Appendix 1: The MATLAB Code Used for this Project

Table of Contents

"main.m"

```
clear;      % clear the workspace
clc;        % clear the command window
close all;  % close all popup windows

% ======================================
% ================ TASK 1 ==============
% ======================================

fprintf("\n--------\n Importing the data...\n--------\n")
rawData = importDatasets();

% [OPTIONAL] Try out different params for the low pass filter
% experimentFilters(rawData, 7, 100)

fprintf("\n--------\n Filtering the data...\n--------\n")
filteredRawData = filterData(rawData);

fprintf("\n--------\n Extracting the time-domain features...\n--------
\n")
processedData = extractFeatures(filteredRawData);

fprintf("\n--------\n Organising the features...\n--------\n")
arrayPerActivity = organiseFeatures(processedData);

fprintf("\n--------\n Labelling the data...\n--------\n")
[unlabelledData, classLabels] = getUnlabelledData(arrayPerActivity);
[labelledData] = getLabelledData(arrayPerActivity);
```

```
% [OPTIONAL] Plot the 7 extracted time domain features of a section
% plotTimeDomain("foot_l_gyro_x", labelledData)

fprintf("\n--------\n Training the ANN using all the features...
\n--------\n")
allFeaturesNNAccuracy = crossValidateNN(labelledData, 5,
 201, "trainscg");
close all

fprintf("\n--------\n Training the SVM using all the features...
\n--------\n")
allFeaturesSVMAccuracy = svmPosterior(labelledData,
 classLabels, "polynomial", 1);


% ======================================
% ================ TASK 2 ==============
% ======================================

% delete vars no longer needed to save memory
clearvars -except labelledData unlabelledData classLabels

fprintf("\n--------\n Finding the top 15 features...\n--------\n")
fifteenFeaturesLabelled = find15Features(labelledData, unlabelledData,
 classLabels);

% [OPTIONAL] Train a NN for the top 15 only
% fprintf("\n--------\n Training the ANN using 15 features...
\n--------\n")
% fifteenFeaturesNNAccuracy = crossValidateNN(fifteenFeaturesLabelled,
 5, 10, "trainscg");

% [OPTIONAL] Train an SVM for the top 15 only
% fprintf("\n--------\n Training the SVM using 15 features...
\n--------\n")
% fifteenFeaturesSVMAccuracy = svmPosterior(fifteenFeaturesLabelled,
 classLabels, "polynomial", 1);

% ======================================
% ================ TASK 3 ==============
% ======================================

fprintf("\n--------\n Finding features from a single segment...
\n--------\n")
[segmentFeaturesLabelled] = extractSegment("thigh_r", labelledData);

% ======================================
% ================ TASK 4 ==============
% ======================================

fprintf("\n--------\n Training the ANN using features from a single
 segment...\n--------\n")
singleSegmentNNAccuracy = crossValidateNN(segmentFeaturesLabelled, 5,
 35, "trainscg");
```

```
fprintf("\n--------\n Training the SVM using features from a single
 segment...\n--------\n")
singleSegmentSVMAccuracy = svmPosterior(segmentFeaturesLabelled,
 classLabels, "polynomial", 1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"importDatasets.m"

```
%{
This function imports all the datasets and deletes all timestamp
 columns
apart from the very first one in each dataset and any columns with
 values of 0.

Returns
- `rawData`    -> a 1x12 struct with 5 fields ("LGW","RA","RD","SiS"
 and "StS")
%}

function [rawData] = importDatasets()
    % initialisations
    previous_is_timestamp = false;
    columnsToDelete = ("yes");

    % array containing the names of the folders. These names will
 match the
    % field names in the struct
    folders = ["LGW","RA","RD","SiS","StS"];
    % loop through each of the folders
    for ff = 1 : length(folders)
        % specify the folder we are interested in during this loop
        myFolder = folders(ff);
        % Get a list of all .dat files in the folder
        filePattern = fullfile(myFolder, '*.dat');
        theFiles = dir(filePattern);
        % We now want to loop through all the files in the current
 folder of
        % interest
        for kk = 1 : length(theFiles)
          baseFileName = theFiles(kk).name;
          fullFileName = fullfile(myFolder, baseFileName);
          fprintf(1, 'Now reading %s\n', fullFileName);
          % read the .dat file into a table variable
          dataset = readtable(fullFileName, "ReadVariableNames",true);
          % read the .dat file also into a cell variable to allow
 retrieval of
          % column details such as data type and units (seconds,
 milliseconds, etc)
          rawDataset = readcell(fullFileName);
          % ---------------------------------------------
          % loop through each column in the dataset to remove those
 with synchronisation time vectors
          % ---------------------------------------------
          for col = 1 : width(dataset)
              thisColumn = dataset(:, col); % Extract this one column
 into its own variable.
              avg = abs(nanmean(thisColumn{:,end}));
              % should delete columns with an average this high as it
 means
              % values in them are a timestamp. Except for a single
 column for
              % each IMU which we keep
              if (col>1) && ((avg > 1000) || avg == 0)
                 columnsToDelete(end+1) =
 dataset.Properties.VariableNames{col};
              end
          end
          % Before moving on to the next .dat file, we want to delete
          % all the columns in 'columnsToDelete' from this file
          if length(columnsToDelete) > 1
              % delete the "yes" element from the array
              columnsToDelete(1) = [];
              % delete the relevant columns from the dataset
              dataset = removevars(dataset,columnsToDelete);
          end
          % restart the columnsToDelete list
          columnsToDelete = ("yes");
          % ---------------------------------------------
          % append the contents of the .dat file to the struct under
 the
          % relevant field name and number
          % ---------------------------------------------
          rawData(kk).(folders{ff}) = dataset;
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"filterData.m"

```
%{
This function takes in the struct containing all the raw data and runs
 it
through a low pass filter.

Arguments:
- `rawData`    -> struct containing the data imported from the folders

Returns:
- `rawData`    -> updated struct containing the filtered data
%}

function rawData = filterData(rawData)
    % create a low pass filter
    filter_Fs = 100;             % filter sampling rate
    filter_Fc = 7;               % cutoff frequency in Hz
    % create the low pass filter
    d = designfilt('lowpassfir', 'FilterOrder', 8, 'CutoffFrequency',
 filter_Fc, 'SampleRate', filter_Fs);
    % ---------------------------------------------
    % Loop through all of the rawData and apply the filter on each
 column
    % ---------------------------------------------
    % array containing the names of the folders. These names will
 match the
    % field names in the struct
    sets = ["LGW","RA","RD","SiS","StS"];
    % loop through each of the folders
    for ff = 1 : length(sets)
        for kk = 1 : length(rawData)
            % sequentially extract a single dataset
            current_dataset = rawData(kk).(sets{ff});
            % Filter the data
            % loop through the columns in the single dataset
            for ii = 1 : width(current_dataset)
                % obtain the relevant column
                colm = table2array(current_dataset(1:end,ii));
                % ignore timestamp columns
                avg = abs(nanmean(colm));
                % if columns is NOT a timestamp one and not a 0 value
 one
                if (avg < 1000) && (avg ~= 0)
                    % apply the filter on the column
                    filtered_Y = filter(d, colm);
                    % update the column in the strut with the filtered
 data
                    rawData(kk).(sets{ff})(:,ii) =
 array2table(filtered_Y);
                end
            end
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"extractFeatures.m"

```
%{
This file takes in the struct containing the filtered data and extract
 all
the time-domain features:
    1. Maximum
    2. Minimum
    3. Mean
    4. Standard deviation
    5. Root means square (RMS)
    6. Zero crossing
    7. Maximum slope changes

    The notations for them: "MAX", "MIN", "AVG", "SD", "RMS", "ZC",
 "MSC".

Arguments:
- `filteredRawData`   -> the struct with the raw data after going
 through
                           the low pass filter.

Returns:
- `processedData`  -> struct with the time-domain features extracted
                        according to the given time window and
 interval
%}

function processedData = extractFeatures(filteredRawData)
    % ----------------------------------------------
    % Loop through all of the filteredRawData, extract max, min, mean,
 standard
    % deviation and RMS.
    fprintf("\nExtracting time domain features...\n")
    % ----------------------------------------------
    % array containing the names of the activities.
    % These names will match the field names in the struct
    sets = ["LGW","RA","RD","SiS","StS"];
    % array containing the names of the time domain features.
    % These names will match the field names in the struct.
    features = ["MAX", "MIN", "AVG", "SD","RMS"];
    % size of the sliding window for extracting features in
 milliseconds
    window_duration = 400;
    % define the time interval required in ms
    timeInterval = 50;
    % loop through each of the folders
    for ff = 1 : length(sets)
        for kk = 1 : length(filteredRawData)
            current_dataset = table2array(filteredRawData(kk).
(sets{ff}));
            current_dataset_without_timestamp =
 current_dataset(:,2:end);
            current_timestamp = current_dataset(:,1);

            % -------------------------------------------------------
            % We should not assume that the timestep between samples
 is fixed.
            % Therefore, we find the average timestep for every
 dataset and
```

```
            % change the window size accordingly.

            % find the average timestep between rows
            avg_timestep = mean(diff(current_timestamp));
            % in case the timestep is in seconds rather than
milliseconds
            if avg_timestep < 1
                avg_timestep = avg_timestep*1000;
            end
            % `window_size` defines the number of readings in each
window
            window_size = int32(window_duration/avg_timestep);
            %--------------------------------------------------------

            % extract time domain features from each dataset,
discarding
            % endpoints to have all windows exactly the length they
should be.
            smean = movmean(current_dataset_without_timestamp,
window_size, 'Endpoints','discard');
            stdD = movstd(current_dataset_without_timestamp,
window_size, 'Endpoints','discard');
            maxV = movmax(current_dataset_without_timestamp,
window_size, 'Endpoints','discard');
            min = movmin(current_dataset_without_timestamp,
window_size, 'Endpoints','discard');
            rms = sqrt(movmean(current_dataset_without_timestamp .^ 2,
window_size, 'Endpoints','discard'));
            % group all the features
            dataset_features = {maxV, min, smean, stdD, rms};
            % assign all the features to a single new struct
            for w = 1 : length(dataset_features)
                % reduce the data using the required time interval.
                % We want to extract only every Nth row to match our
time
                % interval. The 'interval' var defines N.
                [reduced_data, interval] =
reduceData(current_timestamp, dataset_features{1,w}, timeInterval);
                temp_processedData(1).(features{w}) = reduced_data;
            end
            processedData(kk).(sets{ff}) = temp_processedData;
        end
    end

    % ----------------------------------------------
    % Loop through all of the filteredRawData, extract Zero Crossing.
    fprintf("\nManually extracting Zero Crossing...\n")
    % ----------------------------------------------
    % extract zero crossing for the data and add to the same
processedData
    % struct
    % loop through each of the folders
    for ff = 1 : length(sets)
        for kk = 1 : length(filteredRawData)
```

```
            % fprintf("\nIn set number %i and dataset number %i, in
set %s\n", ff, kk, sets(ff))
            current_dataset = filteredRawData(kk).(sets{ff});
            current_dataset_without_timestamp =
current_dataset(:,2:end);
            current_timestamp = table2array(current_dataset(:,1));
            % -------------------------------------------------------
            % We should not assume that the timestep between samples
is fixed.
            % Therefore, we find the average timestep for every
dataset and
            % change the window size accordingly.

            % find the average timestep between rows
            avg_timestep = mean(diff(current_timestamp));
            % in case the timestep is in seconds rather than
milliseconds
            if avg_timestep < 1
                avg_timestep = avg_timestep*1000;
            end
            % `window_size` defines the number of readings in each
window
            window_size = int16(window_duration/avg_timestep);
            % define the half window size according to whether the
window is
            % odd or even
            if rem(window_size,2) == 0
                % if window size is even
                half_window_size = int16(window_size/2);
            else
                % if window size is odd
                half_window_size = int16((window_size-1)/2);
            end
            %-------------------------------------------------------
            % Filter the data
            % loop through the columns in the single dataset
            clearvars zc_dataset
            for ii = 1 : width(current_dataset_without_timestamp)
                % obtain the relevant column
                colm =
table2array(current_dataset_without_timestamp(1:end,ii));
                % calculate zero crossing manually
                % (for each window, 1 if a ZC exists, 0 if not)
                zc = false;
                clearvars zc_column
                % loop through the column
                for r = 1 : interval :length(colm)
                    if length(colm) < (window_size+2)
                        % if the column is smaller than the window
size then ignore
                        fprintf("\nHasal?\n")
                        continue
                    elseif r > (length(colm)-(half_window_size))
```

```
                        % if towards the end of the column, ignore the
 window
                        continue
                    elseif r < (half_window_size+1)
                        % if towards the start of the column, ignore
 the window
                        continue
                    else
                        if rem(window_size,2) == 0
                            % if window size is even
                            g = (window_size-2)/2;
                            h = colm(r-(g+1):r+g);
                        else
                            % if window size is odd
                            h = colm(r-half_window_size:r
+half_window_size);
                        end
                    end
                    % loop through h and see if a ZC exists
                    for rr = 1 : length(h)-1
                        if (h(rr)*h(rr+1))<0
                            zc = true;
                        end
                    end
                    % if we had found a ZC then we want to assign 1,
 if not
                    % then 0. Indexing `sequentialIndex` instead of r
 as r is
                    % increasing by non-1 increments.
                    sequentialIndex = ((r-1)/interval)-3;
                    if zc
                        zc_column(sequentialIndex) = 1;
                    else
                        zc_column(sequentialIndex) = 0;
                    end
                    zc = false;
                end
                % append the zc_column to the existing zc table
                zc_dataset(:,ii) = zc_column;
            end
            processedData(kk).(sets{ff})(1).ZC = zc_dataset;
        end
    end



    % ----------------------------------------------
    % Loop through all of the filteredRawData, extract Maximum Slope
 Change.
    fprintf("\nManually extracting MSC...\n")
    % ----------------------------------------------
    % extract zero crossing for the data and add to the same
 processedData
    % struct
    % loop through each of the folders
    for ff = 1 : length(sets)
        for kk = 1 : length(filteredRawData)
            current_dataset = filteredRawData(kk).(sets{ff});
            current_dataset_without_timestamp =
current_dataset(:,2:end);
            current_timestamp = table2array(current_dataset(:,1));
            % -------------------------------------------------------
            % We should not assume that the timestep between samples
is fixed.
            % Therefore, we find the average timestep for every
dataset and
            % change the window size accordingly.

            % find the average timestep between rows
            avg_timestep = mean(diff(current_timestamp));
            % in case the timestep is in seconds rather than
milliseconds
            if avg_timestep < 1
                avg_timestep = avg_timestep*1000;
            end
            % `window_size` defines the number of readings in each
window
            window_size = int16(window_duration/avg_timestep);
            % define the half window size according to whether the
window is
            % odd or even
            if rem(window_size,2) == 0
                % if window size is even
                half_window_size = int16(window_size/2);
            else
                % if window size is odd
                half_window_size = int16((window_size-1)/2);
            end
            %-------------------------------------------------------
            % Filter the data
            % loop through the columns in the single dataset
            clearvars ms_dataset
            for ii = 1 : width(current_dataset_without_timestamp)
                % obtain the relevant column
                colm =
table2array(current_dataset_without_timestamp(1:end,ii));
                clearvars ms_column
                % loop through the column
                for r = 1 : interval :length(colm)
                    if length(colm) < (window_size+2)
                        % if the column is smaller than the window
size then ignore
                        continue
                    elseif r > (length(colm)-(half_window_size))
                        % if towards the end of the column, ignore the
window
                        continue
                    elseif r < (half_window_size+1)
```

```
                            % if towards the start of the column, ignore
 the window
                            continue
                        else
                            % under normal conditions, take window/2
 values before
                            % r and window/2 values after it
                            if rem(window_size,2) == 0
                                g = (window_size-2)/2;
                                % if window size is evn
                                h = colm(r-(g+1):r+g);
                                timeColum = current_timestamp(r-(g+1):r
+g);
                            else
                                % if window size is odd
                                h = colm(r-half_window_size:r
+half_window_size);
                                timeColum = current_timestamp(r-
half_window_size:r+half_window_size);
                            end
                        end
                        sequentialIndex = ((r-1)/interval)-3;
                        % gradient() finds the slope change between
 consequetive
                        % data points.
                        dydx = gradient(h) ./ gradient(timeColum);
                        % we now need to find the differences between
 consequtive
                        % gradients, then find the maximum value i.e. max
 slope
                        % change
                        maxSlopeChange = max(diff(dydx));
                        ms_column(sequentialIndex) = maxSlopeChange;
                    end
                    % append the ms_column to the existing ms table
                    ms_dataset(:,ii) = ms_column;
                end
                processedData(kk).(sets{ff})(1).MSC = ms_dataset;
            end
        end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"reduceData.m"

```
%{
This function calculates the timestep between each row using the
 timestamp column and
deletes the rows that are not needed from the extracted features
 table.
Arguments
- `timestampColumn`      -> the timestamp column for a single dataset
- `dataTable`            -> table containing an extracted feature (e.g.
 mean)
- `timeInterval`         -> required time interval (delta t) in
 milliseconds

Returns
- `reducedData`          -> the time domain data after reducing it
 using the
given delta t.
- `interval`             -> the number of readings between each
 interval
%}

function [reducedData, interval]=reduceData(timestampColumn,dataTable,
 timeInterval)

    % the max and min time increment steps allowed between
 consequetive
    % readings. If a step beyond the allowed limits is found an error
 is
    % raised. These are in milliseconds.
    maxStepAllowed_ms = 11;
    minStepAllowed_ms = 9;

    % find the average timestep between rows
    avg_timestep = mean(diff(timestampColumn));

    % differentiate between seconds and milliseconds and define the
 interval
    % jump based on that. The 'interval' variable will define the Nth
 row to
    % take from the extracted data
    if (avg_timestep > minStepAllowed_ms) && (avg_timestep <
 maxStepAllowed_ms)
        % timestep is in milliseconds
        interval = int16(timeInterval/avg_timestep);
    elseif (avg_timestep > minStepAllowed_ms/1000) && (avg_timestep <
 maxStepAllowed_ms/1000)
        % timestep is in seconds
        interval = int16((timeInterval/1000)/avg_timestep);
    else
        fprintf("\nError - cannot determine the timestep unit: %f\n",
 avg_timestep)
    end

    % loop through the data and only take the relevant rows (e.g.
 every fifth row)
    for ii = 1 : interval : length(dataTable)
        index = (((ii-1)/5)+1);
        incrementing_reducedData(index,:) = dataTable(index, :);
    end
    % return the reduced data
    reducedData = incrementing_reducedData;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"organiseFeatures.m"

```
%{
This function takes in the extracted time-domain features and
 organises the data:
1. For each single dataset (each body part) we concatenate
 horizontally
    all the features after removing the columns (features) that are
 not
    needed eg magnetometere
2. We then concatetane vertically all training samples
    -> result in having a single matrix per activity that will have
 all
    training samples of all features + parts of body
3. We delete the columns containing magnetometer data as they are not
relevant to the purpose of the classifiers we are building

Arguments:
- `processedData`  -> the struct containing the extracted and reduced
time-domain features

Returns:
- `arrayPerActivity` -> struct containing a single array per
activity without any magnetometer data.
%}

function arrayPerActivity = organiseFeatures(processedData)

    % array containing the names of the activities.
    % These names will match the field names in the struct
    sets = ["LGW","RA","RD","SiS","StS"];
    % array containing the names of the time domain features.
    % These names will match the field names in the struct.
    features = ["MAX", "MIN", "AVG", "SD","RMS", "ZC", "MSC"];
    % define the IMU data to include in the data
    included_readings = ["gyro", "accel", "magnet"];

    % ----------------------------------------------
    % 1. Loop through the processed data and horizonatally concatenate
    % all the features
    % ----------------------------------------------
    for ff = 1 : length(sets)
        for kk = 1 : length(processedData)
            % sample is the struct containing the 7 tables
            % (one for each time feature)
            sample = processedData(kk).(sets{ff});
            % extract each of the 7 features
            sample_max = sample(1).MAX;
            sample_min = sample(1).MIN;
            sample_avg = sample(1).AVG;
            sample_sd = sample(1).SD;
            sample_rms = sample(1).RMS;
            sample_zc = sample(1).ZC;
            sample_msc = sample(1).MSC;
            % concatenate horizontally all of the features for this
dataset
            sample_feature_collection = [sample_max, sample_min,
sample_avg, sample_sd, sample_rms, sample_zc, sample_msc];
            features_collected(kk).(sets{ff}) =
sample_feature_collection;
        end
    end

    % ---------------------------------------------
    % 2. Loop through the features and vertically all the training
samples for
    % each activity. Result in a single table for each activity
    % ---------------------------------------------
    % loop through each of the folders
    for ff = 1 : length(sets)
        for kk = 1 : length(features_collected)
            % sample is the table containing the concatenated features
as
            % a single table
            sample = features_collected(kk).(sets{ff});
            if kk == 1
                growing_activity_data = sample;
            else
                % vertically concatenate all the activity's data
                growing_activity_data =
vertcat(growing_activity_data,sample);
            end
        end
        % array_per_activity is a struct that will contain a table for
each
        % activity.
        array_per_activity(1).(sets{ff}) = growing_activity_data;
        % reset the variable for the next activity
        clearvars growing_activity_data
    end


    % ---------------------------------------------
    % 3. Remove the magnetometer columns from the data
    % ---------------------------------------------
    fprintf("\nExcluding Magnetometer data...\n")
    for ff = 1 : length(sets)
        sample = array_per_activity(1).(sets{ff});
        % DUPLICATE SAMPLE
        update_sample = sample;
        sample_dim = size(update_sample);
        for ii=6 : 6 : sample_dim(2)
            % if deleting magnetometer data, find every sixth column
 and delete the
            % next 3 after it.
            % fprintf("\ncolumn %i of %i\n", ii, sample_dim(2));
            update_sample(:,ii+1) = [];
            update_sample(:,ii+1) = [];
            update_sample(:,ii+1) = [];
            % update the width because the array shrinks as we go
 along
            sample_dim = size(update_sample);
            % manually exit the loop when we reach the end
            if ii == sample_dim(2)
                break
            end
        end
        % update the table for each activity with the new one without
 the
        % magnetometer data
        arrayPerActivity(1).(sets{ff}) = update_sample;
    end
    %{
    Each activity now has one array in arrayPerActivity.
    Each of those array has a size of Nx294
        63 columns from the raw data after removing the timestamps X
        7 extracted time domain features X
        (2/3) getting rid of magnetometer readings (deleting 3 in
 every 9 columns)
        so 63 x 7 x (2/3) = 294

    The N varies with each dataset depending on the original number of
 rows (samples) in
    the raw data.
    %}
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"getUnlabelledData.m"

```
%{
This function takes in the arrayPerActivity struct containing a
single table for each activity.
It labels this data by assigning a unique label to each class, which
 will
eventually be the targets for SVMs.

Arguments
- `arrayPerActivity` -> struct containing a single array per
activity without any magnetometer data.

Returns
- `unlabelledInputs`    -> just the input data
- `classLabels`         -> [Nx1] class labels with a unique string for
 each class
%}

function [unlabelledInputs, classLabels] =
 getUnlabelledData(arrayPerActivity)
    % array containing the names of the activities.
    % These names will match the field names in the struct
    sets = ["LGW","RA","RD","SiS","StS"];
    % redefining the sets in a way that is compatible with SVM class
 labelling
    sets_for_labels = [{'LGW'} {'RA'} {'RD'} {'SiS'} {'StS'}];
    % ---------------------------------------------
    % Label the data by having a single column dedicated for string
 class labels
    % ---------------------------------------------
    for ff = 1 : length(sets)
        % obtain the number of rows in the current dataset
        sample = arrayPerActivity(1).(sets{ff});
        temp_labels = cell(length(sample),1);

        % we want one column of labels, each row labelled using a
 string
        % according to the target class
        temp_labels(:,1) = sets_for_labels(ff);

        % store all the targets as one long 1D array
        if ff == 1
            classLabels = temp_labels;
            unlabelledInputs = sample;
        else
            classLabels = vertcat(classLabels, temp_labels);
            unlabelledInputs = vertcat(unlabelledInputs, sample);
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"getLabelledData.m"

```
%{
This function takes in the arrayPerActivity struct containing a
single table for each activity.
We now want to label this data through one-hot-encoding so will need
 a
complementary matrix with labels. We also want to group all of the
 data
into a single array after labelling it.

Arguments
- `arrayPerActivity` -> struct containing a single array per
                        activity without any magnetometer data.

Returns:
- `labelledData`     -> array containing all samples for all
                        activities labelled by one-hot encoding
%}

function [labelledData] = getLabelledData(arrayPerActivity)

    % array containing the names of the activities.
    % These names will match the field names in the struct
    sets = ["LGW","RA","RD","SiS","StS"];
    % ----------------------------------------------
    % Label the data by adding an extra 5 columns at the end of
    % each activity array. Only the column corresponding to the
    % activity consists of 1's - the rest are all 0's.
    % ----------------------------------------------
    for ff = 1 : length(sets)
        sample = arrayPerActivity(1).(sets{ff});
        temp_labels = zeros(length(sample), 5);
        temp_true_labels = ones(length(sample), 1);
        % horizontally concatenate 5 columns for the labels.
        labelled_activity = horzcat(sample, temp_labels);
        % currently all labels are 0/False
        % depending on the activity, we want to make one column 1/True
        labelled_activity(:,end-(5-ff)) = temp_true_labels;
        labelled_array_per_activity(1).(sets{ff}) = labelled_activity;
    end
    % ----------------------------------------------
    % Vertically concatenate the 5 tables into one long table
    % ----------------------------------------------
    for ff = 1 : length(sets)
        sample = labelled_array_per_activity(1).(sets{ff});
        if ff == 1
            labelledData = sample;
        else
            labelledData = vertcat(labelledData, sample);
        end
    end
    % ----------------------------------------------
    % Split data into inputs and targets for ML
    % ----------------------------------------------
    % Define which features to include in the input set.
    final_inputs_nn = labelledData(:,1:end-5)';    % Take all the
 rows, and all the 294 features as inputs.

    % Define the target set
    final_targets_nn = labelledData(:, end-4:end)'; % Take all the
 rows, and the last 5 columns as outputs.
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"plotTimeDomain.m"

```
%{
Serving as a verification that time-domain features have been
 successfully
extracted, this function extracts all the time domain features
from a certain axis of a certain body segment (e.g. thigh_l_gyro_x)
and plots the first 150 readings of the 7 extracted features
 corresponding
to it.

Arguments
- `keyword`         -> name of a section to plot the time domain
 features
                        (e.g. "foot_l_gyro_x")
- `labelledData`    -> the labelled data
%}

function [] = plotTimeDomain(keyword, labelledData)
    % extract the relevant data from the dataset
    [segment_features_labelled_data, sig_indexes] =
 extractSegment(keyword, labelledData);

    % sig_indexes contains the column indexes of the relevant features
 in order
    % labels.csv contains all the class labels in english in the same
 order
    % that they appear in the data
    feature_labels =
 readtable("labels.csv", "ReadVariableNames",true, 'Delimiter','comma');
    feature_labels_array = table2array(feature_labels);
    relevant_features = feature_labels_array(sig_indexes);

    % plot all the time domain features side by side on a stem plot
    figure;
    for ii=1 : 7
        subplot(2,4,ii)
        stem(segment_features_labelled_data(500:650,ii), 'b')
        title(relevant_features(ii))
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"crossValidateNN.m"

```
%{
This function performs K-Fold Cross Validation by randomising the
samples then taking sequential folds from the data to build a pattern
recognition network model.
It retrieves the classification accuracy from each fold and finds the
mean accuracy accross all the folds.

Arguments
- `labelledData`        -> the data labelled through one-hot encoding
- `folds`               -> the number of folds required
- `hiddenLayerSize`     -> number of neurons required in the hidden
 layer
- `trainingAlgo`        -> the NN training algorithm to use

Returns
- `meanAccuracy`        -> the average classification accuracy of all
                            the models across the N folds.
%}

function [meanAccuracy] = crossValidateNN(labelledData, folds,
 hiddenLayerSize, trainingAlgo)
    % define the number test samples in each fold
    test_element_count = int32(size(labelledData,1)/folds);

    % initialisations
    foldCounter = 1;
    total_accuracy = 0;

    % before runninng cross validation and splitting into inputs and
 targets shuffle the rows
    random_final_labelled_data =
 labelledData(randperm(size(labelledData, 1)), :);

    for ii=0 : test_element_count : size(random_final_labelled_data,1)
        % make sure there are enough elements to make up the fold
        if ii+test_element_count <=
 size(random_final_labelled_data,1)+1
            fprintf("\n******************\nRunning fold %i out of %i
\n",foldCounter, folds)
            % extract the test set
            test_extracted = random_final_labelled_data((ii+1:ii
+test_element_count-1),:);
            % extract the test set (the remaining data), which also
 includes
            % validation data
            train_extracted = random_final_labelled_data;
            train_extracted((ii+1:ii+test_element_count-1),:) = [];

            % Define the input set.
            train_inputs = train_extracted(:,1:end-5)';
            test_inputs= test_extracted(:,1:end-5)';

            % Define the target set
            train_targets = train_extracted(:, end-4:end)';
            test_targets = test_extracted(:, end-4:end)';

            % train the ANN with the extracted data
            acc = nn(train_inputs, test_inputs, train_targets,
 test_targets, hiddenLayerSize, trainingAlgo);
            fprintf("\nAccuracy for this fold is %f\n", acc)

            foldCounter = foldCounter+1;
            % keep track of all the accuracies to find the mean
            total_accuracy = total_accuracy + acc;
        end
    end

    % give a summary of all the folds
    meanAccuracy = total_accuracy/folds;
    fprintf("\n===================\nAverage accuracy across %i folds:
 %f\n===================\n", folds, meanAccuracy);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"nn.m"

```
%{
This function builds a patternet ANN model using given training data.
It then tests it using the given test data and produces a confusion
 matrix
with the results. The patternet created has a single hidden layer.

Arguments:
- `trainInputs`    -> samples to use for training (inc. validation
 data)
- `testInputs`     -> samples to use for testing the model
- `trainTargets`   -> class labels for the training set
- `testTargets`    -> class labels for the test set
- `hiddenLayerSize` -> number of neurons in the single hidden layer
- `trainingAlgo`    -> algorithm to use for training the NN

Returns:
- `accuracy`        -> the classification accuracy of the built model
 when
                        tested with the given unseen test set.
%}

function [accuracy]=nn(trainInputs, testInputs, trainTargets,
 testTargets, hiddenLayerSize, trainingAlgo)

    % Create a Pattern Recognition Network with the defined number of
 hidden layers.
    % `patternnet` is specific for pattern-recognition NNs
    net = patternnet(hiddenLayerSize, trainingAlgo);
%     net.trainParam.showWindow = 0;   % hide the training window
    %{
    patternnet() is specialized for pattern recognition problems.
```

```
   - Default training algo: Scaled conjugate gradient backpropagation
(trainscg).
       * trainscg's goal: minimize a cost function.
   - Default loss cost function: Cross-entropy.
       * This function measures the performance of a classification
model whose
       output varies between 0 and 1.
       * Cross-entropy loss increases as the prediction probability
diverges
       from the output value.
       * Therefore, small values -> good performance, large values ->
bad performance.
   %}

   % Set up Division of Data for Training and Validation.
   % The test subset has already been extracted.
   net.divideParam.trainRatio = 50/100;
   net.divideParam.valRatio = 50/100;

   % Standardise and normalise the input data.
   % standardisation shifts the data such that the center is 0 and
the
   % standard deviation is 1. Function normalises each column by
default.
   % 'range' makes all the values be between 0 and 1 (normalisation)
   trainInputs = normalize(trainInputs, 'range');
   testInputs = normalize(testInputs, 'range');

   % Train the Network
   [net, tr] = train(net, trainInputs, trainTargets);

   % -----------------------
   % Test the Network with the test subset from the current dataset
   % -----------------------
   actualTstOutputs = net(testInputs);

   %  compare the NN's predictions against the training set
   idealTstOutputs = testTargets;
   tstPerform = perform(net, idealTstOutputs, actualTstOutputs);

   sets_for_labels = [{'LGW'} {'RA'} {'RD'} {'SiS'} {'StS'}];

   % we need to convert the targets from Nx5 boolean values into a
single
   % string row/column to be able to run confusionchart
   for yy=1 : size(idealTstOutputs,2)
       % get the 5 1/0 values representing the class label
       current_ideal_class = idealTstOutputs(:,yy);
       current_actual_class = actualTstOutputs(:,yy);
       % find where the '1' is
       [~,I_ideal] = max(current_ideal_class);
       [~,I_actual] = max(current_actual_class);
       % get the corresponding string value of the class label
       idealTstOutputsSimplified(:,yy) = sets_for_labels(I_ideal);
```

```
            actualTstOutputsSimplified(:,yy) = sets_for_labels(I_actual);
        end
        % create the confusion matrix object to show and retrieve
        % classification accuracy from
        plotTitle = sprintf('ANN Confusion Matrix for %i
 features',size(trainInputs,1));
        cm =
 confusionchart(idealTstOutputsSimplified,actualTstOutputsSimplified,...
            'Title', plotTitle,...
            'RowSummary', 'absolute',...
            'ColumnSummary', 'absolute');

        % Calculate the classification accuracy from the confusion matrix
        % Need to first obtain the number of correct classifications, this
 will
        % be equal to the sum of the values in the diagonal of the CM
        confusionMatrixResults = cm.NormalizedValues;
        correct_predictions = 0;
        for ii=1 : length(confusionMatrixResults)
            correct_predictions = correct_predictions +
 confusionMatrixResults(ii,ii);
        end
        accuracy = (correct_predictions/length(testTargets))*100;

        fprintf("\n-------------\nSummary:\n    Hidden layer neurons: %i
\n", hiddenLayerSize)
        fprintf("    Number of features: %i \n", size(trainInputs,1))
        fprintf("    ANN classification accuracy %f\n", accuracy)
        fprintf('    Patternnet performance: %f \n', tstPerform);
        fprintf('    num_epochs: %d, stop: %s\n-------------\n\n',
 tr.num_epochs, tr.stop);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"svmPosterior.m"

```
%{
This function builds a multiclass svm model and evaluates it using
Posterior Probabilities

Arguments
- `labelledData`    -> the data labelled through one-hot encoding
                        (these labels are irrelevant for SVMs and
 will
                        be removed by the code below)
- `svmTargets`      -> the Nx1 unique string class labels for SVM
 targets
- `kernelFunction`  -> the kernel function to use for training the SVM
 model
- `boxConstraint`   -> the C parameter to use for training the SVM
 model.

Returns
- `accuracy`        -> classification accuracy obtained when the
 created
SVM model is tested with the unseen test set
%}

function [accuracy]=svmPosterior(labelledData, svmTargets,
 kernelFunction, boxConstraint)

    % ######### Set aside some of the data for testing ##########

    % extract only the inputs from the labelled data. The SVM targets
 are
    % separately given
    svmInputs = labelledData(:,1:end-5);

    % We want to shuffle both inputs and outputs while preserving the
    % correlation
    p = randperm(length(svmInputs));
    random_final_inputs = svmInputs(p, :);
    random_final_targets = svmTargets(p, :);

    % Standardise and normalise the input data.
    % normalize() normalises the data such that the center is 0 and
 the
    % standard deviation is 1. Function normalises each column by
 default.
    % 'range' makes all the values be between 0 and 1.
    random_final_inputs = normalize(random_final_inputs, 'range');

    % set some percentage of it aside for testing
    test_percent = 30;
    test_element_count =
 uint32((test_percent/100)*length(random_final_inputs));


    % Define which features to include in the input set.
    train_inputs = random_final_inputs(1:end-test_element_count,:);
  % Take all the rows, and all the 10 features as inputs. Could also
 use: inputs = dataSet(:,1:end-2).
    test_inputs= random_final_inputs(end-test_element_count+1:end,:);

    % Define the target set
    train_targets = random_final_targets(1:end-
test_element_count, :);
    test_targets = random_final_targets(end-test_element_count
+1:end,:);


    % ######### Model Training ###################################
```

```
   % Create a SVM Model template to fit into fitcecoc()
   if kernelFunction == "polynomial"
       % if it's a polynomial kernel function then set the order to 2
(i.e. quadratic)
       t =
templateSVM('Standardize',true,'KernelFunction',kernelFunction, 'BoxConstraint',
boxConstraint, 'PolynomialOrder', 2);
   else
       t =
templateSVM('Standardize',true,'KernelFunction',kernelFunction, 'BoxConstraint',
boxConstraint);
   end

   %{
   Train the ECOC classifier using the SVM template.
   - 'FitPosterior':   -> To transform classification scores to class
posterior
                           probabilities (which are returned by
predict or resubPredict)
   - 'ClassNames'      -> To specify the class order.
   - 'Verbose'         -> To display diagnostic messages during
training
   %}
   fprintf("Training SVM binary learners...\n")
   SVMModel =
fitcecoc(train_inputs,train_targets,'Learners',t,'FitPosterior',true,...
       'ClassNames',{'LGW','RA','RD','SiS','StS'},...
       'Verbose',1);

   % Predict the training-sample labels and class posterior
probabilities.
   [label,~,~,Posterior] = resubPredict(SVMModel,'Verbose',1);

   % obtain a random sample from the data
   idx = randsample(size(train_inputs,1),10,1);

   % generate a table showing the predicted label and the
   % posterior probabilites of the sample data.
   % The 5 columns in the 'Posterior' columns correlate to:
{'LGW','RA','RD','SiS','StS'}
   table(train_targets(idx),label(idx),Posterior(idx,:),...
       'VariableNames',{'TrueLabel','PredLabel','Posterior'})


   % ######### Model Evaluation ##################################

   fprintf("\nEvaluating the SVM model...\n\n")
   %{
   Predict the posterior probabilities for each instance in the test
data.
   %}
   [~,~,~,TestSamplePosteriorRegion] = predict(SVMModel,test_inputs);
   % Convert the Nx5 TestSamplePosteriorRegion matrix into an Nx1
 array
   % with the index number of the class with the highest posterior
 prob.
   [~,I] = max(TestSamplePosteriorRegion, [],2);

   % Translate each class number into a string representing the class
   sets_for_labels = [{'LGW'} {'RA'} {'RD'} {'SiS'} {'StS'}];
   for ii=1 : length(I)
       targets_from_posterior_test_prediction(ii,1) =
 sets_for_labels(I(ii,1));
   end

   % Plot Confusion Matrix. This diplays the confusion matrix by
 default
   plotTitle = sprintf('%i Feature Confusion Matrix for SVM based on
 Posterior Prediction',size(svmInputs,2));
   cm =
 confusionchart(test_targets,targets_from_posterior_test_prediction,...
       'Title', plotTitle,...
       'RowSummary', 'absolute',...
       'ColumnSummary', 'absolute');

   % Calculate the classification accuracy from the confusion matrix
   % Need to first obtain the number of correct classifications, this
 will
   % be equal to the sum of the values in the diagonal of the CM
   confusionMatrixResults = cm.NormalizedValues;
   correct_predictions = 0;
   for ii=1 : length(confusionMatrixResults)
       correct_predictions = correct_predictions +
 confusionMatrixResults(ii,ii);
   end
   accuracy = (correct_predictions/length(test_targets))*100;

   fprintf("\n-------------\nSVM model accuracy using %i features: %f
\n", size(svmInputs,2), accuracy)
   fprintf("\nModel binary loss: %s\n-------------\n",
 SVMModel.BinaryLoss)
   % Binary Loss is quadratic since posterior probabilities are
   % being found by all the binary learners
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"find15Features.m"

```
%{
This function takes in the labelled data and uses the minimum
 redundancy
maximum relevance algorithm to find the most significant 15 features
 among
the existing 294 features. It extracts these 15 features and plots a
labelled bar graph of their scores.
Parameters:

Arguments:
- `labelledData`         -> labelled data
- `unlabelledInputs`     -> input data with no labels
- `classLabels`          -> svm-specific target data

Returns:
- `fifteenFeaturesLabelledData`  -> a condensed version of the given
`labelledData` that has samples from only the top 15 features.
%}

function [fifteenFeaturesLabelledData]=find15Features(labelledData,
 unlabelledInputs, classLabels)

    % labels.csv contains all the class labels in english in the same
 order
    % that they appear in the data
    feature_labels =
 readtable("labels.csv", "ReadVariableNames",true, 'Delimiter','comma');

    % run the minimum redundancy maximum relevance algorithm to order
 the 294
    % features according to their relevance
    % the SVM training data is formatted in a way that is suitable as
 an input
    % for the fscmrmr function, so we can use it here.
    [idx,scores] = fscmrmr(unlabelledInputs,classLabels);

    % only consider the top 15 features
    idx = idx(:,1:15);

    % rank the labels according to their relevance. Most important
 goes first.
    sorted_labels = feature_labels(idx,1);

    % give a summary of the most significant features:
    fprintf("\nMost significant 15 features:\n  ")
    sorted_labels

    % take only the top rated 15 rows from the labelled data
    % fifteen_features_inputs_nn = nn_inputs(idx,:);
    class_labels = labelledData(:,end-4:end);
    fifteen_features_unlabelled = labelledData(:,idx);
    fifteenFeaturesLabelledData = horzcat(fifteen_features_unlabelled,
 class_labels);

    % take only the top rated 15 rows from the SVM data
    % fifteen_features_inputs_svm = final_inputs_svm(:,idx);

    % need to convert the table to a cell array to show feature names
 in the
```

```
    % plot
    sorted_labels_cell = table2cell(sorted_labels);
    X = categorical(sorted_labels_cell);
    % categorical sorts the labels alphabetically by default, need to
 reorder
    % to perserve the ascending order of the feature score
    X = reordercats(X,sorted_labels_cell);
    Y = scores(idx);
    % plot the bar chart showing the most significant 15 features
    bar(X,Y)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"extractSegment.m"

```
%{
This function takes in the name of the most significant body segment,
 as
well as the dataset and extracts all of the segment's information.

Arguments
- `keyword`          -> body segment to extract (e.g. 'foot_r')
- `labelledData`     -> the labelled dataset

Returns:
- `segmentFeaturesLabelled`  -> dataset after reducing it to only data
 from that segment.
%}

function [segmentFeaturesLabelled,
 sig_indexes]=extractSegment(keyword, labelledData)
    % labels.csv contains all the class labels in english in the same
 order
    % that they appear in the data
    feature_labels =
 readtable("labels.csv", "ReadVariableNames",true, 'Delimiter','comma');
    ll = 0;
    % sig_indexes will contain the indexes of the features
    % related to the relevant body part i.e. keyword given.
    sig_indexes = zeros(1,1);
    feature_labels_array = table2array(feature_labels);

    % Loop through the feature names
    for ii=1 : size(feature_labels_array,1)
        % Find features that have the word we are looking for
        is_present =
 contains(feature_labels_array(ii),keyword,'IgnoreCase',true);
        if is_present
            % Its index correlates to the same index in the data as
 class labels are in
            % the same order
```

```
                sig_indexes(1,ll+1) = ii;
                ll = ll+1;
            end

        end
        % extract just the segment's columns from the labelled data
        class_labels = labelledData(:,end-4:end);
        segment_features_unlabelled = labelledData(:,sig_indexes);
        segmentFeaturesLabelled = horzcat(segment_features_unlabelled,
 class_labels);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"experimentFilters.m"

```
%{
This function is for finding the optimal params for the low pass
 filter.
It runs a low pass filter with the given params on random feature
 columns
and visualises the impact of the filter on these features in overlayed
 plots.

Arguments:
- `raw_data`    -> raw, unfiltered data
- `cutoffFreq`  -> filter cutoff frequency (in Hz)
- `sampleRate`  -> filter sampling rate

%}

function [] = experimentFilters(raw_data, cutoffFreq, sampleRate)
    sets = ["LGW","RA","RD","SiS","StS"];
    for ii=1 : 8
        subplot(2,4,ii)
        set_index = rem(ii,length(sets))+1;
        % take a random sample of activities and datasets to plot
 before and after
        % applying the low pass filter
        [x, y, fy] = visualiseFilterData(raw_data, cutoffFreq,
 sampleRate, set_index, ii);
        % plot a single features before filtering in vlue
        plot(x, y, 'b')
        hold on
        % plot a single feature after filtering in red
        plot(x, fy, 'r')
        % give each subplot a title
        t = sprintf("%s dataset number %i", sets(set_index), ii);
        title(t)
    end
    % give all the subplots a main title
    mainTitle = sprintf("Single features before (blue) & after (red)
 applying a low pass filter of %i Hz", cutoffFreq);
    sgtitle(mainTitle)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

"visualiseFilterData.m"

```
%{
This function takes in the struct containing all the datasets.
It extracts a column/feature in a given dataset and returns the
 required
data to visualise the effect of a low pass filter on it.
%}

function [x, Y, filtered_Y] = visualiseFilterData(raw_data,
 cutoffFreq, sampleRate, activityIndex, datasetIndex)
    % create a low pass filter with the given params
    d = designfilt('lowpassfir', 'FilterOrder', 8, 'CutoffFrequency',
 cutoffFreq, 'SampleRate', sampleRate);

    % extract time column (x) and one feature (y)
    sets = ["LGW","RA","RD","SiS","StS"];
    current_dataset = raw_data(datasetIndex).(sets{activityIndex});
    x = table2array(current_dataset(1:end,1));
    % ---------------------------------------------
    % Loop through a single dataset
    % ---------------------------------------------
    Y = 0;
    % loop through the columns in the single dataset
    for ii = 1 : width(current_dataset)
        % obtain the relevant column
        colm = table2array(current_dataset(:,ii));
        % ignore timestamp columns
        avg = abs(nanmean(colm));
        % if columns is NOT a timestamp one nor a 0 value one
        if (avg < 1000) && (avg > 0)
            % apply the filter on the column's values
            Y = colm;
            % overwrite the column with the filtered data
        end
    end
    % apply the low pass filter
    filtered_Y = filter(d, Y);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Published with MATLAB® R2020a