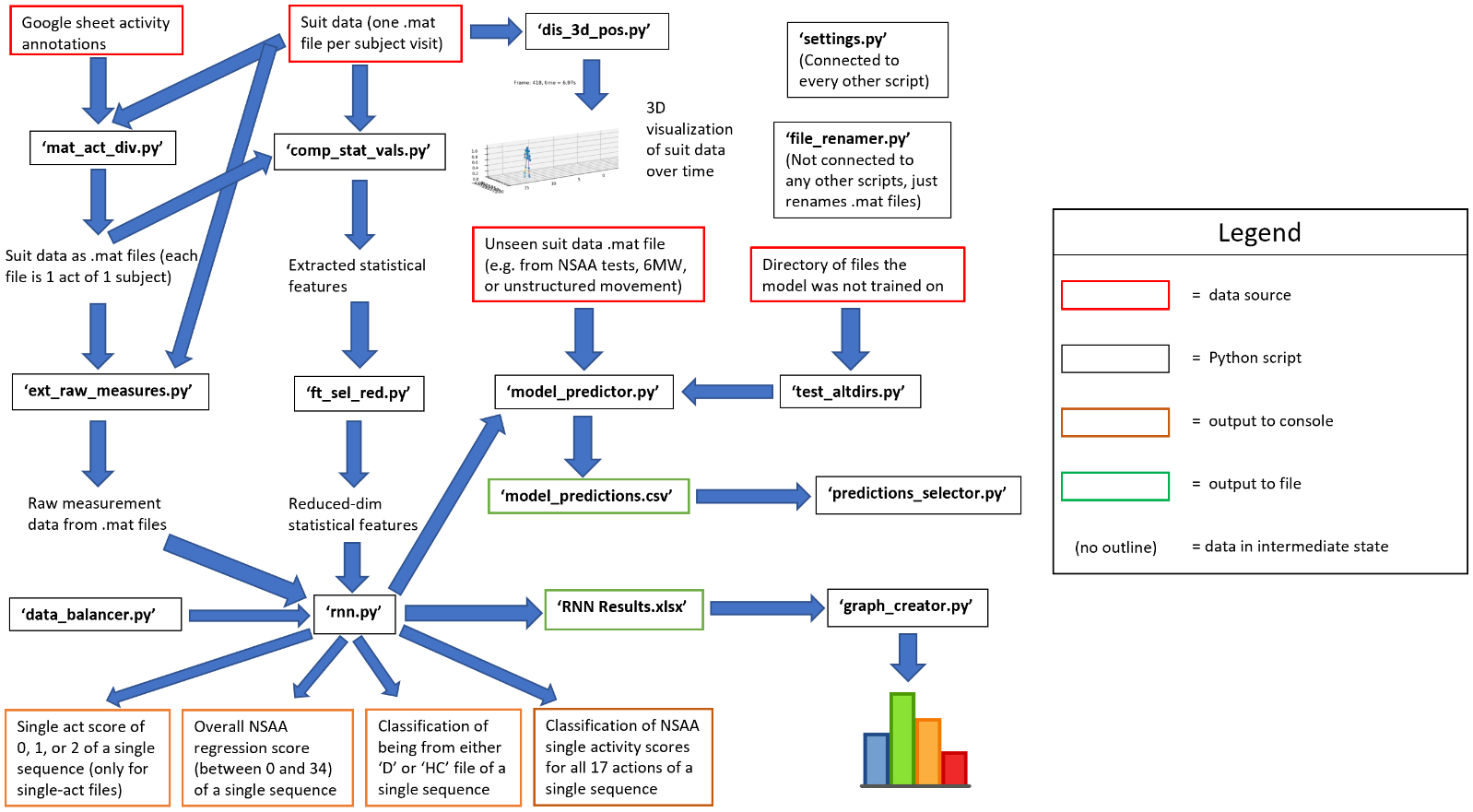
**Script Ecosystem Overview**

**Script Diagram**

Below, we can see a diagram of all the scripts involved in the system for the project. It covers all the inputs that are needed for the project, how they are processed by the various scripts, and what types of outputs are produced by the system. Note that this is only a vague overview, without any details of how the scripts do work (this is discussed further below) nor details of the names and locations of the source files, with the aim more to show the order things should be run in either by the user or by the batch scripts (not included below) and how the outputs relate to each other.



**Script Explanations**

**‘comp\_stat\_vals.py’**

**Overview**

The basic operation of the 'comp\_stat\_vals.py' script can be summarized as follows:

1. Read in a certain .mat file (either a joint angle file, an all-data file, or a datacube object) into Python
2. Apply statistical analysis on its various features (i.e. each column of the 'table' of the .mat file (e.g. joint angle file has table of 22k x 66) (note that this is optional if a joint angle is selected and called with the 'write\_direct\_csv' method, which just translates a joint angle .mat file to .csv format)
3. Write this out to a .csv file with a name corresponding to the read in file; the aim with this is for it to be an easy-to-digest format for the next stage in the analytics pipeline (e.g. a recurrent neural network)

**Statistical extraction**

Each of the statistical analysis analyses that are used are implemented by a distinct function that performs a bit of syntactical help (e.g. the function to calculate mean includes type changing and rounding of numbers). The statistical features that are computed include: mean, variance, mean absolute diff values, FFT, covariance components between axes, mean sum of values across axes, among others.

The bulk of these features have been extracted 'intra-columns'. This is meant by the following: consider a 'JA' (joint angle) file; it's columns correspond to only 1 'measurement', the 'feature' itself ('feature' in this context meaning one of the 17 sensor labels, 22 joint labels, or 23 segment lables, the choice of which depends on which measurement we are referring to), and the 'dimension' of this feature (as we are dealing with 3D position data, this is 3D with each representing the 'X', 'Y' or 'Z' dimension). Many of the statistical features are thus computed on the values within each individual column; for example, the mean for a specific column is computed by averaging all the values for a specific measurement's specific feature's specific dimension (e.g. measurement 'joint angle's feature 'jRightWrist's dimension 'X-dimension'), of which there are ~22k corresponding to ~22k samples in a file.

However, there are also several statistical functions that are applied one-layer up; that is, rather than calculating over a single column representing a single dimension of a feature of a measurement, it calculates over 3 adjacent problems for ALL dimensions of a feature of a measurement. These mainly include operations that calculate features over a 2-dimensional array of data. Finally, the statistical functions that operate on single columns are then reapplied to the calculate the same statistical function over all newly-calculated values. For example, if we are concerned with the variance values for the 'position' measurement over 23 feature names over the 'x'-dimension, then we take the variance of these calculated values to form a new value representing the 'position' measurement, the '(over all features)' feature, and the 'X'-dimension. This process is repeated with statistical functions that operate over the 3 axis dimensions.

The statistical features that are calculated per column (i.e. over a single axis) include:

* Mean
* Variance
* Absolute mean sample difference
* Fast Fourier Transform (1-dimension) largest val

The statistical features that are calculated per set of 3 columns (i.e. over all 3 axes of a feature for a given measurement) include:

* Mean sum of the values of each dimension
* Mean sum of the absolute values of each dimension
* First eigen value of the covariance matrix of the 3 columns
* Second eigen value of the covariance matrix of the 3 columns
* X- to Y-axis covariance (i.e. row 1 col 2 value of the 3x3 covariance matrix)
* X- to Z-axis covariance (i.e. row 1 col 3 value of the 3x3 covariance matrix)
* Y- to Z-axis covariance (i.e. row 2 col 3 value of the 3x3 covariance matrix)
* Fast Fourier Transform (2-dimension) largest 3 values (as 3 separate calculations)
* Proportion of samples outside the mean zone in every dimension

Each of these calculations done for a specific measurement, specific feature, and a single dimension is written as a single value as part of a row with the column title:

"(<measurement name>) : (<feature name>) : (<axis>-axis) : (statistical function)"

...while, when it is subsequently called to repeat the process over all feature names, the column has the title:

"(<measurement name>) : (over all features) : (<axis>-axis) : (statistical function)"

For the calculations done for a specific measurement, specific feature, and over all 3 dimensions, they are again written as a single value as part of a row with the column title:

"(<measurement name>) : (<feature name>) : ((x,y,z)-axis) : (statistical function)"

...while, when it is subsequently called to repeat the process over all feature names, the column has the title:

"(<measurement name>) : (over all features) : ((x,y,z)-axis) : (statistical function)"

The result is then a single row for a whole file with 'n' columns in the row, with each column corresponding to an above label. As we may have many measurements over which to calculate (e.g. 'position', 'velocity', 'angular acceleration', etc.), many features (e.g. 23, 22, or 17 depending on the measurement), 3 dimensions (or 1 dependent on which statistical function we are using), and ~15 statistical functions to compute, a single row for an 'AD' (all data) file can be several thousand columns long. Note that for a joint angle file this is significantly less as we are only concerned with 1 measurement (the 'jointAngle' measurement as this is the only one in the file).

Again note that these values are calculated across each of the samples (e.g. 22k) for each of the single columns or collection of 3 columns (depending on the statistical function in question).

**Running this script**

To run this script and extract the desired .csv outputs, one must do the following (assuming one has access to the downloaded script already):

1. Setup the required packages needed for the script (which are all 'included' at the beginning of the script), including numpy, scipy, pandas, etc (recommended to do so using 'pip' or 'conda' but manually downloading and putting them in 'Lib\Site-packages' is an option too)
2. Change the initial global variable values for 'source\_dir' and 'output\_dir' to the location of the '6minwalk-matfiles' directory of JA and AD .mat files and to the location of the produced .csv files should go, respectively
3. Open the data cube .mat file ('data\_cube\_6mw.mat') in MATLAB and run the 'writetable(excel\_table, "data\_cube\_table.csv)' command. This is necessary to extract the table within the datacube (which is needed in this script to process the datacube correctly) as Python is unable to read a .mat table in the same way as it can a .mat structure.
4. Open up the command prompt (or terminal within an IDE), navigate to the scripts destination, and run the script with arguments for the file type (e.g. 'AD', 'JA', or 'DC) and a file name (either a specific short name like "D2" or "all" which processes all of the specified file type in the current directory); note that additional arguments are available (add '--help' to view these).

Hence, once the sufficient packages are installed, the global variables set, and the data cube table is extracted, the statistical feature values of all the AD files in the subdirectory '6minwalk-matfiles\all\_data\_mat\_files' can be extracted by running the command:

**"python matfiles\_analysis.py ad all"**

This may take a few minutes to complete, but once done, the file will appear in the output directory with the title 'AD\_all\_stats\_features.csv', with each row corresponding to a single file and each column corresponding to the statistical values extracted from the file. Note that each file has the same measurements and features that are taken from their source .mat file, hence all the outputs to the .csv can share the same column labels.

The resultant file will then appear in the output directory specified by 'output\_dir'.

**Optional argument: split files functionality**

An additional argument can be given to the prompt as '--split\_files=<split\_val>' where 'split\_val' is an integer. This integer specifies that, for each file we are concerned with (e.g. all of them for a file type if 'all' is specified or just 1 if something like 'D2' is specified as a filename), we should split the source file samples into sections. For example, with a source 'AD' file with 22k samples (i.e. rows), if 'split\_val' is specified to be '10', we divide this row into 10 parts essentially stacked on top of each other, where each part now has 2.2k rows.

The rest of the statistical extraction process follows as normal, with the exception that each of the statistical functions now operate on the file parts rather than the complete file itself; that's to say, for something like the mean of a single col's values, instead of computing the mean over 22k samples, it would compute 10 means of each part of length 2.2k samples.

This results in 10 rows of values that are outputed to a single .csv rather than just 1 value. Alternatively, if we specify something like 360 for 'split\_val', then for .mat files that correspond to a time length of 360 seconds, then each row in the output .csv file will correspond to the statistical features of a single second of the source .csv.The idea is that this will give us more options when it comes to training ML models on this statistical data.

**Optional argument: check for abnormalities**

This is a brief function that checks output .csv files for what seems to be abnormalities within the file. For example, say we have run the script on 'ad all --split\_files=10'. We then have 10\*number of AD files (= ~150) rows in the 'AD\_all\_stats\_features.csvs' file. This then (if the '--check\_for\_abnormalities=<margin>' optional argument is given) has each of its rows checked to see if any of its features (i.e. columns) fall outside the mean error margin given by 'ratio'; for example, if the mean value of a certain feature is '5' across all rows in the 'AD\_all\_stats\_features.csvs' (i.e. over all file parts) and the 'margin' is set at 0.2, then a value for a certain file part for that feature outside the range of 5-(0.2\*5) and 5+(0.2\*5) (i.e. 4 and 6) will be flagged up.

If enough of these flagged features occur (i.e. above a ratio set by 'abnormality threshold', which is defaulted to 0.3, meaning when more than 30% of features are beyond the mean error margin), then the offending file part will be printed to the console. The ideal is that, with a .csv containing many file parts for a given file (e.g. 360 parts of a single file if it's split up into individual seconds), then we can easily observe strange behaviour given by the source data in the .mat file, which may be strange behaviour on the user's part or a fault in the data collection.

**Optional argument(s): display various additional info**

There also exists 3 (currently) display functions that can infer information about the files in question that corresponds to the display optional argument in question (i.e. arguments 'ad D2 --dis\_3d\_pos' will call the 'display\_3d\_positions' function on the 'AD' 'D2' file only). The behaviour of these are best observed first-hand rather than described, but as a brief summary:

*'display\_3d\_positions()'*: for a given 'AD' file, displays a 3D dynamic representation of an 'AD' file's position values (hence it won't work on a 'JA' or 'DC' file). This runs it real time, and thus will show a 3D figure walking in 3D space over time.

*'display\_3d\_angles()'*: for a given 'JA' file, displays a dynamic 3D representation of the joint angles as they change over time (note: this is somewhat deprecated, and a specific use-case hasn't been formulated yet).

*'display\_diffs\_plot()'*: for a given 'JA' file, displays 66 total subgraphs on one plot (with 3 columns and 22 rows), with each row corresponding to a feature and each row corresponding to a feature's dimension (e.g. X, Y, or Z). Each graph themselves shows a plot of the diffs of the values of that feature's dimension (i.e. before statistical analysis is applied) as it changes over time.

**‘ft\_sel\_red.py’**

**Overview**

One of the consequences of using the 'comp\_stat\_vals' script is that the number of features for a single subject's all data ('AD') file baloons several fold: for a single subject with ~620 columns (with each being one feature of one measurement) and ~22K rows (360s at 60Hz suit sampling rate), this then becomes ~360 rows (given '--split\_size'=1, i.e. 1 row for every 60 source rows) of approximately 4000 columns. Hence our data shape is now (360,4000) for a single file. This is completely impractical to use as training data for a given model for several reasons:

1. The curse of dimensionality means that the models struggle to train at all when dimensionality is this large for the amount of data samples ('360') that we have available.
2. Many of these computed statistical features may hold not that much useful information in them, or at least less useful information compared to other useful statistical features.
3. Even if we were to use all these features, it would take a much longer time to train models for most likely very little gain (with it most likely being worse off than smaller dimensioned data), making it even worse from a practical standpoint.

Hence, for the '\_stat\_features' files that are created by the 'comp\_stat\_vals' script, its more-or-less necessary to reduce the dimensionality to something a lot smaller prior to using this as training data. Note that this isn't done for raw measurement data. This is for three reasons:

1. The dimensionality of these data files is already at a level that is feasible for training (ranging from 51 from sensor measurements to 69 for segment measurements)
2. There are far more rows of data within each of these files; this is due to the fact that, with using 'comp\_stat\_vals' with '--split\_size'=1, we computed stat values over each block of 60 rows and hence reduce the number of actual 'numbers of data' (i.e. numbers that appear in our data set) by 60-fold. This 60-fold increase in data when using raw measurements makes using this 51-69 dimensioned data a lot more feasible in training models.
3. Even though we may be computing many redundant features in 'comp\_stat\_vals', we are much less likely to have features that are as redundant as these in the raw measurements data. This is because every feature corresponds to a single dimension for a sensor, angle, or segment, which is much more likely to hold important information that many of the computed statistical values, and thus there is more of a motivation to keep all of these.

**How it works**

Given a user-specified 'dir' for the directory that we wish to source the stat feature files from, the file type we're interested in (usually set to 'AD'), the 'fn' of the file(s) of which we wish to reduce the dimensions of (set to 'all' to do so over all files in 'dir'), and 'choice' (which is the feature selection/reduction technique to use), the following is undertaken by the script:

1. For a given file name in 'dir', read in the file (e.g. 'AD\_D4\_stat\_features.csv') as a DataFrame and divide it into its 'x' and 'y' components.
2. Normalize each dimension of the data if the relevant optional argument is set.
3. Set the number of features to extract from the data if the relevant optional argument is set. As standard, we use '30', as this generally encompasses a vast amount of the variance inherent to each data file while also being a feasible data width for our RNN models.
4. Based on the 'choice' argument given by the user, use a technique to reduce the dimensionality of the data. This can be done in an unsupervised feature dimensionality reduction manner (e.g. using principal component analysis or Gaussian random projection), unsupervised feature selection manner (e.g. variance thresholding or feature agglomeration), or in a supervised feature selection manner (e.g. by using a random forest for feature selection). 'PCA' has been used up until this point, though at the time of writing, this may be subject to further investigation and experimentation.
5. With the newly-reduced data, call the 'add\_nsaa\_scores' function to add the overall and single-act NSAA scores to each of the rows of reduced-dimensionality data, which is necessary for getting the relevant 'y' labels by the 'rnn' script, which this script feeds into. The information for these scores comes from the 'nsaa\_6mw\_info.xlsx' file, which contains the scores for every subject that has undertaken the NSAA assessment; hence, all that is required is to select the row in this .xlsx file that corresponds to the subject we are currently dealing with.
6. The newly-reduced data, with the NSAA scores appended at the beginning of each row, is then written to the same directory as it was sourced, with the exception that an "FR\_" ("feature reduced") prefix is appended to each file name to differentiate it from the file it came from.
7. Repeat this process for every other file name in 'dir' that is required, which (if 'fn'=all results in all files in 'dir' having their dimensions reduced.

**‘mat\_act\_div.py’**

**Overview**

Along with using the full data files of the suit usage in various models and with varying target outputs (e.g. D/HC classification, overall NSAA score, etc.), we also wish to extract the single activities of .mat files from the NSAA directory. As standard, each .mat file in the NSAA directory contains the suit data of one full assessment for a single subject. This means that each file usually contains the subject performing all 17 activities within the same file which are separated in time, sometimes by only a second or two in the case of the 'climb/descend box' activities and sometimes by up to a minute in the case of the 'get off the floor' activites. Hence, it would be advantageous for us to extract the data of the individual activites from within each file in order to use them for training.

As the data is contained within a very large table and each row is a single time instance of data (collected at 60Hz from the suit, therefore each row is 1/60th of a second's worth of suit data), to create new single-activity files, all we need to do is the following:

1. Determine the start and end rows within the overall file of the activity in question (e.g. if we wished to extract the second activity data that we know starts at 13s and ends at 15s in the subject's assessment, we would need to extract rows 780 to 900 of the overall file)
2. Slice the relevant rows from the table and create a new '.mat'-friendly tree structure within the script
3. Write this data to a 'act\_files' subdirectory of the source directory as a new .mat file with a file name reflecting which activity it represents

From here, we can process these single-activity .mat files in the same way as the standard .mat files, including extracting of raw measurements, computing of statistical values, and training of an RNN.

**How it works**

The key requirement for this script to work is by downloading the relevant Google document data sheet. This contains the manually assessed activity times of each subject, which was done by several members of the research group that analysed each of the videos that corresponds to each subject's .mat files and observed roughly at what times these activites started and ended for each subject. Note that these aren't going to be perfect, which is one flaw of using this sheet, as we can't give the exact start and end times of each activity and so tend to overestimate the amount of time the activity takes (i.e. note down a start time that's most likely before the real time and an end time that's after the true end time). Also, this process is not immune to human error, and therefore it's not impossible to misinterpret what what constitutes a 'complete' activity, which will impact how much use these 'single\_act' files are for us.

This Google sheet that contains all these observations needs to be downloaded prior to running this script and can be found at:

‘https://docs.google.com/spreadsheets/d/1OvkGU6kwmMxD6zdZqXcNKUvur1uFbAx5IND7\_dXibjE"

as a .csv to the relevant directory to the user (usually 'local\_dir + "NSAA\\"'. From here, once this is read in by

the script, there are two functions that are executed:

*'extract\_act\_times'()*: As the name suggests, this function analyses the Google sheet and from here, creates two lists: the first, 'act\_times', is a list of start and end times (in suit frames, i.e. seconds x 60) in a nested structure (e.g. if there are 10 subjects, each performing the 17 activities, and each have a start and end time, then 'act\_times' has a shape (10, 17, 2)); the second, 'ids', contain a list of subject names (e.g. 'D4'), each entry of which corresponds to an entry in 'act\_times'.

*'divide\_mat\_file'*: This function then takes the above two lists and, depending on what 'version' as an arg the user has specified ('V1' as standard; see the table itself for what the different versions correspond to) and what 'fn' arg the user has selected for the subject it whats to divide the activities of, the relevant row within 'act\_times's is retrieved. The complete .mat file for the subject in question (determined by 'fn' arg) is then loaded, the table of data within the .mat file is extracted, and then, for each activity 'pair' (i.e. two numbers that are the start and end row numbers in the table for each activity), the table is sliced for that pair. These rows then 'replace' the rows of the 'whole' .mat file and the .mat file is then rewritten to a different file with a name reflecting the activity it is currently being concerned with in the 'for' loop. This then repeats for each of the 17 activites for the given 'fn' subject.

**‘ext\_raw\_measures.py’**

**Overview**

While the extraction of computed statistical values is an important tool for the data pipeline as an input to the RNN script, it's also necessary to be able to use different types of raw measurement values; in other words, the values that are recorded by the sensors of the body suit and are within the corresponding .mat files. For a given subject's suit data, each measurement (e.g. 'position', 'jointAngle', 'sensorMagneticField', etc.) is inserted into the .mat file's table of values as a column, with the height of the column equal to the number of samples that was taken of the subject (corresponding to the length of time the suit was recording x 60 samples per second). Within this single column, there are vectors of either 51, 66, or 69 values (depending on whether the suit was recording raw sensor values, angle values, or segment values, respectively).

The idea of this script is fairly simple. For a given subject name in a directory (or all the subject names found in that directory) and for a given measurement (or all raw measurements available), the relevant .mat file is opened, and the relevant column is expanded for the given measurement name so that it becomes a matrix of single values rather than a column of vectors (with a matrix of shape '# of samples' x '# of vector vals (e.g. 51, 66, or 69)'). This matrix of data is then to be written to a separate .csv file within a directory that reflects the source directory 'dir' and the measurement name that the matrix contains.

From here, we can then use this data to train an RNN on these raw measurement values with y-labels (i.e. target values) that are determined by the type of file this .csv of data corresponds to (i.e. a 'D' or 'HC' subject), or the overall or single-act NSAA scores that correspond to the subject name of this .csv (e.g. 'D4') that can be found with 'nsaa\_6mw\_info'. In doing this, we provide an alternative to the production of RNN-ready data by 'comp\_stat\_vals' and are able to compare how manually extracted features differ in RNN performance to raw data (and thus the RNN doing its own feature extraction). This is explored further within the discussion of results.

**How it works**

The script runs in a fairly simple way without the necessity of classes or functions and thus just goes through a sequence of steps, which are as follows:

1. Takes in arguments for the directory from which to retrieve the file(s) for raw measurement(s) extraction and checks them for validity (e.g. makes sure 'dir' is one of the allowed types).
2. Retrieves the full file name(s) of the files within 'dir' from which we shall extract the measurements from. If 'fn'=all, retrieves all full file names in 'dir' as a list.
3. Parse the list of measurements that we wish to extract based on the 'measurements' arg that are comma-separated. If 'measurements'=all, then return a list of all extractable measurements available as a list.
4. Create a directory for each raw measurement within 'dir' to store these raw measurements extracted as .csv's.
5. For each file in 'dir', load the .mat file, extract the table of values within its tree structure, removes any 'wrappers' around these values within the table, and for each measurement to extract, select the column from the .mat table that correspond to the measurement, expand it out as 'measure\_data', and write it to a .csv file that reflects the file name and measurement we are currently concerned with.

**‘rnn.py’**

**Overview**

As the central element of the system insofar as it encompasses the learning and prediction models that are relied upon to produce the results, the importance of this script should be self-evident as it contains the class that defines the RNN's architecture, how it trains, predicts, and the instantiation and running of said class. Hence, rather than going through the motivation of writing this script or going through the basics of RNNs and their operation, we instead shall highlight a few important points about the structure of this RNN:

* We chose to use LSTM units instead of traditional neurons mainly due to their ability to learn better and don't suffer the vanishing or exploding gradient problems as severely.
* Other hyperparameters within the RNN itself (number of layers, size of LSTM units, learning rate, etc.) are kept as a constant throughout the experiments. These were found based on prior 'best practices' through prior research projects undertaken by others as well as rudimentary tuning to find 'good enough' parameters.
* The final layer can be either a single node for classification, a single node for regression, or 17 total nodes for single-act classification; hence, the building of the RNN model depends on the arguments passed in to the script.
* The performance of the models that are built here are generally viewed by two means: the console output at the end of the running of the 'rnn.py' script (which provides the info we need to fill in the 'RNN Results.xlsx') or the 'model\_predictor.py' script (which provides info for 'model\_predictions.csv'). See the relevant section in this report for more information of how 'model\_predictor.py’ works.

**How it works**

The structure of the script is fairly complicated and slightly convoluted, with numerous conditional statements needed to handle various data processing edge cases and many possible optional argument combinations that sometimes interact with each other in strange ways that must be handled; hence rather that explaining the structure of the script in detail, it's instead worth going through how exactly the script works upon being instantiated from the command line with arguments. This should give the user the a good grasp of what's going on upon script instantiation:

1. Reads in all required arguments (e.g. source directory, file name(s), output type, etc.) and optional args (e.g. sequence length, sequence overlap, leave out file choice, etc.) and checks each for validity.
2. Preprocesses the data from the source directory and file name(s) chosen; this includes reading in all source '.csv' files, fetching the relevant 'y' labels for the 'x' data from the files, splitting the data into sequences, discarding a proportion of the sequences if necessary, splits into train/test components, etc.
3. Builds the rnn object (instantiated from the 'RNN' class) with the necessary feature length, sequence length, size of LSTM units, number of hidden layers, and so on.
4. Train the RNN on the 'x\_train' and 'y\_train' components, tests on the 'x\_test and 'y\_test' components.
5. Prints out the performance on the test set to the console.
6. Write to a .csv unique to this model the results of the predictions, the arguments used to run the script, and the results that were printed to the console output.

It's also worth touching on a few of the optional arguments. The required arguments should be self-explanatory and in no further need of elaboration; some of the optional arguments and what they are used for are covered in more detail in the experiments and results discussion for the project (such as '--seq\_len', '--seq\_overlap', and '--discard\_prop'), but others aren't and so should be briefly touched upon here:

*'--write\_settings'*: This gives the user the option to store the results of the RNN that are printed to the output to the 'RNN Results.xlsx' file, rather than the user having to manually copy-paste console results to the file in a new row. This is generally used when new experiments with different RNNs are being carried out to save time and minimize the chances of human error.

*'--create\_graph'*: This will create a graph of the true values against the predicted values; as these are done in the continuous numerical domain, this is only really useful for the overall NSAA score output type and is generally written to a new file within the 'Graphs' directory to be used in results' discussions.

*'--epochs'*: A quick way to modify the number of epochs needed to train a model; this over varies based on the type of file being trained; for example, stat values from 'AD' files generally need only about 20 epochs to converge, while we generally use >100 epochs for raw measurements. The epoch value therefore isn't kept as a constant like the other hyperparameters but rather fluctuates as necessary to achieve model convergence.

*‘--other\_dir'*: This argument is set with the name of another source directory in order to also include files from another directory (or directories) in order to train and test the model; it simply loads in additional files into the preprocessing function. The motiviation behind this is further explored in the results discussion of 'model\_predictions.csv'.

*'--leave\_out'*: This is the standard way leave out a specific subject short name (e.g. 'D4') when training the model. This is the workaround instead of removing a subject from the source directory so the model is not exposed to the subject in the training process. This is primarily used in conjunction with 'model\_predictor.py' to test on the left-out subject in question for those particular models. See 'model\_predictions.csv' and its section in the results discussions for more information of using this arg.

*'--balance'*: This is the way that we can either upsample or downsample the data set loaded in by calling the relevant functions within 'data\_balancer.py'. The motiviation for rebalancing the data set and how it works is covered extensively in the README for that script and thus is not worth repeating here.

**‘model\_predictor.py’**

**Overview**

While gaining insights into various types of model parameters, source data types, data preprocessing options, and so on are an important and useful output of the project, one of the primary aims is to be able to test models that have already been built on complete files; for example, we may wish to see how the model performs when tested with a subject it has never seen before and record the results in 'model\_predictions.csv'. Alternatively, we may want to be using models in their 'production' form to help inform specialists about subjects based just on model results. To do this, we need a separate script that both preprocesses a single subject's file(s) for testing, but also to load the relevant models from a specified source.

The 'model\_predictor.py' script was written with this in mind. While it may work with predictions and the preprocessing of data, it's unlike the 'rnn.py' script in that it does not create any models; rather, it uses the models that have been created by 'rnn.py' already. Hence, the script is only useable after 'rnn.py' has created the required models. The arguments to 'model\_predictor.py' primarily serve three purposes: to load the data from the relevant source directories based on the file types (e.g. AD and jointAngle) in '.csv' format (created by either the 'comp\_stat\_vals.py' and 'ft\_sel\_red.py' scripts or the 'ext\_raw\_measures.py' script), to load the models that have been created that have been trained on the directory the file in question is sourced from and with the relevant file types and for all output types, and finally to run the '.csv' data files on the models that have been loaded and aggregate the results to make overall predictions.

**How it works**

The execution of 'model\_predictor.py' runs in a fairly procedural manner; hence, it's more intuitive to describe the program as a sequence of steps that call functions when necessary rather than a series of functions that are connected together as needed (e.g. 'comp\_stat\_vals.py'). The execution is as follows:

1. Checks the validity of each passed in argument.
2. For a given file name, loads in the '.csv' files for each of the file types provided; for example, if fn='D4' and ft='AD,jointAngle,sensorMagneticField', then the 'FR\_AD\_D4\_stat\_features.csv', 'D4\_jointAngle.csv' and 'D4\_sensorMagneticField.csv' files are loaded in (the names of which might slightly vary in practice due to naming conventions).
3. Identify the directories that contain the models that we require to use for the files' assessment; note that these are all contained within the 'output\_files\rnn\_models' directory and have names that reflect how the models were built and on what data. This is done for all three output types as well. For example, if dir='NSAA' and ft='AD', then 'NSAA\_AD\_all\_dhc\_--seq\_len=10\_--seq\_overlap=0.9\_--epochs=300', 'NSAA\_AD\_all\_acts\_--seq\_len=10\_--seq\_overlap=0.9\_--epochs=300' and 'NSAA\_position\_all\_dhc\_--seq\_len=600\_--seq\_overlap=0.9\_--discard\_prop=0.9' are loaded as the directory names containing the models.
4. Preprocesses the data from the '.csv' files so that they will fit into the pre-trained models (e.g. by having the expected batch size and sequence length) along with fetching the requisite 'y labels' for the data in the same way as is done for 'rnn.py'.
5. For each output type and for each of the '.csv' files of the data for the subject in question, put all the data through the model that corresponds to the '.csv's file type (e.g. jointAngle or AD) and its output type in prediction mode and have the predictions collected.
6. For a given output type, average together all predictions made over every sequence prediction for every file type to get a prediction for that output type for the whole file. For example, for the NSAA overall score output type, we average the scores for every sequence from a given file type's predictions, repeat this for the other file types, and finally average these scores to get a prediction of the overall score that takes into account all predictions made for every sequence of all the file types we are assessing on.
7. Outputs these scores to the user and appends these results to a new line within the 'model\_predictions.csv' file, along with the name of the subject in question as well as the file types used, the source directory, etc.

Special attention should be paid to some of the optional arguments. Some are used exclusively by other calling scripts (e.g. '--handle\_dash' and '--file\_num' are exclusively used by the 'test\_altdirs.py' script) and others are fairly simple and self-explanatory (e.g. '--show\_graph' shows the true and predicted overall NSAA scores made for the subject, while '--single\_act' is used when the input to the models are single-act files), there are a few others that require brief explanation:

*'--alt\_dirs'*: provide this with a name of a directory that is not the same as 'dir' to test files on models that haven't been trained on the same directory; for example, if dir='allmatfiles' and alt\_dirs='NSAA' then subject files will be loaded from the 'allmatfiles' directory but tested on models trained on files originating from the 'NSAA' directory. The motivation and results of this are explored in more depth in the results discussion.

*'--use\_seen'*: for a given file name (e.g. matching or deriving from a subject short name like 'D2-009'), the default behaviour of the script is to seek out model directories where the subject has been completely left out of the training and testing process; in other words, the subject who we're assessing is completely new to the models assessing it. This is done by specifically seeking model directories with names containing '--leave\_out=<file name>' (along with the other required directory and file type arguments). Sometimes, we may not want to do this specifically: for example, when we want to compare a subject being tested on a model familiar with the subject with one that isn't. Further results of this are explored in the experiments discussion.

*'--use\_balanced'*: in a similar way that '--use\_seen' seeks out model directories that haven't got something in their names, this optional argument specifically seeks model directories to use that have got '--balanced=<up/down>' in the name (depending on the value given to '--use\_balanced'). Hence, this allows us to test complete files on models that have trained on an up- or down-sampled data set. For more information on the data balancing process, consult the README for 'data\_balancer.py', or for more info on how well this performed on complete files, seek the relevant section in experiment results.

**‘test\_altdirs.py’**

**Overview**

A key motivation of this project is investigating how well, if at all, models that are built on one type of data can be adapted to be used on other types of data. Furthermore, to get a good idea of how well this is done, it's necessary to test numerous files on pre-trained models. And in the case of testing natural movement files on models that are trained on NSAA and 6 minute walk files, this would require running 'model\_predictor.py' manually over 400 times and each time with a different file name from within 'allmatfiles'. To get around this, 'test\_altdirs.py' was created to automate this process.

Crucially, this script only allows 'model\_predictor.py' to work on assessing models' performances on unseen files that also have aren't trained on the same type of data. This allows us to see the strength of the correlation between different types of assessment for subjects wearing the suits and also whether or not predicting the assessment scores by models trained on one type of assessment can be used to infer assessment scores of data in a form that the models haven't been trained on. In other words, can we have subjects just do natural movement activites and then use the models that have been trained on NSAA and 6 minute walk assessments to determine their D/HC classification, NSAA overall scores, etc. just as well as if they had instead done the NSAA and 6 minute walk assessments? The results of this are explored later in the relevant results discussions.

**How it works**

The 'test\_altdirs.py' does the following when run:

1. Reads in the name of a directory from which we wish to source the files that we wish to use for assessment, and also the names of the directories that will have been used to train certain models (for example, supplying 'NSAA\_6minwalk-matfiles' here will ensure that each time 'model\_predictor.py' is then called it retrieves the models that are trained on NSAA and 6 minute walk files).
2. Retrieves a list of .mat file names from within the source directory (i.e. if 'allmatfiles' was passed as the 'dir' argument then the names of all .mat files from within 'allmatfiles' were retrieved and stored in a list)
3. For every file name within this list of file names, create a unique string that corresponds to the input string to run the 'model\_predictor.py' script with certain arguments. This string includes the short file name of the file in question, the file types that the models will have been trained on (for example, if 'allmatfiles' was chosen as 'dir', then this must be 'jointAngle' as this is presently the only type of information that can be extracted from this type of data), the assessment file directory, and the source file directories that were used to train the models.
4. From here, all functionality is passed on to the 'model\_predictor.py', which runs once for every file within the source directory as specified by the 'dir' argument. For further information on how this runs and what it produces, refer to 'README 'model.predictor.py''.

**‘graph\_creator.py’**

**Overview**

There are several different ways that the outputs of experiments can be stored as 'results', along with appearing in several locations. For example, the results of different RNN setups and its tests on the left-out test sets will appear in the 'RNN Results.xlsx' file. Additionally, for each model run (i.e. each row in 'RNN Results.xlsx'), there is a whole file of true and predicted values over the test stored stored in a single '.csv' file with a name that corresponds to the predictions. Meanwhile, the results of whole file predictions (i.e. through the use of 'model\_predictor.py' and it's wrapper script 'test\_altdirs.py') are written as a row per file prediction into the 'model\_predictions.csv'.

However, none of the scripts that write to these files do any sort of plotting or graphing of the data. This is for two reasons:

1. Many times that we are running the scripts, we don't want to see the immediate plotting results or, rather, we can't. For example, when we run the 'model\_predictor.py' script once, it's only concerned with writing a single line to 'model\_predictions.csv', in the same way that 'rnn.py' only writes one line to 'RNN Results.xlsx', so for these to plot any results over several lines, the scripts would need additional user arguments to tell the script which lines it wishes to use for plotting, which adds to the already-high complexity of the scripts. Additionally, we often run 'rnn.py' via a batch script with many slight differences (to easily create a bunch of models to test on) and 'model\_predictor.py' via 'test\_altdirs.py', so stopping to produce a graph for every line that is written to an output file would be very inconvenient and would slow down the process.
2. In separating the functionality, we keep a large degree of modularity amongst the scripts. In other words, the scripts that write the output to the output files ('model\_predictions.csv', 'RNN Results.xlsx, etc.) have nothing to do with the actual plotting of results in graphs. This helps in debugging (i.e. a problem in displaying the data will be isolated to 'graph\_creator.py') and also allows us to choose when we wish to do the plotting (i.e. after the data that we determine we need has been collected, not after a predetermined point in the running of each 'rnn.py' or 'model\_predictor.py' run). Furthermore, this sort of setup opens up the possibility for an easier collaborative effort: if others were to contribute to the output files (e.g. by adding experiment results done on other types of data that is still written to the output files in the same format), then it's possible to use 'graph\_creator.py' as a standalone script without the need to have previously run any of the other scripts.

**How it works**

The direction that 'graph\_creator.py' takes in terms of running entirely depends on the initial argument. Based on this, the script calls one of four functions that process the other given arguments in a certain way. Note that, as each function operates on the arguments given differently, some of them are given generic names such as 'arg\_one' and 'arg\_two'. Also note that, as each function requires different numbers of arguments, every argument other than the first one ('choice') is optional; hence, when 'choice' is set to 'model\_preds\_single\_acts', it won't throw an error when we only give it values for 'arg\_one' and not the other three positional arguments.

Rather than going over things sequentially, we instead go over below each of the functions that are called by their associative 'choice' argument value:

*plot\_trues\_preds()*: this is a very simple function insofar as it just takes in the name of the '.csv' output that is produced by every run of 'rnn.py' that contains the test true and predicted values and are contained within the 'RNN\_outputs' directory. Hence, the only argument needed is 'arg\_one' and this is to be the full name (not including directories and the file extension) of the file we wish to use. This is then read in, the predicted and true values are read in, and these are plotted against each other in 2 dimensions, with a 'y=x' line going through them to signify their 'ideal' positions.

*plot\_model\_preds\_altdirs()*: reads in the 'model\_predictions.csv' as a DataFrame object; we then wish to determine which rows in the DataFrame object that we wish to use. This is then based on rows that have their 'Source dir' column set to the value of 'arg\_one' and the 'Model trained dir(s)' column set to the value of 'arg\_two'. For example, if we wish to plot the rows in 'model\_predictions.csv' where a complete file from a specific source directory (e.g. 'allmatfiles') is then assessed on models trained on 'NSAA' and '6minwalk-matfiles' files, we set arg\_one='allmatfiles' and arg\_two='NSAA,6minwalk-matfiles'. This then selects the lines from the DataFrame object that we are concerned with. From here, with these lines we extract the true and predicted overall NSAA values from both the model trained on NSAA directory files and the model trained on 6minwalk-matfiles directory files. These values are then plotted with the true values along the x-axis and the predicted values along the y-axis and is done for both models. We also extract the 'percentage of correctly predicted D/HC label for sequences' for each file and model this percentage distribution as both cumulative and non-cumulative distributions for both source directories. We then repeat the same process but for the columns representing the percentage of individual acts correctly determined, and finally plots some useful statistical values computed over the lines.

*plot\_model\_preds\_single\_acts()*: this is the second function to read in the 'model\_predictions.csv' file, but as we treat 'single-act' rows in the file differently than those that use alternative directories for assessment, it's easier to keep the functionalities separated. Hence, we first load in the file as a DataFrame object and select only the rows that have '<args.arg\_one> (act' in the name of the short file: this signifies that a single-act file has been assessed on a model, rather than a full source file. From here, for each line we extract from the row's cells the percentage of acts correctly predicted, the percentage of correctly predicted D/HC label for sequences, and difference between the true and predicted overall NSAA score. From these, we take one of the values for each of the single-act files and plot these values against the act-number. This is then repeated 2 more times for the other 2 extracted values over each of the 17 single-act files. This then leads to 3 subplots where the x-axis is the act number (between 1 and 17) and the y-axis is one of percentage of acts, correctly predicted, percentage of correctly predicted D/HC label for sequences, or diff between true/predicted overall NSAA.

*plot\_rnn\_results()*: this is the function that analyses the 'RNN Results' files and is responsible for the majority of graphs that show the performance of different RNN setups (e.g. sequence lengths, overlap proportions, number of features, types of raw measurements, etc.). The 'arg\_one' and 'arg\_two' args take the start and end experiment numbers of the file (once it has been loaded in as a DataFrame object) by looking at the 'Experiment Number' column to decide on which rows of the DataFrame object that we are concerned with. From here, for each row (which is associated with a model that has been created and tested upon), we extract the names of each measurement the model in question has used, the sequence length, and the results that it has produced. Then, based on the fourth provided arg ('xaxis\_choice'), we decide on what to plot along the x-axis: if it's 'seq\_length', then for each measurement (e.g. 'AD', 'jointAngle, etc.), we create a line and plot how well it performed at various sequence lengths with respect to different metrics (e.g. R^2, RMSE, etc.) based on the third provided arg 'out\_type'. If instead it's 'ft' (file type), 'seq\_over' (sequence overlap), or 'features' (number of features used), then a single line to plot is used instead over all the lines from DataFrame selected to plot the aspect of the data specified by the 'xaxis\_choice' against the metric specified by 'out\_type'.

**‘data\_balancer.py’**

**Overview**

One of the inherent problems with the dataset is the lack of 'variance' within the subjects for their overall scores. This is mainly a feature of how the NSAA scores are conducted and the variation of severity of Duchenne muscular dystrophy within the patients. As the individual activity scores range from 0 (can't complete the activity at all) to 2 (completes it perfectly) and as there are 17 activites in total, the overall cumulative score ranges from 0 to 34. However, in reality, most patients in the study have scores ranging between 15 - 24 for moderate Duchenne. When it comes to training a network on the subjects' data and testing it on new files, this causes a problem if the subject has a particularly low overall NSAA score (e.g. 3). In other words, the slight lack of variation in the data we have available may slightly limit the potential on new subjects with particularly extreme cases of Duchenne muscular dystrophy. Thus, this is an important aspect to cover when we wish to improve generalization performance of the models to new subjects.

A classic way in machine learning of helping to get around this is in data balancing. This is traditionally done for classification problems rather than regression problems, as we are doing here. However, we get around this by, for the purposes of rebalancing the data set, considering overall NSAA scores as class labels rather than scores to be regressed on. There are two ways we consider here to balance our data, which are outlined below:

Consider a dataset of 10 sequences of data (i.e. 2D structures of data of shape 'sequence length' x '# features') with scores: [3, 15, 15, 15, 20, 20, 34, 34, 34, 34]. We have 2 ways of approaching this:

*Downsampling*: counts the frequency of each number in the list and finds the lowest frequency; in the above case, it is '1' (as there is only 1 '3' in the list). Next, for each of the labels in the list above, we randomly select '1' sample of each label in the list and, more importantly, the label's corresponding 'x' value (i.e. a single sequence). Thus, we are reduced to a list of 4 sequences and with a label list of [3, 15, 20, 34] (note that there is only 1 of each sample because there was originally 1 '3' label. Hence, we now have a much smaller list, but an even spread of 'y' values for the samples we have remaining.

*Upsampling*: we start off the same, with finding the frequency of each number in the list, but this time considering the highest frequency in the list. In the above case, this would be '4', as there are 4 34's in the list. Next, for each label value in the list, we randomly sample a 'y' and corresponding 'x' value (being a sequence) a total of '4' times for each label. For example, for the '15' labels (i.e. 3 sequences and 3 '15' labels), we randomly pick a pair of 'x' and 'y' values from the 3 available and do this '4' times. Thus, we end up with a much larger list of [3, 3, 3, 3, 15, 15, 15, 15, 20, 20, 20, 20, 34, 34, 34, 34] of 'y' values with corresponding 'x' values (sequences).

Upsampling has the advantage of not discarding any of the data that has been given to us; however, it means that many samples are repeatedly used as 'new' samples, which may lead to unpredicable training results, along with an inflated data set may being more challenging to train on. Downsampling, meanwhile, might give better generalization results than non-resampled data while being a smaller data set (thus making it quicker to train models that achieve better results), but the discarding of data points might leave important insights from the data out of the training process.

**How it works**

The script contains 3 functions: 'ext\_label\_dist', 'downsample', and 'upsample'. The last two functions are more-or-less identical to their respective algorithms that are outlined above, with a few implementation details differing but the overall ideas being the sample; hence, we won't repeat the more-or-less same algorithm here. Instead, it's worth considering how each of the functions are used. The script is never run directly, but rather serves simply as a storage place for several functions that are fetched by 'rnn.py'; hence, it's instead useful to consider exclusively how 'rnn.py' use the functions. Also note that these are only run by the 'rnn.py' script if the '--balance' optional argument is provided.

It's also worth noting the distinction between 'y\_data' and 'y\_data\_balance' when used as parameters for 'downsample' and 'upsample'. 'y\_data' might be, depending on the output type that we are training towards (e.g. D/HC classification, overall NSAA score, or single act scores) a list of 1's and 0's, a list of values between 0 and 34, or a list of lists of 17 values between 0 and 2. Hence, we want a unified way of rebalancing the data that is irrespective of the form that 'y\_data' takes. Hence, 'y\_data\_balance' will \*always\* be the overall NSAA scores for the corresponding 'x\_data'; if, for 'rnn.py', the 'choice' arg is 'overall', then this will be exactly the same as 'y\_data', but for others it will contain the overall NSAA scores that are corresponding to the 'x' and 'y' values. The ‘y\_data\_balance' is then used in the algorithms outlined above to find the indeces of 'x\_data' and 'y\_data' to select to create the new lists of data.

The functions of 'data\_balancer.py' and how they are used by 'rnn.py' are as follows:

*ext\_label\_dist()*: for each file that the 'rnn.py' model is training on, read in the 'nsaa\_6mw\_info.xlsx' file, finds the relevant row in the table corresponding to the file name in question, and returns the overall NSAA score for this file name. This is then used as the label for each of the sequences that are extracted from the file in question, and the process is then repeated for every other file in the source directory, 'dir'.

*downsample()*: if the '--balance' argument is set as 'up', then this function is called that takes in the 'x\_data' and 'y\_data' created from sequences (as 'rnn.py' would normally create) and the additional 'y\_data\_balance' that we have create additionally to use to balance the script, and from these downsamples the data and produces two new lists of 'new\_x\_data' and 'new\_y\_data' via the algorithm outlined above.

*upsample()*: called in the same way as 'downsample()' but via '--balance=up', while taking in the same arguments but instead using the algorithm for upsampling as described above.

With the 'ext\_label\_dist()' and either 'downsample()' or 'upsample()' having been run the requisite number of times ('ext\_label\_dist()' once for every file in the source directory, 'dir', and only once for either of the other two), this data then replaces the original 'x\_data' and 'y\_data' in 'rnn.py', prints the new balanced shapes to the user, adds several output strings to be printed at the end of the script's running to show the before and after data balancing for the distribution of labels, and the execution of 'rnn.py' subsequently continues as usual.

**‘file\_renamer.py’**

**Overview**

One of the primary problems with working with .mat files as part of this project is the lack of standardization of file names as they were collected. We have primarily been dealing with 4 source directories containing .mat files: 'NSAA' (containing NSAA assessments of subjects), '6minwalk-matfiles' and '6MW-matFiles' (containing the 6 minute walk assessments of subjects), and 'allmatfiles' (containing the natural movement files of subjects wearing the suit). Each directory had its own primary way of labelling files but, at the same time, it wasn't necessarily consistent throughout the directory.

This posed a not-insignificant problem in that some of basic characteristics of the file were determined by its file name (e.g. whether it was a 'D' or 'HC' file came from reading its file name, along with what subject the file was associated with). Until development of this script, the solution was having multiple ways of processing every file name within the various scripts that need them. However, there's several flaws in this approach:

1. It was not particularly extensible to new files with new formats being added. If new files were added to one of the source file directories with a slightly different naming format, it would require going deep into several scripts in order to change how they extracted the subject name each new file was associated with, it's D/HC label, etc. This process ends up just adding more 'if...else' clauses to many already-cluttered parts of the scripts.
2. As a result of having to change numerous things in several scripts, the process was more prone to human error. For example, as a result of a small oversight and not correctly reading the 'D' part of a file name that corresponded to subjects with 'D' in their subject name (e.g. 'D5'), the script was incorrectly interpreting the D/HC label as being 'HC' rather than 'D' like it should have been; hence, the model was trained incorrectly due to labelling sequences incorrectly. In comparison, if we would have used 'file\_renamer.py' from the beginning, we would have easily spotted any files that have been renamed incorrectly and correct them before other scripts had the chance to misinterpret their labels.

**How it works**

The basic operation of the 'file\_renamer.py' script can be summarized as follows:

1. Read in the name of a source directory of .mat files of which we wish to standardize the names.
2. Gets the names of all .mat files within the directory and divides them into one of two categories: 'files\_kept' (i.e. the vast majority of files which we don't want to remove) and 'files\_to\_delete' (files which we want to remove from the directory. Note that this is only for certain files that have been previously determined to be too large, too small, or not 'relevant' files to either training or testing models (for example, files that contain 'AllTasks' in the 'allmatfiles' source directory, as these contain the same information as the other files in the directory but concatenated together for a single subject, so there's no need to use these as well as the others.
3. Based on the source directory name, apply a set of regular expression ('regex') rules to each file name that are in 'files\_kept'. These are unique to each directory, as there are some things that we need to check for in some directories but not in other. These regular expressions are a set of substitutions: they search the file name for a certain quality and, if it finds it, replaces it with another, before using this new string as the basis for the next regex. These regexes include: replacing non-capitalized subject names to capitalized versions (e.g. changing 'd4-003.mat' to 'D4-003.mat'), replacing 'NSA' with 'NSAA when found in a file name, changing instances of '-6MW.mat' to '-6MinWalk.mat' (as the type of activities they contain is the same whether it was sourced from '6minwalk-matfiles' or 6MW-matFiles'), and so on
4. With this new list of file names that we are to change 'files\_kept' to, we first remove the files within the source directory based on the file names within 'files\_to\_delete' and then, for each name in 'files\_kept' and its corresponding name in 'new\_files\_names' replace the name of the file in the former with the name in the latter. The result is that all of the files within the specified source directory are automatically changed based on the standard we predefined.

However, it's important to note that this script is not intended to be run more than once, and only at the beginning. Hence, it should be executed before any of the other scripts like 'comp\_stat\_vals.py' or 'ext\_raw\_measures.py' are used. This is because these scripts use the names of the files they are sourced from to create new files with names based on their source names; hence, for 'file\_renamer.py' to be useful, they should be used prior to other files being created that are based on the files that 'file\_renamer.py' wishes to rename.

To this end, 'file\_renamer.py' is only needed to be used once. For this reason, it's also included within 'setup.cmd' as part of the setup process and is applied before any of the other scripts for the above reason.

**‘settings.py’**

**Overview**

The purpose of this file is to hold many of the the variables that are used throughout the rest of the script. In particular, there are many variable names (such as 'source\_dir') that hold the same values throughout all of the scripts. These variables contain values that include directory sources paths, paths to certain files that scripts output information to, lists of sensor names that have been given to us via the 'MVN User Manual', and so on; the common factor is that they are all referenced as being the same values across several different scripts and are thus interpreted as system constants.

In storing these values in a separate file, we achieve three things:

1. It reduces the amount of overall 'clutter' within the scripts, especially when we need to reference large variables such as those holding large lists of strings, which makes the scripts themselves both easier to debug and to maintain.
2. For variables that are supposed to remain static, it reduces the possibility of accidentally changing them to suit the script they are currently being referenced in. For example, we are less likely to accidentally change the name of one of the 'raw\_measurements' when they are only accessed in other scripts and not modified and, if one is changed in 'settings', then this change is reflected out to all other scripts in the same way (e.g. preventing two scripts from each having their own versions of 'raw\_measurements', which could cause conflict in manipulating output files.
3. If they are required to change for whatever reason (e.g. if a new user has their 'local\_dir' in a different location to the default value, or if the batch size to be used across numerous scripts is modified to be something else), then it's much easier to do so in a single 'settings' script rather than tracking down and modifiying each respective variable in each script.

To access these values, each of the scripts calls the necessary variables from settings in the 'import' section of the script. The idea of scripts only importing the variables that it needs was that it enhances clarity (i.e. if 'from settings import \*' was used, we wouldn't as easily be able to see that 'local\_dir' comes from 'settings' as if we used 'from settings import local\_dir'). Additionally, it's also recommended that any user using this project and setup for the first time should first examine the relevant path names (such as 'local\_dir', 'results\_path', etc.) to ensure that the source mat files are contained in the expected location, the scripts can access the necessary output .xlsx and .csv files, and so on.

**‘predictions\_selector.py’**

**Overview**

With so many file predictions being made and stored in the 'model\_predictions.csv', it's become necessary to have a way to sort through them all and return the files that we are most interested in. This is why this script has been built: to filter rows of the table (each corresponding to a complete file prediction made using 'model\_predictor.py' or by extension the 'test\_altdirs.py' script) based on several arguments (e.g. the subject names we're interested in, the directory the subject was trained on, or the alt directories that the models were trained on if they are 'altdirs' rows) and, based on whether '--best' or '--worst' is provided, return the best 'm' rows according to output metric 'n', where these are provided as part of '--best'/'--worst' (e.g. '--best=30,overall').

In essence, this functions similarly to how an SQL query would operate as 'SELECT <a> FROM model\_predictions WHERE <condition>. However, the desire was to do this in Python so the whole pipeline would only require one language for implementation (no accounting for libraries built on top of languages like C++, e.g. for TensorFlow). Furthermore, this is easily possible via extensive use of the 'pandas' library to load in 'model\_predictions.csv' as a DataFrame object, which is excellent for the filtering of rows based on cell values, ordering rows by lowest/highest values in a specified column, and so on to make manipulation of the table as easy as using an SQL query. Additionally, this also means that anyone else running this system only needs to setup a single language/IDE in order to execute all of the scripts.

The idea from building this script is having an easy way to see some of the 'most relevant' rows of the table to the user. Presently, this just takes the form of console output, though easy modification to have these lines written to file is possible. This script is especially useful for when we have many files to 'sift' through in order to get an idea of which are the best or worst performing on a given metric. For example, one particular application could be using the script to look at all the natural movement behaviour files that have been assessed on models build on NSAA and 6-minute walk files (totalling ~400 files) and selecting the best 20 of these according to which predicts the overall NSAA score of that file closest to the true value for that file. This has the potential to help us identify the types of natural behaviour files (e.g. sitting and eating, playing, sitting and moving on the floor, etc.) perform the best according to the metric. Another application could be, for a given subject name from the NSAA directory and on models trained on the same directory but left out of the training set completely, which options make the subject be predicted closest to the correct score (e.g. if the models data are upsampled, downsampled, trained on single-act files, etc.).

**How it works**

The script itself is fairly simple with no functions to call or classes to instantiate; rather, it executes a series of 'groups' of instructions that carries out the above-outlined tasks based on the script arguments. These can be summarised as follows:

1. Loads in the 'model\_predictions.csv' file as a DataFrame object.
2. Filters the rows of the table based on the 'sfn' arg, which removes all rows where the subject name doesn't match the value of 'sfn'; alternatively, if 'sfn'=all, keep all rows at this point.
3. Filters the rows of the table based on the 'sd' arg, which removes all rows whose source directory column is different from the value of 'sd'.
4. If the 'mtd' is given (i.e. if we're concerned with 'altdir' rows), filters the rows of the table based on this arg, which removes all rows whose altdir column is different from the arg value. Note that the this arg is given as comma-separated values, which corresponds to the list values of the column in question.
5. Based on whether the optional 'best' or 'worst' args are given (or both), extracts the first part of the arg(s) as the number of best/worst lines in the table and the second part as the short name of the metric to use to determine which are the best/worst (i.e. by deciding which of the output columns of the table to use to order the rows).
6. For each of the remaining rows of the tables (i.e. after having been filtered by steps 1-4), now filter the columns of the table: the first four columns are kept (the subject name, source dir, model trained dirs, and measurements tested), followed by one of the output columns (the column in question is selected by the second part(s) of the best/worst args). These values are additionally preprocessed: e.g. if 'overall' is selected, then the absolute value of the difference between the true and predicted values in their respective columns are selected, while if we're using the 'percentage of predicted correct sequences' metric, the relevant column for 'percentage of predicted <D, HC> sequences' is used based on the true D/HC label for the row.
7. Creates a list of column names to create a new table of the top 'n' results that include the aforementioned 4 beginning column names from 'model\_predictions.csv', followed by column names of the output metrics with the names of the dir that the models that outputted this metric were trained using.
8. Finally, select the top or bottom (or both) 'n' number of lines based on the selected column metric, depending on which of '--best' or '--worst' has been selected and the number of lines to extract from each of them, having reversed the if needed for percentage metrics ('pacp' and 'ppcs'), before printing out the selected rows to the console as a DataFrame object.

**‘dis\_3d.py’**

**Overview**

One desire for the data that we have received as '.mat' files is to be able to plot the subject portrayed within the file as a real-time 3D plot. The aim of this is to hopefully allow us to do two things:

1. Visualize the subject within the data as doing certain activities in order to provide a reference (along with the console 'Plotting time...' output) as to what activities are taking place at which time
2. In plotting this, easily allow for anomalies within the data file to be detected; for example, if the subject suddenly 'jumps' position or the limbs appear extremely contorted, it might indicate corrupted data which might need to be 'cut out' of the file (or have the whole file discarded)

Though this functionality also exists within the 'comp\_stat\_vals.py' script, it was felt necessary to also provide the functionality as a separate script within the system; hence, a lot of the code that was required by the '--dis\_3d\_pos' optional argument within 'comp\_stat\_vals.py' is repeated for this script.

**How it works**

This script involves a series of basic steps that the data goes through in order to display a dynamic, 3D plot to the user. Hence, we shall explain it here as these steps which include the following:

1. Loads in a '.mat' file corresponding to the 'dir' and 'fn' arguments provided to the script. This is read in as a DataFrame object and is returned from 'preprocessing()' and passed to 'display\_3d\_positions()'.
2. Extracts the values from the 'position' column and reads this in as a 'positions' matrix (of shape '# of samples' x '69'), separates the columns of this new matrix into tuples of x, y, and z axes for each segment within positions for every sample, define connected segments via tuples of pairs of values, sets the boarders of the 3D plot (i.e. the x/y/z mins/maxes), plots the 3D figure from the first sample with connections between points defined by the tuples of pairs of values, and animates it by fetching a new sample to plot every '1/sampling\_rate' sections so the figure is animated in real-time while outputing to console the current time-stamp of the figure in seconds.
3. After ~5 seconds where the data is sourced, extracted, reconfigured to work in 3D, and animated, a new window will appear. This is the 3D plot that runs in real time. Note that one should also see as a console output the time stamp in seconds of where the plot currently is at. There is no current way to pause, slow down, or speed up the plotting, though one can change the viewing perspective by left clicking and dragging with the cursor or zoom in and out by right clicking and dragging with the cursor.

**Additional batch scripts**

Along with the Python scripts that make up the system pipeline, we also make extensive use of several batch scripts for automating some of the tasks and for setup. As these aren't particularly long or complicated, it isn't worth creating a separate README for each, but rather a single README covering all of them along with when we would use them.

*'setup.cmd'*: this script runs the necessary 'pip' package installation commands to setup all the external libraries needed for running the project. Specific versions of the packages are used to match the exact versions used as part of this project to avoid potential complications, although setting up the most recent versions of the packages would most likely work just as well. We also run the necessary system scripts on all setup source directories. This requires that the user has setup the source directories ('NSAA', '6minwalk-matfiles', etc.) in a base directory that matches the name of the 'local\_dir' global variable stored in 'settings.py'. Assuming that, the rest of 'setup' will extract the statistical values from each file in every directory, along with reducing the features of these, standardizing the names of the files, extracting all raw measurements from every 'AD' file, and dividing up files to extract single activities from 'AD' files.

*'model\_pipeline'*: given the necessary arguments given along with the batch script, the 'comp\_stat\_vals.py', 'ft\_sel\_red.py', and 'rnn.py' scripts are run in turn to assess the performance of the RNN model on extracted statistical values. Note that, as we generally will only run 'comp\_stat\_vals.py' once and at the beginning of the project's inception, this batch script isn't particularly required anymore and is kept in more for historical reasons.

*'file\_predictor'*: this is the standard way that we wish to take in a new subject's file from 'inception' (i.e. given to us directly as a '.mat' file) and make predictions about it. This is done by first extracting the statistical values from the file via 'comp\_stat\_vals.py', extracting raw measurements via 'ext\_raw\_measures.py', reducing the feature space of stat values via 'ft\_sel\_red.py', and finally using all of these measures that were previously deemed (in the discussion of experiment results) to be 'useful' as measures (i.e. AD, jointAngle, sensorMagneticField, and position) and assess the file using these measures. While this process can be done just as easily by running each script in turn, this way is easier as it only requires us to run one script manually and with only two arguments (the source directory and the file name). Note that this is done on models that have already been built by 'rnn.py' and will try to use models that have the file left out of the training process, but if it can't find any then uses standard models trained on all files.

*'file\_predictor\_altdirs'*: very much similar to 'file\_predictor' but takes an additional 'altdirs' arg as script input (for more info about this, please consult the README for 'test\_altdirs.py' or the relevant results discussion concerning assessing alternative directories). With this argument, the only difference is that the 'model\_predictor.py' script is called with the optional argument '--alt\_dirs' set to the batch script argument.

*'file\_predictor\_leaveout'*: again, very similar to 'file\_predictor', with the only difference being that, given the file name, models are built to distinctly leave out the file in question from any part of the model training. This ensures that the new file we are working with within this script is completely new to the models that will be predicting from in 'model\_predictor.py'.

**Model prediction set scripts**

In an effort to make the execution of the model predictions sets easier (which often require numerous new models to be created with 'rnn.py' and many separate file predictions to be made with 'model\_predictor.py'), we have created batch scripts to automate this process. This also holds the additonal benefit where any user can inspect what arguments we have run each script with and also enables them to run them for themselves to see if comparable results can be obtained (obviously requiring the setup of all other files via 'comp\_stat\_vals.py' and other necessary scripts beforehand).

The idea is that, for each model prediction set that we are running, all that is needed is therefore to just run the specified '.cmd' script. This will build the requisite models though sometimes it won't build any new models but will instead rely on models built by previous '.cmd' scripts; hence, it's recommended that each model prediction set batch file is to be run in numerical ascending order. Once a given model prediction set batch file has been run, with the necessary models built and file predictions made, the results will appear in 'model\_predictions.csv' as the final rows in the table. It's also worth noting the time discrepancies between some of the '.cmd' files: some will only be calling 'model\_predictor.py' multiple times, which is comparatively quite quick to execute. However, those that call 'rnn.py' many times will take a lot longer; for example, 'model\_predictions\_set\_3.cmd' needs to build 60 separate RNN models, each of which may take 10-15 minutes to run (assuming the user is building using a GPU), which could take 10-15 hours in total to execute the script.

Finally, the scripts don't take any arguments, as the Python script parameters have been decided in advance. For example, prior to executing model predictions sets 3 and up, we decided to test the models on the left-out subjects D3, D9, D11, D17, and HC6 (see the experiments results discussion set for an overview as to why these subjects were chosen). Hence, any changes that would be made to these '.cmd' scripts must modify each instance of the Python script that is called by the batch script in order to correctly alter these chosen script parameters.