**Name:** Daniel Heaton

**CID:** 01524921

**Degree:** MSc Computing (Machine Learning)

# MSc Individual Project – Initial Report

## Introduction

The purpose of this report is essentially three-fold: to provide an overview of the background research conducted for the project that includes both preparatory reading around the theory and practical application of recurrent neural networks and conditional random fields and also how they have been implemented in similar previous studies; to summarise the progress made on the project thus far, including all scripts written and an exploration of the results obtained thus far; and to summarise all of the immediate tasks to be working on, both in the immediate future and also longer term. The hope is that this report will provide a good 'way point' in the overall project that will aid in reassessing the current direction it is being taken in and help keep focus on immediate goals and metrics that should be met in the near future. It should also be noted that some aspects of the project have been left out of this report for brevity's sake; namely, an in-depth discussion of how each script works. However, this instead will be included in the code's accompanying documentation, and additionally it was decided to discuss the individual scripts in the context of the tasks they help to complete rather than the 'ins and outs' of each script. Finally, all the code, other supporting documentation, and detailed paper reviews can be found in the GitHub repository https://github.com/dan-heaton/MSc_indiv_project. Note that this does not include the relevant source data files, as those are not publicly available, and so access to those would be needed before being able to implement of the scripts in the repository.

## Motivation for Project and Purpose of Work

Before outlining the work that has been completed thus far for the project, it's useful to outline some of reasons why the project is being undertaken in the first place and some of the overall aims of the project. Broadly speaking, we wish to investigate the applicability of conditional random fields and recurrent neural networks to the features of human movement data provided by body suit measurements captured from subjects with Duchenne muscular dystrophy. This will hopefully provide not only evidence on the applicability of these models to this sort of data, but also should provide important insights into the data itself and the subjects providing it. These insights from the project will hopefully have direct positive impact in the ability to assess subjects via the north star ambulatory assessment (NSAA) and possibly provide other insights into other features of the condition.

Generally, in this project we are trying to gain insights into body suit data that is captured as '.mat' files (MATLAB data files) through the different measurements that are captured by the suit (joint angles, position, accelerometer values, etc.) of NSAAs or 6-minute walks of the subjects by sequence modelling using CRFs and RNNs. This use of sequence modelling is necessary to model the dependencies through time of measurements, and we are more likely to have a robust model if measurement values are treated as NON-independent with respect to time.

Narrowing in slightly on the scope of the project, a current aim of the project is to build a reasonably good model (with surrounding supporting scripts that are outlined later on) that, when presented with new, unseen '.mat' files of body suit data, can give a reasonably good approximation of individual NSAA activity scores and an overall NSAA score (i.e. the accumulation of all individual activity scores). A prominent limitation currently, however, is the overall lack of data files: we have no more than 50 complete '.mat' files in total of both 6-minute walks and NSAAs. This is primarily down to the fact that data collection is currently an ongoing process and a large repository of previously-collected suit data from subjects with DMD does not appear to exist that's publicly available. Hence, an implicit requirement of the project is to be able to make the most out of the data we have available, such as using it to train a model to predict different things, use different measurements contained within the '.mat' files, look at applying statistical analysis on the raw data, and so on.

Another aim of the project is to be able to use trained model(s) to gain insights into the most influential activities and measurements from the '.mat' files on overall NSAA score and to identify activities that correlate highly with overall assessment. In doing so, it could possibly enable the reduction of 17 activities needed for accurate overall NSAA assessment to far fewer if only a few are needed to correctly assess the subject. The conclusions that we could possibly draw from the project, therefore, hopefully have the potential to aid specialists in the scoring and assessment of subjects scores and possibly provide other insights into the condition.

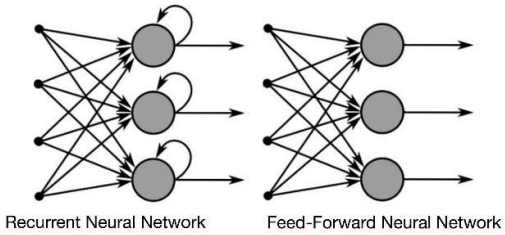
# Overview of Recurrent Neural Networks + Conditional Random Fields

As recurrent neural networks (RNNs) and conditional random fields (CRFs) will form the basis of the engineering aspects of the projects through implementations using Python and supporting libraries, it's necessary to outline some of the theory and applications of them prior to implementing them; in doing so, we can explore why exactly they are useful when adapted to work with the body suit data we're concerned with and to solve the problems we're looking to solve.

## RNNs: Theory and Applications

A problem with standard feedforward neural networks (FFNNs) is that they don't handle the order of data that is fed into the network: each sample fed through and classified or regressed is independent of all other samples. For example, if a convolutional neural network is trained to determine whether images are either of a cat or of a dog, then its assessment of each image's label is independent (rightly so) of the images its seen before. This is appropriate in many situations; however, here we're dealing with time-series data from the suit, where each row of joint angle values, position values, and so on that are contained in a '.mat' file produced by the body suit is a single sample in time. We also know instinctively that these values in real-life are dependent on previous values: for example, for a person in movement, the position of their various body parts are influenced by what they were at a short time ago. FFNNs don't handle order of the values of data that are fed in. For example, if there were inputted numerous rows of data from the body suit, then it would treat each row as independent entities with respect to network predictions.

This lack of memory with FFNNs is something that RNNs attempt to fix in the following way: rather than the values of the hidden nodes of the FFNN being only affected by the values that feed into it (e.g. the network inputs or the values from the previous layer), hidden layers in RNNs are also affected by their own previous values. This difference can be seen in the image below:



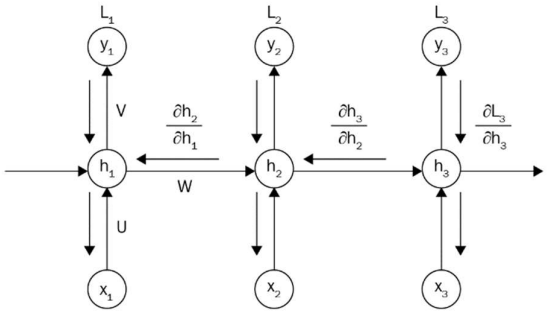

In this sense, the RNN can be seen to contain a memory of sorts that enforces time-dependencies of data that is fed through it. This can also be seen in the general equation that the hidden nodes in an RNN use to calculate the output:

$$h^{(t)} = \varphi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

It's also important to note that, not only is there a weight matrix '$W_{xh}$' that is learned through training to map from previous layer's values to the current one, but there is an additional weight matrix '$W_{hh}$' that is learned to control how much of the same layer's previous values impact the current layer. This requirement is also reflected in the backpropagation through time (BPTT) equation for the updating of this '$W_{hh}$' matrix by gradient descent:

$$\frac{\delta L^{(t)}}{\delta W_{hh}} = \frac{\delta L^{(t)}}{\delta y^{(t)}} * \frac{\delta y^{(t)}}{\delta h^{(t)}} * \left(\sum_{k=1}^{t} \frac{\delta h^{(t)}}{\delta h^{(k)}} * \frac{\delta h^{(k)}}{\delta W_{hh}}\right), \text{ where } \frac{\delta h^{(t)}}{\delta h^{(k)}} = \prod_{i=k+1}^{t} \frac{\delta h^{(i)}}{\delta h^{(i-1)}} = \frac{\delta h^{(t)}}{\delta h^{(t-1)}} * \frac{\delta h^{(t-1)}}{\delta h^{(t-2)}} * \cdots * \frac{\delta h^{(k+1)}}{\delta h^{(k)}}$$

The idea is that the overall loss 'L' is the sum of all the loss functions from times 't=1' to 't=T', and since the loss at time 't' is dependent on the hidden units at all previous time steps, the gradiant is as seen above. Note that its chain rule structure is still very similar to standard backpropagation used in FFNNs, with a primary difference coming from the impact of the all previous values of the hidden layer prior to time 't' on the overall derivative of loss with respect to '$W_{hh}$'. Below, we can also see how these derivative values propagate backwards through time.



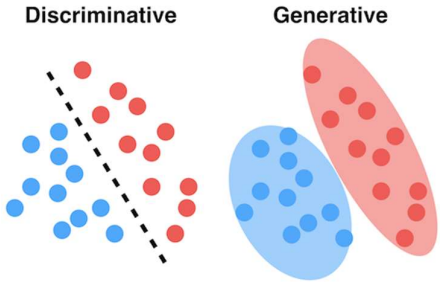

However, a large problem arises from the $\frac{\delta h^{(t)}}{\delta h^{(k)}}$ terms having 't-k' multiplications in the above equation, which therefore multiplies the weight matrix '$W_{hh}$' as many times. If this weight matrix is less than 1, this factor becomes very small, which results in a vanishing gradient, severely impacting the ability of the network to train on data and thus learn anything useful (the opposite happens if the weight matrix is greater than 1, which results in the network having an exploding gradient and never converging). For the purposes of implementing this in Python, long short-term memory (LSTM) cells were chosen to replace the hidden layers with a fixed size; this dramatically limits the potential for a vanishing or exploding gradient through the use of forget gate and thus helps in training the network.

With all that being said, there still must be a practical way of implementing them as part of the project for the RNN architecture to actually be useful for us. Fortunately, there exists numerous open source APIs and Python libraries that handles much of the underlying details of a neural network and simply needs the user to design the architecture. For this project, TensorFlow was chosen to be the API of choice due to previous experience in using it for implementing RNNs in time series data in other work, along with excellent supporting documentation being available and the ability to easily utilize a GPU to help with training the model. Additionally, we also have easy access

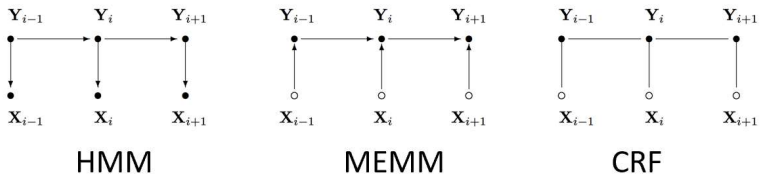
to other libraries that make reading in data from '.csv' and '.xlsx' files and manipulating it as matrix data easy (e.g. 'pandas' and 'numpy') and general-purpose machine learning libraries such as 'sk-learn' to help with other tasks, such as the splitting and shuffling of data for training/testing, the evaluating of various metrics like mean squared error, and so on.

## CRFs: Theory and Applications

The other model that we are considering as part of this project are CRFs. Much like RNNs, they are very capable at modelling sequence data and are also another example of a discriminative model. Discriminative models model the decision boundary between classes; examples of which include neural networks, CRFs, support vector machines, among others. This is in comparison to generative models, which models how the data was generated, which can then be used to make classifications; examples include Naïve Bayes and hidden Markov models (HMMs), which we shall return to shortly. The difference between the two is highlighted in the image below:




CRFs are a discriminative model that is useful in predicting sequences and use contextual information from previous labels, thus increasing the amount of information that the model has to help predict. Crucially, CRFs use logistic regression in the same way that HMMs use Naïve Bayes to predict sequences. CRFs can also be compared with a closely-related discriminative model called the maximum-entropy Markov model (MEMM): where MEMMs use per-state exponential models for conditional probabilities of next states given the current state, CRFs use a single exponential model to determine the joint probability of the entire sequence of labels given the observation sequence. CRFs therefore don't assume observations to be independent of each other, like in HMMs (which do so in order to feasibly consider all possible observation sequences): this is crucial to being applicable for the data we are concerned with [2]. Below we can see how the similarities and differences of how the three models depend on hidden states and observations:




Furthermore, CRFs help to solve a real problem in using MEMMs: the label bias problem. In a CRF, the weights of different features in different states compete against each other: MEMMs computes the probability of the next state given the current state and observation, while a CRF computes all state transitions globally in a single model. This helps to overcome the label bias problem in MEMMs which generates a bias towards states with few successor states. The idea is to drop the local per-state normalization functionality and replace it with global per-state normalization. In other words, it's an undirected graphical model that's globally conditioned on 'X', the observation sequence [2]. This results in the following equation for CRFs when considered as a graph 'G = (V, E)', given 'Y' is a set of labels and vertices in 'G' and are conditioned on 'X':

$$\boldsymbol{P(Y_v \mid X, Y_w, w \neq v) = P(Y_v \mid X, Y_w, w \sim v)}$$

As mentioned previously, CRFs computes likelihoods using principles from logistic regression; however, unlike logistic regression which uses $y = \theta x + b$ as input, CRF uses a feature vector. The feature vector is defined by a set of feature functions where each feature function '$f_j$' can analyse the whole 'X' observation sequence, the current '$y_i$', the previous '$y_{i-1}$', and the current position 'i' in the sequence. The purpose of this is to express characteristics of the sequence that the data point represents. This feature vector $\boldsymbol{F(X, y) = \sum_i f(y_{i-1}, y_i, X, i)}$ for a single feature is thus summed over the entire output sequence. We can then define an arbitrary number of feature functions. In this way, we have a 'global' feature vector that maps the entire sequence; thus, the fully-expanded CRF equation is:

$$P(y \mid X; w) = \frac{exp\left(\sum_i \sum_j w_j f_j(y_{i-1}, y_i, X, i)\right)}{\sum_{y' \in Y} exp\left(\sum_i \sum_j w_j f_j\left(y'_{i-1}, y'_i, X, i\right)\right)}$$

- $\sum i$ = sum over obs sequence
- $\sum j$ = sum over all feature functions
- $\sum y' \in Y$ = sum over all possible label sequence
- $w_j$ = weight for given feature function
- $f_j$ = feature function

Scores '$w_j f_j(y_{i-1}, y_i, X, i)$' will be high if the state transition is probable and low if not. The aim of training the CRF is to find 'w's that model the sequence accurately. Inference with CRF resolves to computing the 'y' sequence that maximises the likelihood: $\hat{y} = \boldsymbol{argmax_y P(y \mid X; w)}$. The minimum of the negative log likelihood is then found by finding $\frac{\delta L}{\delta w}$ and using gradient descent to find the optimal 'w' values for the training set, while the inference algorithm in CRF is based on the Viterbi algorithm.

With all that said, we can see how CRFs are better adapted to sequence modelling that HMMs and MEMMs, given that CRFs are both a discriminative model and also don't suffer the label bias problem. Additionally, it allows us to define our own feature functions. For example, if each observation is a word, we could consider as features the prefixes/suffixes of the word, whether it is capitalized or not, whether it contains a digit or not, etc. Feature functions can also inspect the entire input sequence 'X' at any point during inference. This is equivalent to the constant transition probabilities of HMMs that instead vary across position in the sequence of states, depending on the observation sequence.

Much like RNNs, CRFs are a good model to model sequential data, as we will be analysing in this project. They can also be adapted for regression purposes [5] to help predict north star scores and can also be implemented within other neural networks (e.g. as a convolutional neural network) to function as an RNN to enable a complete model that can be trained via backpropagation [3]. Furthermore, they are a lot faster to train and make inference from when compared with an RNN, though they also require manual feature extraction; this is a primary motivating factor in computing statistical features of the source '.mat' files used in the project, which we shall discuss later.

# Literature Review

An important aspect of this project thus far has been the background research; namely, the examination of other studies that can help to provide insight into how we should carry out this project while exposing us to other types of models and their applications with real-world data. This is, however, not a completed aspect of the project at this time and the process of further reading and reviewing papers shall continue for the months to come, with the idea being that they will help inform some high-level decisions in this project (for example, what dimensionality technique was used in a project that delt with human movement data and why, which could influence the technique chosen for this project). Also note that a more in-depth review of each of the papers mentioned below can be found in the GitHub repository mentioned previously under the directory 'paper_reviews'.

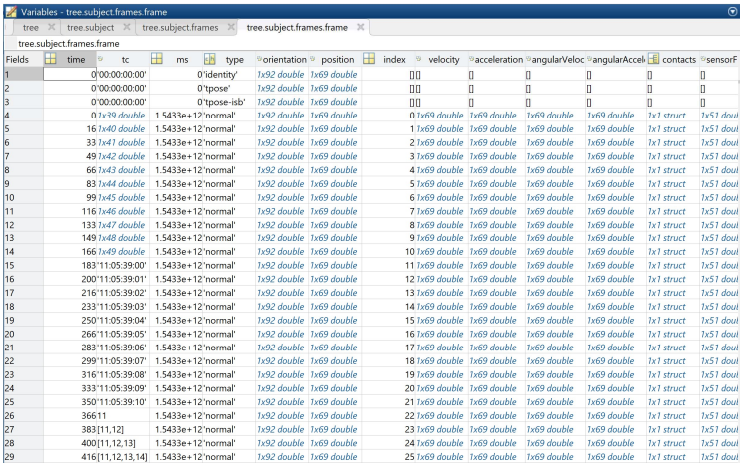
Deep learning is shown to be of high applicability to the more elaborate behaviours in human activity recognition (HAR), as it substitutes the manually designed feature extraction process with its own internal feature extraction, which is a great improvement over manual feature extraction as human activity lacks the robust physiological basis that benefits other fields such as speech recognition [8]. Another problem highlighted was the problem of training long sequences: training a sufficiently large RNN may result in it memorising the entire input-output sequence implicitly, leading to poor generalisation. A proposed solution to this was to introduce 'breaks' into the RNN where internal states are reset to 0, while the resets occur after every training batch with a fixed probability of occurrence; this is a technique to bear in mind if our RNN has trouble generalising when the sequence length is particularly long [8]. The importance of utilizing LSTM units to replace some layers of the RNN to solve problems of input-output weight conflict has also been explored, along with the pros and cons of segmenting what is usually sequence data, with models being trained on segmented data due to the ease of training with corresponding labels but then being able to adapt to work with sequential data [7]. Different ways of utilizing the outputs of RNNs have also been explored: one study takes full advantage of deep RNNs in modelling long-term contextual information of temporal sequences by proposing a hierarchical RNN system for skeletal-based action recognition, where the skeleton is divided into 5 parts. Each part is sent through separate RNNs, the results of which are then further combined in stages until we have only one output. This enables us to capture temporal representations of both low-level body parts and higher-level parts [9]. Bidirectional RNNs are also used here so each part of each sequence utilizies both past and future context [9]. Both of these architectural features need to be kept in mind as a possible implementation in our model to model both low- and high-level features as well as both past and future context.

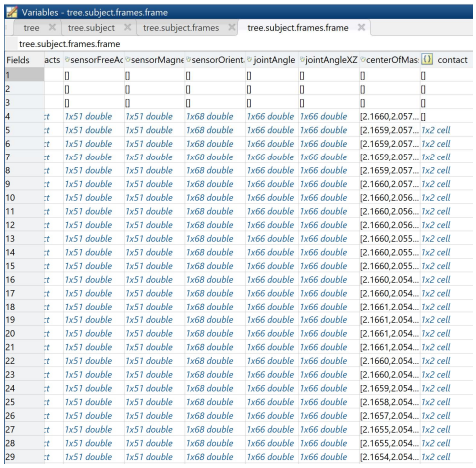
When later implementing a CRF to work with features extracted from the suit data, its worth considering using a variation called the mHCRF (modified hidden conditional random field), as this is mathematically defined as having a global optimum, which eliminates the possiblility of the HCRF being stuck in a local minimum at training time. The 'hidden' aspect of this is also capable of incorporating a single sequence label into the optimization of observation conditional probabilities, while it being a 'modified' HCRF eliminates the possibility of bad parameter initialization; these models have shown to outperform CRFs on human movement data due to its ability of discriminative learning of hidden state structures [1]. The addition of hidden states to a CRF and its advantages were further covered in [6], where the use of hidden states in a discriminative multi-class random field model showed its usefulness in incorporating long-range dependencies while also modelling latent structures of sequence data as done in HMMs. Another advantage of using CRFs is that they can be implemented as part of a post-processing step of a neural network; for example, CRF inference can be used to refine weak and coarse pixel-level label predictions from a convolutional neural network. The full algorithm of mean-field in dense CRFs is broken down into steps and implemented as CNN layers, which can in turn be reformulated as an RNN [3]. CRFs have also seen applications in the detection and classification of human movement patterns in smart homes, showing an improvement over using HMMs for the same task due to their ability to handle the substantial interdependencies between features of movement given by the sensors [4]. Furthermore, adapting CRF from classification to regression tasks has also shown to be possible and shows strong results especially when working alongside other discriminative models [5].

# Forms of the Source Data Files

Before moving on to how the data is actually used to train and test an RNN model, it's worth outlining the forms the data can take, what it actually represents, and how we treat these different forms differently. This is necessary so that the RNN model itself is able to treat these data files in the same way (though giving noticeably different results depending on the nature of the data file); this avoids the problem of having to create a unique RNN script for every different type of data file.

All the data that we are concerned with is captured by subjects wearing the Xsens MVN inertial motion capture system (i.e. the body suit). Each single file that we are provided is captured by one instance of a single subject wearing the suit (i.e. one subject's visit to the hospital). The files themselves are stored as '.mat' files within a tree structure containing the data measurements themselves, among other metadata including segment, sensor, and joint names, the time and date the data was captured, and other relevant metadata. An example of what the data values within a '.mat' file looks like can be seen below as the aforementioned 'data values' within the '.mat' file:




Note that each row is a single sampled captured by the bodysuit and 60 of these rows correspond to the data captured over 1 second (as the sensors sample at 60 Hz). The columns, meanwhile, primarily consist of measurements that are captured by the suit's 17 inbuild sensors. Note that there are different aspects that are measured by the suit depending on the measurement; for example, for 'position' the sensors measure the positions of 23 segments on the body suit which, given it's measuring in 3 dimensions, correspond to 69 values for a given time instance (i.e. a single 'row' of data), while 'jointAngle' measures using the 22 joints of the suit, which gives 66 total values. The result is that, over a single time instance of $1/60^{th}$ of a second, approximately 739 distinct values are captured by the suit. This is what we mean when we refer to an '**all data**' file (or **AD file** for short): it contains all the possible data captured by the suit for an instance of the subject wearing the suit.

In contrast to this, we are also provided with '.mat' files that correspond to the same subject's instance of wearing the suit but that only contain the 'jointAngle' measurements. The idea behind this is that, as it's believed that joint angles will be an important measurement for model training in several scenarios (classification of file type, regression of overall NSAA score, etc.), the '**joint angle'** files (or **JA files** for short) provide a simple jumping-off point to train some preliminary models (the results of which we shall see in the 'Results' section. The form of a JA file that corresponds to that of the AD file seen above is the following:

Variables - jointangle

jointangle

21809x66 double

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.1561 | -0.3084 | 3.2762 | 0.5116 | -0.1452 | 1.4568 | 0.5116 | -0.1452 | 1.4568 | 0.3834 | -0.1101 | 1.0927 | 5.7799 |
| 2 | 1.1480 | -0.2871 | 3.3059 | 0.5082 | -0.1358 | 1.4699 | 0.5082 | -0.1358 | 1.4699 | 0.3808 | -0.1031 | 1.1025 | 5.7892 |
| 3 | 1.1396 | -0.2685 | 3.3226 | 0.5045 | -0.1275 | 1.4773 | 0.5045 | -0.1275 | 1.4773 | 0.3781 | -0.0968 | 1.1081 | 5.8042 |
| 4 | 1.1307 | -0.2525 | 3.3379 | 0.5007 | -0.1204 | 1.4841 | 0.5007 | -0.1204 | 1.4841 | 0.3753 | -0.0915 | 1.1131 | 5.8272 |
| 5 | 1.1233 | -0.2397 | 3.3519 | 0.4975 | -0.1146 | 1.4902 | 0.4975 | -0.1146 | 1.4902 | 0.3729 | -0.0872 | 1.1178 | 5.8556 |
| 6 | 1.1177 | -0.2314 | 3.3666 | 0.4951 | -0.1109 | 1.4968 | 0.4951 | -0.1109 | 1.4968 | 0.3710 | -0.0844 | 1.1227 | 5.8862 |
| 7 | 1.1149 | -0.2292 | 3.3799 | 0.4938 | -0.1100 | 1.5027 | 0.4938 | -0.1100 | 1.5027 | 0.3701 | -0.0837 | 1.1271 | 5.9151 |
| 8 | 1.1146 | -0.2292 | 3.3908 | 0.4937 | -0.1100 | 1.5075 | 0.4937 | -0.1100 | 1.5075 | 0.3700 | -0.0837 | 1.1307 | 5.9427 |
| 9 | 1.1163 | -0.2323 | 3.4020 | 0.4944 | -0.1114 | 1.5125 | 0.4944 | -0.1114 | 1.5125 | 0.3706 | -0.0848 | 1.1344 | 5.9688 |
| 10 | 1.1196 | -0.2378 | 3.4112 | 0.4958 | -0.1139 | 1.5166 | 0.4958 | -0.1139 | 1.5166 | 0.3716 | -0.0867 | 1.1375 | 5.9936 |
| 11 | 1.1243 | -0.2453 | 3.4169 | 0.4979 | -0.1173 | 1.5192 | 0.4979 | -0.1173 | 1.5192 | 0.3731 | -0.0892 | 1.1395 | 6.0170 |
| 12 | 1.1255 | -0.2468 | 3.4254 | 0.4984 | -0.1180 | 1.5229 | 0.4984 | -0.1180 | 1.5229 | 0.3735 | -0.0897 | 1.1423 | 6.0402 |
| 13 | 1.1275 | -0.2509 | 3.4319 | 0.4992 | -0.1198 | 1.5258 | 0.4992 | -0.1198 | 1.5258 | 0.3741 | -0.0911 | 1.1445 | 6.0635 |
| 14 | 1.1303 | -0.2570 | 3.4369 | 0.5004 | -0.1226 | 1.5281 | 0.5004 | -0.1226 | 1.5281 | 0.3750 | -0.0932 | 1.1461 | 6.0869 |
| 15 | 1.1338 | -0.2647 | 3.4409 | 0.5019 | -0.1261 | 1.5299 | 0.5019 | -0.1261 | 1.5299 | 0.3761 | -0.0958 | 1.1475 | 6.1103 |
| 16 | 1.1379 | -0.2734 | 3.4437 | 0.5037 | -0.1299 | 1.5311 | 0.5037 | -0.1299 | 1.5311 | 0.3775 | -0.0987 | 1.1485 | 6.1338 |
| 17 | 1.1425 | -0.2829 | 3.4458 | 0.5057 | -0.1342 | 1.5321 | 0.5057 | -0.1342 | 1.5321 | 0.3789 | -0.1019 | 1.1492 | 6.1568 |
| 18 | 1.1475 | -0.2933 | 3.4472 | 0.5078 | -0.1389 | 1.5327 | 0.5078 | -0.1389 | 1.5327 | 0.3805 | -0.1054 | 1.1497 | 6.1789 |
| 19 | 1.1534 | -0.3054 | 3.4478 | 0.5103 | -0.1443 | 1.5330 | 0.5103 | -0.1443 | 1.5330 | 0.3824 | -0.1095 | 1.1499 | 6.1993 |
| 20 | 1.1597 | -0.3185 | 3.4488 | 0.5131 | -0.1501 | 1.5335 | 0.5131 | -0.1501 | 1.5335 | 0.3844 | -0.1139 | 1.1503 | 6.2183 |
| 21 | 1.1669 | -0.3335 | 3.4491 | 0.5162 | -0.1569 | 1.5337 | 0.5162 | -0.1569 | 1.5337 | 0.3867 | -0.1190 | 1.1504 | 6.2348 |
| 22 | 1.1745 | -0.3475 | 3.4491 | 0.5194 | -0.1632 | 1.5337 | 0.5194 | -0.1632 | 1.5337 | 0.3891 | -0.1237 | 1.1504 | 6.2503 |
| 23 | 1.1815 | -0.3601 | 3.4491 | 0.5224 | -0.1688 | 1.5338 | 0.5224 | -0.1688 | 1.5338 | 0.3914 | -0.1279 | 1.1505 | 6.2654 |
| 24 | 1.1880 | -0.3716 | 3.4497 | 0.5252 | -0.1740 | 1.5341 | 0.5252 | -0.1740 | 1.5341 | 0.3935 | -0.1318 | 1.1507 | 6.2800 |
| 25 | 1.1937 | -0.3813 | 3.4521 | 0.5277 | -0.1783 | 1.5352 | 0.5277 | -0.1783 | 1.5352 | 0.3953 | -0.1351 | 1.1515 | 6.2945 |
| 26 | 1.1985 | -0.3894 | 3.4545 | 0.5297 | -0.1820 | 1.5363 | 0.5297 | -0.1820 | 1.5363 | 0.3969 | -0.1378 | 1.1523 | 6.3088 |
| 27 | 1.2023 | -0.3957 | 3.4566 | 0.5314 | -0.1848 | 1.5372 | 0.5314 | -0.1848 | 1.5372 | 0.3981 | -0.1399 | 1.1531 | 6.3230 |
| 28 | 1.2054 | -0.3996 | 3.4581 | 0.5327 | -0.1866 | 1.5379 | 0.5327 | -0.1866 | 1.5379 | 0.3991 | -0.1413 | 1.1536 | 6.3371 |
| 29 | 1.2075 | -0.4018 | 3.4600 | 0.5337 | -0.1876 | 1.5388 | 0.5337 | -0.1876 | 1.5388 | 0.3998 | -0.1420 | 1.1542 | 6.3511 |

As can be seen above, the table now looks very similar to how a corresponding '.csv' file would also look, which lends itself to simplicity in reading the data into Python scripts and using it to train a model. Note that the dimensions are '21809x66', with '21809' corresponding to ~363 seconds worth of data over 66 dimensions (i.e. 3 dimensions of 22 joint angles).

These joint angles files are also contained within what we call a '**data cube**' (or '**DC**' for short). This single '.mat' file contains the joint angle data for 25 subjects and a table that contains information about them, as seen below:

data_cube | data_cube.joint_angles | excel_table

data_cube.joint_angles

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 21600x66 d... | 21600x66 d... | 21600x66 d... | 21600x66 d... | 21000x66 d... | 21600x66 d... | 21600x66 d... | 21600x66 d... | 3630x66 do... | 21600x66 d... |

The image above shows how the data cube contains cells that contains a whole joint angle file within them (that look similar to that seen above), while the table below shows the table contained in the data cube that contains useful information about the respective joint angle files. Hence, due to the useful structure and provided table, when we look to train an RNN model on raw joint angle data, we use the data cube as standard.

data_cube | data_cube.joint_angles | excel_table

25x7 table

| | 1 DMD_HC | 2 ID | 3 FileName | 4 WalkDurationIr | 5 StartFrame | 6 EndFrame | 7 Notes |
|---|---|---|---|---|---|---|---|
| 1 | 'DMD' | 'D2' | 'All-D2-6Mi... | 360 | 90 | 21689'' | |
| 2 | 'DMD' | 'D3' | 'All-D3-6Mi... | 360 | 50 | 21649'' | |
| 3 | 'DMD' | 'D4' | 'All-D4-6Mi... | 360 | 40 | 21639'' | |
| 4 | 'DMD' | 'D5' | 'd5-6MinW... | 360 | 70 | 21669'' | |
| 5 | 'DMD' | 'D6' | 'All-D6-6Mi... | 350 | 2590 | 23589'' | |
| 6 | 'DMD' | 'D7' | 'All-D7-6Mi... | 360 | 1000 | 22599'' | |
| 7 | 'DMD' | 'D9' | 'D9-019-6... | 360 | 3700 | 25299'' | |
| 8 | 'DMD' | 'D10' | 'D10-021-6... | 360 | 500 | 22099'' | |
| 9 | 'DMD' | 'D11' | 'D11b-001-... | 60.5000 | 1170 | 4799 | 'Kid fell ov... |
| 10 | 'DMD' | 'D12' | 'D12-014-6... | 360 | 610 | 22209'' | |
| 11 | 'DMD' | 'D14' | 'D14-014-6... | 360 | 1375 | 22974'' | |
| 12 | 'DMD' | 'D15' | 'D15-013-6... | 360 | 4 | 21603'' | |
| 13 | 'DMD' | 'D17' | 'D17-021-6... | 360 | 200 | 21799'' | |
| 14 | 'DMD' | 'D18' | 'D18-017-6... | 360 | 900 | 22499'' | |
| 15 | 'DMD' | 'D19' | 'D19-014-6... | 360 | 1130 | 22729'' | |
| 16 | 'HC' | 'HC1' | 'All-HC1-6... | 360 | 170 | 21769'' | |
| 17 | 'HC' | 'HC2' | 'All-HC2-6... | 360 | 175 | 21774'' | |
| 18 | 'HC' | 'HC3' | 'All-HC3-6... | 360 | 200 | 21799'' | |
| 19 | 'HC' | 'HC4' | 'All-HC4-6... | 360 | 170 | 21769'' | |
| 20 | 'HC' | 'HC5' | 'All-HC5-6... | 360 | 210 | 21809'' | |
| 21 | 'HC' | 'HC6' | 'All-HC6-6... | 320 | 50 | 19249'' | |
| 22 | 'HC' | 'HC7' | 'All-HC7-6... | 360 | 65 | 21664'' | |
| 23 | 'HC' | 'HC8' | 'All-HC8-6... | 360 | 2450 | 24049'' | |
| 24 | 'HC' | 'HC9' | 'All-HC9-6... | 360 | 350 | 21949'' | |
| 25 | 'HC' | 'HC10' | 'All-HC10-... | 180 | 10900 | 21700 | 'We have u... |

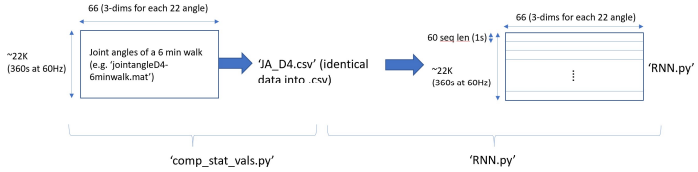
Finally, an important distinction to make is what sort of real-world activity each '.mat' file actually shows. The first type is '**6minwalk**', which as the name suggests is 6 minutes worth of data (so approximately 21600 samples) of the subject walking around a given area more-or-less continuously. This is provided to us as 'AD', 'JA' and 'DC' files, so for a given subject we can chose how we wish to interpret their data. The other type we are currently concerned with is '**NSAA**', which are files usually between a length of 3 and 10 minutes that contain the subject carrying out the 17 activities that are set as part of the North Star Ambulatory Assessment. These files, however, are only provided to us as 'AD' files and do not come in 'JA' or 'DC' form (though we can still extract the raw joint angle measurements from an NSAA file via the 'ext_raw_measurements.py' script; more on this later).

With the forms that the data can take now summarized, we can move onto what we actually do with this data prior to it being used to train an RNN.

# The Data Pipeline

What is referred to as the ‘data pipeline’ is shorthand for three Python scripts that read from data files, manipulate data, and write to new files. The aim of the pipeline is to convert the data that is specified by the user of the script (via arguments passed to the Python scripts) into a format that is usable by the RNN model. The specifics of what each file does are not covered here for the sake of brevity, so here we instead focus on the different shapes and forms the data goes through depending on whether it came in as an ‘AD’ or ‘JA’/’DC’ file (as the ‘DC’ simply contains multiple ‘JA’ files, we treat both ‘DC’ and ‘JA’ files the same way).

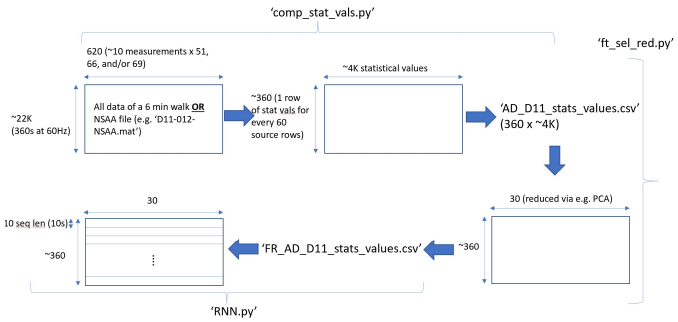
Below, we can see the pipeline with respect to ‘JA’ or ‘DC’ file(s) as input to the pipeline.




This is what is done when we are dealing with **raw joint angle** data. Let’s say we wish to feed in the ‘JA’ file for subject ‘D4’, as seen in the diagram above. The data comes in the form of a 22K x 66 matrix, as can be seen in a previous image, which is loaded by the ‘comp_stat_vals’ Python script and simply written back out in exactly the same way to a ‘.csv’ format. In essence, we are removing some of the metadata from the ‘.mat’ file and then transforming the data values into ‘.csv’ format. The reason we do this is because the ‘RNN’ is expecting to load data from ‘.csv’ format when dealing with ‘AD’ files, so this essentially standardises the process so the ‘RNN’ works in the same way for each different original data file type.

In the above case, this data is then loaded from ‘JA_D4.csv’ and has sequences made out of it. In this context, we refer to a **sequence** as a 2-dimensional grouping of data that the RNN will treat as a single sample of data. In the above case, the sequence is a ’60 x 66’ matrix of values: each row of 66 values are fed into the RNN one after another, repeating for a total of 60 times before the output of the RNN is observed; the RNN is essentially ‘reset’ before the next matrix of values is considered, hence the sequences are essentially independent of each other. In this case, the data is time dependent of each other only in 1 second intervals (given a 60 Hz sampling rate of the body suit), while outside of these sequences the data is treated independently of each other (in the same way as each image being classified by a convolutional neural network is independent of each other). Note the 1 second sequence length size is just an initial value and will be subjected to much more experimentation in the future.

Next, we can see below the pipeline with respect to ‘AD’ files(s) as input to the pipeline:




The first point to notice is that the data starts with far more dimensions than ‘JA’ or ‘DC’ files. This is because the ‘AD’ files consider ALL the measurements and not just one measurement (the joint angles). With using them as input to ‘comp_stat_vals’, we process them differently; namely, we don’t just write them directly to a .csv, but rather calculate statistical values. These operations (which including finding the mean, variance, first/second eigenvalues of covariance matrices, fast Fourier transform values, mean sum absolute values, and so on) operate on only a limited number of rows of the data at a time. For example, in the above case, we select a given number of

rows of the 22K starting samples at a time (in this case, 60 to correspond to 1 second's worth of data), calculate statistical features over every single one of its 620 dimensions (along with between some of the 620 dimensions), and compute this as a row of approximately 4000 statistical values. This is then repeated for every other 60-sample part of the original 22K to produce a total of approximately 360 rows each of 4000 statistical values.

Much like the sequence length that the RNN processes, the length of time to compute the statistical features over is a parameter that is set as an argument to the script (i.e. a hyperparameter that we set ourselves). Hence, the size of '60' is just an initial value and we intend to experiment with different settings in the future. Something to bear in mind, though, is that if we increase this parameter so the statistical values are calculated over a longer time period, we will reduce the number of rows that are outputted from this script, hence reducing the amount of data available at the next stage of the pipeline (i.e. the data going into the feature reduction script outlined below).

With this data having its statistical values computed and outputted to a new file (in this case of the above image, 'AD_D11_stats_values.csv'), we then load this .csv into a new Python script called 'ft_sel_red'. The purpose of this is to simply reduce the dimensionality of the data while preserving as much useful information as possible and rewriting this to a new file. While there are several options in feature reduction and feature selection (including principal component analysis, random forest feature selection, feature agglomeration, among others), we are currently just using PCA as standard based on its prevalence in other studies, though this will be subject to further investigation. Hence, when run on the input data and with a target reduction size set to 30 dimensions (again, this is a hyperparameter set as a script argument and subject to further experimentation), the data is transformed from a dimensionality of '360'x'~4000' to '360'x'30', which is far more conducive to being used as training data to the RNN in the next stage, as this reshaping helps minimize the effects of the 'curse of dimensionality'.

Finally, this is fed into the RNN and sliced up into sequences of length that we defined in the same way as in 'JA'/'DC' as previously described. A choice of '10' is used here as the sequence length due to the limited size of the data ('360'x'30') in comparison to the case of reading from raw joint angle files ('22000'x'66'). Also, it's worth noting that even though the sequence length is only '10' here, as each of the rows here have a full 60 rows worth of data from the original AD file (which corresponds to 1 second), each sequence from an AD file to the RNN has 10 seconds worth of data encoded into it as statistical values. In comparison, the raw joint angle data has a sequence length of 60 and encodes only 1 second's worth of data. This evidently plays a role in the difference in their respective results, which we shall discuss shortly.

# Recurrent Neural Network (RNN) – Setups and Variations

Now that the data has been prepared as a series of sequences of data either from a single AD, JA, or DC file, or multiple of each, we now look at what the recurrent neural network model we have built actually does. It's important to note that each of the variations works with every type of input file, be they raw joint angle files or AD files capturing either 6minwalk or NSAA data; that is, after all, a primary purpose of using the pipeline. It's also necessary to note that the trained model in every variation operates on a sequence-by-sequence basis even for testing; that is to say, it doesn't classify or provide a regression score for a complete file but rather for a sequence of a pre-specified length within said file. The classifications or regression values of each of the sequences of the test file can then be aggregated together to provide a classification or score for the whole file (how this is exactly carried out is subject to further research). Finally, it's worth noting that each variant is implemented within one Python script called 'RNN', and the necessary architectural differences are setup based on arguments passed into the script.

The three main variants that we have developed at present are described as follows:

- **Classification of sequences of being from 'DMD' or 'HC' subjects**: the purpose of this variation is to classify sequences as being from a file that is from either a 'DMD' or 'HC' subject. Consider the previously outlined case where 'FR_AD_D11_stats_values.csv' is fed in from the pipeline to the 'RNN' at the end. When it is

loaded into the 'RNN' script, it's divided into '36' sequences of length '10' to give a data shape of (36, 10, 30): this is the 'x' data in the sense of a neural network. For the 'y' data, we simply look at the title of the file this data originated from to provide a label of either '1' or '0' depending on the nature of the file name: since it's a 'D' file due to it being about patient 'D11' it gets a label of 1, as can be seen in the corresponding code, `y_label = 1 if file_name.split("_")[2][0] == "D" else 0`. This is then repeated for each sequence to obtain a list of '1's of length 36. We then repeat this process for all other files pushed through the data pipeline, some of which will be from 'D' subjects and others from 'HC' subjects. The result, in the case of doing this over all AD files for files corresponding to NSAA instances, is an input shape of (742, 10, 30) for 'x' and (742, ) for 'y', which contains a mixture of 1's and 0's for each sequence. There is also only a single neuron output for the network that contains categorical value of either 0 or 1 for a given sequence: this output will go to 0 if the sequence inputted is predicted to be from a 'HC' subject or 1 if predicted to be from a 'D' subject.

- **Overall NSAA regression score**: here, the RNN is tasked with taking a sequence and trying to predict the overall NSAA score of the subject that it comes from. This score corresponds to the accumulation of the scores of the 17 individual activities done by the assessment. As each activity is scored either a 0, 1 or 2 (with 2 being a perfect score), the overall NSAA score will range from a 0 from a subject with severe DMD and a 34 for a subject that shows no symptoms. Using the above example, we start with a shape of (742, 10, 30) over all the NSAA AD input files. For each of these 742 sequences, we then check a table that is provided as part of the NSAA dataset: this table provides a list of the subjects, their testing details, and crucially their individual NSAA scores (17 of these between 0 and 2) and overall NSAA score (1 of these between 0 and 34). This overall score is then used for every sequence from a given file that is inputted to the 'RNN'. Using this, we then obtain 742 values between 0 and 34, with each value being the overall NSAA score of the source file of its corresponding 'x' component of shape (10, 30). From here, we again have in the RNN architecture a single output neuron, but this time it outputs a regression value (rather than a classification value in the previous case) between 0 and 34; hence, each sequence that passes through the RNN will result in it making an estimate of the overall NSAA score of the patient that the sequence originates from.

- **Classification of NSAA single activity scores for all 17 actions**: in a similar vein to predicting the overall NSAA scores, the RNN is also able to train towards predicting individual activity scores; that is to say, given a single sequence, the RNN will output an array of 17 values, each being either a 0, 1, or 2, that corresponds to its prediction of the individual activity scores of the subject that file corresponds to. Again, given the same 'x' data fed through the pipeline, the corresponding 'y' values to train and test on are obtained from the table that is provided with the NSAA dataset to obtain the necessary array of 17 values for each sequence. Hence the data fed into the 'RNN' now has a shape (in the case of all NSAA AD files being used) of (742, 10, 30) for 'x' and (742, 17) for 'y'. To account for this, the RNN is modified to predict 17 individual classification labels of either 0, 1, or 2 for 17 output neurons. It should also be noted that experiments have not been carried out using this architectural setup yet and the results of which might be somewhat unreliable, as it might not be possible for a single sequence that we test with to be able to predict the scores of activities that it may or may not contain any data about; hence, further experimentation is required.

Currently, all 3 variants have many of the same hyperparameters, including the test-to-train ratio of data (0.2), number of units in each LSTM cell (128), number of hidden layers (2), and learning ratio of the 'adam' optimisation algorithm (0.001). The reasoning behind this was that differences in experiment results will hopefully be down to differences in source data file types (e.g. raw joint angle files of 6minwalks vs AD files of 6minwalks) rather than potentially different architectures (e.g. if one had more hidden layers, increased performance may be resulting from this rather than the source of the data going into the RNN, which we want to be able to compare). There are some differences though: a policy decided on early was that the RNN should train until its loss more-or-less converges. This required different number of epochs dependent on the source data type. For example, if the data going into the RNN came from raw joint angles, it only needed approximately 20 training epochs to converge, whereas data from AD statistical value files needed between 200-300 to converge. Finally, it's also worth pointing out the main difference in training for training classification (either for single or multiple output nodes) and regression was the different loss functions used, in that classification used binary cross entropy and regression used mean-squared error.

# Results of Experiments Undertaken

With the data pipeline and RNN architectures defined and implemented in Python scripts, we were then able to begin testing with varying sources of data and with different RNN architectures. Presently, only 8 of these experiments have been tested and there are numerous more still to undertake, but they are enough to start to draw some conclusions and point the project in new directions.

| Description | Results |
|---|---|
| Raw joint angles from DC to perform D/HC classification | Test Accuracy = 99.88% |
| Raw joint angles from DC to perform overall NSAA score regression | Mean Squared Error = 0.4762, Mean Absolute Error = 0.4037 |
| Raw joint angles from JA to perform D/HC classification | Test Accuracy = 100.0% |
| Raw joint angles from JA to perform overall NSAA score regression | Mean Squared Error = 0.1675, Mean Absolute Error = 0.2919 |
| Stat values from 6minwalk-matfiles\AD to perform D/HC classification | Test Accuracy = 82.81% |
| Stat values from 6minwalk-matfiles\AD to perform overall NSAA score regression | Mean Squared Error = 29.4065, Mean Absolute Error = 3.56 |
| Stat values from NSAA\AD to perform D/HC classification | Test Accuracy = 92.97% |
| Stat values from NSAA\AD to perform overall NSAA score regression | Mean Squared Error = 28.7121, Mean Absolute Error = 2.9016 |

There are several things that can be noted straight away from looking at the results. The first is the difference between using raw joint angles for classification and NSAA overall score regression and using extracted statistical values from the same subject. We first compare row 1 and row 5, which are data from the same subjects that train a model to perform the same tasks with markedly different results. The only difference between the two is that, in the first case, the model is trained on raw joint angles (provided by the datacube file) for '6minwalk' data, while the second uses extracted statistical values of many measurements for the same subjects. In using just the raw data values, we achieve a 99.88% accuracy (i.e. for each sequence of 60 rows of 66 joint angle values, the model can predict with 99.88% accuracy whether it came from a 'D' file or an 'HC' file); however, looking at more measurements (e.g. position, accelerometer values, etc.), performing manual feature extraction via computing of the statistical values and then reducing the dimensionality, and then training the model provides a much worse classification accuracy of 82.81%. This can also be seen when the same data sources are then used to train the RNN to perform regression for the overall NSAA score: the raw joint angle data gives a much smaller mean absolute error of 0.4037 (meaning that it predicts a score of between 0 and 34 which on average is 0.4037 away from the true value in either direction), compared with a much worse MAE of 3.56 from 6minwalk AD statistical value files.

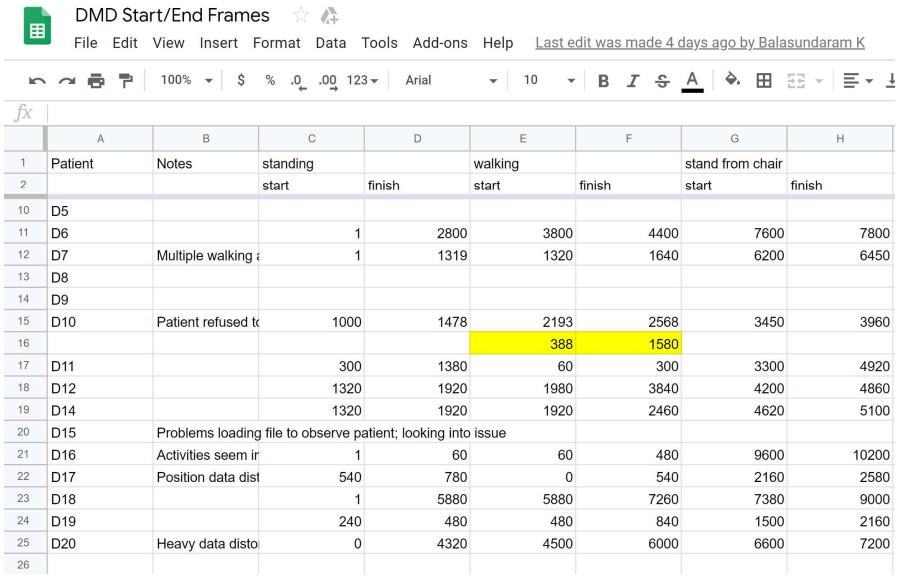
This may seem counterintuitive at first glance, as the former is just using data direct from the provided '.mat' files, while with the latter we actually process it further to ideally extract more important features from the data. However, a prominent benefit of using neural networks is that they are noted to perform better with raw data rather than manually extracted features. This most likely goes a fair way towards explaining this discrepancy in accuracies and mean errors. It should also be noted that training the RNN with raw joint angle data requires far fewer training epochs (~20) than for statistical values (~100). This will primarily be due to the larger amount of raw joint angle data that is fed through the network; in the case of training on all files within the data cube, the 'x' input shape is (8470, 60, 66) while for the corresponding AD files we only have (552, 10, 30) samples due to how computing statistical values dramatically reduces the raw amount of available data. This decrease in amount of available data also might help to account for the reduction in accuracy when using data sourced from AD files.

A further observation can be made about the experiments concerning the raw joint angle files (rows 1 to 4 in the table) in that they were performing much better than we were expecting them to be: by simply considering only

the joint angle measurement of a subjects suit data, given 1 second's worth of an input sequence to the RNN, it can correctly classify whether the frame comes from a healthy control subject or one with DMD to a very high accuracy of 99.88% and predict the overall NSAA score to within 0.4037 of the true value of between 0 and 34. This is extraordinarily high, much better than the ability of medical professionals and, most notably, this is only the first iteration of the experiments with raw measurement files. Hence, it is more likely that an oversight was made in the coding of the RNN or an incorrect assumption about its performance was somehow made. This investigation into the cause of this is an important next step in the project and further expanded upon later on.

# Data Preprocessing Work and Tools Used

One of the disadvantages of the AD files that have been given to us that show the NSAA activities specificially is that all the activities for each subject are provided in the same '.mat' file and the table that provides information of these subjects' trials with the body suit (the 'KineDMD data updates Feb 2019.csv' fle) contain information about the overall cumulative NSAA score as well as the individual activity scores (among other meta data), but not the specific times of occurrence of each activity. These activity times are needed in order to 'divide up' the original AD files into AD files that contain only the timeframe of a specific activity, which would be useful to have for this project (as expanded upon later on when discussing next steps of the project), as well as other group members' projects. Hence, we were tasked as a group to work together to create a table of each subject's predicted times for each activity. A portion of the results, contained in a shared Google sheet called 'DMD Start/End Frames', can be seen below:


DMD Start/End Frames

File Edit View Insert Format Data Tools Add-ons Help Last edit was made 4 days ago by Balasundaram K

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Patient | Notes | standing | | walking | | stand from chair | |
| 2 | | | start | finish | start | finish | start | finish |
| 10 | D5 | | | | | | | |
| 11 | D6 | | 1 | 2800 | 3800 | 4400 | 7600 | 7800 |
| 12 | D7 | Multiple walking a | 1 | 1319 | 1320 | 1640 | 6200 | 6450 |
| 13 | D8 | | | | | | | |
| 14 | D9 | | | | | | | |
| 15 | D10 | Patient refused to | 1000 | 1478 | 2193 | 2568 | 3450 | 3960 |
| 16 | | | | | 388 | 1580 | | |
| 17 | D11 | | 300 | 1380 | 60 | 300 | 3300 | 4920 |
| 18 | D12 | | 1320 | 1920 | 1980 | 3840 | 4200 | 4860 |
| 19 | D14 | | 1320 | 1920 | 1920 | 2460 | 4620 | 5100 |
| 20 | D15 | Problems loading file to observe patient; looking into issue | | | | | | |
| 21 | D16 | Activities seem in | 1 | 60 | 60 | 480 | 9600 | 10200 |
| 22 | D17 | Position data dist | 540 | 780 | 0 | 540 | 2160 | 2580 |
| 23 | D18 | | 1 | 5880 | 5880 | 7260 | 7380 | 9000 |
| 24 | D19 | | 240 | 480 | 480 | 840 | 1500 | 2160 |
| 25 | D20 | Heavy data disto | 0 | 4320 | 4500 | 6000 | 6600 | 7200 |
| 26 | | | | | | | | |

This is only a portion of the table, and there are many more columns that aren't displayed, but the above can be interpreted fairly simply. For example, for the 'D11' subject's NSAA AD file, the file is observed via a '3D dynamic plotting' function in 'comp_stat_vals' (to be discussed shortly) to show the subject do the 'standing' activity between frames 300 and 1380 (corresponding to between 5s to 23s) and the 'stand from chair' activity between frames 3300 and 4920. There were several rules that were followed to extract these frame times by human observation:

- As activities were often repeated by the subject, we consider only the first completed activity as the start and end points in the sheet; for example, if the subject tried to do the activity 'hop on right food' several times without success before being able to do so, we only count the last successful attempt

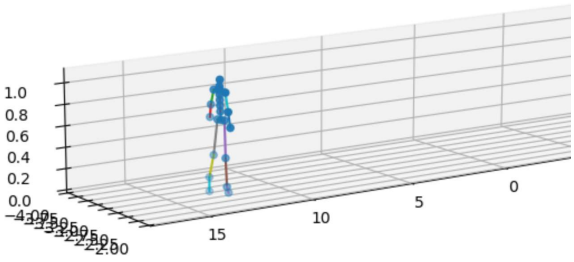
- Ideally, try to give a small amount of 'leniancy' on the start/end times of the activity; the idea behind this is that no part of the activity is therefore 'missed' when it's divided up into smaller AD files

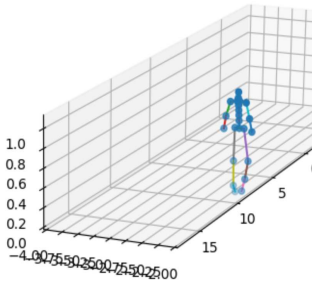
- Some of the activities were either not seemingly performed or performed alongside another activity, or performed very subtlely; for example, the 'nod head activity' was particularly tricky to detect in many subjects. In these cases, a best guess on the time of occurrence of the activity was made; the checking of how we divided up these files is another task of the project to investigate how accurate the above sheet is.

So given that we have the 'rules of thumb' for our annotation work, it was next necessary to actually 'visualize' these AD files. As there was no easy way of 'running' one of these AD files in '.mat' format, the project necessitated a function that could use the thousands of 'position' values of the AD file ('position' being one of the measurements in an AD file along with 'jointAngle', among others) to feed into a function that animated a stick figure as it moved around three-dimensional space. This was the work of the 'display_3d_positions' function in the 'comp_stat_vals' Python script and, when the script was run with the required arguments, e.g. for the 'D11' subject as `python comp_stat_vals.py NSAA AD D11 --dis_3d_pos`, the script does the following:

1. Load the position values from the AD file for subject 'D11' into a massive array
2. Group each of the columns corresponding to 'x', 'y' and 'z' dimensions of all segments together
3. Send these to the 'animate' function that animated it at a rate of 60Hz (so the animation would appear as real-time) and closed upon completing the plotting of all values for that file

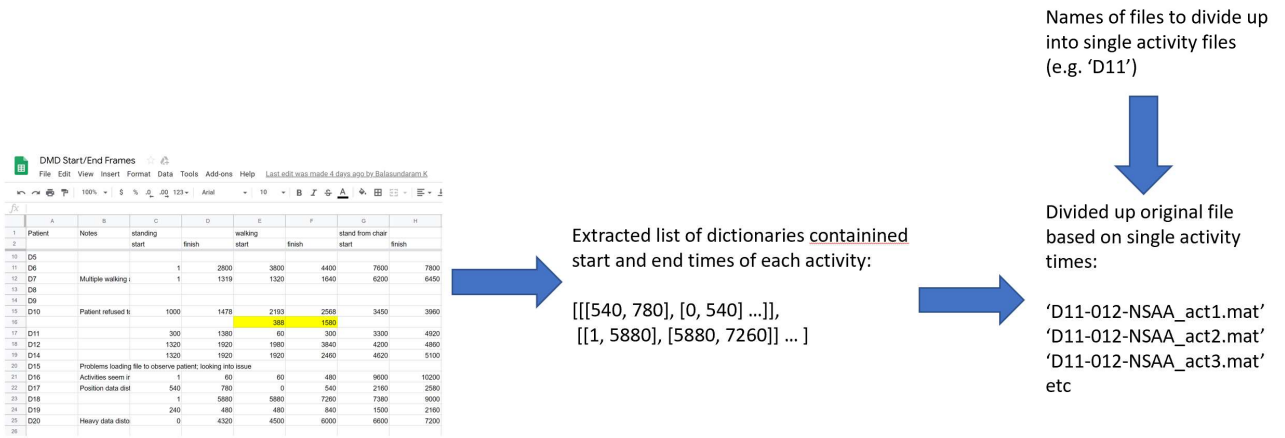
The result of this was 3D data in a new window that looked like the following:




Additionally, the current running time of the animation function was also printing to the console; this enabled us, by observing the movements of the stick figure and what activites it seemed to perform, to get a reasonably accurate idea of the times of occurrence of each activity.

With the Google sheet now being complete with the help of the above plotting functionality, we could now use the sheet as a reference tool for a script that could 'divide up' a given source AD file: the 'mat_act_div' script. The rough functionality of the script can be summarised in the diagram below:



When the script is run with an argument given to be the name(s) of the file(s) to 'divide up' based on the annotated Google sheet, it does two things:

1. Reads in the Google sheet that we had previously annotated, locates the relevant row given the provided arguments, and extracts a list of the activity times (both start and end times) as pairs of integers
2. Loads in the relevant AD file for the given argument(s) and, using the pairs of integers in the above list, slices up the file into 17 non-overlapping parts and writes these to a subdirectory of the file(s)'s source directory with names specific for the activity and source file (see the names in the bottom-right image above)

Currently, these new 'divided up' AD files are not being used, as they've yet to be verified that they are 'correct' (to do this, we need to observe each of these new files in the 'display_3d_position' function described above to see if they do indeed contain the expected activities); this is one of the next immediate tasks to complete as part of the project. However, once we have done this, these files will be able to be used to train an RNN to predict single activity NSAA scores (i.e. classification of sequences of these single-activity AD files as being one of a 0, 1, or 2 score; more on this later in the 'Next Steps and Further Work' section).

# Next Steps and Further Work

Having obtained an initial framework and results that will serve as a good basis for the next stages of the projected, we now turn our attention to what comes next. The immediate next pieces of work to do for the project are first outlined, followed by a higher-level interpretation of the stages of the project to move onto after this.

The immediate next tasks to complete include the following:

- **Add more documentation to the project**: presently, all 5 of the scripts that presently make up the project require significantly more commenting and explanation that they currently have. In doing this, it will not only help if and when we need to modify them later on, but also will help in explaining the scripts to others as and when needed. Additionally, another important task includes the adding of more scripts to the 'README.txt' file; currently, it only contains information on the 'comp_stat_vals' script, and we require more on the other scripts so that if and when others wish to use the scripts, it will be easier for them to do so.

- **Implement sequence overlap for the RNN data**: presently, we have only trained and tested our RNN model on sequences that do not overlap each other; that is to say, there is no row data within a sequence that is then used in any another sequence. For example, if we chose to have a 50% overlap of sequences, that would provide two advantages: 1.) it would help diminish the effects of NSAA actions in NSAA AD files that appear across two or more sequences essentially being 'cut', due to the new overlapping sequence being more likely to capture it in full within at least one sequence, and 2.) it would increase the overall amount of data available to the model (in the case of 50% overlap, it doubles the amount of data, and in the case of 75% overlap, quadruples the available data).

- **Investigate the high-accuracy of preliminary results**: we've seen from the results given thus far that the model performs surprisingly well when just given the raw joint-angle data (e.g. classifies a sequence as being from either a 'D' file (i.e. from a Duchenne's subject) or from a 'HC' file (i.e. from a healthy control subject) with 99.88% accuracy and predicts the overall NSAA assessment score to within 0.4037 of the true value). Based on comparisons with other results, these are unexpectedly high results that necessitate further investigation. Possible causes could be class imbalance, a too narrow scope of training and testing (i.e. which would mean the model fails to perform on new patients' data), or a simple engineering oversight.

- **Use other forms of raw data to train and test the RNN model:** from looking at the results, even accounting for potentially 'too good' results from raw joint angle data, there is a clear discrepancy between using 'raw' data and data that has the extracted statistical values via the 'comp_stat_vals.py' script. This is consistent with our prior knowledge of neural networks in that they often work better with raw data rather than manually crafted features as they develop their own features [7]. Hence, it's logical to look at raw data other than just joint angles to see if we can achieve comparable results. To this end, we shall be using the 'ext_raw_measurements.py' script to look at other data that is captured by the body suits (e.g. 'angularAcceleration', 'orientation', 'sensorMagneticField', etc.) and use these in turn to train and test on an RNN to see if we achieve comparable results to that of joint angles. If so we might be able to consider an ensemble approach to classification and regression later on with multiple models, each being trained on a different type of raw data.

- **Use the 'divided-up' source files by activity to train an RNN for a new output type**: with the 'mat_act_div.py' script having divided up the original '.mat' file into parts based on the predicted activity times as outlined in the Google annotations sheet, we can now use these in the pipeline in exactly the same way as the original data files. However, we shall now know, when dividing them up into sequences to put through the RNN, not only the patient name that the sequence corresponds to but <u>also</u> what activity it

corresponds to and its score of either 0, 1, or 2. Hence, another option to setup the RNN should be to be able to train the model to predict a class of 0, 1, or 2. The result would be a model that, given an unseen sequence of data for a certain activity, can classify it as being of an activity that deserves a 0, 1, or 2 score.

These are some of the immediate tasks to be working on, which will obviously involve expanding the current table of results that currently only includes 8 test cases. However, there are additional approaches to be explored and more in-depth tasks to be carried out later on after these are completed that include the following:

- **Continue background research alongside further experimentation**: it's necessary to continue with background research surrounding the project, primarily around studies involving adapting sequence modelling to human activity recognition. This should provide a good indication as to good hyperparameter settings for our models, possibly variations of models to consider (such as the bidirectional variant of RNNs as seen in [9]), and other considerations to maintain when dealing with human activity recognition data. Hence, a preliminary aim is to find, read, and make notes from 3 papers per week for the next month that are both highly cited (>50 citations) and relevant to this project.

- **Explore different hyperparameter setups for the RNN**: we've been keeping the hyperparameters of the model more-or-less constant throughout the experiments thus far (with changing the number of 'epochs' only as needed to achieve reasonably-good results, i.e. until the training loss more-or-less converges). However, once we decide on an approach that works (e.g. through statistical values of AD files, raw joint angles, or another measurements' raw data) and a variation of the output that we wish to focus on (e.g. prediction of overall NSAA score via regression, classification of all 17 activities' scores within a single sequence, etc.), we should then look towards optimising the network for the task at hand. This might be done via manual experimentation (not ideal as even though it's simple it's also time-consuming) or a more thorough method such as grid search or Bayesian optimization.

- **Create CRF model with selected features from the stat values of an AD file**: with the feature selection functionality of 'ft_sel_red.py' script, we can now construct a CRF model as we have a way of finding a variety of features that will account for the majority of the variance within the data without it being too high dimensional. The first step of this would be to run a feature selection method (e.g. random forest feature selection) over each of the 'all data' files in turn to find the most frequently occurring features. The top features from each file can then be combined together and from this, the most frequent of those can be found, which would thus be implied to be the most important features over all the files. We could then return to the stat values of each AD file and extract these specific features and use these to train a CRF model (by doing this process, we ensure that each file has the same features extracted before being used in the model and not just its own features that are determined to be the most useful). Though the CRF is a more computationally simplistic model when compared with an RNN, it has the potential to be used alongside one to improve results, as is the case in [3].

- **Investigate the performance of various model setups when used in natural movement data:** presently, we have only trained the models on movement data in one of two scenarios: 6-minute walking data and NSAA scenarios (i.e. where all 17 activities are carried out). To be more useful to assessors and medical professionals, its important that the model can make judgements and assessments based on real-world movement data (i.e. movement data of the subjects in natural environments). This will improve its applicability and also lend credence to the claims that it can correctly assess the subjects.

- **Look into the most applicable variation of feature selection/reduction to use in our project**: as we just use PCA as our standard feature reduction technique currently, it will be worth further consulting other research papers on what technique they used to reduce too-high-dimensioned data prior to feeding it into a neural network. This might also be done in collaboration with other members of the group, as they are investigating feature selection in the context of the suit data we are working with, so the results of their research might be impactful to the eventual permanent choice of feature selection/reduction for us.

# Predicted Timeline for Further Work

Given that there is approximately 3 months (8th June – 6th Sep; 13 weeks) left until the project is to be completed, it's worth making a rough timeline of expected progress for the foreseeable future. The idea is that this will serve as a measurement of actual progress as it's made in comparison to where we expect to be. Note that this is only an approximation, as there will be many other tasks and sub-tasks within the project that haven't been accounted for yet, along with some tasks that are known but the actual time to complete may be incredibly variable (e.g. hyperparameter tuning of the RNN may take a day or upwards of a week). Finally, we don't include under each week the requirement to research and take notes from 3 papers per week, as that will be for every week and isn't necessary to be included in the table below.

| Week | Tasks |
|---|---|
| 1 | **Implement sequence overlap for the RNN data; use other forms of raw data to train and test the RNN model; run experiments w/ both sequence overlapping and other forms of raw data** |
| 2 | **Document much of currently-done work, including commenting all scripts and adding 'README's for all scripts; use the 'divided-up' source files by activity to train an RNN for a new output type** |
| 3 | **Given left-out .mat files, implement a script to load a pre-trained model and make predictions on a new unseen data file (i.e. how the model would be used in the real-world); experiment with different architectures, for different outputs, and different data sources** |
| 4 | **Create a CRF model with selected features from the stat values of an AD file; investigate (and ideally implement) the possibility of combining CRF and RNN models to create a more robust model and carry out experiments with this newly combined model, comparing it to RNN by itself** |
| 5 | **Explore different hyperparameter setups for the RNN; from all the results of the experiments done w/ the RNN and/or CRF, determine the ideal setup; with the ideal data setup (e.g. from raw joint angles) and the ideal hyperparameters, investigate the performance of the model when used with natural movement data** |

Given this approximate timeline to complete the tasks and experiments that we envisage for the foreseeable future, this accounts for approximately 5 weeks of work. This would leave a further 8 weeks before the project ends which, if we account for approximately 3 weeks for report writing and any other tasks needed to round-out the project, gives the project a total of approximately 5 weeks worth of flexibility. This allows us to pivot to a slightly different direction at a later point if necessary, along with enabling us to expend more effort in further background research if required in order to inform certain architectural decisions later on.

# References


[1] Zhang, J., and Gong, S. Action categorization with modified hidden conditional random field. In *Pattern Recognition, VOL 43, NO. 1*, pages 197-203, 2010

[2] Lafferty, J., McCallum, A., and Pereira, F. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning 2001*, pages 282-289, 2001.

[3] Zheng, S., Jayasumana, S., and Romera-Paredes, B. Conditional Random Fields as Recurrent Neural Networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1529-1537, 2015.

[4] Nazerfard, E., Das, B., and Holder, L. B. Conditional random fields for activity recognition in smart environments. In *Proceedings of the 1st ACM International Health Informatics Symposium*, pages 282-286, 2010.

[5] Radosavljevic, V., Vucetic, S., and Obradovic, Z. Continuous Conditional Random Fields for Regression in Remote Sensing. In *ECAI*, pages 809-814, 2010.

[6] Wang, S. B., Quattoni, A., and Morency, L. P. Hidden conditional random fields for gesture recognition. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), VOL 2*, pages 1521-1527, 2006.

[7] Inoue, M., Inoue, S., and Nishida, T. Deep recurrent neural network for mobile human activity recognition with high throughput. *Artificial Life and Robotics, VOL 23, NO.* 2, pages 173-185, 2018.

[8] Hammerla, N. Y., Halloran, S., and Plötz, T. Deep, convolutional, and recurrent models for human activity recognition using wearables. *arXiv preprint arXiv:1604.08880*, 2016.

[9] Du, Y., Wang, W., and Wang, L. Hierarchical recurrent neural network for skeleton based action recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1110-1118, 2015.

# Appendix I – 'comp_stat_vals.py'

```
import sys
sys.path.append("..")
import scipy.io as sio
import numpy as np
from numpy import linalg as la
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
from tkinter import*
import os
from os.path import isfile
import argparse

sampling_rate = 60      #In Hz

#Note: CHANGE THESE to location of the 3 sub-directories' encompassing directory local to the user
source_dir = "C:\\msc_project_files\\"
sub_dirs = ["6minwalk-matfiles\\", "6MW-matFiles\\", "NSAA\\"]
output_dir = "..\\output_files\\"

#Lists of constants that dictate the allowed source file types to analyse and the dimensions of the data
file_types = ["JA", "AD", "DC"]
axis_labels = ["X", "Y", "Z"]

#Below 3 lists are labels for the 23 segments, 22 joints, and 17 sensors, as dictated by the 'MVN User Manual'
segment_labels = ["Pelvis", "L5", "L3", "T12", "T8", "Neck", "Head", "RightShoulder", "RightUpperArm",
                  "RightForeArm", "RightHand", "LeftShoulder", "LeftUpperArm", "LeftForeArm", "LeftHand",
                  "RightUpperLeg", "RightLowerLeg", "RightFoot", "RightToe", "LeftUpperLeg", "LeftLowerLeg",
                  "LeftFoot", "LeftToe"]

joint_labels = ["jL5S1", "jL4L3", "jL1T12", "jT9T8", "jT1C7", "jC1Head", "jRightT4Shoulder", "jRightShoulder",
                "jRightElbow", "jRightWrist", "jLeftT4Shoulder", "jLeftShoulder", "jLeftElbow", "jLeftWrist",
                "jRightHip", "jRightKnee", "jRightAnkle", "jRightBallFoot", "jLeftHip", "jLeftKnee",
                "jLeftAnkle", "jLeftBallFoot"]

sensor_labels = ["Pelvis", "T8", "Head", "RightShoulder", "RightUpperArm", "RightForeArm", "RightHand",
                 "LeftShoulder", "LeftUpperArm", "LeftForeArm", "LeftHand", "RightUpperLeg", "RightLowerLeg",
                 "RightFoot", "LeftUpperLeg", "LegLowerLeg", "LeftFoot"]

#Mapping used to map a given measurement name to the number of x/y/z values found in the .mat file
measure_to_len_map = {"orientation": 23, "position": 23, "velocity": 23, "acceleration": 23, "angularVelocity": 23,
                      "angularAcceleration": 23, "sensorFreeAcceleration": 17, "sensorMagneticField": 17,
                      "sensorOrientation": 22, "jointAngle": 22, "jointAngleXZY": 22}

#Mapping used to select lists of labels names to use based on the length of the numbers contained in data array
seg_join_sens_map = {len(segment_labels): segment_labels, len(joint_labels): joint_labels, len(sensor_labels):
sensor_labels}


"""
Section below covers a few general-purpose mathematical functions used for several file object types
for statistical analysis
"""
def mean_round(nums):
    """
        :param list of values of which we wish to find the mean/average:
        :return a float value rounded to 2 decimal places of the mean of 'nums':
    """
    return round(float(np.mean(nums)), 4)

def variance_round(nums):
    """
        :param list of values of which we wish to find the variance:
        :return a float value rounded to 2 decimal places of the variance of 'nums':
    """
    return round(float(np.var(nums)), 4)

def compute_diffs(nums):
    """
        :param list of values of which we wish to find the diffs between each successive value:
        :return list of diff values of len = len(nums)-1, where each value is difference between value
        in the 'nums' list and the next value in the list:
    """
    return [(nums[i+1]-nums[i]) for i in range(len(nums)-1)]

def mean_diff_round(nums):
    """
        :param list of values of which we wish to find the mean diff values:
        :return mean value of the absolute values of the diffs list (see 'compute_diffs'):
    """
    return round(float(np.mean(np.absolute(compute_diffs(nums)))), 4)

def fft_1d_round(nums):
    """
        :param list of values of which we want to find the 1-dimensional FFT for 3 values
        :return largest FFT value from list of 3 values rounded to 4 decimal place
    """
```

```
    fft_1d = np.fft.rfft(nums, n=3)
    fft_1d.sort()
    return round(np.real(np.flip(fft_1d)[0]), 4)

def covariance_round(nums):
    """
        :param list of values of 2D array of numbers to find covariance of (shape = # of dimensions of data (e.g. 3 for
        x,y,z data) x # of samples):
        :return covariance values of rounded float values of shape = # of dimensions of data x # of dimensions of data,
        for 'top' right triangle at 1D list:
    """
    return [[round(float(n), 8) for n in num] for num in np.cov(nums.tolist())]

def fft_2d_round(nums):
    """
        :param list of 2D values of which we want to find the 2-dimensional FFT for 3 values:
        :return list of 3 values returned from the 2D FFT operation, in descending order and each rounded
        to 4 decimal places
    """
    fft_2d = np.fft.fft2(nums, s=(1, 3))
    fft_2d.sort()
    fft_2d = np.flip(fft_2d)
    return [round(np.real(fft_2d[0][i]), 4) for i in range(len(fft_2d[0]))]

def mean_sum_vals(nums):
    """
        :param list of values of 2D array of numbers of which we wish to find the mean of the sums of each dimension
        :return mean of the sum of each dimension of the nums 2D array (i.e. sum along each x, y, and z of 2D feature
        values), rounded to 4 decimal places
    """
    return round(float(np.mean([np.sum(nums[i]) for i in range(len(nums))])), 4)

def mean_sum_abs_vals(nums):
    """
        :param list of values of 2D array of numbers of which we wish to find the mean of the sums of each dimension
        :return same as 'mean_sum_vals', with difference that each value in 2D input array is 'absoluted' first
    """
    return round(float(np.mean([np.sum(np.abs(nums[i])) for i in range(len(nums))])), 4)

def cov_eigenvals(nums, i):
    """

    :param list of values of 2D array of numbers of which we wish to find the eigenvals of covariance matrix, and an
    index to indicate which of the eigenvals we wish to return
    :return one of the two largest (rounded to 4 decimal places) eigenvalues of the covariance matrix of the
    2D array of numbers (i.e. the top 2 of 3 eigenvals), depending on the index 'i'
    """
    vals = la.eig(covariance_round(nums))[0].tolist()
    vals = list(vals)
    vals.sort(reverse=True)
    top_vals = vals[:2]
    return round(top_vals[i], 4)



def prop_outside_mean_zone(nums, percen=0.1):
    """
        :param array of 2D numbers of which we wish to find the proportion of the samples in the array that are outside
        'percen' boundaries around the mean zone (e.g. if mean zone is 5 for one dim, the sample would have to have
        a value not between 4.5 and 5.5 for that dimension's value to be considered 'outside'; the same just also
        be true for its other 2 dimensions)
        :return the proportion of the samples within 'nums' that are outside the mean zone (i.e. ALL 3 of its
        lie beyond their respective mean zones
    """
    #Mean values of the array of nums for each of its x, y, and z dimensions; used to calculate if a sample is
    #outside the mean zone
    x_mean, y_mean, z_mean = np.mean(nums[0]), np.mean(nums[1]), np.mean(nums[2])
    nums_outside = 0
    nums = np.swapaxes(nums, 0, 1)
    for num in nums:
        #Each sample must have all 3 of its dimensions within the 'mean zone' to have 1 added to
        if not x_mean*(1-percen) < num[0] < x_mean*(1+percen):
            if not y_mean*(1-percen) < num[1] < y_mean*(1+percen):
                if not z_mean*(1-percen) < num[2] < z_mean*(1+percen):
                    nums_outside += 1
    return round(nums_outside/len(nums), 4)



def extract_stats_features(f_index, file_part, split_file, file_name, measurement_names, is_recursive_call=False):
    """

    :param 'f_index' for a number the file_part represents of the complete file, 'file_part' being the data itself that
    we will be extracting the statistical features of, 'split_file' being the total number of parts within the complete
    file that 'file_part' comes from, 'file_name' being the name of the file that 'file_part' comes from,
    'measurement_names' being a list of measurements (e.g. angularAcceleration, position, jointAngles, etc.) we wish to
    extract the statistical features from, and 'is_recursive_call' used to handle the recursive case
    :return: dictionary of data containing the complete statistical features extracted for a single file part that
    corresponds to a single line written to an output .csv
    """
    data = {}
    # Adds a 'y' label for a given file
    data['file_type'] = "HC" if "HC" in file_name else "D"
    #For each measurement category
    for i in range(len(file_part)):
        seg_join_sens_labels = seg_join_sens_map[measure_to_len_map[measurement_names[i]]]
```

```
        num_features = 1 if is_recursive_call else measure_to_len_map[measurement_names[i]]
        #For each feature (e.g. segment, joint, or sensor)
        for j in range(num_features):
            #For each x,y,z dimension
            for k in range(len(file_part[i][j])):
                #Gives each statistical value calculated for a specific measumrent/feature/axis combination a
                #label based on these values that are shared amongst all statistical extracted values
                if is_recursive_call:
                    stat_name = "(" + measurement_names[i] + ") : (Over all features) : (" + axis_labels[k] + "-axis)"
                else:
                    stat_name = "(" + measurement_names[i] + ") : (" + seg_join_sens_labels[j] + \
                                ") : (" + axis_labels[k] + "-axis)"
                data[stat_name + " : (mean)"] = mean_round(file_part[i][j][k])
                data[stat_name + " : (variance)"] = variance_round(file_part[i][j][k])
                data[stat_name + " : (abs mean sample diff)"] = mean_diff_round(file_part[i][j][k])
                data[stat_name + " : (FFT largest val)"] = fft_1d_round(file_part[i][j][k])
            #For column labels for statistical values computed over all 3 dimensions, gives a shared label part based
            #just on the measurement name and feature being computed in question that's shared over all
            #statistical extracted values
            if is_recursive_call:
                xyz_stat_name = "(" + measurement_names[i] + ") : (Over all features) : ((x,y,z)-axis) : "
            else:
                xyz_stat_name = "(" + measurement_names[i] + ") : ( " + seg_join_sens_labels[j] + ") : ((x,y,z)-axis) : "
            data[xyz_stat_name + "(mean sum vals)"] = mean_sum_vals(file_part[i][j])
            data[xyz_stat_name + "(mean sum abs vals)"] = mean_sum_abs_vals(file_part[i][j])
            data[xyz_stat_name + "(first eigen cov)"] = cov_eigenvals(file_part[i][j], 0)
            data[xyz_stat_name + "(second eigen cov)"] = cov_eigenvals(file_part[i][j], 1)
            data[xyz_stat_name + "(x- to y-axis covariance)"] = covariance_round(file_part[i][j])[0][1]
            data[xyz_stat_name + "(x- to z-axis covariance)"] = covariance_round(file_part[i][j])[0][2]
            data[xyz_stat_name + "(y- to z-axis covariance)"] = covariance_round(file_part[i][j])[1][2]
            data[xyz_stat_name + "(FFT 1st largest val)"] = fft_2d_round(file_part[i][j])[0]
            data[xyz_stat_name + "(FFT 2nd largest val)"] = fft_2d_round(file_part[i][j])[1]
            data[xyz_stat_name + "(FFT 3rd largest val)"] = fft_2d_round(file_part[i][j])[2]
            data[xyz_stat_name + "(proportion samples outside mean zone)"] = prop_outside_mean_zone(file_part[i][j])

    #Handles recursive case of function when computing statistics over all the measurements' features
    if len(file_part[0]) == 1:
        return data

    #Concatenates samples along the 'features' axis (i.e. so data now:
    # '# measurements' x 1 x '# axes' x '(# samples x # features)' to form 'new_file_part'
    new_file_part = np.reshape(file_part, (len(file_part), 1, len(file_part[0][0]), -1))
    data_over_all = extract_stats_features(f_index=f_index, file_part=new_file_part, split_file=split_file,
                                           file_name=file_name, measurement_names=measurement_names, is_recursive_call=True)
    #Adds the statistical values that are computed in the recursive case (i.e. computed inter-features rather than
    #intra-features) to the original dictionary
    data.update(data_over_all)

    # Creates and returns a DataFrame object from a single list version of the dictionary (so it's only 1 row),
    # and creates either a .csv with the default output_name (w/ .csv name from the AD short file name)
    # or with a given name and, if the given name already exists, appends it to the end of existing file
    return pd.DataFrame([data], columns=data.keys(), index=[file_name + "_(" +
                                                            str(f_index + 1) + "/" + str(split_file) + ")"])



def check_for_abnormality(file_names, error_margin=1, abnormality_threshold=0.3):
    """
        :param 'file_names' to be the names of the files to check for abnormality, 'error_margin' to be the proportion
        of the mean of a feature to compute the upper and lower bound (e.g. error_margin=0.2 for a feature of mean
        '5' would mean a file part's value has to be within '4' and '6' for that feature to be considered a 'normal'
        value), and 'abnormality_threshold' to be the portion of features for a file part outside of the normal ranges
        for the file part to be considered 'abnormal'
        :return: no return, but prints the names of the file parts that are considered to be abnormal
    """
    for m in range(len(file_names)):
        file_df = pd.read_csv(file_names[m], index_col=0).iloc[:, 1:]
        col_means = [np.mean(file_df.iloc[:, i]) for i in range(len(file_df.iloc[0]))]
        for i in range(len(file_df)):
            features_out_of_range = 0
            for j in range(len(file_df.iloc[0])):
                feature, mean = abs(file_df.iloc[i, j]), abs(col_means[j])
                lb, ub = mean - mean*error_margin, mean + mean*error_margin
                if feature < lb or feature > ub:
                    features_out_of_range += 1
            if features_out_of_range / len(file_df.iloc[0]) > abnormality_threshold:
                print(file_df.index.values[i], "outside", error_margin, "mean error margin for >",
                      (abnormality_threshold*100), "% of its features")



class AllDataFile(object):

    def __init__(self, ad_file_name, sub_dir):
        """
            :param 'ad_file_name' being the string name of the complete name of the source file (i.e. full file
            path name), with 'short_fn' being the string name of the unique identifier of an 'All-' data matlab file
            (e.g. 'HC3'), and 'sub_dir' being the name of the sub-directory within 'output_dir' to place the
            extracted statistical values when 'write_statistical_features' is called
            :return no return, but sets up the DataFrame 'df' attribute from the given matlab file
        """
        self.ad_file_name = ad_file_name
        split_name = ad_file_name.split("\\")[-1].split("-")
        splits = [[s for s in split_name if "HC" in s], [s for s in split_name if "D" in s or "d" in s]]
        self.ad_short_file_name = splits[1][0] if not splits[0] else splits[0][0]
        self.ad_sub_dir = sub_dir
```

```
    # Loads the data from the given file name and extracts the root 'tree' from the file
    ad_data = sio.loadmat(ad_file_name)
    tree = ad_data["tree"]
    # Corresponds to location in matlabfile at: 'tree.subjects.frames.frame'
    #Try-except clause here to catch when 'D6' doesn't have a 'sensorCount' category within 'tree.subject.frames'
    print("Extracting data from AD file " + self.ad_file_name + "....")
    try:
        frame_data = tree[0][0][6][0][0][10][0][0][3][0]
    except IndexError:
        frame_data = tree[0][0][6][0][0][10][0][0][2][0]
    # Gets the types in each column of the matrix (i.e. dtypes of submatrices)
    col_names = frame_data.dtype.names
    # Extract single outer-list wrapping for vectors and double outer-list values for single values
    frame_data = [[elem[0] if len(elem[0]) != 1 else elem[0][0] for elem in row] for row in frame_data]
    df = pd.DataFrame(frame_data, columns=col_names)
    self.df = df


def display_3d_positions(self):
    """
        :param no params, but relies on the 'df' attribute that is set up at the 'AllDataFile' object's instantiation
        :return no return, but plots position values for the given file object on a 3D dynamic plot,
        with the necessary components connected, and outputs a dynamic accumulated time in seconds to the console:
    """
    #Disregard first 3 samples (usually types 'identity', 'tpose' or 'tpose-isb')
    positions = self.df.loc[:, "position"].values[3:]
    #Extracts position values for each dimension so 'xyz_pos' now has shape:
    # # of dimensions (e.g. 3 for x,y,z position values) x # of samples (~22K) x # of features (23 for segments)
    xyz_pos = [[[s for s in sample[i::3]] for sample in positions] for i in range(3)]
    #Xyz_pos_t now has dimensions:
    # # of features (23 for segments) x # of samples (~22K) x # of dimensions (e.g. 3 for x,y,z position values)
    xyz_pos_t = np.swapaxes(xyz_pos, 0, 2)
    #List of tuples that define how much segment labels are connected on the human body (e.g. segments 0 and 1 are
    #connected as are 0 and 15) so as to dynamically drawn lines between them on the 3D plot
    stick_defines = [(0, 1), (0, 15), (0, 19), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6),
                     (7, 8), (8, 9), (9, 10), (11, 12), (12, 13), (13, 14), (15, 16),
                     (16, 17), (17, 18), (19, 20), (20, 21), (21, 22)]
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    #Calculates the maximums and means alone each x, y, and z dimension to use to set the 3D boundaries
    maxs = [max([max(xyz_pos[i][j]) for j in range(len(xyz_pos[i]))]) for i in range(len(xyz_pos))]
    mins = [min([min(xyz_pos[i][j]) for j in range(len(xyz_pos[i]))]) for i in range(len(xyz_pos))]
    ax.set_xlim(mins[0], maxs[0])
    ax.set_ylim(mins[1], maxs[1])
    ax.set_zlim(mins[2], maxs[2])

    xy_ratio = (maxs[1]-mins[1])/(maxs[0]-mins[0])
    xz_ratio = (maxs[2]-mins[2])/(maxs[0]-mins[0])
    ax.get_proj = lambda: np.dot(Axes3D.get_proj(ax), np.diag([1, xy_ratio, xz_ratio, 0.25]))

    colors = plt.cm.jet(np.linspace(0, 1, len(xyz_pos[0][0])))
    lines = sum([ax.plot([], [], [], '-', c=c) for c in colors], [])
    pts = sum([ax.plot([], [], [], 'o', c=c) for c in colors], [])

    stick_lines = [ax.plot([], [], [], 'k-')[0] for _ in stick_defines]

    #Defines the 'init' function object to setup the points and lines defining a person in 3d space
    def init():
        for line, pt in zip(lines, pts):
            line.set_data([], [])
            line.set_3d_properties([])
            pt.set_data([], [])
            pt.set_3d_properties([])
        return lines + pts + stick_lines

    #Function object that updates the plot based on the next sample of 3D coordinates for each feature (i.e.
    #each 'point' on the walking figure in the plot)
    def animate(i):
        if i >= len(xyz_pos[0]):
            exit()
        if (i*5)%sampling_rate == 0:
            print("Plotting time: " + str(int((i*5)/sampling_rate)) + "s")
        i = (5 * i) % xyz_pos_t.shape[1]
        for line, pt, xi in zip(lines, pts, xyz_pos_t):
            x, y, z = xi[:i].T
            pt.set_data(x[-1:], y[-1:])
            pt.set_3d_properties(z[-1:])
        for stick_line, (sp, ep) in zip(stick_lines, stick_defines):
            stick_line._verts3d = xyz_pos_t[[sp, ep], i, :].T.tolist()
        fig.canvas.draw()
        if (i+5) >= int(len(xyz_pos[0])):
            plt.close()
            print("Animation ended...")
        return lines + pts + stick_lines

    #Plot the figure in 3D space, with an update interval (defined in miliseconds) to be in real time
    anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len(xyz_pos[0]),
                                   interval=1000 / sampling_rate, blit=False, repeat=False)
    plt.show()


def file_info(self):
    """
        :param string name of the unique identifier of an 'All-' data matlab file (e.g. 'HC3'):
        :return no return, but sets up the DataFrame 'df' attribute from the given matlab file: name
    """
```

```
        print("\nAd " + self.ad_file_name + " keys:", ", ".join([("'" + key + "'") for key in self.df]), "\n")
        for k in list(self.df.keys())[:3]:
            print(k, ":", (self.df[k]))


    def write_statistical_features(self, measurements_to_extract=None, output_name=None, split_file=1, split_size=None):
        """
            :param 'measurements_to_extract' to be an optional list of measurement names from the AD file that we wish to
            apply statistical analysis (otherwise, defaults to 'measurement_names'), along with 'output_name' that
            defaults to the short name of the AD file, which writes to an individual file for the object (i.e.
            a single line .csv for the object); specify shared 'output_name' to instead append to an existing .csv;
            'split_file' and 'split_size' optionally given as 2 different ways to break up a file into parts
            :return no return, but writes to an output .csv file the statistical values of the certain
            measurements to an output .csv file
        """
        #Sets the default measurements to extract from the AD file object if none are specified as args
        measurement_names = measurements_to_extract if measurements_to_extract else [
            "position", "sensorFreeAcceleration", "sensorMagneticField", "velocity", "angularVelocity",
            "acceleration", "angularAcceleration"]

        # Extracts values of the various measurements in different layout so 'f_d_s_data' now has shape:
        # (# measurements (e.g. 3 for position, accel, and megnet field) x # of features (e.g. 23 for segments)
        # x # of dimensions (e.g. 3 for x,y,z position values) x # of samples (~22K))
        extract_data = self.df.loc[3:, measurement_names].values
        f_d_s_data = np.zeros((len(measurement_names),
                               max(len(segment_labels), len(joint_labels), len(sensor_labels)), 3,
                               len(self.df.loc[:])-3))
        for i in range(len(measurement_names)):
            for j in range(int(len(self.df.loc[3:, measurement_names[i]].values[0])/3)):
                for k in range(len(axis_labels)):
                    for m in range(len(self.df.loc[3:])):
                        f_d_s_data[i, j, k, m] = extract_data[m, i][(j*3)+k]

        #If 'split_size' is given as command line argument and 'split_file' isn't, set 'split_s' to the given number
        #multiplied by sampling rate (i.e. if given num is 5 and 'sampling_rate' is 60, 'split_s' is 300, i.e. the
        #number of rows in each 'split_file') and 'split_file' is set to the number of these sizes that fit in the
        #original file (i.e. how many parts we can get out of a file given a 'split_s')
        if split_file == 1 and split_size:
            split_s = int(sampling_rate * split_size)
            split_file = int(len(f_d_s_data[0, 0, 0])/split_s)
        else:
            split_s = int(len(f_d_s_data[0, 0, 0])/split_file)

        written_names = []
        for f in range(split_file):
            fds = f_d_s_data[:, :, :, (split_s*f):(split_s*(f+1))]
            # Creates a dictionary of extracted statistical values for a given file part
            df = extract_stats_features(f_index=f, file_part=fds, split_file=split_file,
                                        file_name=self.ad_short_file_name, measurement_names=measurement_names)

            #Creates the relevant sub-directories within 'output_files' to store the created .csv
            ad_output_dir = output_dir + self.ad_sub_dir + "\\"
            if not os.path.exists(ad_output_dir):
                os.mkdir(ad_output_dir)
            ad_output_dir += "AD\\"
            if not os.path.exists(ad_output_dir):
                os.mkdir(ad_output_dir)

            #Sets output name of file (and related print statement) to whether or not the user is storing it in
            #an 'all' .csv or a .csv determined by the name of the writing file
            if not output_name:
                output_complete_name = ad_output_dir + "AD_" + self.ad_short_file_name + "_stats_features.csv"
                print("Writing AD", self.ad_file_name, "(", (f+1), "/", split_file, ") statistial info to",
                      output_complete_name)
            else:
                output_complete_name = ad_output_dir + "AD_" + output_name + "_stats_features.csv"
                print("Writing AD", self.ad_file_name, "(", (f + 1), "/", split_file, ") statistial info to",
                      output_complete_name)
            #Writes just the data to file if the file already exists, or both the data and headers if the file
            #doesn't exist yet
            if isfile(output_complete_name):
                with open(output_complete_name, 'a', newline='') as file:
                    df.to_csv(file, header=False)
            else:
                with open(output_complete_name, 'w', newline='') as file:
                    df.to_csv(file, header=True)
                written_names.append(output_complete_name)
        #Returns the complete names of the file(s) that have been written or appended to a .csv
        return written_names




class DataCubeFile(object):

    def __init__(self, sub_dir, dis_data_cube=False):
        """
            :param 'dis_data_cube' is specified to True if wish to display basic Data Cube info to the screen
            :return no return, but reads in the datacube object from .mat file, the datacube table from the .csv,
            extracts the names of the joint angle files in the datacube file, and instantiates JointAngleFile objects
            for each file contained in the datacube .mat file

            IMPORTANT NOTE: as there's no conceivable way currently to read a matlab table into Python (unlike structs),
            it's required that the matlab table is exported to a .csv before creating DataCubeFileObject. Hence, with
            the data_cube .mat file open in matlab, run writetable(excel_table, "data_cube_table.csv") in matlab for the
            following to work
```

```
        """

        self.dc_sub_dir = sub_dir

        try:
            self.dc_table = pd.read_csv(source_dir + "data_cube_table.csv")
            self.dc_short_file_names = self.dc_table.values[:, 1]
            self.dc_file_names = self.dc_table.values[:, 2]
        except FileNotFoundError:
            print("Couldn't find the 'data_cube_table.csv file. Make sure to run the "
                  "'writetable(excel_table, \"data_cube_table.csv\") in matlab with data_cube.mat file open")
            sys.exit()
        self.dc = sio.loadmat(source_dir + "data_cube_6mw", matlab_compatible=True)["data_cube"][0][0][2][0]

        self.ja_objs = []
        for i in range(len(self.dc_short_file_names)):
            print("Extracting data from data cube file (" + str(i+1) + "/" + str(len(self.dc_short_file_names)) +
                  "): " + self.dc_short_file_names[i] + "...")
            self.ja_objs.append(JointAngleFile(ja_file_name=self.dc_file_names[i], sub_dir=self.dc_sub_dir,
                                               dc_object_index=i, dc_short_file_name=self.dc_short_file_names[i]))

        if dis_data_cube:
            self.display_info()



    def display_info(self):
        """
            :param no params
            :return no return, but displays some basic info about the Data Cube attribute
        """
        print("\nData cube keys:", ", ".join([("'" + key + "'") for key in self.dc]), "\n")
        print("'__header__': ", self.dc["__header__"])
        print("'__version__': ", self.dc["__version__"])
        print("'__globals__': ", self.dc["__globals__"])
        print(self.dc["data_cube"])


    def write_statistical_features(self, output_name="all", split_file=1, split_size=None):
        """
        :param 'output_name', which defaults to 'All', which is the short name of the output .csv stats file that the
        objects will share...specify a different name if desired; 'split_file' and 'split_size' optionally given as 2
        different ways to break up a file into parts
        :return no return, but creates a single .csv output file that contains the statistical analysis of each joint
        angle file contained within the data cube via calls to 'JointAngleFile.write_statistical_features()' method
        for each of the joint angle objects extracted at initialization
        """
        written_names = []
        for i in range(len(self.ja_objs)):
            written_names += self.ja_objs[i].write_statistical_features(output_name=output_name, split_file=split_file,
                                                                        split_size=split_size)
        return written_names


    def write_direct_csv(self, output_name="all"):
        """
        :param no params
        :return for each JointAngleFile object within the datacube, call their 'write_direct_csv' method with their
        'output_name' set to their respective short name
        """
        written_names = []
        for i in range(len(self.ja_objs)):
            written_names += self.ja_objs[i].write_direct_csv()
        return written_names


class JointAngleFile(object):

    def __init__(self, ja_file_name, sub_dir, dc_object_index=-1, dc_short_file_name=None):
        """
            :param 'ja_file_name' being the string name of the complete name of the source file (i.e. full file
            path name), with 'short_fn' being the string name of the unique identifier of an 'joint angle' data
            matlab file (e.g. 'D2'), and 'sub_dir' being the name of the sub-directory within 'output_dir' to place
            the extracted statistical values when 'write_statistical_features' is called, and 'dc_object' which is
            set to non-zero if object is created as part of a DataCubeFile:
            :return no return, but sets up the DataFrame 'df' attribute from the given matlab file
        """
        self.ja_file_name = ja_file_name
        split_name = ja_file_name.split("\\")[-1].split("-")
        splits = [[s for s in split_name if "HC" in s], [s.upper() for s in split_name if "D" in s.upper()]]
        if not dc_short_file_name:
            self.ja_short_file_name = splits[1][0][splits[1][0].index("D"):] \
                if not splits[0] else splits[0][0][splits[0][0].index("HC"):]
        else:
            self.ja_short_file_name = dc_short_file_name
        self.ja_sub_dir = sub_dir
        self.is_dc_object = True if dc_object_index != -1 else False
        #Loads the JA data from the datacube file instead of the normal JA files if passed from DataCubeFile class
        if dc_object_index != -1:
            self.ja_data = sio.loadmat(source_dir + "data_cube_6mw",
                                       matlab_compatible=True)["data_cube"][0][0][2][0][dc_object_index]
        else:
            #'Try-except' clause included to handle some slight naming inconsistencies with the JA filename syntax
            try:
                self.ja_data = sio.loadmat(ja_file_name)['jointangle']
            except FileNotFoundError:
                self.ja_data = sio.loadmat(ja_file_name + "-TruncLen5Min20Sec")['jointangle']
```

```
    print("Extracting data from JA file " + self.ja_file_name + "....")
    #x/y/z_angles arrays have shape (# of samples in JA data file x # of features (i.e. # of features, 22))
    self.x_angles = np.asarray([[s for s in sample[0::3]] for sample in self.ja_data])
    self.y_angles = np.asarray([[s for s in sample[1::3]] for sample in self.ja_data])
    self.z_angles = np.asarray([[s for s in sample[2::3]] for sample in self.ja_data])
    #xyz_angles array has shape: '# of samples in JA data file' x '# of features (i.e. # of features, 22)'
    # x '# of dimensions of data (i.e. 3 for x,y,z)'
    self.xyz_angles = np.asarray([[[x, y, z] for x, y, z in
                                   zip([s for s in sample[0::3]], [s for s in sample[1::3]],
                                       [s for s in sample[2::3]])] for sample in self.ja_data])


def display_3d_angles(self):
    """
        :param no params
        :return no return, but displays the angles of each feature in 3D as it changes over
        time (i.e. with sample num)
    """
    ja_3d_data = self.xyz_angles
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    plt.ion()
    for i, sample in enumerate(ja_3d_data):
        try:
            ax.scatter(sample[:, 0], sample[:, 1], sample[:, 2])
            plt.pause((1/sampling_rate))
            ax.clear()
        except TclError:
            print("Window closed on frame:", (i+1))
            sys.exit()
    plt.close()


def display_diffs_plot(self):
    """
        :param no params
        :return no return, but displays '# of features (e.g. 22 features)' x '# of dimensions (e.g. 3 for x,y,z)'
        subplots for plot of diffs for each feature's dimension over time (e.g. 2,3 graph shows the diffs of
        successive joint angles values for the the 2nd feature's z-axis)
    """
    plt.rcParams.update({'font.size': 5})
    #Creates diff lists for all features and dimensions of the joint angle file object, with shape:
    #(# of dimensions (e.g. 3 for x,y,z) x # of features (e.g. 22 features) x # of samples - 1)
    x_y_z_diffs = np.asarray([[compute_diffs(self.xyz_angles[:, i, j]) for i in range(len(self.xyz_angles[0]))]
                              for j in range(len(self.xyz_angles[0][0]))])

    i_dim = len(x_y_z_diffs)
    j_dim = len(x_y_z_diffs[0])
    #Makes sure all subplots share the same axes for easier comparison
    fig, axes = plt.subplots(j_dim, i_dim, sharex='all', sharey='all')
    for i in range(i_dim):  # For each x/y/z dimension
        for j in range(j_dim):  # For each of the joint angles
            #Uses 'time_increments' for x_axis points, with 1 increment (in 's') corresponding to one diff sample
            #for each subplot
            time_increments = np.arange(0, (len(x_y_z_diffs[i][j]) / sampling_rate), (1 / sampling_rate))
            axes[j, i].plot(time_increments, x_y_z_diffs[i][j])
            axes[j, i].set_ylabel(joint_labels[j] + ": " + axis_labels[i], rotation=0, size=6, labelpad=25)

    plt.subplots_adjust(hspace=0.1, top=0.95, bottom=0.03, right=0.99, left=0.07)
    plt.gcf().set_size_inches(20, 20)
    fig.suptitle("Plot of rates of change of joint angles for: \"" + self.ja_file_name +
                 "\" (Row = joint angle, Column = dimension (x, y, or z) of angle): "
                 "X-axis = time(s), Y-axis = angle change per sample", size=13)
    plt.show()


def write_statistical_features(self, output_name=None, split_file=1, split_size=None):
    """
        :param 'output_name' defaults to the short name of the joint angle file, which writes to an individual
        file for the object (i.e. a single line .csv for the object); specify shared 'output_name' to
        instead append to an existing .csv; 'split_file' and 'split_size' optionally given as 2 different ways
        to break up a file into parts
        :return: no return, but writes to an output .csv file the statistical values of the joint angle
        measurement to an output .csv file
    """
    x_angles, y_angles, z_angles = self.x_angles, self.y_angles, self.z_angles
    #Angles has shape: (# of dimensions of features (3 for x,y, and z) x # features (22 here) x # samples (~22K),
    #hence angles[i][j] selects list of sample values for dimension 'i' for feature 'j'
    angles = np.swapaxes(np.asarray([x_angles.T, y_angles.T, z_angles.T]), 0, 1)

    #If 'split_size' is given as command line argument and 'split_file' isn't, set 'split_s' to the given number
    #multiplied by sampling rate (i.e. if given num is 5 and 'sampling_rate' is 60, 'split_s' is 300, i.e. the
    #number of rows in each 'split_file') and 'split_file' is set to the number of these sizes that fit in the
    #original file (i.e. how many parts we can get out of a file given a 'split_s')
    if split_file == 1 and split_size:
        split_s = int(sampling_rate * split_size)
        split_file = int(len(angles[0, 0])/split_s)
    else:
        split_s = int(len(angles[0, 0])/split_file)

    written_names = []
    for f in range(split_file):
        ang = angles[:, :, (split_s * f):(split_s * (f + 1))]
        #Creates a dictionary of extracted statistical values for a given file part
        df = extract_stats_features(f_index=f, split_file=split_file, file_name=self.ja_short_file_name,
                                    measurement_names=["jointAngle"], file_part=[ang])
```

```
            file_prefix = "DC" if self.is_dc_object else "JA"

            #Creates the relevant sub-directories within 'output_files' to store the created .csv
            dc_ja_output_dir = output_dir + self.ja_sub_dir + "\\"
            if not os.path.exists(dc_ja_output_dir):
                os.mkdir(dc_ja_output_dir)
            dc_ja_output_dir += file_prefix + "\\"
            if not os.path.exists(dc_ja_output_dir):
                os.mkdir(dc_ja_output_dir)

            #Sets output name of file (and related print statement) to whether or not the user is storing it in
            #an 'all' .csv or a .csv determined by the name of the writing file
            if not output name:
                output_complete_name = dc_ja_output_dir + file_prefix + "_" + self.ja_short_file_name +
"_stats_features.csv"
                print("Writing", file_prefix, self.ja_file_name, "(", (f+1) , "/", split_file,
                      ") statistial info to", output_complete_name)
            else:
                output_complete_name = dc_ja_output_dir + file_prefix + "_" + output_name + "_stats_features.csv"
                print("Writing", file_prefix, self.ja_file_name, "(", (f+1), "/",
                      split_file, " ) statistial info to", output_complete_name)

            #Writes just the data to file if the file already exists, or both the data and headers if the file
            #doesn't exist yet
            if isfile(output_complete_name):
                with open(output_complete_name, 'a', newline='') as file:
                    df.to_csv(file, header=False, line_terminator="")
            else:
                with open(output_complete_name, 'w', newline='') as file:
                    df.to_csv(file, header=True, line_terminator="")
            written_names.append(output_complete_name)
        #Returns the complete names of the file(s) that have been written or appended to a .csv
        return written_names


    def write_direct_csv(self, output_name=None):
        """
            :param 'output_name', which, if specified, ensures the file is written to a specified file name rather
            than a name dependent on the source file name (e.g. 'D2')
            :return: no return, but instead directly writes a JointAngleFile object from a .mat file to a .csv
            without extracting any of the statistical features
        """
        df = pd.DataFrame(self.ja_data, index=[self.ja_short_file_name for i in range(len(self.ja_data))])
        headers = [("(" + joint_labels[i] + ") : (" + axis_labels[j] + "-axis)")
                   for i in range(len(joint_labels)) for j in range(len(axis_labels))]
        file_prefix = "DC" if self.is_dc_object else "JA"

        # Creates the relevant sub-directories within 'output_files' to store the created .csv
        dc_ja_output_dir = output_dir + "direct_csv\\"
        if not os.path.exists(dc_ja_output_dir):
            os.mkdir(dc_ja_output_dir)
        dc_ja_output_dir += file_prefix + "\\"
        if not os.path.exists(dc_ja_output_dir):
            os.mkdir(dc_ja_output_dir)

        if not output_name:
            output_complete_name = dc_ja_output_dir + file_prefix + "_" + self.ja_short_file_name + ".csv"
            print("Writing", file_prefix, self.ja_file_name, "to", output_complete_name)
        else:
            output_complete_name = dc_ja_output_dir + file_prefix + "_" + output_name + ".csv"
            print("Writing", file_prefix, self.ja_file_name, "to", output_complete_name)
        if isfile(output_complete_name):
            with open(output_complete_name, 'a', newline='') as file:
                df.to_csv(file, header=False)
        else:
            with open(output_complete_name, 'w', newline='') as file:
                df.to_csv(file, header=headers)
        return output_complete_name



def class_selector(ft, fn, fns, sub_dir, is_all, split_files, is_extract_csv=False, split_size=None):
    """
        :param 'ft' is the type of data file (i.e. one of 'AD', 'JA', or 'DC'), 'fn' is the full name of the data file
        (i.e. the full-directory path of the file) if 'class_selector' is dealing with a single file, 'fns' is the
        full file names if 'class_selector' is dealing with multiple files(i.e. the 'all'
        command line argument is given for 'fn'), 'sub_dir' is the sub-directory to write the file(s)
        in question in their extracted stats .csv format, 'is_all' is set to true if 'all is given as 'fn' command line
        argument (else false), 'split_files' is set at the value given by '--split_files' command line argument (else 1),
        'is_extract_csv' is true if calling 'write_direct_csv' on JointAngleFile object (else false), and 'split_size'
        is the value set by the '--split_size' command line argument (else None)
        :return list of names that have already been written to output .csv; however, the more important operation is
        the creation of various file objects and the calling on their respective 'write_statistical_features' methods
        with arguments governed by the passed in arguments to 'class_selector'
    """
    names = []
    if ft == "AD":
        if is_all:
            for f in fns:
                names += AllDataFile(f, sub_dir).write_statistical_features(split_file=split_files, split_size=split_size)
        else:
            names.append(AllDataFile(fn, sub_dir).write_statistical_features(
                split_file=split_files, split_size=split_size))
    elif ft == "JA":
        if not is_extract_csv:
            if is_all:
```

```
                fns = [fn for fn in fns if "jointangle" in fn]
                for f in fns:
                    names += JointAngleFile(f, sub_dir).write_statistical_features(
                        output_name="all", split_file=split_files, split_size=split_size)
            else:
                names += JointAngleFile(fn, sub_dir).write_statistical_features(
                    split_file=split_files, split_size=split_size)
        else:
            if is_all:
                fns = [fn for fn in fns if "jointangle" in fn]
                for f in fns:
                    names += JointAngleFile(f, sub_dir).write_direct_csv()
            else:
                names += JointAngleFile(fn, sub_dir).write_direct_csv()
    else:
        if not is_extract_csv:
            names += DataCubeFile(sub_dir).write_statistical_features(split_file=split_files, split_size=split_size)
        else:
            names += DataCubeFile(sub_dir).write_direct_csv()
    return list(dict.fromkeys(np.ravel(names)))



def del_files(names):
    for name in names:
        try:
            os.remove(name)
            print("'" + str(name) + "' successfully deleted")
        except FileNotFoundError:
            print("'" + str(name) + "' doesn't exist, unable to delete...")



"""Section below encompases all the command line arguments that can be supplied to the program, with those beginning
with '--' being optional arguments and the others being required arguments (with the exception of 'fn' argument not
being necessary if 'ft' is 'DC')"""
parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained within "
                                "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
parser.add_argument("ft", help="Specify type of file we wish to read from, being one of 'JA' (joint angle), "
                               "'AD' (all data), or 'DC' (data cube).")
parser.add_argument("fn", nargs="?", default="DC",
                    help="Specify the short file name to load; e.g. for file 'All-HC2-6MinWalk.mat' or "
                         "jointangleHC2-6MinWalk.mat, enter 'HC2'. Specify 'all' for all the files available "
                         "in the default directory of the specified file type. Optional for 'DC' file type.")
parser.add_argument("--dis_3d_pos", type=bool, nargs="?", const=True,
                    help="Plots the dynamic positions of an AllDataFile object over time. "
                         "Only works with 'ft' set as an 'AD' file name.")
parser.add_argument("--dis_diff_plot", type=bool, nargs="?", const=True,
                    help="Plots the diff plots of all features and axes of a JointAngleFile object over time."
                         "Only works with 'ft' set as a 'JA' file name.")
parser.add_argument("--dis_3d_angs", type=bool, nargs="?", const=True,
                    help="Plots the 3D positions of all features' joint angles over time. "
                         "Only works with 'ft' set as a 'JA' file name.")
parser.add_argument("--split_files", type=int,
                    help="Splits each of the files into '--split_files' number of parts to do statistical analysis on, "
                         "with each retaining the original file's file label ('HC' or 'D').")
parser.add_argument("--split_size", type=float, nargs="?", const=1,
                    help="Splits each of the files into multiple parts, with each part being of size in rows "
                         "'--split_size' x 'sampling_rate', so '--split_size' is the desired time frame in seconds for "
                         "the length of the split files (hence there are 'len of file in seconds' / '--split_size' "
                         "number of splits (i.e. '--split_size'=1 sets 60 frames (due to sampling rate) to each label)."
                         "Has no effect if '--split_files' called in same command.")
parser.add_argument("--check_for_abnormalities", type=float, nargs="?", const=True,
                    help="Checks for abnormalities within the currently-working-on file within it's parts. In other words,"
                         "if '--split_files=5', checks whether any of the 5 file parts is significantly different from "
                         "any of the other parts.")
parser.add_argument("--extract_csv", type=bool, nargs="?", const=True,
                    help="Directly writes a JA file from .mat to .csv format (i.e. without stat extraction).")
parser.add_argument("--del_files", type=bool, nargs="?", const=True,
                    help="Deletes the created file(s) as soon as they're created.")
parser.add_argument("--combine_all", type=bool, nargs="?", const=True,
                    help="Combines all the written files into a single file with sub-title containing '_ALL'")
args = parser.parse_args()


#Gets default values for 'split_files' and 'split_size' if optional arguments are not provided
split_files = args.split_files if args.split_files else 1
split_size = args.split_size if split_files == 1 else None

#Section below sets 'source_dir' to the subdirectory that we shall be pulling .mat files from, along with setting
#up 'file_names' to be either the names of the files in subdirectory or a string if we are working with the data cube.
if args.dir + "\\" in sub_dirs:
    source_dir += args.dir + "\\"
else:
    print("First arg ('dir') must be a name of a subdirectory within the source dir and must be one of "
          "'6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
    sys.exit()
file_names = []
if args.dir == "6minwalk-matfiles":
    if args.ft.upper() == "AD":
        source_dir += "all_data_mat_files\\"
        file_names = os.listdir(source_dir)
    elif args.ft.upper() == "JA":
        source_dir += "joint_angles_only_matfiles\\"
        file_names = os.listdir(source_dir)
    elif args.ft.upper() == "DC":
```

```
        source_dir += "joint_angles_only_matfiles\\"
        file_names = "DC"
elif args.dir == "6MW-matFiles":
    if args.ft.upper() == "AD":
        file_names = [f for f in os.listdir(source_dir) if f.endswith(".mat")]
    else:
        print("Second arg must be 'AD', as '6MW-matFiles' doesn't have joint angle or data cube files in them.")
        sys.exit()
else:
    if args.ft.upper() == "AD":
        # Only 'matfiles' subdirectory of 'NSAA' applicable for analysis with this script
        source_dir += "matfiles\\"
        file names = [f for f in os.listdir(source dir) if f.endswith(".mat")]
    else:
        print("Second arg must be 'AD', as 'NSAA\matfiles' doesn't have joint angle or data cube files in them.")
        sys.exit()


#Only do the statistical analysis on files if none of the optional 'display' arguments have been given, as these do not
#require any statistical value extraction to display their results
if not args.dis_3d_pos and not args.dis_diff_plot and not args.dis_3d_angs:
    names = []
    if args.ft in file_types:
        if file_names != "DC":
            #Handles the 'AD'/'JA' case when file name is NOT 'all' (i.e. just a single file)
            if any(args.fn in fn for fn in file_names):
                file_name = source_dir + [fn for fn in file_names if args.fn in fn][0]
                names = class_selector(args.ft, file_name, fns=None, sub_dir=args.dir, is_all=False,
                                       split_files=split_files, is_extract_csv=args.extract_csv, split_size=split_size)
            elif args.fn == "all":
                file_names = [source_dir + fn for fn in file_names]
                names = class_selector(args.ft, None, fns=file_names, sub_dir = args.dir, is_all=True,
                                       split_files=split_files, is_extract_csv=args.extract_csv, split_size=split_size)
            else:
                print("Third arg ('fn') must be one of the file names for the '" + args.ft + "' file type, or 'all'")
                sys.exit()
        #Handles the 'data cube' case
        else:
            if not args.extract_csv:
                names = class_selector(args.ft, None, fns=None, sub_dir = args.dir, is_all=True,
                                       split_files=split_files, is_extract_csv=False, split_size=split_size)
            else:
                names = class_selector(args.ft, None, fns=None, sub_dir = args.dir, is_all=True,
                                       split_files=split_files, is_extract_csv=True, split_size=split_size)
    else:
        print("Second arg ('ft') must be one of the accepted file types ('JA', 'AD', or 'DC').")
        sys.exit()
    #Calls the 'check_for_abnormalities' function on the created .csv's if argument is specified
    if args.check_for_abnormalities:
        check_for_abnormality(names, error_margin=args.check_for_abnormalities)
    #Combines all the written output files into one file and deletes the individual files the 'ALL' was sourced from
    if args.combine_all:
        all_name = output_dir + args.dir + "\\" + args.ft + "\\" + args.ft + "_ALL_stats_features.csv"
        print("Combining all files into one and writing to " + all_name + "...")
        for i, name in enumerate(names):
            print("Adding", name, "to 'ALL' .csv...")
            df = pd.read_csv(name, index_col=0)
            if i == 0:
                with open(all_name, 'w', newline='') as file:
                    df.to_csv(file, header=True)
            else:
                with open(all_name, 'a', newline='') as file:
                    df.to_csv(file, header=False)
        #Get rid of the non-'ALL' files
        del_files(names)
    #If '--del_files' optional argument given, delete the files that have just been created
    if args.del_files:
        del_files(names)



#Final 3 optional arguments are called only if specified and, in the process, the script does not perform any
#of the above statistical analysis on the files given in argument; once the file object is created, its required
#display method is called
elif args.dis_3d_pos:
    if args.ft != "AD":
        print("Second arg ('ft') must be 'AD' for calling 'display_3d_positions' method.")
        sys.exit()
    else:
        if any(args.fn in fn for fn in file_names):
            file_name = source_dir + [fn for fn in file_names if args.fn in fn][0]
            AllDataFile(file_name, args.dir).display_3d_positions()
        else:
            print("Third arg ('fn') must be the short name of an all data file (e.g. 'D2', 'HC5').")
            sys.exit()
elif args.dis_diff_plot:
    if args.ft != "JA":
        print("Second arg ('ft') must be 'JA' for calling 'display_diffs_plot' method.")
        sys.exit()
    else:
        if any(args.fn in fn for fn in file_names):
            file name = source dir + [fn for fn in file names if args.fn in fn][0]
            JointAngleFile(file_name, args.fn, args.dir).display_diffs_plot()
        else:
            print("Third arg ('fn') must be the short name of a joint angle file (e.g. 'D2').")
            sys.exit()
elif args.dis_3d_angs:
```

```
if args.ft != "JA":
    print("Second arg ('ft') must be 'JA' for calling 'display_3d_angles' method.")
    sys.exit()
else:
    if any(args.fn in fn for fn in file_names):
        file_name = source_dir + [fn for fn in file_names if args.fn in fn][0]
        JointAngleFile(file_name, args.fn, args.dir).display_3d_angles()
    else:
        print("Third arg ('fn') must be the short name of a joint angle file (e.g. 'D2').")
        sys.exit()
```

# Appendix II – 'ft_sel_red.py'

```
import pandas as pd
import numpy as np
import os
import sys
import argparse
from sklearn.preprocessing import normalize
from sklearn.decomposition import PCA
from sklearn.random_projection import GaussianRandomProjection
from sklearn.cluster import FeatureAgglomeration
from sklearn.feature_selection import VarianceThreshold
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel

#Note: CHANGE THESE to location of the 3 sub-directories' encompassing directory local to the user that's needed to
#map to the .csvs containing the NSAA information
nsaa_table_path = "C:\\msc_project_files\\"

#Note: CHANGE THIS to location of the source data for the feature selection/reduction to use
source_dir = "..\\output_files\\"

sub_dirs = ["6minwalk-matfiles\\", "6MW-matFiles\\", "NSAA\\", "direct_csv\\"]
sub_sub_dirs = ["AD\\", "JA\\", "DC\\"]
choices = ["pca", "grp", "agglom", "thresh", "rf"]

#Default number of components to preserve with the feature selection/reduction options; overridden if
#user supplies value for '--num_features'
num_components = 30

parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained within "
                                "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
parser.add_argument("ft", help="Specify type of .mat file that the .csv is to come from, being one of 'JA' (joint "
                               "angle), 'AD' (all data), or 'DC' (data cube).")
parser.add_argument("fn", help="Specify the short file name of a .csv to load from 'source_dir'; e.g. for file "
                               "'All_D2_stats_features.csv', enter 'D2'. Specify 'all' for all the files available "
                               "in the 'source_dir'.")
parser.add_argument("choice", help="Specifies the choice of feature reduction or feature selection to carry out on "
                                   "the input .csv data")
parser.add_argument("--no_normalize", type=bool, nargs="?", const=True,
                    help="Include this argument if user specifically doesn't want to normalize the 'x' data.")
parser.add_argument("--num_features", type=int,
                    help="Specify an 'int' here to define the features to reduce the data to.")
parser.add_argument("--dis_kept_features", type=bool, nargs="?", const=True, default=False,
                    help="Specify this if user wishes to print to console the kept features after a feature selection "
                         "choice is made. Has no impact if an unsupervised feature reduction choice is made (e.g. "
                         "'pca', 'grp', or 'agglom').")

#TODO: add optional args to choose # dimensions to keep
#TODO: add option to print out the kept dimensions of feature selection techniques
args = parser.parse_args()

if args.dir + "\\" in sub_dirs:
    source_dir += args.dir + "\\"
else:
    print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
          "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', or 'direct_csv'.")
    sys.exit()

if args.ft.upper() + "\\" in sub_sub_dirs:
    source_dir += args.ft + "\\"
else:
    print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
          "\'JA', or \'DC\'.")
    sys.exit()

match_fns = [s for s in os.listdir(source_dir) if s.split(".")[0].split("_")[1].upper() == args.fn.upper()]
if match_fns:
    full_file_names = [source_dir + match_fns[0]]
else:
    if args.fn == "all":
        match_fns = [s for s in os.listdir(source_dir)]
        full_file_names = [source_dir + match_fns[i] for i in range(len(match_fns))]
    else:
        print("Third arg ('fn') must be the short name of a file (e.g. 'D2' or 'all') within", source_dir)
        sys.exit()

if args.choice in choices:
    choice = args.choice
else:
    print("Fourth arg ('choice') must be the name a feature selection/reduction and must be one of 'pca' (to carry "
          "out PCA on the file), 'grp' (to reduce dimensionality through a Gaussian random projection), 'agglom' "
          "(to carry out feature agglomeration), 'thresh' (to do features selection based on a minimal variance "
          "threshold for each features), and 'rf' (to fit the data to a random forest and use this to select "
          "the most useful features).")
    sys.exit()


def ft_red_select(full_file_name):
```

```
    print("Reducing dims of " + full_file_name + "...")
    df = pd.read_csv(full_file_name)
    col_names = df.columns.values[2:]
    #Splits the loaded file into the 'y' parts (the original .mat source file column and file label) and 'x' parts (all
    #the statistical values extracted via 'comp_stat_vals.py')
    x = df.iloc[:, 2:].values
    y = df.iloc[:, :2].values

    #Normalize the data
    if not args.no_normalize:
        x = normalize(x)

    #Overrides the number of features to keep in feature selection/reduction if optional argument is chosen
    global num_components
    if args.num_features:
        num_components = args.num_features

    #Given the argument choice of feature selection/reduction, creates the relevant object, fits the 'x' data to it,
    #and reduces/transforms it to a lower dimensionality
    new_x = []
    print("Original 'x' shape:", np.shape(x))
    if choice == "pca":
        pca = PCA(n_components=num_components)
        new_x = pca.fit_transform(x)
    elif choice == "grp":
        grp = GaussianRandomProjection(n_components=num_components)
        new_x = grp.fit_transform(x)
    elif choice == "agglom":
        #Find out
        agg = FeatureAgglomeration(n_clusters=num_components)
        new_x = agg.fit_transform(x)
    elif choice == "thresh":
        #Below threshold gives ~26 components upon application
        vt = VarianceThreshold(threshold=0.00015)
        new_x = vt.fit_transform(x)
        kept_features = list(vt.get_support(indices=True))
        if args.dis_kept_features:
            print("Kept features: ")
            for i in kept_features:
                print(col_names[i])
    elif choice == "rf":
        y_labels = [1 if s == "D" else 0 for s in y[:, 1]]
        clf = RandomForestClassifier(n_estimators=10000, random_state=0, n_jobs=-1)
        print("Fitting RF model....")
        clf.fit(x, y_labels)
        sfm = SelectFromModel(clf, threshold=-np.inf, max_features=num_components)
        print("Selecting best features from model...")
        sfm.fit(x, y_labels)
        kept_features = list(sfm.get_support(indices=True))
        if args.dis_kept_features:
            print("Kept features: ")
            for i in kept_features:
                print(col_names[i])
        new_x = x[:, kept_features]

    print("Reduced 'x' shape:", np.shape(new_x))
    return new_x, y



def add_nsaa_scores(file_df):
    #To make sure that accepted parameter is as a DataFrame
    file_df = pd.DataFrame(file_df)
    mw_tab = pd.read_excel(nsaa_table_path + "6MW-matFiles\\6mw_matfiles.xlsx")
    mw_cols = mw_tab[["ID", "NSAA"]]
    mw_dict = dict(pd.Series(mw_cols.NSAA.values, index=mw_cols.ID).to_dict())

    nsaa_matfiles_tab = pd.read_excel(nsaa_table_path + "NSAA\\matfiles\\nsaa_matfiles.xlsx")
    nsaa_matfiles_cols = nsaa_matfiles_tab[["ID", "NSAA"]]
    nsaa_matfiles_dict = dict(pd.Series(nsaa_matfiles_cols.NSAA.values, index=nsaa_matfiles_cols.ID).to_dict())

    mw_dict.update(nsaa_matfiles_dict)
    nss = [mw_dict[i.upper()] for i in [j.split("_")[0] for j in file_df.iloc[:, 0].values]]

    file_df.insert(loc=0, column="NSS", value=nss)

    nsaa_acts_tab = pd.read_excel(nsaa_table_path + "NSAA\\KineDMD data updates Feb 2019.xlsx")
    nsaa_acts_file_names = nsaa_acts_tab.iloc[2:20, 0].values
    nsaa_acts = nsaa_acts_tab.iloc[2:20, 53:70].values
    nsaa_acts_dict = dict(zip(nsaa_acts_file_names, nsaa_acts))
    nsaa_labels = nsaa_acts_tab.iloc[1, 53:70].values

    label_sample_map = []
    for i in range(len(nsaa_labels)):
        inner = []
        for j in range(len(file_df.index)):
            fn = file_df.iloc[j, 1].split("_")[0]
            if fn in nsaa_acts_dict:
                inner.append(nsaa_acts_dict[fn][i])
            else:
                #If patient isn't found in the 'KineDMB' table, assume its a healthy control patient and thus all
                #scores for all activities are perfect (i.e. '2').
                inner.append(2)
        label_sample_map.append(inner)
    for i in range(len(nsaa_labels)):
        file_df.insert(loc=(i+1), column=nsaa_labels[i], value=label_sample_map[i])
```

```
    return file_df


for full_file_name in full_file_names:
    new_x, y = ft_red_select(full_file_name)

    #Recombine the now-reduced 'x' data with the source file name and label columns
    new_df = pd.DataFrame(np.concatenate((y, new_x), axis=1))

    #Add a column of NSAA scores to the DataFrame by referencing the external .csvs
    try:
        new_df_nsaa = add_nsaa_scores(new_df)
    except KeyError:
        print(full_file_name + " not found as entry in either '6mw_matfiles.xlsx' or 'nsaa_matfiles.xlsx', skipping...")
        continue

    #Writes the new data to the same directory as before with the same name except with 'FR_' on the front
    split_full_file_name = full_file_name.split("\\")
    split_full_file_name[-1] = "FR_" + split_full_file_name[-1]
    new_full_file_name = "\\".join(split_full_file_name)
    new_df_nsaa.to_csv(new_full_file_name)
```

# Appendix III – 'RNN.py'

```
import sys
import os
import pandas as pd
import numpy as np
import tensorflow as tf
import argparse
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
import pyexcel as pe

#Does not print to display the many warnings that TensorFlow throws up (many about updating to next version or
#deprecated functionality)
tf.logging.set_verbosity(tf.logging.ERROR)


parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained within "
                                "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
parser.add_argument("ft", help="Specify type of .mat file that the .csv is to come from, being one of 'JA' (joint "
                               "angle), 'AD' (all data), or 'DC' (data cube).")
parser.add_argument("fn", help="Specify the short file name of a .csv to load from 'source_dir'; e.g. for file "
                               "'All_D2_stats_features.csv', enter 'D2'. Specify 'all' for all the files available "
                               "in the 'source_dir'.")
parser.add_argument("choice", help="Specify the choice of what the network should be training towards. Specify 'dhc' "
                                   "for simple binary classification of sequences, 'overall' for single-target "
                                   "regression to predict the overall north star scores for sequences, or 'acts' for "
                                   "multi-target classification of possible NSAA activities that have taken place "
                                   "in the sequence and what their corresponding individual NSAA would be.")
parser.add_argument("--seq_len", type=float, nargs="?", const=1,
                    help="Option to split a source file (be it a raw joint-angle file or a JA/DC/AD stats features file) "
                         "into multiple parts for training purposes, where '--seq_len' is the number of 'rows' of data "
                         "to correspond to each file label/score(s).")
parser.add_argument("--write_settings", type=bool, nargs="?", const=True, default=False,
                    help="Option to write the hyperparameters of the RNN directly to the RNN results .csv file in "
                         "the appropriate row.")
args = parser.parse_args()

#If no optional argument given for '--split_size', defaults to split size = 10, i.e. defaults to splitting files into
#10 second increments
if not args.seq_len:
    args.seq_len = 10.0

#Location at which to store the created model
model_path = "\\tmp\\model.ckpt"

#Note: CHANGE THIS to location of the source data for the RNN to use
source_dir = "..\\output_files\\"
output_dir = "..\\output_files\\RNN_outputs\\"
sub_dirs = ["6minwalk-matfiles\\", "6MW-matFiles\\", "NSAA\\", "direct_csv\\"]
sub_sub_dirs = ["AD\\", "JA\\", "DC\\"]
choices = ["dhc", "overall", "acts"]
choice = None

#Note: CHANGE THIS to location of the 3 sub-directories' encompassing directory local to the user that's needed to
#map to the .csvs containing the NSAA information
nsaa_table_path = "C:\\msc_project_files\\"

"""RNN hyperparameters"""
x_shape = None
y_shape = None
sampling_rate = 60
#Defines the number of sequences that correspond to one 'x' sample
sequence_length = int(args.seq_len)
batch_size = 64
num_lstm_cells = 128
num_rnn_hidden_layers = 2
learn_rate = 0.001
num_epochs = 300
test_ratio = 0.2
num_acts=17


def preprocessing(test_ratio):
    """
        :return given a short file name for 'fn' command-line argument, finds the relevant file in 'source_dir' and
        adds it to 'file_names' (or set 'file_names' to all file names in 'source_dir'), reads in each file name in
        'file_names' as .csv's into dataframes, create corresponding 'y-labels' (with one 'y-label' corresponding to
        each row in the .csv, with it being a 1 if it's from a file beginning with 'D' or 0 otherwise), and shuffling/
        splitting them into training and testing x- and y-components
    """
    file_names = []
    x_data, y_data = [], []

    global source_dir
    if args.dir + "\\" in sub_dirs:
        source_dir += args.dir.upper() + "\\"
    else:
        print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
```

```
                "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', or 'direct_csv'.")
            sys.exit()

        if args.ft + "\\" in sub_sub_dirs:
            source_dir += args.ft + "\\"
        else:
            print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
                "\'JA', or \'DC\'.")
            sys.exit()

        if args.fn.lower() != "all":
            if any(args.fn in s for s in os.listdir(source_dir)):
                file names.append([s for s in os.listdir(source dir) if args.fn in s][0])
            else:
                print("Cannot find '" + str(args.fn) + "' in '" + source_dir + " '")
                sys.exit()
        else:
            file_names = [fn for fn in os.listdir(source_dir)]

        global choice
        if args.choice in choices:
            choice = args.choice
        else:
            print("Must provide a choice of 'dhc' for simple binary classification of sequences, 'overall' for single-target "
                "regression to predict the overall north star scores for sequences, or 'acts' for multi-target "
                "classification of possible NSAA activities that have taken place in the sequence and what their "
                "corresponding individual NSAA would be.")
            sys.exit()

        #Ensures that only the written files from feature select/reduct script are used if present (i.e. if the directory
        #we are concerned with is not within 'direct_csv')
        if args.dir != "direct_csv":
            file_names = [fn for fn in file_names if fn.startswith("FR_")]

        for file_name in file_names:
            print("Extracting '" + file_name + "' to x_data and y_data....")
            data = pd.read_csv(source_dir + file_name)
            if choice == "overall":
                #Only had the overall and individual NSAA scores if not already within the file
                if data.columns.values[1] != "NSS":
                    data = add_nsaa_scores(data.values)
                #Gets the overall NSAA score of the first row of the file, which will be the same throughout the file
                data = data.values
                if args.dir != "direct_csv":
                    y_label = data[0, 1]
                else:
                    y_label = data[0, 0]
            elif choice == "dhc":
                data = data.values
                if args.dir != "direct_csv":
                    y_label = 1 if file_name.split("_")[2][0] == "D" else 0
                else:
                    y_label = 1 if file_name.split("_")[1][0] == "D" else 0
            else:
                if data.columns.values[1] != "NSS":
                    data = add_nsaa_scores(data.values)
                data = data.values
                if args.dir != "direct_csv":
                    y_label = data[0][2:19]
                else:
                    y_label = data[0][1:18]
            num_data_splits = int(len(data) / sequence_length)
            for i in range(num_data_splits):
                if args.dir == "direct_csv" and choice == "dhc":
                    x_data.append(data[(i * sequence_length):((i + 1) * sequence_length), 1:])
                elif args.dir == "direct_csv":
                    x_data.append(data[(i * sequence_length):((i + 1) * sequence_length), 19:])
                else:
                    x_data.append(data[(i * sequence_length):((i + 1) * sequence_length), 21:])
                y_data.append(y_label)

        global x_shape
        global y_shape
        x_shape = np.shape(x_data)
        y_shape = np.shape(y_data)
        print("X shape =", x_shape)
        print("Y shape =", y_shape)
        return train_test_split(x_data, y_data, shuffle=True, test_size=test_ratio)



    def add_nsaa_scores(file_df):
        #To make sure that accepted parameter is as a DataFrame
        file_df = pd.DataFrame(file_df)
        mw_tab = pd.read_excel(nsaa_table_path + "6MW-matFiles\\6mw_matfiles.xlsx")
        mw_cols = mw_tab[["ID", "NSAA"]]
        mw_dict = dict(pd.Series(mw_cols.NSAA.values, index=mw_cols.ID).to_dict())

        nsaa_matfiles_tab = pd.read_excel(nsaa_table_path + "NSAA\\matfiles\\nsaa_matfiles.xlsx")
        nsaa_matfiles_cols = nsaa_matfiles_tab[["ID", "NSAA"]]
        nsaa_matfiles_dict = dict(pd.Series(nsaa_matfiles_cols.NSAA.values, index=nsaa_matfiles_cols.ID).to_dict())

        mw_dict.update(nsaa_matfiles_dict)
        #Adds column of overall NSAA scores at position 0
        nss = [mw_dict[i] for i in [j.split("_")[0] for j in file_df.iloc[:, 0].values]]
        file_df.insert(loc=0, column="NSS", value=nss)
```

```
    nsaa_acts_tab = pd.read_excel(nsaa_table_path + "NSAA\\KineDMD data updates Feb 2019.xlsx")
    nsaa_acts_file_names = nsaa_acts_tab.iloc[2:20, 0].values
    nsaa_acts = nsaa_acts_tab.iloc[2:20, 53:70].values
    nsaa_acts_dict = dict(zip(nsaa_acts_file_names, nsaa_acts))
    nsaa_labels = nsaa_acts_tab.iloc[1, 53:70].values

    label_sample_map = []
    for i in range(len(nsaa_labels)):
        inner = []
        for j in range(len(file_df.index)):
            fn = file_df.iloc[j, 1].split("_")[0]
            if fn in nsaa_acts_dict:
                inner.append(nsaa acts dict[fn][i])
            else:
                #If patient isn't found in the 'KineDMB' table, assume its a healthy control patient and thus all
                #scores for all activities are perfect (i.e. '2').
                inner.append(2)
        label_sample_map.append(inner)
    for i in range(len(nsaa_labels)):
        file_df.insert(loc=(i+1), column=nsaa_labels[i], value=label_sample_map[i])
    return file_df



def create_batch_generator(x, y=None, batch_size=64):
    n_batches =len(x)//batch_size
    #Ensures 'x' is a multiple of batch size
    x = x[:n_batches*batch_size]
    if y is not None:
        #Ensures 'y' is a multiple of batch size
        y = y[:n_batches*batch_size]
    #For each 'batch_size' chuck of x, yield the next part of the 'x' and 'y' data (i.e. the next 'batch size' samples)
    for ii in range(0, len(x), batch_size):
        if y is not None:
            yield x[ii:ii+batch_size], y[ii:ii+batch_size]
        else:
            yield x[ii:ii+batch_size]



def write_to_csv(trues, preds, open_file=True):
    print("\nWriting true and predicted values to .csv....")
    df = pd.DataFrame()
    df["Sequence Number"] = np.arange(1, len(trues)+1)
    df["Trues"] = trues
    if choice != "acts":
        df["Predictions"] = preds
    else:
        df["Predictions"] = ["[" + " ".join(str(num) for num in preds[i]) + "]" for i in range(len(preds))]
    #Empty column to keep a space between the trues/predictions and RNN settings
    df[""] = ""
    settings = ["X shape = " + str(x_shape), "Y shape = " + str(y_shape), "Test ratio = " + str(test_ratio),
                "Sequence length = " + str(sequence_length), "Features length = " + str(len(x_train[0][0])),
                "Num epochs = " + str(num_epochs), "Num LSTM units per layer = " + str(num_lstm_cells),
                "Num hidden layers = " + str(num_rnn_hidden_layers), "Learning rate = " + str(learn_rate)]
    df["Settings"] = pd.Series(settings)
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)
    full_output_name = output_dir + "RNN_trues_preds.csv"
    if os.path.exists(full_output_name):
        os.remove(full_output_name)
    df.to_csv(full_output_name, index=False, float_format="%.2f")
    if open_file:
        os.startfile(full_output_name)
    if args.write_settings:
        settings = ["" for i in range(7)] + settings
        sheet = pe.get_sheet(file_name="..\\RNN Results.ods")
        sheet.row += settings
        sheet.save_as("..\\RNN Results.ods")



class RNN(object):
    def __init__(self, features_length, seq_len, lstm_size, num_layers, batch_size, learning_rate, num_acts):
        """
        :param sets the hyperparameters as 'RNN' object attributes, builds the RNN graph, and initializes
        the global variables
        """
        self.features_length = features_length
        self.seq_len = seq_len
        self.lstm_size = lstm_size
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.num_acts = num_acts

        self.g = tf.Graph()
        with self.g.as_default():
            tf.set_random_seed(123)
            self.build()
            self.saver = tf.train.Saver()
            self.init op = tf.global variables initializer()

    def build(self):
        """
        :return: no return, but sets up the complete RNN architecture to be called upon initialization
        """
```

```
        #Placeholders to hold the data that is fed into the RNN (where each batch has shape 'seq_len' x
        # 'features_length' for 'x' data and a single '1' or '0' for 'y' data)
        tf_x = tf.placeholder(tf.float32, shape=(self.batch_size, self.seq_len, self.features_length), name='tf_x')
        if choice != "acts":
            tf_y = tf.placeholder(tf.float32, shape=(self.batch_size), name='tf_y')
        else:
            tf_y = tf.placeholder(tf.float32, shape=(self.batch_size, self.num_acts), name='tf_y')
        tf_keepprob = tf.placeholder(tf.float32, name='tf_keepprob')

        #Defines several hidden RNN layers, with 'self.num_layers' layers, 'self.lstm_size' number of cells per
        #layer, and each implementing dropout functionality
        cells = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.DropoutWrapper(tf.contrib.rnn.BasicLSTMCell(
            self.lstm size), output keep prob=tf keepprob) for i in range(self.num layers)])
        #Sets the initial state for the RNN layers
        self.initial_state = cells.zero_state(self.batch_size, tf.float32)
        #With the RNN layer architecture defined in 'cells', sets up the layers to feed from input 'x' placeholder
        #into 'cells' and with the above defined 'initial_state'
        lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, tf_x, initial_state = self.initial_state)


        #Defines the cost based on the sigmoid cross entropy with the RNN output and the 'y' labels, along with
        #the Adam optimizer for the optimizer of choice with a learning rate set by 'self.learning_rate'
        if choice != "acts":
            logits = tf.layers.dense(inputs=lstm_outputs[:, -1], units=1, activation=None, name='logits')
            logits = tf.squeeze(logits, name='logits_squeezed')
        else:
            logits = tf.layers.dense(inputs=lstm_outputs[:, -1], units=self.num_acts, activation=None, name='logits')
        if choice == "overall":
            cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=tf_y, predictions=logits), name='cost')
            predictions = {'cost': cost}
        elif choice == "dhc":
            # Adds an output layer that feed from the final values emitted from the 'cells' layers with a single neuron
            # to classify for a binary value
            y_proba = tf.nn.sigmoid(logits, name='probabilities')
            cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf_y, logits=logits), name='cost')
            predictions = {'probabilities': y_proba, 'labels': tf.cast(tf.round(y_proba), tf.int32, name='labels')}
        else:
            cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=tf_y, predictions=logits), name='cost')
            predictions = {'labels': tf.cast(tf.round(logits), tf.int32, name='labels'), 'cost': cost}
        optimizer = tf.train.AdamOptimizer(self.learning_rate)
        train_op = optimizer.minimize(cost, name='train_op')


    def train(self, x_train, y_train, num_epochs):
        """
        :param the training data for both 'x' and 'y' data is provided, along with the number of training epochs
        that the training will run for
        :return no return, but for each epoch, fetches the next batch of data from the 'x' and 'y' training sets with
        a dropout probability and initial state, and feeds it into the RNN for training following the architecture
        and hyperparameters of 'build()', while printing the loss at every 20 iterations for a given epoch
        """
        with tf.Session(graph=self.g) as sess:
            sess.run(self.init_op)
            iteration = 1
            for epoch in range(num_epochs):
                state = sess.run(self.initial_state)
                for batch_x, batch_y in create_batch_generator(x_train, y_train, self.batch_size):
                    feed = {'tf_x:0': batch_x, 'tf_y:0': batch_y, 'tf_keepprob:0': 0.5, self.initial_state: state}
                    loss, _, state = sess.run(['cost:0', 'train_op', self.final_state], feed_dict=feed)
                    if iteration % 20 == 0:
                        print("Epoch: %d/%d Iteration: %d | Train loss: %.5f" % (epoch+1, num_epochs, iteration, loss))
                    iteration += 1
            self.saver.save(sess, model_path)
        print("\n\n")


    def predict(self, x_test, return_proba=False):
        """
            :param feeds in the 'x' testing data with which we wish to compute the predicted values (1 or 0) for
            each of the 2-dimensional samples (where each sample is 'self.seq_length' x 'self.features_length'), with
            an option to return the labels rather than the probabilities for assessment purposes
            :return: list of predicted values that are the result of feeding through the 'x' testing data through the
            now-trained RNN model
        """
        preds = []
        with tf.Session(graph = self.g) as sess:
            self.saver.restore(sess, model_path)
            test_state = sess.run(self.initial_state)
            for ii, batch_x in enumerate(create_batch_generator(x_test, None, batch_size=self.batch_size), 1):
                feed = {'tf_x:0': batch_x, 'tf_keepprob:0': 1.0, self.initial_state: test_state}
                if return_proba:
                    pred, test_state = sess.run(['probabilities:0', self.final_state], feed_dict=feed)
                elif choice == "overall":
                    pred, test_state = sess.run(['logits_squeezed:0', self.final_state], feed_dict=feed)
                else:
                    pred, test_state = sess.run(['labels:0', self.final_state], feed_dict=feed)
                preds.append(pred)
        return np.concatenate(preds)


#Extracts the training and testing data, builds the RNN based on the hyperparameters initially set, and trains the model
x_train, x_test, y_train, y_test = preprocessing(test_ratio=test_ratio)
rnn = RNN(features_length=len(x_train[0][0]), seq_len=sequence_length, lstm_size=num_lstm_cells,
                  num_layers=num_rnn_hidden_layers, batch_size=batch_size, learning_rate=learn_rate, num_acts=num_acts)
```

```
rnn.train(x_train, y_train, num_epochs=num_epochs)

preds = rnn.predict(x_test)
#Ensures the true 'y' values are the same length and the predicted values (so 'preds' and 'y_true' have the same shape)
y_true = y_test[:len(preds)]


if choice == "overall":
    mse = mean_squared_error(y_true=y_true, y_pred=preds)
    print("\n\nMean Squared Error = " + str(round(mse, 4)))
    mae = mean_absolute_error(y_true=y_true, y_pred=preds)
    print("Mean Absolute Error = " + str(round(mae, 4)))
elif choice == "dhc":
    #Calculates and prints the accuracy to the user
    accuracy = round(((np.sum(preds == y_true) / len(y_true)) * 100), 2)
    print("\n\nTest Accuracy = " + str(accuracy) + "%")
else:
    ind_sum, all_sum = 0, 0
    for i in range(len(preds)):
        ind_sum += np.sum(preds[i] == y_true[i])
        all_sum = all_sum + 1 if list(preds[i]) == list(y_true[i]) else all_sum
    ind_accuracy = round(((ind_sum / (len(y_true)*num_acts)) * 100), 2)
    print("\n\nIndividual Activity Accuracy = " + str(ind_accuracy) + "%")
    all_accuracy = round(((all_sum / len(y_true)) * 100), 2)
    print("All Activities Accuracy = " + str(all_accuracy) + "%")


write_to_csv(trues=y_true, preds=preds)
```

# Appendix IV – 'mat_act_div.py'

```
#Note: for this to work, we must *first* download the google sheets and place in the 'NSAA' file directory that is
#local to the user. Hence, the user must first download the google sheet at:
#'https://docs.google.com/spreadsheets/d/1OvkGU6kwmMxD6zdZqXcNKUvurluFbAx5IND7_dXibjE', place it in the location of
#their NSAA directory, and change the global variable 'source_dir' below to reflect the path to their 'NSAA' directory

import argparse
import pandas as pd
import numpy as np
import sys
import os
import scipy.io as sio

parser = argparse.ArgumentParser()
parser.add_argument("version", help="Specify the version that we wish to consider (i.e. the visit number of the "
                                    "patients). Must be either 'V1' or 'V2'.")
parser.add_argument("fn", help="Specify the patient ID to split the .mat file of")
args = parser.parse_args()

source_dir = "C:\\msc_project_files\\NSAA\\"



def extract_act_times():
    data = pd.read_excel(source_dir + "DMD Start_End Frames.xlsx")
    patient_ids = data["Patient"].values.tolist()
    patient_ids = list(zip(patient_ids, np.arange(1, len(patient_ids)+1)))

    patient_ids_v1v2 = [[]]
    for pair in patient_ids:
        if pair[0] != "V2":
            patient_ids_v1v2[-1].append(pair)
        else:
            patient_ids_v1v2.append([])

    if args.version == "V1" or args.version == "V2":
        v1 v2 index = 0 if args.version == "V1" else 1
        id_pairs = patient_ids_v1v2[v1_v2_index]
        #Remove 'nan' pairs
        id_pairs = [id_pair for id_pair in id_pairs if str(id_pair[0]) != "nan"]
        ids = [id[0] for id in id_pairs]
        if args.fn in ids:
            #Retrieves the relevant row from the table and only includes the columns containing the
            act_times = data.iloc[id_pairs[ids.index(args.fn)][1]-1, 2:36].values
            #'Pair up' each start and end time for each activity within 'act_times'
            act_times_outer = [[[act_times[i], act_times[i+1]] for i in range(0, len(act_times), 2)]]
            return act_times_outer, ids
        elif args.fn == "all":
            act_times_outer = []
            for id in ids:
                act_times = data.iloc[id_pairs[ids.index(id)][1] - 1, 2:36].values
                act_times_outer.append([[act_times[i], act_times[i+1]] for i in range(0, len(act_times), 2)])
            return act_times_outer, ids
        else:
            print("Second arg ('fn') must be a patient ID within the specified version heading.")
            sys.exit()
    else:
        print("First arg ('version') must be one of 'V1' or 'V2'.")
        sys.exit()


def divide mat file(act times outer, ids):
    #Filters the file names within 'NSAA\matfiles\' by the chosen version
    global source_dir
    source_dir += "matfiles\\"
    if args.version == "V1":
        version_fns = [fn for fn in os.listdir(source_dir) if not fn.endswith("2.mat") and "V2" not in fn]
    else:
        version_fns = [fn for fn in os.listdir(source_dir) if fn.endswith("2.mat") or "V2" in fn]

    if args.fn != "all":
        version_fns = [fn for fn in version_fns if args.fn in fn]
    else:
        version_fns = [fn for fn in version_fns if fn != "act_files"]

    if not os.path.exists(source_dir + "act_files\\"):
        os.mkdir(source_dir + "act_files\\")
    else:
        for fn in os.listdir(source_dir + "act_files\\"):
            os.remove(source_dir + "act_files\\" + fn)
    for file_name in version_fns:
        try:
            if len(act_times_outer) == 1:
                act_times = act_times_outer[0]
            else:
                act_times = act_times_outer[ids.index(file_name.split("-")[0])]
        except ValueError:
            print("Skipping " + file_name + " as not a row in table for version " + args.version + "...")
            continue
```

```
        #For a given patient name, if it doesn't have the necessary times in their respective row in the table already
        #filled in with numbers, skip it and continue to the next...
        exit_loop = 0
        for act in act_times:
            for a in act:
                if "nan" in str(a) or "-1" in str(a):
                    exit_loop = 1
        if exit_loop == 1:
            print("Skipping " + file_name + " as missing values in table for corresponding entry for version " +
                  args.version + "...")
            continue

        print("Dividing up " + file name + "...")
        full_file_name = source_dir + file_name
        file = sio.loadmat(full_file_name)

        for i, pair in enumerate(act_times):
            inner_table = file["tree"][0][0][6][0][0][10][0][0][2][0]
            new_inner_table = inner_table[pair[0]:pair[1]]
            np.delete(file["tree"][0][0][6][0][0][10][0][0][2], 0)
            file["tree"][0][0][6][0][0][10][0][0][2] = [new_inner_table]
            new_file_name = str(file_name.split(".mat")[0]) + "_act" + str(i+1) + ".mat"
            sio.savemat(source_dir + "act_files\\" + new_file_name, file)
            file["tree"][0][0][6][0][0][10][0][0][2] = [inner_table]



act_times, ids = extract_act_times()
divide_mat_file(act_times, ids)
```

# Appendix V – 'ext_raw_measurements.py'

```
import argparse
import sys
import os
import scipy.io as sio
import pandas as pd

#Note: CHANGE THESE to location of the 3 sub-directories' encompassing directory local to the user
source_dir = "C:\\msc_project_files\\"
sub_dirs = ["6minwalk-matfiles\\", "6MW-matFiles\\", "NSAA\\"]
measurements = ["orientation", "position", "velocity", "acceleration", "angularVelocity", "angularAcceleration",
                "sensorFreeAcceleration", "sensorMagneticField", "sensorOrientation", "jointAngle", "jointAngleXYZ"]

axis_labels = ["X", "Y", "Z"]

#Below 3 lists are labels for the 23 segments, 22 joints, and 17 sensors, as dictated by the 'MVN User Manual'
segment_labels = ["Pelvis", "L5", "L3", "T12", "T8", "Neck", "Head", "RightShoulder", "RightUpperArm",
                  "RightForeArm", "RightHand", "LeftShoulder", "LeftUpperArm", "LeftForeArm", "LeftHand",
                  "RightUpperLeg", "RightLowerLeg", "RightFoot", "RightToe", "LeftUpperLeg", "LeftLowerLeg",
                  "LeftFoot", "LeftToe"]

joint_labels = ["jL5S1", "jL4L3", "jL1T12", "jT9T8", "jT1C7", "jC1Head", "jRightT4Shoulder", "jRightShoulder",
                "jRightElbow", "jRightWrist", "jLeftT4Shoulder", "jLeftShoulder", "jLeftElbow", "jLeftWrist",
                "jRightHip", "jRightKnee", "jRightAnkle", "jRightBallFoot", "jLeftHip", "jLeftKnee",
                "jLeftAnkle", "jLeftBallFoot"]

sensor_labels = ["Pelvis", "T8", "Head", "RightShoulder", "RightUpperArm", "RightForeArm", "RightHand",
                 "LeftShoulder", "LeftUpperArm", "LeftForeArm", "LeftHand", "RightUpperLeg", "RightLowerLeg",
                 "RightFoot", "LeftUpperLeg", "LegLowerLeg", "LeftFoot"]

#Mapping used to map a given measurement name to the number of x/y/z values found in the .mat file
measure_to_len_map = {"orientation": 23, "position": 23, "velocity": 23, "acceleration": 23, "angularVelocity": 23,
                      "angularAcceleration": 23, "sensorFreeAcceleration": 17, "sensorMagneticField": 17,
                      "sensorOrientation": 22, "jointAngle": 22, "jointAngleXZY": 22}

#Mapping used to select lists of labels names to use based on the length of the numbers contained in data array
seg join sens map = {len(segment labels): segment labels, len(joint labels): joint labels, len(sensor labels):
sensor_labels}


parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained within "
                                "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
parser.add_argument("fn", help="Specifies the short name (e.g. 'D11') of the file that we wish to extract the specified "
                               "raw measurements. Specify 'all' for all the files available in the 'source_dir'.")
parser.add_argument("measurements", help="Specifies the measurements to extract from the source .mat file. Separate "
                                         "each measurement to extract by a comma.")
args = parser.parse_args()


if args.dir + "\\" in sub_dirs:
    if args.dir == "6minwalk-matfiles":
        source_dir += args.dir + "\\all_data_mat_files\\"
    elif args.dir == "6MW-matFiles":
        source_dir += args.dir + "\\"
    else:
        source_dir += args.dir + "\\matfiles\\"
else:
    print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
          "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', or 'direct_csv'.")
    sys.exit()

file_names = os.listdir(source_dir)
if any(args.fn in fn for fn in file_names):
    full_file_names = [source_dir + [fn for fn in file_names if args.fn in fn][0]]
elif args.fn == "all":
    full_file_names = [source_dir + file_name for file_name in file_names if file_name.endswith(".mat")]
else:
    print("Second arg ('fn') must be the short name of a file (e.g. 'D2' or 'all') within", source_dir)
    sys.exit()

measures = []
for measure in args.measurements.split(","):
    if measure in measurements:
        measures.append(measure)
    else:
        print("'" + measure + "' not a valid 'measurement' name. Must be one of:", measurements)
        sys.exit()

for measure in measures:
    if not os.path.exists(source_dir + measure):
        os.mkdir(source_dir + measure)

for full_file_name in full_file_names:
    print("Extracting", measures, "from '" + full_file_name + "'...")
    mat_file = sio.loadmat(full_file_name)
    tree = mat_file["tree"]
    try:
        frame_data = tree[0][0][6][0][0][10][0][0][3][0]
    except IndexError:
```

```
    frame_data = tree[0][0][6][0][0][10][0][0][2][0]
col_names = frame_data.dtype.names
# Extract single outer-list wrapping for vectors and double outer-list values for single values
frame_data = [[elem[0] if len(elem[0]) != 1 else elem[0][0] for elem in row] for row in frame_data]
df = pd.DataFrame(frame_data, columns=col_names).iloc[3:]

for measure in measures:
    measurement_names = seg_join_sens_map[measure_to_len_map[measure]]
    headers = ["(" + measurement_name + ") : (" + axis + "-axis)"
               for measurement_name in measurement_names for axis in axis_labels]

    measure_data = [list(data) for data in df.loc[:, measure].values]
    short file name = full file name.split("\\")[-1].split("-")[0]
    measure_df = pd.DataFrame(measure_data, index=[short_file_name for i in range(len(measure_data))])

    new_file_name = source_dir + measure + "\\" + full_file_name.split("\\")[-1].split(".mat")[0] + \
                    "_" + measure + ".csv"
    print("Writing '" + new_file_name + "' to '" + source_dir + measure + "\\'")
    measure_df.to_csv(new_file_name, header=headers)
```