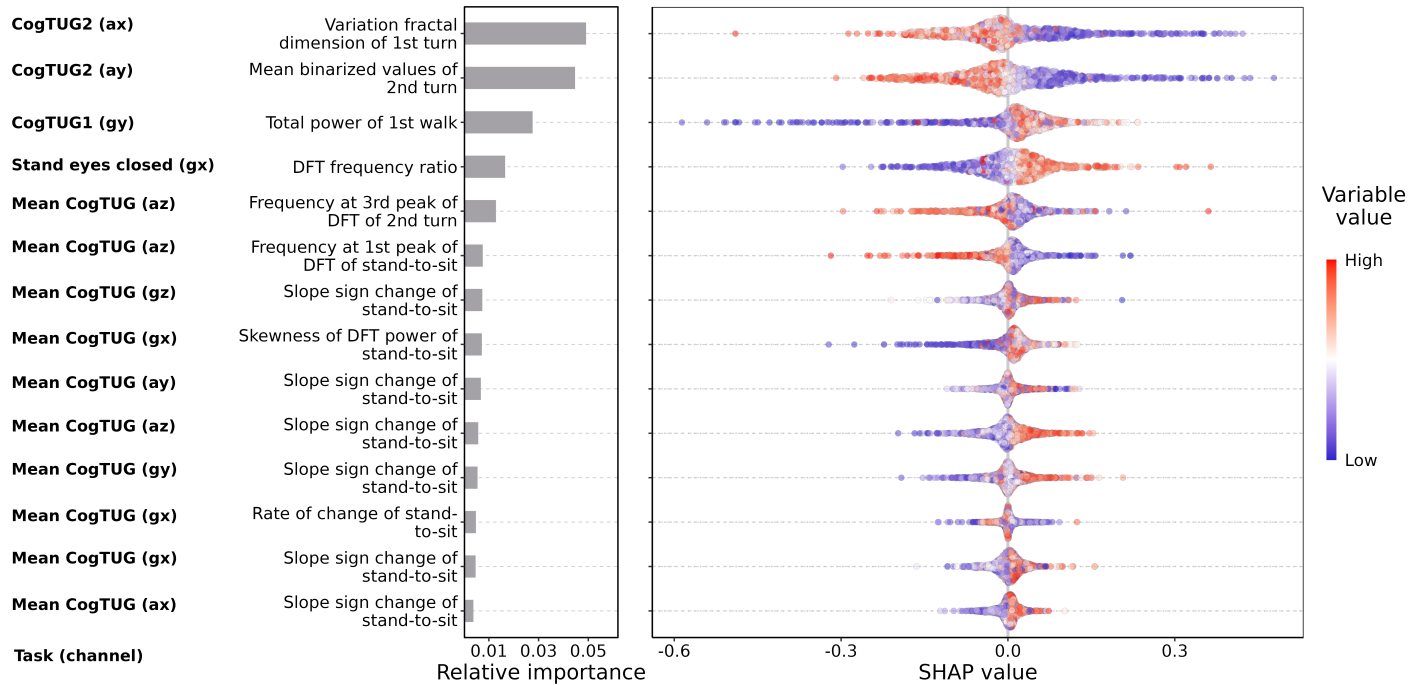
size = 12, family = "sans"), axis.text.y = element_blank(),
legend.title = element_text(size = 15, family = "sans"),
legend.text = element_text(colour = "black", size = 12,
family = "sans"), legend.title.align = 0.5) +
guides(fill = guide_colourbar("Variable\nvalue\n", ticks = FALSE,
barheight = 10, barwidth = 0.5), color = "none") +
xlab("SHAP value") + ylab("")

```

```

plot_all = arrangeGrob(p_imp, p_shap, ncol = 2)
ggsave(plot_all, file = "figures/HY_early_HC_imp_plot_bar.png",
dpi = 500, width = 14, height = 7)

```



5.2.3 Moderate PD participants and controls

```

load("./rdata/var_reduct_HY_mild_HC_splits.RData")
top_models = c("DeepLearning_model_1684292205921_1319234", "DRF_model_1684292205921_1381265",
"DeepLearning_model_1684292205921_1554632", "DeepLearning_model_1684292205921_1670051",
"DeepLearning_model_1684292205921_1793667", "DRF_model_1684334187433_12138",
"DRF_model_1684334187433_123741", "DeepLearning_model_1684334187433_322304",
"GBM_model_1684334187433_393672", "DRF_model_1684334187433_511971",
"DRF_model_1684334187433_636647", "DRF_model_1684334187433_759936",

```

```

    "DeepLearning_model_1684334187433_948250", "DeepLearning_model_1684334187433_1043046",
    "DeepLearning_model_1684334187433_1120152", "DeepLearning_model_1684334187433_1207087",
    "DeepLearning_model_1684334187433_1281223", "DRF_model_1684334187433_1321702",
    "DeepLearning_model_1684334187433_1468319", "DRF_model_1684334187433_1528798",
    "DeepLearning_model_1684431909716_60276", "DeepLearning_model_1684431909716_140703",
    "DRF_model_1684431909716_148995", "DRF_model_1684431909716_226293",
    "DRF_model_1684431909716_297316")
save_var_imp(data, all_splits, top_models, "HY_mild_HC")

top_models=["DeepLearning_model_1684292205921_1319234", "DRF_model_1684292205921_1381265", \
"DeepLearning_model_1684292205921_1554632", "DeepLearning_model_1684292205921_1670051", \
"DeepLearning_model_1684292205921_1793667", "DRF_model_1684334187433_12138", \
"DRF_model_1684334187433_123741", "DeepLearning_model_1684334187433_322304", \
"GBM_model_1684334187433_393672", "DRF_model_1684334187433_511971", \
"DRF_model_1684334187433_636647", "DRF_model_1684334187433_759936", \
"DeepLearning_model_1684334187433_948250", "DeepLearning_model_1684334187433_1043046", \
"DeepLearning_model_1684334187433_1120152", "DeepLearning_model_1684334187433_1207087", \
"DeepLearning_model_1684334187433_1281223", "DRF_model_1684334187433_1321702", \
"DeepLearning_model_1684334187433_1468319", "DRF_model_1684334187433_1528798", \
"DeepLearning_model_1684431909716_60276", "DeepLearning_model_1684431909716_140703", \
"DRF_model_1684431909716_148995", "DRF_model_1684431909716_226293", \
"DRF_model_1684431909716_297316"]
data_columns = ["gytIAV_walk2.t5", "aztSSC_sTs2.t7", "gxfMNP_sTs2.t7", "gytSSI_walk2.t7", \
"az_lsp5_sTs1.t6t7", "oars_a.demo", "oars_i.demo", "oars.t.demo", \
"response"]
save_var_shap(data_columns, top_models, "HY_mild_HC")

imp_data = read.csv("./files/HY_mild_HC/top_imp_scores.csv")
imp_data = subset(imp_data, select = -X)
imp_data <- imp_data[order(imp_data$relative_importance,
    decreasing = TRUE), ]
load("./rdata/var_reduct_HY_mild_HC_splits.RData")

rownames(imp_data) = imp_data$variable
top_features_names = c("Frequency at 5th peak of LSP of sit-to-stand",
    "Slope sign change of stand-to-sit", "Simple square interval of 2nd walk",
    "DFT mean power of stand-to-sit", "OARS total score",
    "Integrated absolute value of 2nd walk", "OARS total ADLs score",
    "OARS total IADLs score")
tasks_channels = c("Mean CogTUG (az)", "CogTUG2 (az)", "CogTUG2 (gy)",
    "CogTUG2 (gx)", "---", "TUG2 (gy)", "---", "---")
task_pos = c(1, 2, 3.1, 4, 5.12, 6.14, 7.15, 8.14)

```

```

imp_data$Subset <- factor(sapply(rownames(imp_data), function(x) strsplit(x,
  split = "\\.")[[1]][2]))
p_imp = ggplot(imp_data, aes(x = reorder(variable, relative_importance),
  y = relative_importance)) + geom_bar(stat = "identity",
  width = 0.3, fill = "#A4A2A6") + coord_flip(ylim = c(0.002,
  max(imp_data$relative_importance) + 0.005), xlim = c(1,
  8), clip = "off") + theme_bw() + theme(panel.grid.major = element_blank(),
  panel.grid.minor = element_blank(), panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),
  panel.border = element_rect(color = "black"), legend.text = element_text(color = "black",
  size = 10, family = "sans"), plot.margin = ggplot2::margin(0,
  65, 0, 150, "pt"), axis.title.x = element_text(color = "black",
  size = 15, family = "sans"), axis.title.y = element_text(color = "black",
  size = 15, family = "sans"), axis.text.x = element_text(color = "black",
  size = 12, family = "sans"), axis.text.y = element_text(color = "black",
  size = 12, family = "sans"), legend.title = element_blank(),
  legend.position = "top", legend.margin = ggplot2::margin(b = -10)) +
  scale_y_continuous(breaks = seq(0.01, 0.05, 0.01)) +
  scale_x_discrete(labels = lapply(rev(top_features_names),
  str_wrap, width = 25)) + xlab("") + ylab("Relative importance") +
  ggtitle("") + annotate("text", label = rev(tasks_channels),
  x = task_pos, y = -0.14, fontface = "bold", family = "sans",
  hjust = 0, size = 4) + annotate("text", label = "Task (channel)",
  x = 0, y = -0.12, fontface = "bold", family = "sans",
  size = 4) #+

test_data = as.data.frame(matrix(nrow = 0, ncol = length(data)))
colnames(test_data) = colnames(data)
for (split_num in 1:5) {
  for (iter_num in 1:5) {
    test = read.csv(paste0("./files/HY_mild_HC/train_test_files/test_split",
      split_num, "_iter", iter_num, ".csv"))
    test_data = rbind(test_data, test)
  }
}
shap_data = read.csv("./files/HY_mild_HC/top_shap_values.csv")
shap_data = subset(shap_data, select = -c(response))
data_ = subset(test_data, select = -c(PDGP, response))
data_ <- apply(data_, 2, function(x) rank(x))

```

```

data_ <- apply(data_, 2, function(x) rescale(x, to = c(0,
  100)))

var_diff <- apply(shap_data, 2, function(x) max(x) - min(x))
var_diff <- var_diff[order(-var_diff)]
df <- data.frame(variable = melt(shap_data)$variable, shap = melt(shap_data)$value,
  variable_value = melt(data_)$value)

top_vars <- factor(names(var_diff))
imp_scores = c()
for (variable in top_vars) {
  score = imp_data$relative_importance[imp_data$variable ==
    variable]
  imp_scores = c(imp_scores, score)
}

df <- df[df$variable %in% top_vars, ]
df$variable <- factor(df$variable, levels = top_vars[order(imp_scores)],
  labels = top_vars[order(imp_scores)])

p_shap = ggplot(df, aes(x = shap, y = variable)) + geom_hline(yintercept = top_vars,
  linetype = "dotted", color = "grey80") + geom_vline(xintercept = 0,
  color = "grey80", size = 1) + geom_point(aes(fill = variable_value),
  color = "#6C6C82", shape = 21, alpha = 0.5, size = 2,
  position = "auto", stroke = 0.1) + scale_fill_gradient2(low = "#2B26CB",
  mid = "white", high = "red", midpoint = 50, breaks = c(0,
  100), labels = c("Low", "High")) + theme_bw() +
  theme(plot.margin = ggplot2::margin(18, 0, 0, -60, "pt"),
  panel.grid.major = element_blank(), panel.grid.minor = element_blank(),
  panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),
  panel.border = element_rect(color = "black"), axis.line = element_line(color = "black"),
  axis.title.x = element_text(colour = "black", size = 15,
  family = "sans"), axis.title.y = element_text(colour = "black",
  size = 15, family = "sans"), axis.text.x = element_text(colour = "black",
  size = 12, family = "sans"), axis.text.y = element_blank(),
  legend.title = element_text(size = 15, family = "sans"),
  legend.text = element_text(colour = "black", size = 12,
  family = "sans"), legend.title.align = 0.5) +
  guides(fill = guide_colourbar("Variable\nvalue\n", ticks = FALSE,
  barheight = 10, barwidth = 0.5), color = "none") +

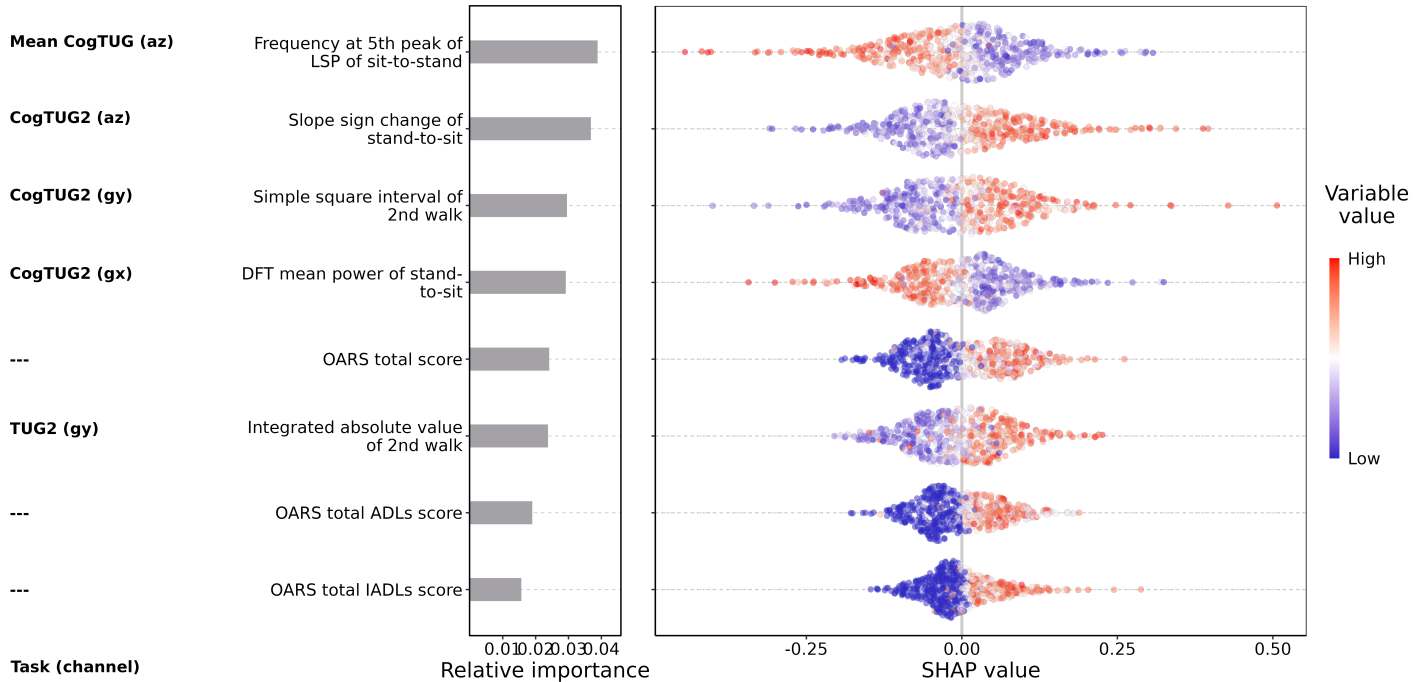
```

```

xlab("SHAP value") + ylab("")

plot_all = arrangeGrob(p_imp, p_shap, ncol = 2)
ggsave(plot_all, file = "figures/HY_mild_HC_imp_plot_bar.png",
       dpi = 500, width = 14, height = 7)

```



5.2.4 Severe PD participants and controls

```

load("./rdata/var_reduct_HY_severe_HC_splits.RData")
top_models = c("GBM_model_1684381314577_68957", "DeepLearning_model_1684381314577_261294",
  "DRF_model_1684381314577_323560", "DeepLearning_model_1684381314577_524854",
  "GBM_model_1684381314577_603296", "DRF_model_1684381314577_717893",
  "DRF_model_1684381314577_847473", "DeepLearning_model_1684381314577_1060632",
  "DeepLearning_model_1684381314577_1192576", "DeepLearning_model_1684381314577_1317450",
  "DeepLearning_model_1684381314577_1446169", "DRF_model_1684381314577_1504662",
  "DRF_model_1684381314577_1622170", "GBM_model_1684381314577_1753250",
  "GBM_model_1684381314577_1869370", "DeepLearning_model_1684421828027_111476",
  "DeepLearning_model_1684421828027_206688", "GBM_model_1684421828027_271687",
  "GBM_model_1684421828027_408646", "GLM_model_1684421828027_529561",
  "DeepLearning_model_1684421828027_734593", "DeepLearning_model_1684421828027_847931",
  "DeepLearning_model_1684421828027_963294", "DeepLearning_model_1684421828027_1095195",

```

```

"GLM_model_1684421828027_1158212")
save_var_imp(data, all_splits, top_models, "HY_severe_HC")

top_models=["GBM_model_1684381314577_68957", "DeepLearning_model_1684381314577_261294", \
"DRF_model_1684381314577_323560", "DeepLearning_model_1684381314577_524854", \
"GBM_model_1684381314577_603296", "DRF_model_1684381314577_717893", \
"DRF_model_1684381314577_847473", "DeepLearning_model_1684381314577_1060632", \
"DeepLearning_model_1684381314577_1192576", "DeepLearning_model_1684381314577_1317450", \
"DeepLearning_model_1684381314577_1446169", "DRF_model_1684381314577_1504662", \
"DRF_model_1684381314577_1622170", "GBM_model_1684381314577_1753250", \
"GBM_model_1684381314577_1869370", "DeepLearning_model_1684421828027_111476", \
"DeepLearning_model_1684421828027_206688", "GBM_model_1684421828027_271687", \
"GBM_model_1684421828027_408646", "GLM_model_1684421828027_529561", \
"DeepLearning_model_1684421828027_734593", "DeepLearning_model_1684421828027_847931", \
"DeepLearning_model_1684421828027_963294", "DeepLearning_model_1684421828027_1095195", \
"GLM_model_1684421828027_1158212"]
data_columns = ["aztSSI_sTs2.t6", "gxtIAV_walk1.t7", "aztIAV_sTs2.t6t7", "oars.t.demo", \
"response"]
save_var_shap(data_columns, top_models, "HY_severe_HC")

imp_data = read.csv("./files/HY_severe_HC/top_imp_scores.csv")
imp_data = subset(imp_data, select = -X)
imp_data <- imp_data[order(imp_data$relative_importance,
    decreasing = TRUE), ]
load("./rdata/var_reduct_HY_severe_HC_splits.RData")

rownames(imp_data) = imp_data$variable
top_features_names = c("Integrated absolute value of 1st walk",
    "OARS total score", "Simple square interval of stand-to-sit",
    "Integrated absolute value of stand-to-sit")
tasks_channels = c("CogTUG2 (gx)", "---", "CogTUG1 (az)",
    "Mean CogTUG (az)")
task_pos = c(1.05, 2, 3.05, 4.05)

imp_data$Subset <- factor(sapply(rownames(imp_data), function(x) strsplit(x,
    split = "\\.")[[1]][2]))
p_imp = ggplot(imp_data, aes(x = reorder(variable, relative_importance),
    y = relative_importance)) + geom_bar(stat = "identity",
    width = 0.2, fill = "#A4A2A6") + coord_flip(ylim = c(0.005,
    max(imp_data$relative_importance) + 0.005), xlim = c(1,
    4), clip = "off") + theme_bw() + theme(panel.grid.major = element_blank(),

```

```

panel.grid.minor = element_blank(), panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),
panel.border = element_rect(color = "black"), legend.text = element_text(color = "black",
  size = 10, family = "sans"), plot.margin = ggplot2::margin(0,
  65, 0, 150, "pt"), axis.title.x = element_text(color = "black",
  size = 15, family = "sans"), axis.title.y = element_text(color = "black",
  size = 15, family = "sans"), axis.text.x = element_text(color = "black",
  size = 12, family = "sans"), axis.text.y = element_text(color = "black",
  size = 12, family = "sans"), legend.title = element_blank(),
legend.position = "top", legend.margin = ggplot2::margin(b = -10)) +
scale_y_continuous(breaks = seq(0.01, 0.15, 0.03)) +
scale_x_discrete(labels = lapply(rev(top_features_names),
  str_wrap, width = 25)) + xlab("") + ylab("Relative importance") +
ggtitle("") + annotate("text", label = rev(tasks_channels),
x = task_pos, y = -0.4, fontface = "bold", family = "sans",
hjust = 0, size = 4) + annotate("text", label = "Task (channel)",
x = 0.22, y = -0.34, fontface = "bold", family = "sans",
size = 4) #+

test_data = as.data.frame(matrix(nrow = 0, ncol = length(data)))
colnames(test_data) = colnames(data)
for (split_num in 1:5) {
  for (iter_num in 1:5) {
    test = read.csv(paste0("./files/HY_severe_HC/train_test_files/test_split",
      split_num, "_iter", iter_num, ".csv"))
    test_data = rbind(test_data, test)
  }
}

shap_data = read.csv("./files/HY_severe_HC/top_shap_values.csv")
shap_data = subset(shap_data, select = -c(response))
data_ = subset(test_data, select = -c(PDGP, response))
data_ <- apply(data_, 2, function(x) rank(x))
data_ <- apply(data_, 2, function(x) rescale(x, to = c(0,
  100)))

var_diff <- apply(shap_data, 2, function(x) max(x) - min(x))
var_diff <- var_diff[order(-var_diff)]
df <- data.frame(variable = melt(shap_data)$variable, shap = melt(shap_data)$value,
  variable_value = melt(data_)$value)

top_vars <- factor(names(var_diff))

```



```

imp_scores = c()
for (variable in top_vars) {
  score = imp_data$relative_importance[imp_data$variable ==
    variable]
  imp_scores = c(imp_scores, score)
}

df <- df[df$variable %in% top_vars, ]
df$variable <- factor(df$variable, levels = top_vars[order(imp_scores)],
  labels = top_vars[order(imp_scores)])

p_shap = ggplot(df, aes(x = shap, y = variable)) + geom_hline(yintercept = top_vars,
  linetype = "dotted", color = "grey80") + geom_vline(xintercept = 0,
  color = "grey80", size = 1) + geom_point(aes(fill = variable_value),
  color = "#6C6C82", shape = 21, alpha = 0.5, size = 2,
  position = "auto", stroke = 0.1) + scale_fill_gradient2(low = "#2B26CB",
  mid = "white", high = "red", midpoint = 50, breaks = c(0,
  100), labels = c("Low", "High")) + theme_bw() +
  theme(plot.margin = ggplot2::margin(18, 0, 0, -60, "pt"),
  panel.grid.major = element_blank(), panel.grid.minor = element_blank(),
  panel.grid.major.y = element_line(color = "grey80",
  size = 0.3, linetype = 2), strip.background = element_blank(),
  panel.border = element_rect(color = "black"), axis.line = element_line(color = "black"),
  axis.title.x = element_text(colour = "black", size = 15,
  family = "sans"), axis.title.y = element_text(colour = "black",
  size = 15, family = "sans"), axis.text.x = element_text(colour = "black",
  size = 12, family = "sans"), axis.text.y = element_blank(),
  legend.title = element_text(size = 15, family = "sans"),
  legend.text = element_text(colour = "black", size = 12,
  family = "sans"), legend.title.align = 0.5) +
  guides(fill = guide_colourbar("Variable\nvalue\n", ticks = FALSE,
  barheight = 10, barwidth = 0.5), color = "none") +
  xlab("SHAP value") + ylab("")

plot_all = arrangeGrob(p_imp, p_shap, ncol = 2)
ggsave(plot_all, file = "figures/HY_severe_HC_imp_plot_bar.png",
  dpi = 500, width = 14, height = 7)

```