Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Recurrent Neural Networks Applied for Human Movement in Subjects with Duchenne Muscular Dystrophy

Author:
Daniel Heaton

Supervisor:
Aldo Faisal

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing (Machine Learning) of Imperial College London

September 2019

Abstract

Your abstract.

Contents

Part I

Background and Basis of Research

Chapter 1

Project Overview and Background

1.1 Motivation for Project and Purpose of Work

Modern machine learning algorithms and methodologies has seen great success in regards to being applied to biomedical data in order to provide insights that assist specialists and help diagnose potential conditions that may afflict subjects. One example of this is DeepMind's recent progress with AI-assisted eye scans to detect over 50 different types of diseases potentially in a subject's eye as accurately as world-leading expert doctors [1]. Deep learning techniques have also been utilized by HeartFlow to help build 3D models of subjects' hearts and assess the impacts of blockages on blood flow to the heart [2]. Based on these breakthroughs, among others, in recent years, it is very probable that AI-assistance will become the new norm in various clinics and hospitals around the world within the next decade [3]. Taking note of the prominent applicability of artificial intelligence to the analysis of human movement in particular, Imperial College London have undertaken a research initiative in collaboration with Great Ormond Street Hospital to investigate the applicability of various AI techniques to the analysis of subjects with Duchenne muscular dystrophy [4], a form of muscular dystrophy that predominantly affects young males under the age of 12 and severely impacts their movement ability to varying degrees. The hope is that great strides in treatments can be achieved by utilizing artificial intelligence to inform and make decisions on a subject-by-subject basis and potentially draw insights about the condition.

With regards to this project specifically that is undertaken as part of this research initiative, we wish to investigate the applicability of recurrent neural networks (RNNs) to the features of human movement data provided by body suit measurements captured from subjects with Duchenne muscular dystrophy (DMD). 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 of the subjects providing it. These insights from the project will hopefully have a direct positive impact in the ability to assess subjects via the North Star Ambulatory Assessment (NSAA) and possibly provide insights into other features of the condition. Generally, in this project we are trying to gain insights about the body suit data that

is captured as '.mat' files (MATLAB data files) through the different measurements that are captured by the 17 sensors of the suit (joint angles, position, accelerometer values, etc.) of NSAAs or 6-minute walks assessments of the subjects by means of sequence modelling using 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.

The overall goal of the project is therefore to provide a deliverable that includes a complete system that works with varying forms of suit data, learns from it, and provides insights about it, while hopefully being able to be adapted to new subjects, new NSAA assessments of existing subjects, and help inform specialists of the severity of their condition. A significant hope, therefore, is to provide a software solution that positively and directly impacts the lives of those with DMD through hopefully making their assessments easier and more accurate.

1.2 Aims and Objectives

With the overall aim of the project outlined and with the motivation for undertaking the work provided above, we now turn our attention to covering some of the main aims of the project. These include:

- Building 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 for each of the 6-minute walk and NSAA assessments. This is primarily due to the fact that the data collection is currently an ongoing process and a large repository of previously-collected suit data from other 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.

- Being 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 practical undertaking

of the assessments through minimizing the amount of testing the subjects have to do.

- Investigating the impact of training models on different types of source data directly. For example, we'd like to see whether or not it's possible to train models on natural movement behaviour data sets to the same standard as if we were using NSAA data sets when training towards overall and individual NSAA scores. If this were to be the case, then there exists a real possibility of not requiring the NSAA assessments to be completed by subjects at all, and instead simply requiring the subject to undertake natural movement instead, which may be significantly easier and/or more practical for subjects.

- Building models that are trained only on one 'version' of assessments of subjects and attempt to generalise to subsequent versions. Here, by 'version' we mean an assessment of a subject that takes place at a certain time, with subsequent versions being the same assessment but taken 6-months later on. For example, a subject's initial NSAA assessment would be stored as the subject name 'D4', and when that subject returns 6-months later to undertake their subsequent NSAA assessment the resultant data file would be stored as 'D4V2'. The hope is therefore to train models on non-'V2' files and generalise to newly presented 'V2' files. This should provide an advisory tool for any specialists wishing to assess how a subject's conditioned has progressed during the time between assessments.

- Looking into how possible it is to build models that generalise well to new subjects and the system settings needed to achieve this; by this, we mean models that are able to assess subjects that they have never come across before during training (which differs to the previous bullet point, which looks at new data from existing subjects). Therefore, if this were to be the case then we would be able to extend the applicability of this system to not only new assessments of the existing subjects but brand new subjects to the overall research initiative. There are numerous techniques that we would have to look into if the models have a problem generalizing, and so a large amount of the model predictions sets will focus on this aim.

- Package all the scripts and models necessary for a specialist or any other researcher wishing to use any of the built tools in a way that is easy to use and gives intuitive output. This requires us to construct the system in a way where it is possible to be uses by others outside of the development environment in order to be practically applicable to achieve the aims outlined above.

With our overall project aims outlined above, it's also useful to cover some of the objectives that we intend to achieve in order to complete these aims, many of which will be investigated within their own experiment set or model predictions set. These include:

- Comparing models built from different measurements (e.g. joint angles, acceleration values, computed statistical values, etc.) on their performance of evaluating unseen sequences of data to an accurate D/HC classification and overall/single-act regression of NSAA scores.
- Evaluating the ideal values for different sequence setups for the data going into the model with respect to the performance for various output types. This includes finding the ideal sequence length, sequence overlap, and discard proportion of frames within the sequences.
- Investigating the ideal number of features needed for the raw measurements and computed statistical values to train a model. This will involve a trade-off, with more features providing more of the inherent variance within the data and fewer features making it easier for the model to learn from.
- Looking into how well models performed when evaluated on files from a different source directory than their own. For example, we'd like to investigate the potential to models built from NSAA files and assess subject files from the natural movement behaviour data set. If this is possible, then with the finished models we could use these to assess a subject based solely on their natural movement, which might be much more practical than requiring the subject to undertake the NSAA assessment.
- Investigating how well models perform when they are familiar with the subject as opposed to when the model has never seen the subject before in training (even if it was trained on different data from the same subject than was used for assessing the subject).
- Assessing the applicability of generalisation techniques that includes downsampling the data, adding Gaussian noise to the data set, concatenation of features for multiple measurement types, and the leaving-out of anomalies within the subjects.

Chapter 2

Basis of Research Project

2.1 Duchenne Muscular Dystrophy

The data that we shall be working with for this project is from subjects who have varying severities of Duchenne muscular dystrophy (DMD). DMD is a genetic disorder that is characterized by progressive muscle degeneration and weakness and is caused by the absence of dystrophic, a protein that helps keeps muscle cells intact [5]. This leads to increasing levels of disability and is a progressive condition, meaning that it gets worse over time. It's classified as a rare disease, with around 2,500 patients in the UK and an estimated 300,000 sufferers worldwide [6]. There are currently no known cures for any form of muscular dystrophy (MD), though there are treatments available to help manage the conditions [7].

DMD is one of the more severe forms of MD and generally affects boys in their early childhoods, and those with the condition generally only live into their 20s or 30s. The muscle weakness starts in early childhood and symptoms are usually first noticed between the ages of 2 and 5. The weakness mainly affects the muscles near the hips and shoulders, so among the first signs of the disorder are when the child has difficulty getting up off the floor, walking, or running. The weakness progresses to eventually affect all muscles used for moving and also those involved in breathing and the heart muscle. Many are confined to a wheelchair by 12 years of age, with those in their late teens generally losing the ability to move their arms and experiencing progressive problems with breathing.

With the aim to increase the life span and movement options of sufferers of DMD, a range of treatments are available. These range from steroids to increase muscle strength, physiotherapy to assist with mobility, and surgery to correct postural deformities. And thanks to advances in cardiac and respiratory care, life expectancy for sufferers is increasing and many are able to survive into their 30s and 40s with careers and families, with there even being cases of men with the condition living into their 50s. Additionally, there is ongoing research looking into ways to repair the genetic mutations and damaged muscles associated with various forms of MD.

2.2 The North Star Ambulatory Assessment

The North Star Ambulatory Assessment (NSAA) is a 17-item rating scale that is used to measure functional motor abilities in subjects with DMD and is generally used to monitor the progression of the disease and the effects of treatments. The tests are to be completed without the use of any thoracic braces or any equipment assistance that may help the subject to complete the activities. To carry out the assessments, the assessor conducting the assessments needs a mat, a stopwatch, a box step, a size-appropriate chair, and at least 10-metres of pathway [8].

To carry out the assessment, the assessor gets the subject to carry out 17 sequential tasks. Each task is graded as follows: '2' if there is no obvious modification of activity, '1' if the subject uses a modified method but achieves the goal independent of physical assistance from another, and '0' if the subject is unable to complete the activity independently. The 17 activities involved in the assessment with the requests to the subject are given below:

1. **Stand**: "Can you stand up tall for me for as long as you can and as still as you can?"
2. **Walk**: "Can you walk from A to B (state to and where from) for me?"
3. **Stand up from chair**: "Stand up from the chair, keeping your arms folded if you can"
4. **Stand on one leg – right**: "Can you stand on your right leg for as long as you can?"
5. **Stand on one leg – left**: "Can you stand on your left leg for as long as you can?"
6. **Climb box step – right**: "Can you step onto the top of the box using your right leg first?"
7. **Climb box step – left**: "Can you step onto the top of the box using your left leg first?"
8. **Descend box step – right**: "Can you step down from the box using your right leg first?"
9. **Descend box step – left**: "Can you step down from the box using your left leg first?"
10. **Gets to sitting**: "Can you get from lying to sitting?"
11. **Rise from floor**: "Get up from the floor using as little support as possible and as fast as you can (from supine)."
12. **Lifts head**: "Lift your head to look at your toes keeping your arms folded."

13. **Stand on heels**: “Can you stand on your heels?”
14. **Jump**: “How high can you jump?”
15. **Hop right leg**: “Can you hop on your right leg?”
16. **Hop left leg**: “Can you hop on your left leg?”
17. **Run (10m)**: “Run as fast as you can to. . . .(give point).”

The NSAA assessment has been shown to be a quick, reliable, and clinically relevant method to measure the functional motor ability of ambulant children with DMD, and is also considered to be suitable to be used in research. It has also been shown to have high intra-observer reliability and high inter-observer reliability [9]. This means that NSAA is generally fairly reliable so as to be used as part of research assuming adequate training is provided to assessors. Furthermore, the hierarchy of items within NSAA was shown to be supported by clinical expert opinion, with items in the NSAA assessment being listed based on their level of difficult which is agreed upon by most experts, while a questionnaire-based study shows that clinicians generally consider NSAA as clinically relevant [10].

2.3 The KineDMD Research Initiative

The project undertaken as described in this report is part of a wider research initiative known as the ‘KineDMD’ study, conducted by Imperial College London in collaboration with Great Ormond Street Hospital (GOSH). The study involves around 20 DMD subjects (‘D’) subjects and 10 healthy control (‘HC’) subjects who participate in the study for 12 months, who are assessed wearing a sensor suit on selected days during clinical assessments at GOSH, along with using fitness tracker bracelets in the form of Apple watches throughout the trial, which collect data of everyday movements while the subjects are at home or school. A broad aim of the research initiative is to make use of AI to make sense of the data patterns collected from the suit and watches for each of the subjects which, from there, would aid doctors in being able to monitor disease progression with more precision [11]. The initiative has been funded with £320,000 through the Duchenne Research Fund to develop and test the bodysuit that captures the motions of subjects suffering with DMD [4].

The hope is that insights found would cut down the time taken to test new treatments and thus drive down the costs of future clinical trials. A further aim of the initiative is that the developed suit and associated AI techniques and research projects undertaken as part of the initiative will help determine whether any new treatment regimes are working, which would be able to help inform doctors on future treatments. This is particularly useful for specialists, as the condition can be difficult to treat due to relatively slow progression and each subject responds uniquely; furthermore, many of the assessments are done by ‘eye’ instead of using measurement and

objective methods. The initiative hopes that bringing AI techniques into the assessments will take a lot of the human fallibility elements out of the assessments and give a more informed perspective that is better able to understand the progression of the condition in the subjects.

Chapter 3

Overview of Recurrent Neural Networks

3.1 Outline

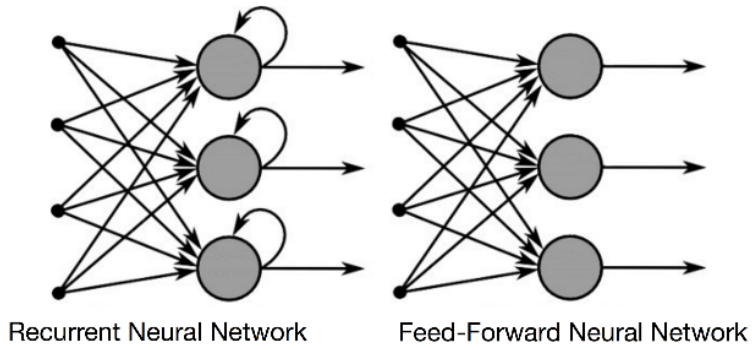
As recurrent neural networks (RNNs) will form the basis of the engineering aspect of the project through implementations using Python and supporting libraries, it's necessary to outline some of the theory and applications prior to implementing them; in doing so, we can explore why exactly they are useful when adapted to work with the body suit data we're concerned with and to solve the problems we're looking to solve. We begin by exploring the shortcomings of feedforward neural networks (FFNNs) and how using RNNs in their place works to overcome them (and, in particular, why they're applicable here). We then touch on the mathematical basis of how the hidden nodes' states change with respect to time and the input, along with how we use long short-term memory (LSTM) units to help deal with the vanishing/-exploding gradient problems. Finally, we look at how we can implement this type of network within a Python script by means of the TensorFlow framework, including how we build the model, train it, and test it on unseen data.

3.2 Motivation, Architecture, and Training

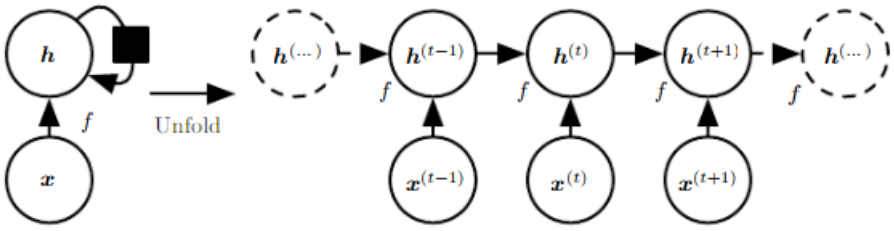
A problem with FFNNs is that they don't handle the order of data that is fed into the network: each sample fed through and classified or regressed is independent of all other samples. For example, if a convolutional neural network is trained to determine whether images are either of a cat or of a dog, 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 ('frame') 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 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 frames of data from the body suit, then it would treat each as independent entities with regard to the network's 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. In contrast with FFNNs, RNNs share their weight parameters across different time steps: this allows a sequence to extend and apply the model to examples of different forms and generalize across them (which allows a sentence like "I went to Nepal in 2009" and "In 2009 I went to Nepal" to recognize the year, '2009', in the same way)[12]. The resultant core architectural difference between the two 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. If we considered a sequential model where the state of a hidden node h^t is modified by not only the input x but also the layer's previous state h^{t+1}, we can essentially 'unfold' this dependency with respect to time to get:



In other words, the output of a hidden node at time t is given as h^t and is a function f of the input at this time from the previous layer x^t and the output of the same layer at the previous time step h^{t-1}. This can therefore be seen as the 'memory'

aspect of an RNN: values from previous parts of a sequence carry over to influence the subsequent parts. It should be noted, however, that this is only within sequences and subsequent sequences are considered to be independent of each other (in the same way that images fed through a convolutional neural network are independent of each other); hence, the choosing of the correct time-dependency in the form of the sequence length is a key aspect of experimentation which we further look into in our experimentation. This idea of unfolding in time can be extended to apply to multiple layers and multiple nodes per layer and can be seen in the general equation that the hidden nodes in an RNN use to calculate their output:

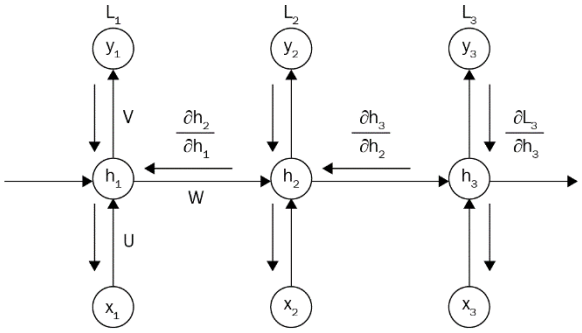
$$h^{(t)} = \varphi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

Note that the above function ϕ_h is equivalent to f in the above diagram, as in both cases they represent the activation function for layer h. We interpret this as the output of a hidden layer at time step t being a combination of the previous hidden layer output and the current input to the hidden layer, with both being modified by their respective weight matrices. It's also important to note that, not only is there a weight matrix W_{xh} that is learned through training to map from the 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. Alternatively, a way to look at it is the W_{hh} matrix controls the influence each left-to-right arrow in the unfolded sequence image has on the state it points to, while the W_{xh} controls how much influence up down-to-up arrow in the unfolded sequence image has on the state it points to.

This requirement of using the additional weight matrix for the states of the hidden layers is also reflected in the backpropagation through time (BPTT) equation for the updating of the W_{hh} matrix via gradient descent:

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

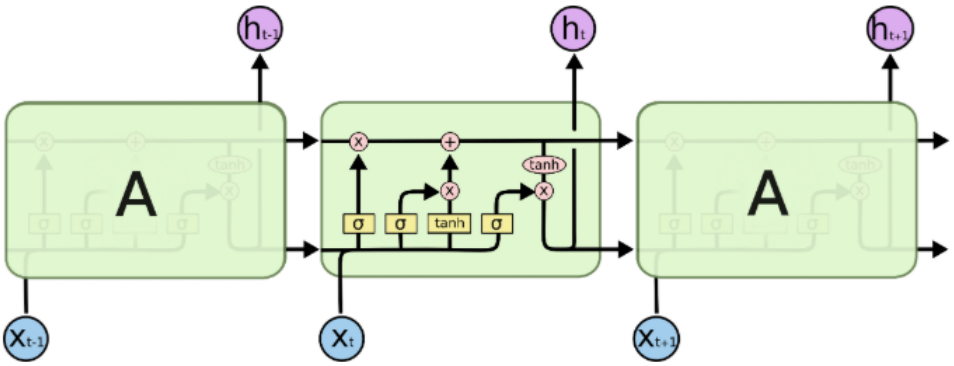
The idea is that the overall loss L is the sum of all the loss functions from times $t = 1$ to $t = T$, and since the loss at time t is dependent on the hidden units at all previous time steps, the gradient is as seen above. Note that its chain rule structure is still very similar to standard backpropagation used in FFNNs, with a primary difference coming from the impact of 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 $\frac{\partial h^t}{\partial h^k}$ 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). To counteract this, a common way to implement sequence models is by using gated RNNs and, for this project, we chose to use LSTM units.

3.3 The Long Short-Term Memory RNN Architecture

The idea of using gated RNNs, which includes the LSTM architecture, is that we are able to create paths through time that have derivatives that neither vanish nor explode and involve connection weights that may change at each time step. Gated RNNs are also automatically able to decide when to clear a hidden state (i.e. set it to 0), and a core idea of LSTMs is to introduce loops within themselves to produce paths where the gradient can flow for long durations of time, while the weight on this internal path loop is conditioned on context, rather than fixed as in the standard RNN. The weight of this path is controlled by another unit and thus the time scale can be changed based on the input sequence [12]. A diagram of a single LSTM network cell and how it interacts with the wider RNN can be seen in the image below, with the LSTM unit itself being the central part of the three green boxes below:



The central idea of this input is the horizontal line running through the top of the unit which allows the data to run along the unit relatively unchanged if required. The gates, represented in the above small yellow boxes within the central green box, allow the unit to let information in (we shall denote these as gates 1 to 4, where gate 1 is the leftmost yellow box and gate 4 is the rightmost box). These gates are there to protect and control the cell state [13].

Gate 1 is the 'forget gate', which decides which information to 'throw away' from the cell state and is a combination of h_{t-1} and x_t and outputs a number via the sigmoid function σ between 0 (which signifies to 'get rid of this completely') and 1 ('keep this completely'). This could see applicability with a word sequence (i.e. a sentence) where we wish forget older parts of a sequence in order to make a more accurate assessment of the next word of the sequence based on more recently occurred words. The output of this gate f^t is given as:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Gate 2 is known as the 'input gate', which decides the values we'll update within the cell in order to store new information in the cell state. This is used in conjunction with Gate 3, which is a 'tanh' gate that creates a vector of new candidate values that can be added to the state. The equations governing the outputs of each of these two parts are given as:

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C) \end{aligned}$$

These two are multiplied together to get the new information that we wish to store in the cell, which replaces the old information that we have lost via the 'forget gate'.

With the information we wish to discard having been forgotten and the new information that we wish to replace it with having been calculated, we then turn to modifying the old cell state C_{t-1} into the new cell state C_t. This is done by multiplying the previous cell state by the forget gate output to forget the things we decided earlier to forget, followed by adding the new proposed candidate values scaled by how much we wish to update each value, and is given by the equation that governs the new cell states information:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Now that we have the cell state obtained, we finally decide how much of this information to output from the cell (i.e. a filtered version of the cell). We then use

Gate 4 (the 'output gate') to decide which parts of the cell we shall output, which we multiply by the 'tanh' of the new cell state C_t (which makes sure the cell state outputs between -1 and 1). This ensures that we only output the parts we decided to output (e.g. in the case of the language model it allows one to only output information pertinent to verbs if that is what comes next in the sequence). The output of the output gate and of the cell itself is thus given as:

$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

In using LSTMs as part of our architecture, we enable the models to condition themselves on sequences with some parts forgotten within the cell and also being able to chose which parts of hidden states it wishes to output to the next layer and the next state of the same hidden layer. As a result, this architecture of the hidden units is much more conducive to modelling long-term relationships within a sequence and also being able to train via backpropagation with much reduced effects from the vanishing and exploding gradient problems.

3.4 Implementing an RNN using TensorFlow

With all that said, there still must be a practical way of implementing RNNs using LSTM units 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 that 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. Further justification can be found in the 'System Choices' chapter of the report.

A detailed guide on building an RNN using TensorFlow is beyond the scope of this report; however, it was felt worthwhile to outline some of the central elements of the models that are built in 'rnn.py' and how they relate to the concepts outlined above. While we shall give a detailed breakdown of the script itself in the 'Script Ecosystem Overview' chapter of the report, we now turn our attention to specific sections of code within 'rnn.py' that are particularly significant to the architectural makeup of the models:

- The input shape of the model is setup with a placeholder variable that sets the input size equal to (x, y, z), where x is the batch size (i.e. number of sequences per training batch), y is the sequence length (i.e. the number of frames of data that is 'pushed through' the model per sequence; generally either 60 for

raw measurement data or 10 for computed statistical values), and z is the dimensionality of the frames itself (e.g. 66 for joint angle data).

```
tf_x = tf.placeholder(tf.float32, shape=(self.batch_size, self.seq_len, self.features_length), name='tf_x')
```

The equivalent is also done for the y data, depending on the output type the model is training towards.

- We define the LSTM cells, with their size and number of cells (i.e. equivalent to the number of nodes per hidden layer and number of hidden layers, respectively, if we were using a 'traditional' RNN) in a single line of code: we define multiple *BasicLSTMCell* objects with a size set as *self.lstm_size* (a hyperparameter that we can tune), a dropout percentage given as *tf_keepprob*, and create multiple of these in a loop, with the number of these given as *self.num_layers*. These multiple cells (i.e. hidden layers of the model) are then used to create a *MultiRNNCell* object, which acts as a wrapper for all the hidden layers of the model:

```
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)])
```

- Finally, we set up the model architecture so that the input x held in the placeholder *tf_x* feeds into the 'cells' (i.e. the hidden layers), which modifies the initial state of the model throughout the application of the input sequence x to the hidden layers to give us an output *lstm_outputs* and a final state of the layers, *self.final_state*:

```
lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, tf_x, initial_state = self.initial_state)
```

- These steps are defined within the *build()* method for the 'RNN' class which is called upon by the constructor of the object at object creation. Hence, when we create an RNN object as...

```
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)
```

...it results in setting the attributes of the RNN object, including most of the hyperparameters that influence the architecture of the object, and calling the *build()* method of the object that uses many of these hyperparameters. Thus, the above statement sets up the computational graph that defines our RNN model which is now ready to have data inputted through it for the training process.

- To train the model, the method *train()* is called by the 'rnn' object, which results in splitting the training data components *x_train* and *y_train* into batches, whereupon each batch-pair (i.e. a batch of *x_train* with a batch of *y_train*) is placed into a dictionary that matches train components to batches (i.e. it would

match a batch of *x_train* to the *tf_x* placeholder variable described above, which ensures that the *x_train* batch is used as input to the model). Each of these dictionaries are then fed through via the *session.run()* method that specifies we are training the model (which hence calls upon the optimizer within *build()* to train the model) and takes in the dictionary to train the model on each batch:

```
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)
```

- A similar process is then used when we wish to test the model via the *predict()* method called by the 'rnn' object. Much like *train()*, it splits the *x_test* data into batches, which it adds to the 'feed' dictionary (setting the dropout probability to 0% this time via *tf_keepprob:0: 1.0* and the session to use the *test_state* of the model) and calls the *sess.run()* method to push this dictionary through the model to get the predictions made on the batch. We retrieve the predictions based on the output type we are working towards and adds the predictions obtained to the list of predicted values for the *x_test* input:

```
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" or choice == "indiv":
        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)
```

While there are, however, many other steps in the process of using the 'rnn.py' to build the models, these are some of the crucial steps where we applied knowledge of RNNs and their usefulness to sequence modelling to create a deep learning solution in TensorFlow. And with easy access to other libraries that make reading in data from '.csv' and '.xlsx' files and manipulating it as matrix data easy (e.g. via '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 sequences for training/testing, the evaluating of various metrics like mean squared error, and so on), we have all the resources needed to build RNN models using LSTM units using the applicable preprocessing steps to tailor it towards working with our data pipeline and produce results that can be observed and compared within experiment sets and model predictions sets.

Part II

System Preparation: Choices and Setup

Chapter 4

System Choices: Language, IDE, and Libraries

4.1 Python

As with most software-related projects, one of the primary choices that must be made is what programming language to implement the components of the system in, along with what development environment it is to be built using. Both of these have a large impact in the time and ease it will take to develop the system, as well as how optimal it will be running in its final variation. For the choice of programming language, we chose to use **Python (3.6)** to build all scripts from for the following reasons:

- It takes very few lines to implement many things when compared with other languages like Java; for example, the code below shows how a neural network can be implemented in Python in only 9 lines using the Keras library (a wrapper for TesorFlow):

```
num_extra_hidden_layers = 3
num_hidden_nodes = 30
ann = Sequential()

ann.add(Dense(units= num_hidden_nodes, activation= 'sigmoid', input_dim= num_dimensions))
for i in range(num_extra_hidden_layers):
    ann.add(Dense(units = num_hidden_nodes, activation= 'sigmoid'))
ann.add(Dense(units= 1, activation= 'sigmoid'))

ann.compile(optimizer= 'adam', loss = 'binary_crossentropy', metrics= ['accuracy'])
ann.fit(x= x_train, y= y_train, batch_size= 10, nb_epoch= 30)
```

  This enables faster development and easier testing of new ideas and concepts than other languages, as we can afford to care less about problems of tricky syntax (e.g. with using C++) and can instead move towards development 'at the speed of thought'.

- The open-source nature of Python's community means that there are very often libraries that others have built that fit the profile of what we need, so we don't need to 'reinvent the wheel' by creating our own version of it; this can be seen

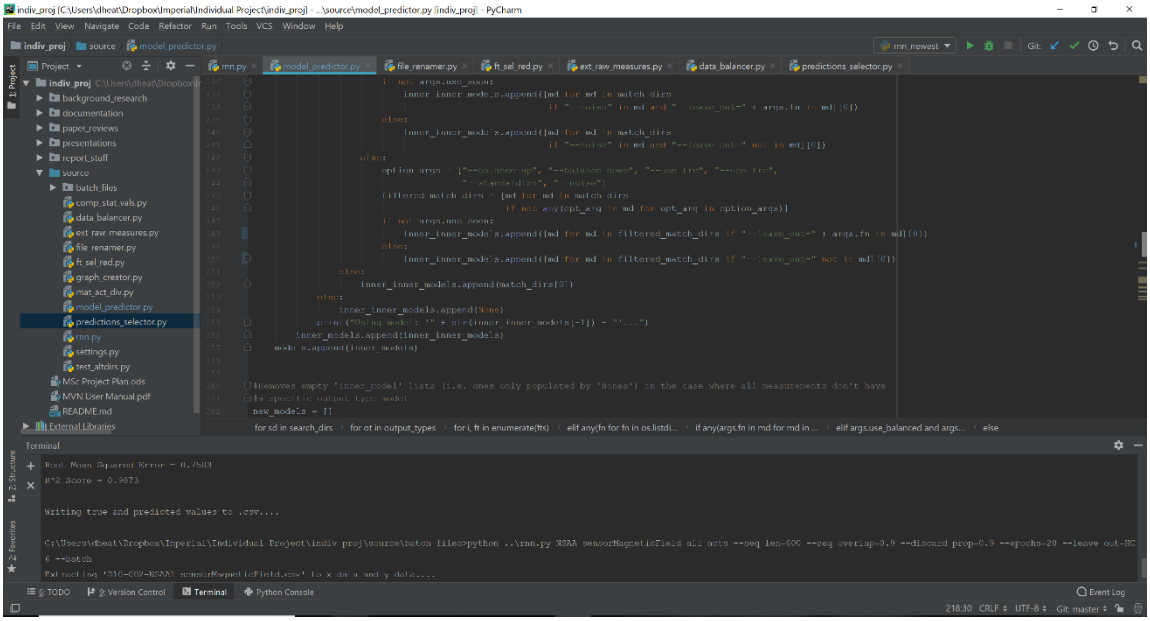
in the system's reliance on external functions from 'scikit-learn' to compute metrics, along with 'pandas' to handle the reading and writing to and from '.csv' files, as opposed to writing our own functions to carry out this functionality.

- Python has seen extensive use for building and testing machine learning and deep learning models by research and business communities; thus, it is easily the most well-developed with regards to open-source libraries such as TensorFlow, with TensorFlow's Python API being most complete of its various language implementations [14].

- Many of the frameworks and libraries that power the system we have built (e.g. NumPy and SciPy) build upon lower-layer Fortran and C implementations for fast and vectorized operations for multidimensional arrays, which helps overcome the inferior speed of a scripting language like Python when compared to these lower-level languages [15].

- Further development of the system is easier, as being written in a language like Python (which is close to English) lends itself to the easier understanding of what is going on within each script. Coupled with variables having intuitive names and commenting where necessary, this helps with anyone undertaking edits or rewrites to one or more of the scripts in the future.

4.2 Integrated Development Environment

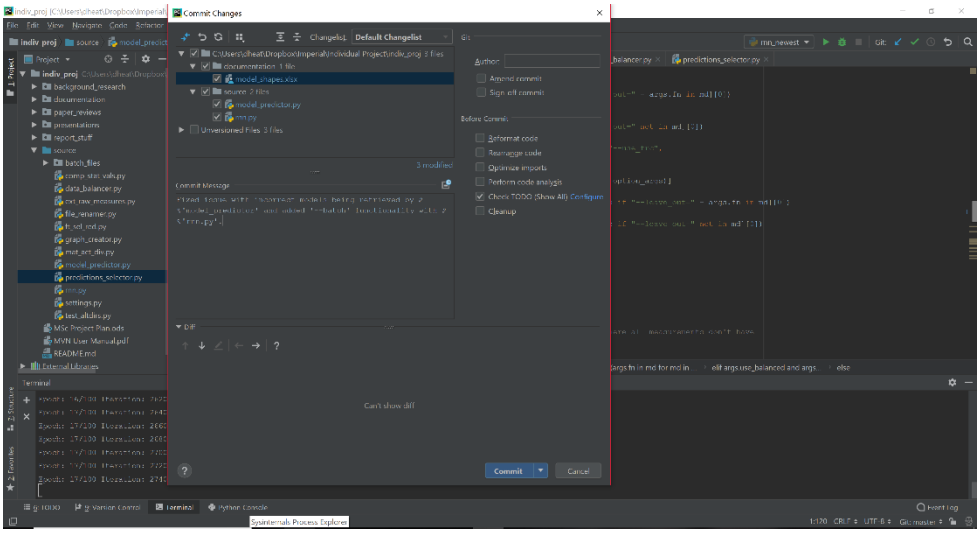


With regards to where we shall develop the Python programs for this project, we have chosen to use the **PyCharm Community Edition 2016** integrated development environment. We can see above a screenshot of how the IDE is setup for this project. This has been chosen for the following reasons over a text editor or another IDE:

- *Multiple program tabs*: This enables the easy transitions between different scripts. This is particularly useful when we are making changes to multiple scripts simultaneously (e.g. adding the same optional argument to both 'rnn.py' and 'mode_predictor.py' that ensures certain models built by 'rnn.py' are then retrieved correctly by 'model_predictor.py').

- *In-built terminal*: This allows us to run programs from command line within the IDE itself without having to open a separate terminal window and navigating to the script directory every time it's reopened. This is a major quality of life improvement, as most of the scripts are run from the command line with required and optional arguments.

- *Good compatibility with git*: This has two main benefits. The first is that changes made to scripts and their specific lines of change from a previous commit are highlighted in blue, which helps with accounting for modifications made when writing commit messages and keeping track of all recent changes made. The other benefit is the GUI approach to committing to the GitHub repo (as can be seen below) which is a much easier way to make regular commits and also highlights easier more subtle changes made (such as writing lines to output files).



- *Previous experience*: We've used it before for many previous Python projects, including in two professional roles, for a final-year undergraduate individual project, and for many pieces of coursework involving the use of several machine learning and deep learning libraries such as 'scikit-learn' and 'TensorFlow'. Hence, this previous experience and the resultant familiarity with the environment helps make development of this project a more expedient and easier experience.

- Debugging and error handling: The layout of PyCharm makes for writing and immediate testing and modifying of code very simple, with debug options

showing locations of compilation errors very easily and with clarity. This minimizes the time lost in development due to basic syntax errors and other basic software-engineering-related issues.

- Package implementation: It's easy to add additional packages via 'Available Packages' in the 'Project Interpreter', which is useful as we need to add many additional libraries, from TensorFlow to simple libraries like 'pyexcel'.

4.3 TensorFlow

For programming in Python, there are numerous options for which library we can use to implement our central RNN models. One option is the 'Keras' wrapper that wraps the TensorFlow framework. Although this is a lot simpler to use and has fewer aspects to manually code, there are numerous advantages that TensorFlow has over this that includes the following:

- More extensive and highly detailed documentation and examples for TensorFlow.
- Higher amount of direct control over the RNN models with things such as weights and optimizers.
- Better performance with TensorFlow through threading and queues to speed up the training process.
- Availability of the TensorBoard visualization tool to help understand our models.

Thus, using TensorFlow will hopefully lead to more successful RNN models that can better learn from raw measurements and computed statistical values that can thus better operate on newly-presented subject data. Preparatory work for using this framework includes prior use as the engine for an undergraduate individual project, along with reading Chapter 14 and 16 of [15] where we learned:

- The benefits of using TensorFlow for neural network training performance in utilizing GPU cores, where using a high-end GPU resulting in 15 times more floating-point calculations per second than using an equivalently-priced CPU.
- Concepts of graphs, sessions, ranks, tensors and operations, which gave clarity to the concepts of the computational graph structure used by TensorFlow.
- The 'placeholder' concept of TensorFlow where a variable is an 'empty' variable that expects data input (in our case, these 'placeholders' will be implemented for *x_train/x_test*).
- How aspects specific to an RNN works in TensorFlow, such as implementing layers as LSTM cells and initial/final states for the variables.

With this obtained knowledge from the above textbook, along with examination of other examples found primarily on GitHub or the TensorFlow documentation website, we felt confident enough in our knowledge of the TensorFlow library that, coupled with prepared input data, our RNN models could now be created.

Chapter 5

System Setup: Software and Data Preparation

5.1 Overview

A necessity to either continue further work on this system, validate the results we obtained in experimentation and elaborate upon in this report, or produce one's own results through system use is to setup the required components to run the system. Before being able to modify or run parts of the system, there are several things that must be setup beforehand. This includes:

- Obtaining relevant system resources.
- Downloading the requisite data sets and setting them up in the correct directories.
- Setting up Python and 'pip', along with the IDE and necessary packages.
- Running all the setup scripts that are run via the'setup.cmd' script.
- Setup of TensorFlow to use a GPU.

Once these steps are all completed, the editing of scripts, building of models, testing of files, and so on can be done by the user. In this part of the report, we shall be going through all the necessary steps to run the system on a different workstation; the hope is that, with the steps completed in this section, any user with the necessary computational resources can build models, reproduce results, and carry out additional experiments using the suit data captured that is used as part of the project.

5.2 Necessary System Resources

As this system works with large amounts of data and requires a heavy computational workload in order to build and test models, among other tasks, anyone running parts of this system requires a workstation setup with the necessary resources

to execute many of the scripts, store the data, and so on. The vast majority of the work done on this project has been undertaken on a 'Dell XPS 15 (9570)' laptop with the following specs:

- *CPU*: 'Intel Core i7-8750H'
- *RAM*: '16GB DDR4, 2666MHz'
- *GPU*: 'Nvidia GeForce GTX 1050Ti with 4GB GDDR5'
- *Storage*: '512GB SSD'

A system with similar specs should be adequate to run the system; however, the following is ideal:

- *CPU*: At least a 7th gen Intel i5 or i7 (or AMD equivalent). A lot of the data preparation, computing of statistical values, reading from and writing to '.csv's, and so on are done using the CPU (i.e. anything the system does that's not including the training, testing, and assessing of models); hence, a good CPU will enable the user to run these scripts in reasonable time.
- *RAM*: Minimum of 10GB needed; some of the larger datasets we look at (e.g. when multiple raw measurements' data are combined into one data set for a data shape of (16000, 60, 180)) require in excess of 8GB of memory just for the data, not including resources required for the IDE and other parts of the script being run. Any less than 10GB, therefore, may risk system instability or limit the user from carrying out certain parts of the experimentation outlined in this report. Additionally, DDR4 is recommended so as to increase the speed at which data is able to be written and read from memory.
- *GPU*: We make heavy use of TensorFlow running using the inbuilt GPU; the alternative would be to use the CPU, which by our estimation is approximately 10x slower to train models than using the GPU. Hence, to realistically build models in this system we need a good GPU. Ideally, the user would use an 'Nvidia' card as that is the easiest to setup with TensorFlow and, preferably, the card would have many CUDA cores (the 1050ti has 768) to enable faster parallel processing when training models.
- *Storage*: As of writing, the total storage required for all data sets contained within the 'local directory' (including '.mat' files for NSAA assessments, 6-minute walks, natural movement behaviour, and the intermediate data extracted from all files via the data pipeline) amounts to over 170GB, while the system itself contained within the 'project directory' requires approximately 725MB of space; hence at least this much storage is required, ideally on an SSD to enable fast read speed from storage (which will happen a lot during the setup scripts). See the chapter on the 'Project and Local Directories' for more information about what these specific directories contain.

5.3 Data Sets Setup

With a workstation obtained with the requisite resources, the next step is obtaining the data sets required by the system. There are two approaches that can be taken: the first involves being able to access the link shown below (which should be possible for anyone with Imperial College London credentials), which contains all the data used as part of this project (i.e. that also contains all the intermediate data constructed via the scripts that make up the data pipeline). Hence, a user that downloads all the data from this link would not need to run the Python scripts contained within the 'setup.cmd' script, only the pip installation commands. The second approach should be taken if the user either doesn't have Imperial College London credentials or otherwise can't access the files, or if one wishes to run the pipeline 'from scratch' (i.e. computing ones own intermediate data files via the data pipeline); this does require the user to fully run the 'setup.cmd' script. Note that the below explanations assume the user already has access to the complete 'project directory' if one is reading this report; however if not, consult the chapter on 'Project and Local Directories' for guidance on how to access this.

The first approach is as follows:

1. Setup a base directory in the user's storage directory; the default name for this that has been used thus far has been 'C:\msc_project_files\'; however, a more intuitive name can be chosen by the user if desired. This becomes the user's 'local directory' for the project.

2. Download each of the directories contained within the OneDrive link: `https://imperiallondon-my.sharepoint.com/:f:/g/personal/djh18_ic_ac_uk/Euymu00dXG1Cmm` `e=L5C62Z` and place each directory in the 'msc_project_files' directory (e.g. 'left-out', 'allmatfiles', etc.) within the user's created 'local directory'.

3. Modify the requisite line within '<project directory>\source\settings.py' to point to this new location. For example, if the user has setup the 'local directory' as 'example' in 'C:\', then they should modify the 'local_dir' variable (line 7) to now contain: 'local_dir="C:\example\". In doing this, it ensures that all other scripts that need to access the data files in 'example' are correctly pointed to it.

4. Once the steps outlined in the section below on Python, pip, PyCharm, and the necessary packages have been undertaken, open '<project directory>\source\batch\setup.cmd' for editing in a text editor, comment out line 19 and onward (as these create the intermediate data from the pipeline which we now already have), save the file, and run it to setup the Python packages.

The second approach is as follows:

1. Setup a base directory in the user's storage directory; the default name for this that has been used thus far has been 'C:\msc_project_files\'; however, a more

intuitive name can be chosen by the user if desired. This becomes the user's 'local directory' for the project.

2. Obtain permission from the owners of the data sets used as part of the KineDMD research initiative to access and download the directories.

3. Create another directory within the 'local directory' called 'output_files'. This shall contain a number of things produced by the scripts and by the models, including the '.csv' files of computed statistical values, the constructed models themselves, among other parts of the system.

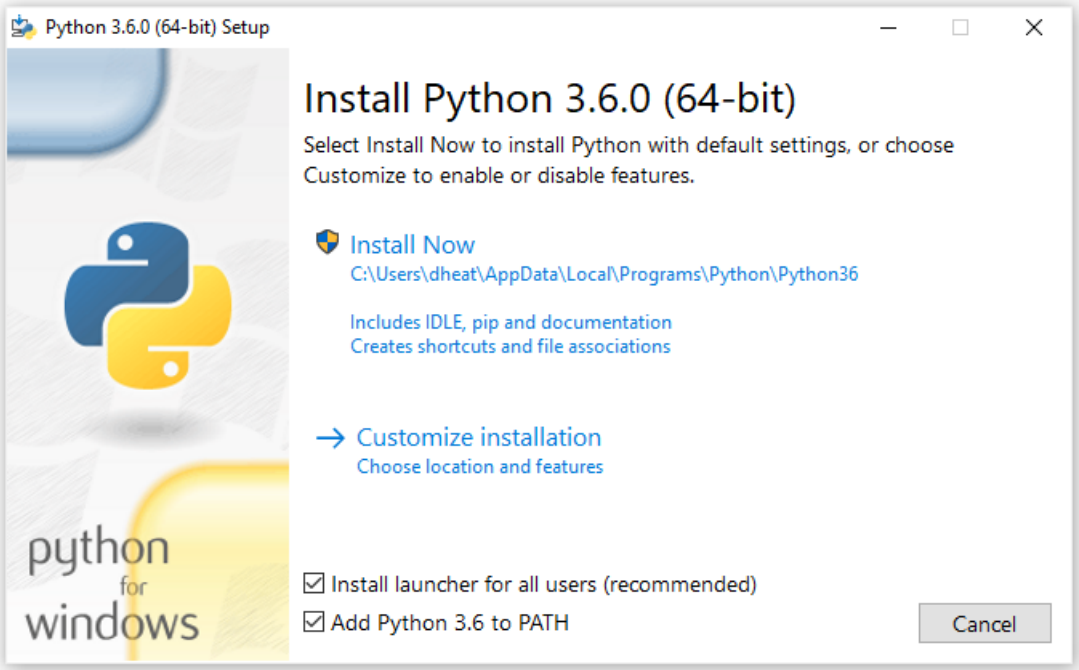
4. The user should have links to the following data sets (though if they don't the requisite permission for each must be obtained by the relevant parties): 'NSAA', 'NMB', 'allmatfiles', '6MW-matFiles', and '6minutewalk-matfiles'. Each of these directories should then be downloaded and directly placed within 'local_dir'.

5. Once the steps outlined in the section below on Python, pip, PyCharm, and the necessary packages have been undertaken, run the 'setup.cmd' in full to obtain the necessary Python packages for the project and compute the intermediate data used as part of the project.

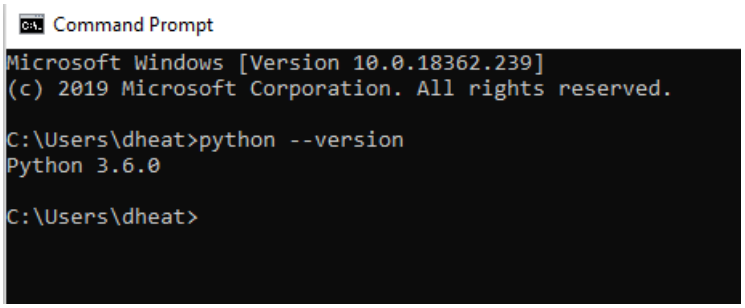
With these steps done, the data should now be in a form in which all the scripts that form the system should be able to access with the necessary directories constructed.

5.4 Setup of Python, Pip, Necessary Packages, and PyCharm

We now turn our attention to the setting up of the language, the 'pip' tool for package installation, and the libraries required to run the scripts. The first step is the installation of Python; the version this system runs on is '3.6.0' and, while installing any subsequent versions should be acceptable, we shall install this version to avoid any possible complications to do with the language further down the line. Version '3.6.0' can be downloaded from `https://www.python.org/downloads/release/python-360/` and by selecting the relevant installer from 'Files'. With the installation window open, ensure that the 'Add Python 3.6 to PATH' box is checked; this shall ensure that the user is able to run Python commands from the command line or terminal:



With this installed, we can ensure that Python is setup correctly with the required version by entering 'python –version' at the command line:



It should be noted that the installation window for the user should also say that it is installing 'pip'. This can be asserted to have been installed for the user by entering 'pip' at the command line, which should show output of something like this:

```
C:\Users\dheat>pip

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependencies.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  help                        Show help for commands.
```

If, for whatever reason, 'pip' has not been installed, follow the steps outlined at `https://pip.pypa.io/en/stable/installing/`, which includes the modification of the PATH environment variable to ensure that we are able to use 'pip' correctly.

Once 'pip' has been installed and/or asserted to be setup ready to use, we are able to install the required packages. Navigate to the '<project directory>\source\batch_files' directory and execute the 'setup.cmd' script. This will setup the required packages to run the project. Not only that, but it will ensure that the versions of each of the packages installed for this system will also be setup by the user; this should minimize the chances that the user will encounter dependency issues between packages (different versions of 'tensorflow' have been shown to interact badly with certain versions of 'numpy', for example). It should be noted that 'setup.cmd' will also run all the Python scripts within the 'source' directory that are used to setup the files for other scripts such as 'rnn.py' and 'model_predictor.py', so if the user doesn't wish to run these at this time, the recommended way is to comment out each of these lines. This can be done by commenting out each of these setup lines (line 19 and onward) by inserting 'REM ' at the beginning of each line. Note that this means that 'setup.cmd' must be run later when the user wishes to prepare the data for the system and if the user doesn't already have the intermediate data.

5.5 Installation of PyCharm (IDE)

If the user is intending to do any long-term modifications to the system, or if they simply want a more convenient place to launch the scripts from, then it is recommended that they install an IDE for the project; specifically, PyCharm Community Edition. Using this provides several advantages to the user:

- An in-built terminal to run the scripts of the system with the necessary arguments.
- Easy interaction with Git to work from the project code in GitHub/GitLab.
- Syntax assistance when editing any files.
- Multiple tabs to help with editing multiple files simultaneously along with the script variable explorer.

To setup the IDE, the following steps should be undertaken:

1. Download the community edition of PyCharm, the link for which can be found at: `https://www.jetbrains.com/pycharm/download/#section=windows`.
2. In the 'Installation Options' window, it's recommended that the user selects the 'Add "Open Folder as Project"' option (to allow opening the 'project directory' as a PyCharm project) and the '.py' association (so Python files open in PyCharm as default).

3. Launch PyCharm and select 'Do not import settings'.

4. From the 'Customize PyCharm' window, click 'Skip Remaining and Set Defaults'.

5. Prior to continuing with PyCharm setup, we first must setup Git if the user doesn't have it already. The following section applies to Windows, but the equivalent can easily be done for iOS or Linux.

   (a) Download Git from `https://git-scm.com/download/win`.

   (b) Run the installation, making note of where Git is installed, with the default settings.

   With this now completed (or not, depending on whether Git is already installed), the user should launch PyCharm and in the 'Welcome to PyCharm' window, click 'Configure' and 'Settings', followed by navigating to 'Version Control' and Git. In the 'Path to Git executable', enter the location of the 'git.exe' program that was just installed (or previously installed). This can be found within the 'Git\bin\' directory within the location where Git was setup. Click 'Apply' and 'OK'.

6. In the 'Welcome to PyCharm' window, select 'Check out from Version Control' and Git. This is because we will be directly installing the 'project directory' directly from 'GitHub'. Note that it's recommended doing this even if the source directory has already been downloaded and setup elsewhere, as doing it this way ensures that it is easily to modify and commit to git if any changes to the scripts are to be made. Within this new window, enter the URL: `https://github.com/dan-heaton/MSc_indiv_project` within the URL window and click 'Clone' to clone the repository. Alternatively, if one wishes to clone from GitLab instead, swap the URL above with: `https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project`.

7. The final step is to associate PyCharm with the Python interpreter we have previously installed. To do so, with the project opened in PyCharm, navigate to 'File' and 'Settings'. Under 'Project: MSc_indiv_project' and 'Project Interpreter', click the settings icon and 'Add...'. Under 'System Interpreter' the Python executable should already be detected within 'Python36\' as 'python.exe'. However, modify this path to the 'python.exe' file if has not done so already. Click 'OK', 'Apply', and 'OK'. In the main PyCharm window, it may take a few seconds to configure this interpreter but, once done, the user will be able to edit and run the scripts of the system within PyCharm.

5.6 Configuring of TensorFlow to Use the GPU

The following section works on the assumption that the user is working on a workstation containing a GPU. This is more or less a necessity to build models using

'rnn.py': while a reasonable GPU with >700 CUDA cores builds a typical model in 5-15 minutes, building these using an equivalently-priced CPU would take 1-3 hours. While the necessary steps to undertake the complete setup of a GPU are somewhat arduous, there exists a useful guide to doing so at `https://www.tensorflow.org/install/gpu`, along with a CUDA installation guide at `https://docs.nvidia.com/cuda/` for multiple OS's. This includes details on setting up the CUDA toolkit and the CuDNN (CUDA Deep neural Network library), which are required to run TensorFlow on the GPU. With this setup, TensorFlow with subsequently run as default in all scripts using the GPU to create models as opposed to using the CPU.

Part III

System Overview and Explanation

Chapter 6

Project and Local Directories: Overview and Explanations

6.1 Background

Prior to undertaking an in-depth discussion of the individual scripts, their uses, and the various experiment and model prediction sets we undertook as part of this project, it's worth giving a brief explanation of the two types of directories that we are concerned with for this project. There are two directories that we are concerned with: the 'project directory' (containing the source files, the documentation used and written to, among other things) and the 'local directory' (containing the source '.mat' files that we use as inputs to scripts and some of the outputs of the models, including the models themselves). Below, we cover each of the two in turn, how to access and/or set them up, and so on. The aim is thus to educate any users on how the project is laid out and how each part of the system connects to each other.

6.2 The Project Directory

The project directory is the directory containing all of the source code, the information required about the NSAA subjects ('nsaa_6mw_info.xlsx'), the documents containing the results of the experiment sets and model prediction sets, and other reports and presentations constructed throughout the duration of the project. Hence, this is where the majority of the deliverables of the project lie, with the exception of the majority of the models that were created throughout the project and the data that is required to train the models. The reason we keep these apart (i.e. why project directory and local directory are not synonymous) is as follows:

- The local directory requires >170GB of storage for all the data sets that is used in the project. However, we've been making extensive use of storing the project directory within DropBox and as a Git repository and, as it would be not possible and very impractical, respectively, to store the local directory on both DropBox and within GitHub or GitLab, we chose to keep these separate.

- When we run several of the data pipeline scripts (e.g. 'comp_stat_vals.py' or 'ext_raw_measures.py') we end up producing a lot of new files within the local directory). Therefore, to avoid constant DropBox synchronizations or many new or deleted files to appear in each Git commit, it was felt that it would be easier to simply keep the data aspects apart from the source code and other documentation.
- Permission may be required to deal with certain data directories contained within the local directory, as this contains data about ongoing subjects of a research initiative that is not freely available. Hence, it was the desire to keep the project directory available to whoever wished to access it, without predicating access on also being able to access the data required to populate the local directory (e.g. if one wished simply to browse or borrow ideas from the scripts within the project directory); this also enables and encourages an easier transition to applying the project to other data sets possibly for other domains.

To access the complete project, the advisable way to obtain it would be to clone it via GitHub. The repo can be accessed at `https://github.com/dan-heaton/MSc_indiv_project` and cloned via `https://github.com/dan-heaton/MSc_indiv_project.git`. Alternatively, one can access it via GitLab at `https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project` and clone with `https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project.git`. The current name for the project directory as used in its local form for development has been 'indiv_proj; however, one could rename this freely without requiring any other changes to the scripts. However, it's recommended that the directories without the project directory should not be changed (e.g. 'source' or 'documentation'), as doing so would require rewriting of several of the scripts that hardcoded paths to access things outside of its own directory.
Broadly speaking, the directories can be broken down as the following:

- **background_research**: this folder contains some preparatory programs that were written (with help from sources cited in the scripts themselves) to familiarize oneself with the building of RNNs with the chosen libraries in order to make using them for the actual project later that much easier; hence, these aren't used by the rest of the project at all and are included for historical documentation reasons only.
- **documentation**: contains a number of files pertaining to the outputs of the data pipeline for the project. This includes the following:
  - *'RNN Results.xlsx'*, which covers the performances of various model setups on test data (i.e. a large proportion of the experimentation that covers different types of raw measurements, sequence lengths, overlaps, etc., source their results from here). These are what form the basis of our later discussion on the experiment sets.
  - *'model_predictions.csv'*, which (unlike 'RNN Results.xlsx') shows the performance of using 'model_predictor.py' to assess the performance of pretrained models on whole data files. These provide the information needed for the model predictions sets, as discussed later.

  - *'model_shapes.xlsx'*, which is just to be used by 'model_predictor.py' to set the sequence length to the correct value (and is not particularly relevant to the user).
  - *'nsaa_6mw_info.xlsx'*, which contains a table of the subject names and their corresponding single-act and overall NSAA scores (this provides the necessary *y_labels* for many RNN models).
  - *'nsaa_17subtasks_matfiles.csv'*, which is the Google annotations sheet that was collaboratively created by others within the wider research initiative that contains the file names and times within said files where the 17 NSAA activities are performed by each of subjects. This is determined by watching the source videos of the '.mov' files of the subjects performing the activities and recording at what times in the video these activities occur, along with making use of the 'dis_3d_pos.py' script; this sheet is then used by 'mat_act_div.py' to create the single-act files used in later model predictions sets.
    We also have several other subdirectories in 'documentation':
  - **Graphs**: all graphs created by 'graph_creator.py' are placed in here. These source from 'RNN Results.xlsx' and 'model_predictions.csv' to create graphs that are easier to display the results of groups of experiments done than it would be to display the same information using a table. We see many of these graphs later on within the discussion of the experiment sets.
  - **Script Explanations**: collection of 'READMEs' for each of the scripts within 'project_files\source'. The idea is that, if one wishes to find out what each script does, why it was written, etc., then reading its relevant 'README' should provide sufficient detail. Much of these READMEs form the basis of our script overview later on.

- **paper_reviews**: contains paper reviews done of research papers that are believed to be relevant to the project. These predominantly focus on the use of RNNs when applied to real-world human movement data, and each paper consists of a slightly-shortened bullet-pointed version of the paper and then a section of the most significant points from these bullet points. Hence, these papers are useful in justifying decisions taken with respect to model choices, experimentation directions, etc.

- **presentations**: contains a collection of presentations that have been created to display to group members about the project's progress thus far (which were kept in order to aid in final report writings).

- **report_stuff**: contains several initial reports and other documentations of project progress thus far, and also 'MSc Project Plan.ods', which is where the already-completed and upcoming task lists are stored; this is particularly useful if one wishes to see what is currently being worked on or has recently been completed. The vast majority of the contents of this directory, however, are contained within this report.

- **source**: contains all of the scripts that are needed by the project pipeline to run. This includes the core Python scripts (such as 'comp_stat_vals.py' and 'rnn.py'), along with some 'supporting' scripts, such as 'settings.py' (to contain global variables that are used across several scripts). For information about how to run each of these scripts, run the script of interest through the command line/terminal with the '–help' optional argument set (e.g. 'python comp_stat_vals.py –help'). This will display each of the arguments that are available to be set, the significance of each, how they interact with other arguments (if relevant), etc.

  - **Batch files**: within this, we contain the batch scripts that are used to automate some of the running of the scripts. For further info about the significance of any or all of the scripts, consult the 'README(s)' for the relevant scripts in '<project directory>\documentation\Script Explanations\', the script ecosystem overview in 'plans_and_presentations', or the diagram of the scripts and their connections to each other found at the beginning of the 'Script Ecosystem Overview' chapter. Along with some of the simpler automation of the tasks, we also run each of the model predictions sets from their respective batch files, as many of them require building many models and testing many different combinations of files, which require many runs of 'rnn.py' and 'model_predictor.py'; hence, the automation of this makes the process of replication hopefully much easier for the user.

6.3 The Local Directory

The local directory is considered to have two purposes: to store the data sets that we make use of in this project, and to store the direct outputs of the 'rnn.py' scripts that include the models themselves and the '.csv' output predictions that are written directly on a sequence by sequence basis (e.g. for a model created with 'rnn.py' using a test set of 1000 sequences, there will exist within this '.csv' each 1000 predicted value and true value, depending on the output type set by the user). There are three reasons why we include this 'rnn.py' output within the local directory:

- As we create many models as part of this project, the size of this directory has become an issue and thus we would prefer to keep it separate from the project directory for space reasons. We therefore want to keep a lot of this data 'clutter' apart from the what is considered the 'core' of the project with the project directory.

- We also want to keep a consistent philosophy with which we consider to be 'intermediate data', which is data that exists as a product of one script and that is used by other scripts: in this case, the models produced are intermediate data in that they are created by 'rnn.py' and used by 'model_predictor.py'. This holds for other forms of intermediate data such as data produced by

'comp_stat_vals.py' and 'ext_raw_measures.py', and so we wish to do the same for the models and 'rnn.py' predictions output.

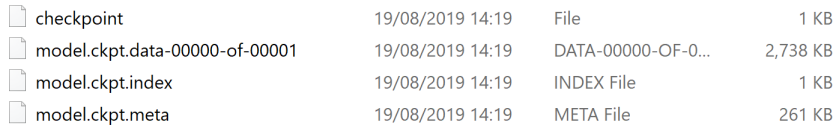
- The majority of these models are only used once as part of one particular experiment set or model predictions set, and therefore they don't form a part of the 'complete' system (with the exception of the final selected models that are contained within the 'source' directory, though this is only a small number of the overall number of created models). Thus, we keep these models separate from the ones constituting the completed system at the end of all experimentation; in other words, the models that are intended to be used by a user to do assessments with specific subjects are contained within the project directory, while the models created during all experimentation are contained within the local directory.

To access the complete local directory of files that we have been used for this project, use the OneDrive link given as `https://imperiallondon-my.sharepoint.com/:f:/g/personal/djh18_ic_ac_uk/Euymu00dXG1Cmmeoz3xxq24BekH57ZuDmU9uZtoSr60xfg?e=L5C62Z` where one can find a directory given as 'msc_project_files'. This is the local directory as used during the development of the project. Download and store it while modifying the local variable in 'settings.py' (as directed in the 'System Setup' chapter). Note that the total directory is in excess of 170GB, so sufficient space may need to be made for it beforehand.

6.3.1 The Local Directory: 'rnn.py' Outputs

We first look at the two sub-directories within the local directory containing the outputs of 'rnn.py':

- **output_files\rnn_models**: this contains all models that have been produced by 'rnn.py' throughout the course of the project, including the final models used that are contained within the project directory. Each model's contents are the product of the TensorFlow library working through 'rnn.py' and each model consists of a directory that looks something like this:



These files constitute a single model in the eyes of 'model_predictor.py', which is the only script that makes used of these models. For instance, within these files contains the model input and output shapes, the numbers of hidden layers, other hyperparameter settings, and the weights of each of the neuron connections. In other words, it's a fully trained model that is ready to be used by 'model_predictor.py' to be used on a whole subject's data file.

The names of the models may seem unnecessarily long and complex, but they

are in fact simply the exact arguments used to invoke the instance of 'rnn.py' that creates this specific model, excluding the always-necessary 'python rnn.py' parts of the argument sequence. There are two reasons why we do this:

  - This creates the names of the directories automatically, which takes much of the human-element of labelling each directory out of the process based on what experiment set or model prediction set it is used for.
  - In having them written by name based on the 'rnn.py' arguments, we can ensure that 'model_predictor.py' will always use the correct models during experimentation by writing a simple set of rules within 'model_predictor.py'. For instance, if we want 'model_predictor.py' to use models that have been created with added Gaussian noise, we can ensure that it uses only model directories that contain '–noise' within their names. Hence, it's an easier way to determine the correct models to use rather than analysing the contents of the model directory (e.g. which would mean looking into the 'model.ckpt.meta' file) to determine if it's a model we need to use.

- **output_files\RNN_outputs**: each model that is built by 'rnn.py' that is also tested on a test partition of data (i.e. if the '–no_testset' argument is not set) writes a separate '.csv' file to this directory. Each file contains, for a given created model, the arguments used to invoke it (in the form of the '.csv' file name), the 'overall' results of the test set on the model (e.g. MAE of data for the 'overall' output type, accuracy of data for the 'dhc' output type, etc.), the individual predictions made for each test sequence versus their true values, and the model hyperparameter settings. An example of this can be seen below:

Sequence Number	Predictions	Trues	Results	Settings
1	33.96	34	NSAA,allmatfiles jointAngle all overall --seq_len=600 --seq_overlap=0.9	X shape = (25533, 60, 66)
2	34.03	34	Mean Squared Error = 0.4382, Mean Absolute Error = 0.3014, Root M	Y shape = (25533,)
3	33.97	34		Test ratio = 0.2
4	2.61	3		Sequence length = 60
5	34.11	34		Features length = 66
6	2.93	3		Num epochs = 20
7	25.4	26		Num LSTM units per layer = 128
8	34.09	34		Num hidden layers = 2
9	33.77	33		Learning rate = 0.001
10	34.1	34		
11	34	34		
12	27.9	28		
13	27.27	26		
14	34.05	34		

The contents of this file are entirely optional to use, however, as all the requisite information about the test set performance (that is to be inputted into 'RNN Results.xlsx' and then used as part of experiment sets) are also produced as console output, which is easier to copy over for the user. However, this files serve as a log of model performance through time as we continue to create more models if we wish to use them as a reference at any point.

6.3.2 The Local Directory: Data Sets

With the directories containing model outputs having been covered, we shall now look at the directories containing the raw data sets, what is contained within each directory, and what 'type' of data these directories contain. It should be noted that

the 'source' scripts assume that each of these directories are a constant (i.e. that any other users don't modify the names of the directories); any changes made to the names here require modifications to the necessary variables within 'settings.py' (e.g. the 'sub_dirs' variable).

Prior to looking at each data set in turn, it's preferable to briefly discuss the general locations of raw source data (e.g. as source '.mat' files) and intermediate data. This becomes particularly relevant when we shall shortly be discussing the contents of each data directory, and knowing what produced intermediate data and where is conducive towards understanding the data pipeline. Below, we can see how each script within the data pipeline produces data and where the data is stored:

1. **'comp_stat_vals.py'**: This script takes the data stored as source '.mat' files from within the data directories within the local directory and outputs data to the corresponding path within 'output'. For example, if 'comp_stat_vals.py' intends to operate on 'NSAA' (based on the 'dir' argument passed to it), it sources its data from '<local directory>\NSAA\matfiles' as '.mat' files and produce the computed statistical values in '<local directory>\output_files\NSAA\AD' as '.csv' files. This functions in the same way for other source data sets (e.g.' 6minwalk-matfiles') where the path to the computed statistical value files is more-or-less the same as the source directory path with 'output_files' appended after the path to the local directory.

2. **'ext_raw_measures.py'**: Unlike computed statistical values, the produced raw measurement files are instead stored within subdirectories of the source data set directory that they are sourced from, as opposed to a subdirectory of 'output_files'. For example, if 'ext_raw_measures.py' intends to extract the raw measurements from the '6minwalk-matfiles' directory, it retrieves files from '<local directory>\6minwalk-matfiles\all_data_mat_files' and writes each measurement extracted per file to a sub-directory with a name matching the measurement name (e.g. 'D4's joint angle data is written to '<local directory>\6minwalk-matfiles\all_data_mat_files\jointAngle' as 'D4-6MinWalk-jointAngle.csv' while its position data is written to '<local directory>\6minwalk-matfiles\all_data_mat_files\position as 'D4-6MinWalk-position.csv', and so on).

3. **'mat_act_div.py'**: It should be noted first that, as this script extracts single-act files from complete-act files, it therefore only expects to be used on the 'NSAA' directory, as only files from 'NSAA' contain NSAA activities. When activities are divided, they are placed either within 'act_files' or 'act_files_concat' (depending on whether '–single_act_concat' was set for the script) within the 'NSAA' directory itself, much like 'ext_raw_measures.py'. For example, 'mat_act_div.py' would pull source '.mat' files from '<local directory>\NSAA\matfiles' and write single-act files (non-concatenated) to '<local directory>\NSAA\matfiles\act_files' as source '.mat' files but only containing single-activities within each. Note that if 'ext_raw_measures.py' then operates on these single-act files, they get written to a subdirectory within here, much like how it would operate on complete-act files; hence, drawing the joint angle files from the above case would write

the files to '<local directory>\NSAA\matfiles\act_files\jointAngle'. Similarly, when we compute the statistical values of single-act files, they get written to the 'output_files' directory; in the above case, the computed statistical values of the single-act NSAA files would be written to '<local dir>\output_files\NSAA\AD\act_files'.

4. **'ft_sel_red.py'**: To simplify the process of storing the feature-reduced variants of the '.csv' outputs of 'comp_stat_vals.py', we decided to store the files in the exact same directory as the files they operate on. For example, if we wish to reduce the dimensions of the computed statistical value files for the 'NMB' source data directory, we would write the feature-reduced files to '<local directory>\output_files\NMB\AD', where the computed statistical values are already stored. The difference here, however, is that the feature-reduced equivalents will have 'FR_' appended to the front of each of the files. For example, the computed statistical values for the 'D4' subject whose data is from 'NMB' will be stored within the above path as 'AD_D4_stats_features.csv', and when 'ft_sel_red.py' operates on this subject, it will take the data from this file and write to a file stored in the above path as 'FR_AD_D4_stats_features.csv' (alternatively, if the feature-reduced-concatenation option is set within 'ft_sel_red.py' it will instead be stored as 'FRC_AD_D4_stats_features.csv'). This naming convention ensures that 'rnn.py' fetches the feature reduced variants of files to ensure that models of input nodes size ¿4000 isn't required.

5. **'rnn.py'**: Although this is considered part of the data pipeline, this simply fetches the data from all of the above locations dependent on the arguments given, while we have already discussed the output locations of 'rnn.py' in the previous section.

With the relationship between the data pipeline scripts and the data set directories having been established, we now move on to examining the data that's contained within each of these directories. The data directories are as follows:

- **6minwalk-matfiles**: This contains the 6-minute walk data of many (but not all) of the subjects within two sub-directories within this directory: 'all_data_mat_files' and 'joint_angles_only_matfiles'. The former contains the source '.mat' files for all available measurements for the subjects' walking assessments, while the latter contains source '.mat' files with only joint angles. Therefore, what we consider 'JA' and 'DC' files ('Joint Angle' and 'Data Cube', respectively) that are referenced in experiment set 1 comes from 'joint_angles_only_matfiles', while in most other cases when we use the data of 6-minute walk assessments, we use the 'all_data_mat_files' directory.

- **6MW-matFiles**: This directory also contains some of the 6-minute walk assessment files for some of the subjects and, while some of the files overlap with '6minwalk-matfiles', it also contains assessment data that is not found in the other directory. Additionally, we don't see any 'joint angle only' data within this data set, and so all source '.mat' files are stored directly within this directory.

- **allmatfiles**: This contains the natural movement behaviour as source '.mat' files that contains only the joint angle data (much like '6minwalk-matfiles\joint_angles_only_ma files). As we only had the natural movement in 'joint angle only' form for quite a while (until we were given the 'NMB' data set), we had to make use of this for several of the later model predictions sets, though we later received the natural movement behaviour in true 'AD' form (i.e. containing all the measurements data for each subject) as 'NMB'.

- **left-out**: This is a small sample of data files that have been excluded from the main data sets that can act as data that is left-out of any of the data sets used to train models. It should be noted the difference between assessing using 'model_predictor.py' on files from left-out as opposed to 'left-out' subjects (as used in many model predictions sets): assessing a file from the 'left-out' directory will assess a model that is familiar with the subject (through having been trained on other files of the same subject), while assessing a complete left-out subject will assess a model that is not familiar with any files for the specific subject.

- **NMB**: This is the complete 'AD' data for the natural movement behaviour, as opposed to 'allmatfiles' which only contains the joint angle data in source '.mat' file form. As a result of receiving this data late in the project lifecycle, we are only able to use this data set in later model predictions sets. It should be noted the sheer number of files per subject: many of the subjects have up to 30 files captured from each of them that will have captured a variety of 'natural' activities, such as sitting, playing, eating, and so on. The data contained within these files, therefore, is much more 'unstructured' than either the 6-minute walk or NSAA assessment data.

- **NSAA**: This contains the data for each of the subjects' full NSAA assessments, and the source '.mat' files specifically are contained within the 'NSAA\matfiles' subdirectory. It should be noted that, for some subjects, their assessments are split over several files. We account for this when extracting raw measurements and computing statistical values by simply concatenating these source files with respect to time, while we select the correct file to use to get the single-act files via 'mat_act_div.py' by referencing the file name columns within the Google annotations sheet.

Chapter 7

Reference Documents Explanation

7.1 Background

As we can see from the system overview diagram shown at the beginning of the 'Script Ecosystem Overview' chapter, as well as having many Python and batch scripts that play integral roles within the system we also make heavy use of several documents that provide much needed information for training models and serve as places to store the results of experiments. These include the Google sheet of single-act annotations, the reference document containing the overall and individual scores of each subject used in the project, the 'model_predictions.csv' file, and the 'RNN Results.xlsx' file. Each of these files in turn can be found within the project source directory within the '<project directory>\documentation' directory. In the section below, we will discuss file each in turn, from the information they contain to how and where they are used within the system (i.e. what other scripts depend on them and what scripts feed into the documents). The aim here, therefore, is to give a more concrete understanding of what these documents contain prior to seeing them referenced in the results discussion and general system overview sections.

7.2 Google Annotations Sheet ('nsaa_17subtasks_matfiles.csv')

Step up R					Step down R			
filename	start	finish	completed	notes	filename	start	finish	completed
D2-010-NSAA	12981	13281	yes		D2-010-NSAA	13881	14181	yes
D3-006-NSAA	8404	8846	yes		D3-006-NSAA	9096	9256	special
d4-004-nsaa1a	17236	17304	yes		d4-004-nsaa1a	17048	17124	yes
d5-007-nsaa1	7217	7414	yes		d5-007-nsaa1	8000	8244	yes
D6-003-NSAA	-1	-1	no		D6-003-NSAA	-1	-1	no
D7-002-NSAA	8897	9007	special	was holding rails; actual steps	D7-002-NSAA	9655	9767	special
D9-002-NSAA	7098	7184	yes		D9-002-NSAA	7299	7386	yes
D10-002-NSAA1	1776	1776	no	can't do it	D10-002-NSAA1	1776	1776	no
D11-012-NSAA	14100	14220	yes		D11-012-NSAA	14520	14640	yes
D12-003-NSAA	7800	8520	yes		D12-003-NSAA	8520	8760	yes
D14-002-NSAA	8520	8640	yes		D14-002-NSAA	8820	8940	yes
D15-007-NSAAX	19570	19656	yes		D15-007-NSAAX	19777	19930	yes
D17-002-NSAA1	4680	4800	yes		D17-002-NSAA1	4860	5040	yes
D18-001-NSAA1	10908	11196	yes		D18-001-NSAA1	11770	11848	yes
D19-002-NSAA1	4740	4920	yes		D19-002-NSAA1	5160	5280	yes
D20-001-NSAA1	11283	11403	yes		D20-001-NSAA1	11523	11643	yes
D4V2-004-NSAA	12427	13182	special	help given; 12625 end frame without help	D4V2-004-NSAA	13369	13444	yes
D5V2-010-NSAA1	7900	8000	yes		D5V2-010-NSAA1	8000	8100	yes
HC1-004-NSAAAttempt2-NaturalBehaviour	5476	5569	yes		HC1-004-NSAAAttempt2-NaturalBehaviour	5570	5671	yes
HC2-007-PUL-NSAA	77765	77848	yes		HC2-007-PUL-NSAA	77845	77921	yes
HC03-004-NSAA	7798	7891	yes		HC03-004-NSAA	7892	7967	yes
HC4-013-NSAA	8222	8342	yes		HC4-013-NSAA	8462	8582	yes
HC5-009-nsaa	0	0	yes		HC5-009-nsaa	0	0	yes
HC6-004-NSAA	0	0	yes		HC6-004-NSAA	0	0	yes
HC7I-008-NSAA	7030	7116	yes		HC7I-008-NSAA	7237	7303	yes
HC9-003-NSAA1	5961	6351	special	tries to balance on step with one leg rather than step on to it with both legs	HC9-003-NSAA1	6489	6685	special
HC10-15-NSAA	2936	3038	yes		HC10-15-NSAA	3290	3506	special
HC11-004-NSAA1	5258	5348	yes		HC11-004-NSAA1	5358	5452	yes
HC12-002-NSA1	18917	19132	yes		HC12-002-NSA1	19561	19684	yes
HC13-001-NSA1	19263	19350	yes	not done as per the protocol; left foot doesnâ€™t rest on the step	HC13-001-NSA1	19445	19512	yes

As part of the wider research initiative, we collaboratively undertook to analyse

and record the times of each activity undertaken by the subjects as part of the initiative. The information that was collected into this document, therefore, was intended to be used as part of several projects that relied on the start and end times of each activity undertaken by the subjects. As a result, the subjects' corresponding activity videos were divided into three parts and each of us determined the activity times for our given subjects.

The process to undertake these annotations was as follows:

1. Load either the source video that corresponds to the '.mat' file (which is provided in the data sets as '.mov' files) OR use the corresponding '.mat' file with the 'dis_3d_pos.py' script to load a basic 3D dynamically updating image through the 'matplotlib' library (more on this in the 'Script Ecosystem Overview' chapter).

2. For each activity of the NSAA assessment set (e.g. 'raise from floor', 'run', 'step up using right foot', etc.), observe the time in seconds (or frames) the activity starts and finishes in the file. Note that we only count the **first completed** activity within the source file and we try to be slightly accommodating of the start time (for example, if the activity started between 3s and 4s in the file, we record it as having started at 3s to ensure we capture the complete activity with a bit of 'slack').

3. For the subject in question and for the activity in question, record the start and end times in **frames** in the 'start' and 'end' columns for the activity (note that if the time was observed in seconds, simply multiply this by 60 as the suit samples at 60Hz), along with recording the name of the file that the activity occurred in (as this will be the same name as the corresponding '.mat' file the suit data for this video will be in, just with a '.mat' file extension instead of '.mov').

In the image above, we can see a snapshot of several activities that have been recorded for some of the subjects, including the file names containing the activity in question for the subjects, along with the start and end times within the respective files. It should be noted that not every activity could be drawn from each of the subjects' files. This could be due to the subject simply not performing the activity they were told to perform or were otherwise unable to perform the activity. In these cases, the start and end times will be marked with either a '-1' or '0', with a '0' sometimes signifying an incomplete activity annotation done by the annotator.

For our project, the main use of this document is as a tool for the 'mat_act_div.py' script. For each of the source files that the script wishes to divide up, it will look in the table for the name of the subject in question for that source file, find the row corresponding to that file and, for each activity, get the start and end times for that activity and extract the corresponding frames from the source file (making sure its name matches that of the activity's 'filename' column entry). So while these single-act '.mat' files will be further processed by other scripts (such as 'comp_stat_vals.py'

and 'ext_raw_measures.py'), 'mat_act_div.py' is the only file that directly relies on the Google annotations sheet, and no file within the system modifies it in any way.

7.3 NSAA Scores Reference Document ('nsaa_6mw_info.xlsx')

Class	ID	Visit	6MWDiste	NSAA	Act 1	Act 2	Act 3	Act 4	Act 5	Act 6	Act 7	Act 8	Act 9	Act 10	Act 11	Act 12	Act 13	Act 14	Act 15	Act 16	Act 17
DMD	D2	1	250	15	1	1	1	1	2	1	1	1	1	1	1	0	0	1	0	1	1
DMD	D3	1	275	19	1	1	1	2	2	1	1	0	1	2	1	2	0	1	1	1	1
DMD	D4	1	310	20	1	1	1	2	2	2	2	1	1	1	0	2	0	1	1	1	1
DMD	D5	1	442	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
DMD	D6	1	395	30	2	2	2	2	2	2	2	2	2	2	1	2	1	2	2	1	1
DMD	D7	1	375	26	2	2	2	2	2	1	1	1	1	1	1	2	2	2	1	1	2
DMD	D9	1	335	22	2	1	1	2	2	2	2	1	1	1	0	2	0	2	1	1	1
DMD	D10	1	125	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
DMD	D11	1	175	27	2	2	2	1	2	2	2	2	2	2	1	1	1	2	1	1	1
DMD	D12	1	240	18	1	1	1	2	2	1	1	1	1	1	1	1	0	1	1	1	1
DMD	D14	1	340	28	2	2	2	2	2	2	2	2	2	2	1	0	2	2	1	1	1
DMD	D15	1	300	28	2	2	2	2	2	2	2	2	2	1	1	2	1	2	1	1	1
DMD	D16	1	100	23	2	2	1	2	2	1	1	1	1	2	0	2	1	2	1	1	1
DMD	D17	1	475	33	2	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2
DMD	D18	1	267	24	2	2	1	2	2	1	1	1	1	2	1	1	1	2	1	2	1
DMD	D19	1	182	24	2	2	2	2	2	2	2	1	1	2	1	1	1	2	0	0	1
DMD	D20	1	401	29																	
DMD	D4v2	2	301	14																	
DMD	D5v2	2	465	33																	
HC	HC1	1	485	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC2	1	525	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC3	1	490	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC03	1	490	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC4	1	469	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC5	1	450	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC6	1	450	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC7	1	442	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC8	1	450	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC9	1	425	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC10	1	453	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC11	1	463	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC12	1	463	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
HC	HC13	1	463	34	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

The next script in the system is the reference file which we use to create the y labels used by every model that is built in the system. As a result of it being the only place that contains the y label information for each of the subjects, it's referenced by several scripts including 'ft_sel_red.py', 'rnn.py', and 'model_predictor.py'. The script itself was a collaboration of several other scripts found within the source data directories for 'NSAA' and '6minwalk-matfiles', including 'KineDMD data updates Feb 2019.xlsx' and 'nsaa_matfiles.xlsx'. Some of these scripts contain information about certain subjects that others don't; hence, rather than having the scripts checking each of the files in turn, it was felt that it would be easier to combine the information from all of them into one file.

Each of the models that are built by the 'rnn.py' script are built to target one output type, which will be either 'dhc' (the D/HC label), 'overall' (the overall NSAA score), 'acts' (the 17 individual activity scores), or 'indiv' (the score of the individual activity assuming the input are single-act files) for each of the sequences the model is training and subsequently assessing on (see the 'Data Forms and Types' chapter for more information on each of these output types). For the 'dhc' output type, we don't need to reference this file, as this can be determined simply by the file name. For example, in preprocessing the data into x and y components within 'rnn.py', if the given data that is currently being preprocessed for a given file is from a file called 'D4_position.csv', then we know that the 'dhc' label would be 'D' (or 1 when being fed into the network), while if the data came from the 'HC6_position.csv' file the label would be 'HC' (or 0).

However, for the other types of y labels that we use to train our 'rnn.py' to target the other output types, it's not as simple as observing the name of the file. This is where the NSAA score reference file comes in. As we can see in the above image, it contains the information for every subject we have (that was collected by the initial assessors of the subjects) that includes their individual activity scores and their overall NSAA score. Hence, by finding the relevant row within the document, we get all the information we need for the other output types. For example, let's say the preprocessing function of 'rnn.py' is extracting data from the file corresponding to subject 'D16'.

Once the data file has been separated into their sequences with necessary sequence overlap and an optional proportion of the sequence dropped (more on these later in the discussion of experiment sets), we then need to get the corresponding label for this sequence. If the model is being trained for the 'overall' output type, then we check the table for the value in the 'D16' subject's 'NSAA' column, which corresponds to '23'. This is the y label that each sequence extracted from the 'D16' source file would get if we are training for the 'overall' output type. Alternatively, if the model output type was instead 'acts', the y label would be a list of values '[2, 2, 1, 2, 2, 1, 1, 1, 1, 2, 0, 2, 1, 2, 1, 1, 1]' or, if the source file was a specific single act file for the subject, e.g. 'D16_position_act4.csv', then it would get the label '2' (both of which are determined by the above table. This process of label extraction is identical across multiple scripts, be it for training models in 'rnn.py' or getting the true values of assessing subjects in 'model_predictor.py', and so on.

7.4 Results for Experiment Sets ('RNN Results.xlsx')

Experiment Numb	Description	'comp_stat_vals'/ext_raw_measures' input	'ft_sel_red' arguments
1	Stat values from NSAA\AD to perform D/HC classification	python comp_stat_vals.py NSAA AD all --split_size=1	python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize
2	Stat values from NSAA\AD to perform overall NSAA score regression	python comp_stat_vals.py NSAA AD all --split_size=1	python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize
3	Raw joint angles from DC to perform D/HC classification	python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv	(Not used)
4	Raw joint angles from DC to perform overall NSAA score regression	python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv	(Not used)
5	Raw joint angles from JA to perform D/HC classification	python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv	(Not used)
6	Raw joint angles from JA to perform overall NSAA score regression	python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv	(Not used)
7	Stat values from 6minwalk-matfiles\AD to perform D/HC classification	python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1	python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize
8	Stat values from 6minwalk-matfiles\AD to perform overall NSAA score regression	python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1	python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize
9	Stat values from NSAA\AD to perform single-acts classification	python comp_stat_vals.py NSAA AD all --split_size=1	python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize
10	Raw joint angles from DC to perform single-acts classification	python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv	(Not used)
11	Raw joint angles from JA to perform single-acts classification	python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv	(Not used)
12	Stat values from 6minwalk-matfiles\AD to perform single_acts classification	python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1	python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize

(continued)

'rnn' arguments	Results	Settings	
python rnn.py NSAA AD all dhc --seq_len=10	Test Accuracy = 92.97%	X shape = (742, 10, 30)	Y shape = (742,)
python rnn.py NSAA AD all overall --seq_len=10	Mean Squared Error = 28.7121, Mean Absolute Error = 2.9016	X shape = (742, 10, 30)	Y shape = (742,)
python rnn.py direct_csv DC all dhc --seq_len=60	Test Accuracy = 99.88%	X shape = (8470, 60, 66)	Y shape = (8470,)
python rnn.py direct_csv DC all overall --seq_len=60	Mean Squared Error = 0.4762, Mean Absolute Error = 0.4037	X shape = (8470, 60, 66)	Y shape = (8470,)
python rnn.py direct_csv JA all dhc --seq_len=60	Test Accuracy = 100.0%	X shape = (2143, 60, 66)	Y shape = (2143,)
python rnn.py direct_csv JA all overall --seq_len=60	Mean Squared Error = 0.1675, Mean Absolute Error = 0.2919	X shape = (2143, 60, 66)	Y shape = (2143,)
python rnn.py 6minwalk-matfiles AD all dhc --seq_len=10	Test Accuracy = 82.81%	X shape = (552, 10, 30)	Y shape = (552,)
python rnn.py 6minwalk-matfiles AD all overall --seq_len=10	Mean Squared Error = 29.4065, Mean Absolute Error = 3.56	X shape = (552, 10, 30)	Y shape = (552,)
python rnn.py NSAA AD all acts --seq_len=10	Individual Activity Accuracy = 92.92%, All Activities Accuracy = 79.69%	X shape = (742, 10, 30)	Y shape = (742, 17)
python rnn.py direct_csv DC all acts --seq_len=60	Individual Activity Accuracy = 99.77%, All Activities Accuracy = 98.5%	X shape = (8470, 60, 66)	Y shape = (8470, 17
python rnn.py direct_csv JA all acts --seq_len=60	Individual Activity Accuracy = 99.97%, All Activities Accuracy = 99.74%	X shape = (2143, 60, 66)	Y shape = (2143, 17
python rnn.py 6minwalk-matfiles AD all acts --seq_len=10	Individual Activity Accuracy = 85.48%, All Activities Accuracy = 73.44%	X shape = (552, 10, 30)	Y shape = (552, 17)

With the two main reference files used by the project covered, we now move onto the first of the two documents containing the results of the various experiment sets and model predictions sets that we run. The first of those, 'RNN Results.xlsx', contains the information of building various models within the 'Experiment Set's section of the results discussion which we cover later on. These include testing

different measurements with which to build models, examining different sequence lengths and sequence overlap proportions, and so on. What separates this document from the 'model_predictions.csv' document that we shall discuss shortly is that 'RNN Results.xlsx' contains the results obtained from just analysing the testing data sets supplied to the 'rnn.py': this is generally 20% of the source data set that is fed into the 'rnn.py' script. Hence, the results contained in each cell of the 'Results' column for this document covers the performance of various model setups on this testing data. In contrast, for most of the 'model_predictions.csv' sets these are the results of feeding in complete files of subjects to be assessed on prebuilt models via the 'model_predictor.py' script. Therefore, 'RNN Results.xlsx' gets its results from the direct console output of building models in 'rnn.py', while 'model_predictions.csv' gets its results from running 'model_predictor.py' on models already built in 'rnn.py' and assessing on complete files.

Another difference from 'model_predictions.csv' is that 'RNN Results.xlsx' has its information manually inserted into the table, as opposed to having it automatically done in 'model_predictions.csv' via the 'DataFrame.writecsv()' method called in 'model_predictor.py'. The reason this is manually done is that, for each model that is created that we wish to write to a row of in 'RNN Results.xlsx', there is additional information that 'rnn.py' has no knowledge about and therefore cannot write to the table. This includes a short description of the model that has been created and what prior scripts were needed to have been run in order to preprocess the data that is required for this model (such as 'comp_stat_vals.py' or 'ext_raw_measures.py'). Therefore, when we wish to write a row of data to 'RNN Results.xlsx', we manually fill in parts of the table ourselves and copy/paste in other parts of the console output (here, into the 'Results' column). Additionally, along with the description, necessary prior scripts with arguments to have been run, and the results of the model, we also output the general model configuration that includes the shape of the data and several of the more significant hyperparameters. The idea is that anyone examining our results in the results discussion section can refer back to this table fairly easily and see exactly how the models were built.

The way these results from 'RNN Results.xlsx' are generally assessed is via the 'graph_creator.py' script. This script reads directly from the file, grabs the requested rows, and plots one or more of the columns against each other depending on the arguments (for example, one configuration of 'graph_creator.py' could plot the results of the MAE of the overall NSAA score against the sequence lengths of the corresponding rows, which we do so in experiment set 8). This is therefore the primary way with which we use the 'RNN Results.xlsx' file in this report, though it also serves the purpose of providing results which can be compared to via other users using the system.

7.5 Results for Model Predictions Sets ('model_predictions.csv')

N/A	D11 (downsampled)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1]
N/A	D17	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
N/A	D17 (downsampled)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
N/A	HC6	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
N/A	HC6 (downsampled)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
N/A	D3	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1, 1, 1]
N/A	D3 (feature concat = 60)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1, 1, 1]
N/A	D3 (feature concat = all)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1, 1, 1]
N/A	D9	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
N/A	D9 (feature concat = 60)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
N/A	D9 (feature concat = all)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
N/A	D11	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1]
N/A	D11 (feature concat = 60)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1]
N/A	D11 (feature concat = all)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1]
N/A	D17	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
N/A	D17 (feature concat = 60)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
N/A	D17 (feature concat = all)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
N/A	HC6	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
N/A	HC6 (feature concat = 60)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
N/A	HC6 (feature concat = all)	NSAA	N/A	['position', 'sensorMag' ------ NSAA P	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

(continued)

58.82%	D	D	100.0%	0.0%	27	24
94.12%	D	HC	50.0%	50.0%	33	28
70.59%	D	HC	50.0%	50.0%	33	25
100.0%	HC	HC	25.0%	75.0%	34	31
58.82%	HC	D	75.0%	25.0%	34	23
52.94%	D	D	91.74%	8.26%	19	25
23.53%	D	D	77.23%	22.77%	19	28
41.18%	D	D	87.95%	12.05%	19	29
64.71%	D	D	100.0%	0.0%	22	25
41.18%	D	D	80.94%	19.06%	22	28
41.18%	D	D	73.75%	26.25%	22	25
82.35%	D	D	94.84%	5.16%	27	27
82.35%	D	D	69.53%	30.47%	27	26
70.59%	D	D	63.12%	36.88%	27	27
94.12%	D	HC	22.66%	77.34%	33	27
94.12%	D	D	79.69%	20.31%	33	26
88.24%	D	D	81.25%	18.75%	33	26
100.0%	HC	HC	16.8%	83.2%	34	31
52.94%	HC	D	85.94%	14.06%	34	26
100.0%	HC	HC	13.28%	86.72%	34	32

The final document that we make heavy use of in the system is the 'model_predictions.csv' document. This is where any assessments made by the 'model_predictor.py' script is stored (and by extension 'test_altdirs.py' which calls 'model_predictor.py' numerous times) and as such corresponds to the model predictions sets that are discussed later in the 'Experiments and Results Discussion' chapter. As all the information that we wish to write to the file is known by the 'model_predictor.py' at run time, this processed is automated via the script itself and does not require the user to enter information into the table manually. As the primary differences between the two documents have already been outlined, it just remains to outline the types of information that is recorded in the table along with how we make use of this when assessing various model setups and investigating performance of different variations of subject files in the model predictions sets.

Each assessment made by 'model_predictor.py' writes one row of information to this table. This contains the name of the subject we are assessing on, along with extra strings that signify different types of models that subject was assessed on or any other preprocessing steps that were taken for the subject file(s) (see the relevant model prediction set descriptions to see to what each of these additional strings correspond). The name of the directory that the subject file(s) are sourced from is also included, along with any other directories that act as alternative directories from which to source files (see model prediction set 1 for an example of this). We also include the different measurements extracted from this subject's file(s) that were used to act as input to the necessary models (the exact models that are chosen therefore depend on the types of measurement that are extracted from the subject file in question).

The rest of the columns contain the assessed results of the subject for all the output types we are testing the subject on (which is generally the 'overall', 'acts', and 'dhc' output types). For example, the 'Predicted 'D/HC Label" reflects the results of the assessing the subject on models built for the 'dhc' output type, while the 'True 'Overall NSAA Score" and 'Predicted 'Overall NSAA Score" reflects the results of assessing on models built for the 'overall' output type. Therefore, we use all the information from subject assessments on various types of models to create tables comparing different subjects and setups (as shown in many model predictions sets) or to be used via 'graph_creator.py', which can access 'model_predictions.csv' in a similar way to 'RNN Results.xlsx' in order to create graphs over various rows of the table, based on the arguments passed to 'graph_creator.py'.

Chapter 8

Data Forms and Types

8.1 Background

Before moving onto 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 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 measurement ('input type') and model type we are targeting ('output type').

8.2 The Source '.mat' Files

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). Further information on how this suit works can be found in the "MVN User Manual.pdf" file in the base 'project directory'. Each single file have 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. To access this information we are concerned with (i.e. non-metadata), we can first open the '.mat' file in MATLAB before navigating to 'tree.subject.frames.frame'. An example of what the data values within a '.mat' file look like at this point can be seen below as the aforementioned 'data values' within the '.mat' file:

Variables - tree.subject.frames.frame

tree.subject.frames.frame

Fields	time	tc	ms	type	orientation	position	index	velocity	acceleration	angularVeloc	angularAccel	contacts	sensorF
1	0	'00:00:00:00'	0	'identity'	1x92 double	1x69 double	[] []	[]	[]	[]	[]	[]	
2	0	'00:00:00:00'	0	'tpose'	1x92 double	1x69 double	[] []	[]	[]	[]	[]	[]	
3	0	'00:00:00:00'	0	'tpose-isb'	1x92 double	1x69 double	[] []	[]	[]	[]	[]	[]	
4	0	1x39 double	1.5433e+12	'normal'	1x92 double	1x69 double	0	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
5	16	1x40 double	1.5433e+12	'normal'	1x92 double	1x69 double	1	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
6	33	1x41 double	1.5433e+12	'normal'	1x92 double	1x69 double	2	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
7	49	1x42 double	1.5433e+12	'normal'	1x92 double	1x69 double	3	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
8	66	1x43 double	1.5433e+12	'normal'	1x92 double	1x69 double	4	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
9	83	1x44 double	1.5433e+12	'normal'	1x92 double	1x69 double	5	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
10	99	1x45 double	1.5433e+12	'normal'	1x92 double	1x69 double	6	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
11	116	1x46 double	1.5433e+12	'normal'	1x92 double	1x69 double	7	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
12	133	1x47 double	1.5433e+12	'normal'	1x92 double	1x69 double	8	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
13	149	1x48 double	1.5433e+12	'normal'	1x92 double	1x69 double	9	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
14	166	1x49 double	1.5433e+12	'normal'	1x92 double	1x69 double	10	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
15	183	'11:05:39:00'	1.5433e+12	'normal'	1x92 double	1x69 double	11	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
16	200	'11:05:39:01'	1.5433e+12	'normal'	1x92 double	1x69 double	12	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
17	216	'11:05:39:02'	1.5433e+12	'normal'	1x92 double	1x69 double	13	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
18	233	'11:05:39:03'	1.5433e+12	'normal'	1x92 double	1x69 double	14	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
19	250	'11:05:39:04'	1.5433e+12	'normal'	1x92 double	1x69 double	15	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
20	266	'11:05:39:05'	1.5433e+12	'normal'	1x92 double	1x69 double	16	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
21	283	'11:05:39:06'	1.5433e+12	'normal'	1x92 double	1x69 double	17	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
22	299	'11:05:39:07'	1.5433e+12	'normal'	1x92 double	1x69 double	18	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
23	316	'11:05:39:08'	1.5433e+12	'normal'	1x92 double	1x69 double	19	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
24	333	'11:05:39:09'	1.5433e+12	'normal'	1x92 double	1x69 double	20	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
25	350	'11:05:39:10'	1.5433e+12	'normal'	1x92 double	1x69 double	21	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
26	366	11	1.5433e+12	'normal'	1x92 double	1x69 double	22	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
27	383	[11,12]	1.5433e+12	'normal'	1x92 double	1x69 double	23	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
28	400	[11,12,13]	1.5433e+12	'normal'	1x92 double	1x69 double	24	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul
29	416	[11,12,13,14]	1.5433e+12	'normal'	1x92 double	1x69 double	25	1x69 double	1x69 double	1x69 double	1x69 double	1x1 struct	1x51 doul

Variables - tree.subject.frames.frame

tree.subject.frames.frame

Fields	acts	sensorFreeAc	sensorMagne	sensorOrient	jointAngle	jointAngleXZ	centerOfMas	contact
1		[]	[]	[]	[]	[]	[]	[]
2		[]	[]	[]	[]	[]	[]	[]
3		[]	[]	[]	[]	[]	[]	[]
4	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.057...	[]
5	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1659,2.057...	1x2 cell
6	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1659,2.057...	1x2 cell
7	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1659,2.057...	1x2 cell
8	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1659,2.057...	1x2 cell
9	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.057...	1x2 cell
10	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.056...	1x2 cell
11	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.056...	1x2 cell
12	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.056...	1x2 cell
13	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.055...	1x2 cell
14	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.055...	1x2 cell
15	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.055...	1x2 cell
16	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.054...	1x2 cell
17	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.054...	1x2 cell
18	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1661,2.054...	1x2 cell
19	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1661,2.054...	1x2 cell
20	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1661,2.054...	1x2 cell
21	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1661,2.054...	1x2 cell
22	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.054...	1x2 cell
23	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1660,2.054...	1x2 cell
24	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1659,2.054...	1x2 cell
25	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1658,2.054...	1x2 cell
26	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1657,2.054...	1x2 cell
27	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1655,2.054...	1x2 cell
28	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1655,2.054...	1x2 cell
29	:t	1x51 double	1x51 double	1x68 double	1x66 double	1x66 double	[2.1654,2.054...	1x2 cell

Note that each row is a single sample (what we call a 'frame') 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 in 3D space which, given that 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 1/60th 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 joint angle 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

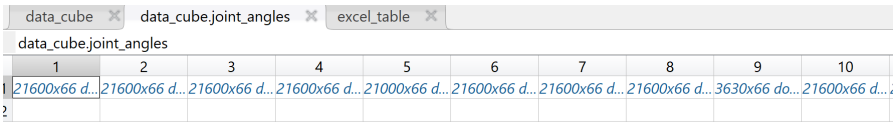
jointangle

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 (21809, 66), 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:



The image above shows how the data cube contains cells which each contains a whole joint angle file's worth of data 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 (metadata) 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 for the first several experiment sets (though we later extract the joint angle data from 'AD' files as is done for other raw measurements in later experiment sets and model predictions sets).

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 frames) 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 second 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 that in the 'Script Ecosystem

Overview' chapter). The final type is the natural movement behaviour data set provided to us in the 'allmatfiles' directory (which contains the joint angle data for the natural movement behaviour data of the subjects) and in the 'NMB' (which contains all the measurements from the natural movement behaviour files and which we received late in the project, hence why we used 'allmatfiles' for a long time). These directory contains numerous files from the same subjects as outlined previously performing various natural movement behaviours, such as sitting and talking, eating lunch, playing, and so on; while this isn't used in most of the experiment sets to begin with, we do use this later in some of the model predictions sets.

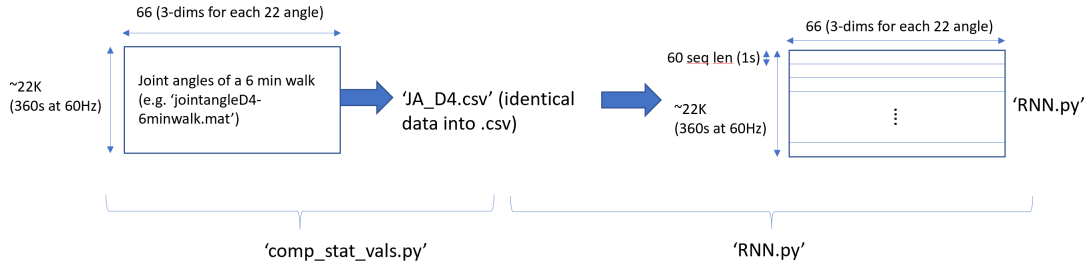
With the forms that the data can take now summarized, we can move onto what we actually do with this data prior to it being used to train an RNN.

8.3 The Data Pipeline

What is referred to as the 'data pipeline' is shorthand for four Python scripts ('comp_stat_vals.py', 'mat_act_div.py', 'ext_raw_measures.py', and 'ft_sel_red.py') that read from data files, manipulate data, and write to new files in the form of 'intermediate data'. 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 ultimately usable by the RNN model. The specifics of what each script does are not covered here for the sake of brevity (see 'Script Ecosystem Overview' for this), so 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). Refer back to the subsection 'The Local Directory: Data Sets' within the 'Project and Local Directories' chapter for information about the source and destination folders used for the variations outlined below.

8.3.1 Variation 1: Joint angles from 'JA' and 'DC'

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



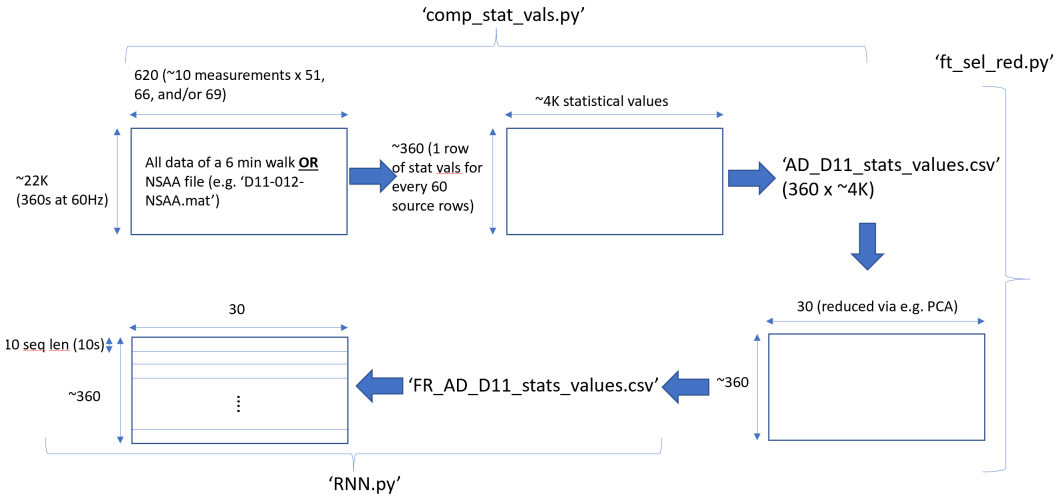
This is what is done when we are dealing with raw joint angle data. Let's say we wish to feed in the 'JA' file for subject 'D4', as seen in the diagram above. The

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.py' 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 a '.csv' format when dealing with all files regardless of its setup, so this essentially standardises the process so the RNN works in the same way for each different original data file type. An important aspect to note, though, is that this is also what is done with the 'JA' or 'DC' files: in the case of 'AD' files, to get the raw measurements we instead use 'ext_raw_measures.py'.

In the 'JA'/'DC' case, this data is then loaded by the RNN 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, 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).

8.3.2 Variation 2: Computed statistical values from 'AD' files

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



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.py', 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 frames 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-frame part of the original 22K to produce a total of approximately 360 rows, each containing 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 a value that has been determined to produce enough data while calculating over enough rows. A thing 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).

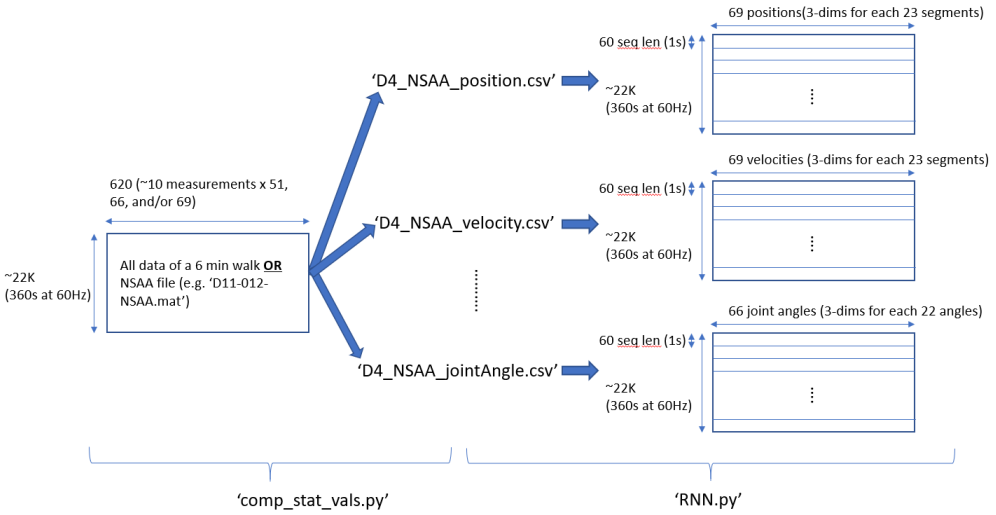
With this data having its statistical values computed and outputted to a new file (in this case of the above image, 'AD_D11_stats_values.csv'), we then load this .csv into a new Python script called 'ft_sel_red()'. The purpose of this is to simply reduce the dimensionality of the data while preserving as much useful information as possible and to then rewrite 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 use PCA as standard based on its prevalence in other studies. 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 is subject to experimentation in experiment set 7), the data is transformed from a dimensionality of (360, 4000) to (360, 30), which is far more conducive to being used as training data for 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, 30) in comparison to the case of reading from raw joint angle files (22000, 60) 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's worth of data embedded within them), 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.

8.3.3 Variation 3: Extracted raw measurements from 'AD' files

Below we can see the pipeline with respect to using 'AD' files as a source of raw measurements:



Unlike the previous two variations of the data pipeline, here we don't make use of either 'comp_stat_vals.py' or 'ft_sel_red.py'. This is because in this setup we neither compute statistical values, nor extract exclusively joint angles, nor do we need to do any dimensionality reduction. Rather, we wish to extract any raw measurements that we need from a given file. This is done by the 'comp_stat_vals.py' script: it takes the name of a file to extract the raw measurements from (or multiple names to carry out this process on) and the names of the raw measurements we wish to extract. From here, the script will seek out the columns of relevance within the tree structure of the corresponding source '.mat' file for the given subject(s), as each vector of a certain raw measurement at each time instance is stored in its own column of the 'tree.subject.frames.frame' table which, when expanded, becomes a matrix of values for the raw measurement. This matrix of values (in the case of the 'position' measurement having a rough shape of (22000, 69)) is then written to a new '.csv' file with a name reflecting the subject name the measurements are from along with the measurement name (e.g. 'D4_NSAA_position.csv').

From here, the 'rnn.py' script is able to build models from these other measurement types in the same way it has done so for the joint angle data and the computed statistical values from the first two variations of the data pipeline by providing a different raw measurement name (e.g. 'sensorMagneticField' or 'acceleration') as opposed to just either 'AD' for computed statistical values (note here that 'AD' references to the computed statistical values; see the Glossary for further clarification about the two ways in which 'AD' is referenced) or 'jointAngle'. After this, the following steps are identical, from the creating of sequences to the training and testing

of models. It should also be noted that, like the 'JA' and 'DC' files, the extracted raw measurement values from 'AD' each represent 1/60th of a second's worth of data; hence, when they are used to create sequences of 60 rows of raw measurement data, this represents 1 second's worth of data in the same way it does for the 'JA'/'DC' files even though it comes from the 'AD' files: it's only when computed statistical values are outputted that the data originating from 'AD' files that it comes to represent 1 second for each row rather than only 1/60th of a second.

8.4 Output Types

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 models we have built actually do. It's important to note that each of the variations works with every type of input file, be they from raw joint angle files or 'AD' files capturing either 6-minute walk 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 file being assessed can then be aggregated together to provide a classification or regression 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.py', and the necessary architectural differences are setup based on arguments passed into the script.

The three main variants (or 'output types') that we have developed are described as follows:

- *'dhc' - 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 'rnn.py' 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. 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 subject assessments, 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' - 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 in the subject's 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 for a subject with severe DMD and a 34 for a subject that shows no symptoms (i.e. a 'HC' subject). 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 contains the subject information ('nsaa_6mw_info.xlsx', as described in the previous chapter): 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 as 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 subject that the sequence originates from.

- *'acts' - 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 the sequence originates from. Again, given the same x data fed through the pipeline, the corresponding y values to train and test on are obtained from the 'nsaa_6mw_info.xlsx' table 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.

Currently, all 3 variants have many of the same hyperparameters, including the train-test ratio of data (0.2; excluding the '–no_testset' optional argument being set which sets it to 0), the number of units in each LSTM cell (128), the number of hidden layers (2), and the 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 6-minute walks vs 'AD' files of 6-minute walks) 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 is what we want

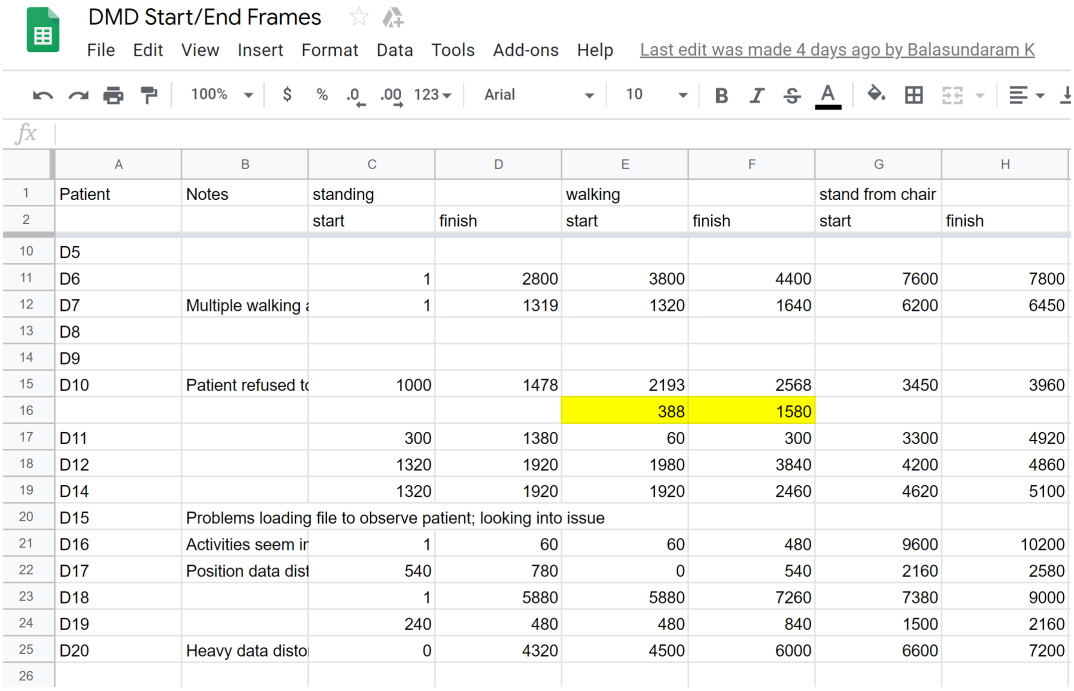
to be able to compare). There are some differences though: a policy decided upon early was that the RNN should train until its loss more-or-less converges. This required different numbers of epochs dependent on the source data type. For example, if the data going into the RNN came from raw joint angles, it only needed approximately 20 training epochs to converge, whereas data from 'AD' computed statistical value files needed between 200-300 to converge. Finally, it's also worth pointing out the main difference in training for 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.

It should also be noted that there is one more output type that is slightly unlike the others and that is the 'indiv' output type. This setups the models to only have one output neuron that computes a score of between 0 and 2, much like the 'overall' output type (though models trained for the 'overall' output type can range from between 0 and 34 for an output value). However, for the 'indiv' output type, we are only concerned with 'single-act' input file data, i.e. files that contain data of a certain raw measurement (or computed statistical values) for a single activity for a certain subject that are produced by the 'mat_act_div.py' script. Hence, the aim of this output type is to compute, for a given subject's single-act file that has a corresponding true value associated with it (that is contained within the 'nsaa_6mw_info.xlsx' table), the predicted value between 0 and 2. Note that this is a regression task, and so the model can predict anywhere between 0 and 2, with the final prediction made by 'model_predictor.py' for a given subject based on the average of these predictions for a sequence and then rounding this average to the closest integer to get the predicted value for that complete subject. We should also note that the reason we don't consider this output type with the others is that it operates on a different file type: while the other three output types can be obtained for any input file, the 'indiv' output type can only be done for single-act files. This makes intuitive sense, as it's somewhat pointless to ask a model to predict a single act score corresponding to a specific activity for a sequence from a subject which could have come from anywhere within the source file (i.e. any one of the 17 activities or the 'in-between' activities data).

8.5 Additional Data Preprocessing Work and Tools Used

One of the disadvantages of the NSAA files provided 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 ('nsaa_6mw_info.xlsx') 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 via the 'mat_act_div.py' script, which would be useful to have for this project as well as other research initiative 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 ir	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								

While we have touched on the above file within the 'Google Annotations Sheet' section of the 'Reference Documents' Chapter, it's worth at this point clarifying a few things about how it was put together. The above image 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 the 'dis_3d_pos.py' script (elaborated upon in the 'Script Ecosystem Overview' chapter) 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 in order 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 in the table.
- Ideally, we try to give a small amount of 'leniency' 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 seemingly not performed or performed alongside another activity, or performed very subtly; 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 further continuation work that can be done for the project and is covered later.

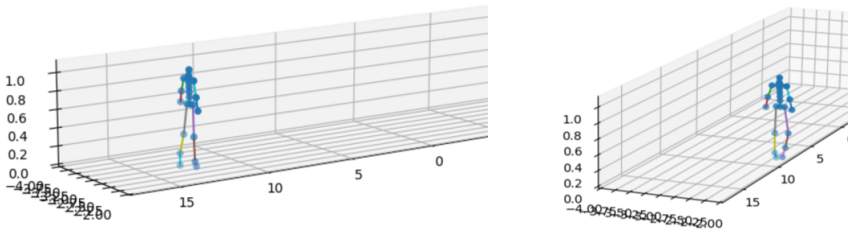
So given that we have the 'rules of thumb' for our annotation work, it was next necessary to actually 'visualize' these 'AD' files. As there was no easy way of 'running' one of these 'AD' files in '.mat' format and as we (at that point) had no access to the '.mov' video files that corresponded to each source '.mat' file, the project necessitated a script 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 'dis_3d_pos.py' Python script and, when the script is run with the required arguments, e.g. for the 'D11' subject , the script does the following:

1. Loads 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.

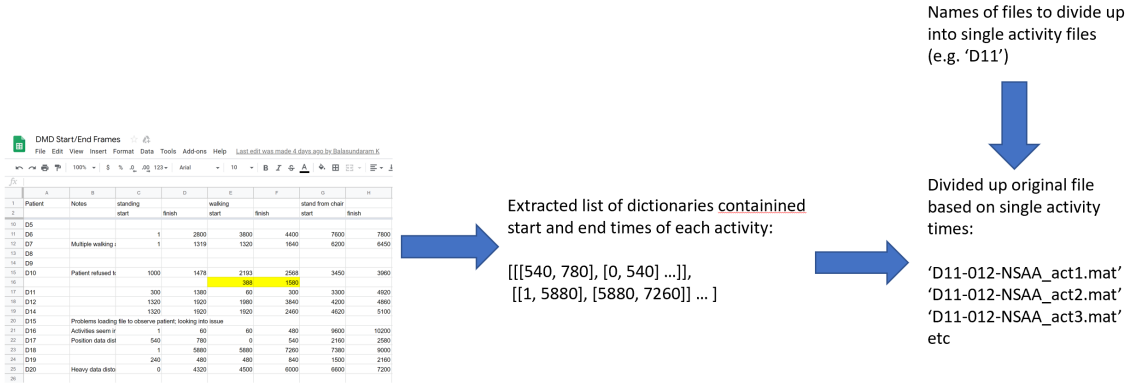
3. Send these to the 'animate' function that animated it at a rate of 60Hz (so the animation would appear as real-time) and closed upon completing the plotting of all values for that file.

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 activities it seemed to perform, to get a reasonably accurate idea of the times of occurrence of each activity.

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



When the 'mat_act_div.py' script is then 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 and the names of the files that contains the activities.

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 data directory with names specific to the activity and subject source file (see the names in the bottom-right image above).

While these divided-up 'AD' files are not used to tune the models for the general cases of using the system, these are used instead in later model prediction sets to train RNN models to predict single-activity NSAA scores and evaluate the significance of particular activities with respect to predicting overall subject assessment.

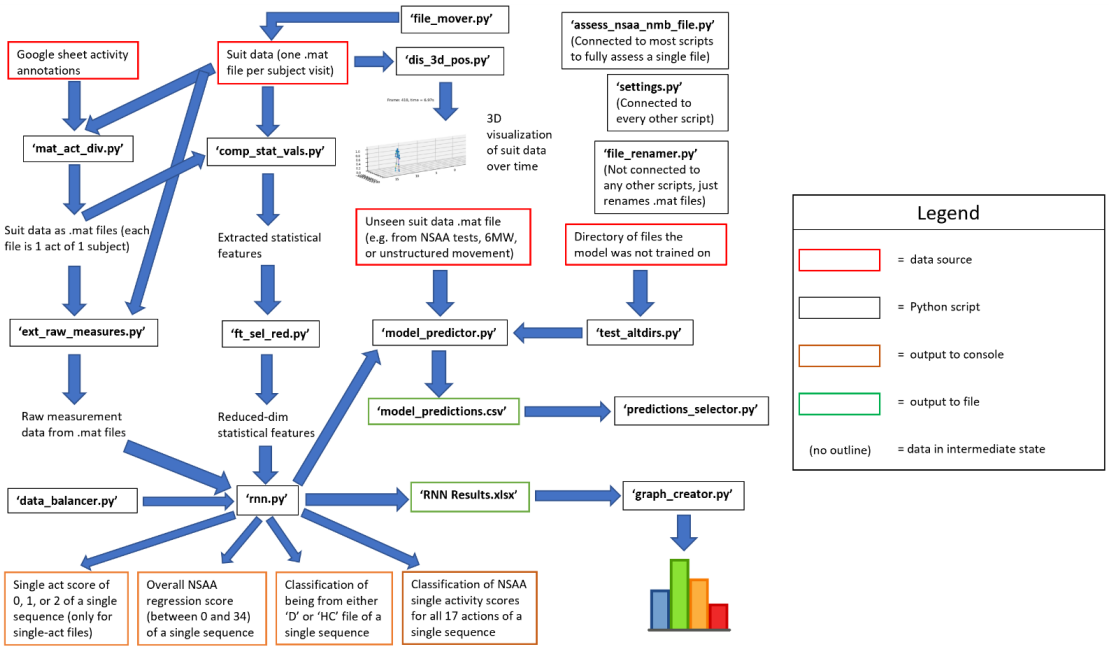
Chapter 9

Script Ecosystem Overview

9.1 Overview and Script Diagram

In this chapter, we shall be covering all of the scripts that are used as part of the system. By 'system' here, we mean the complete set of scripts and supporting documents that are used to carry out all the experiment sets and model predictions sets in an effort to realise the aims of the project, and that encompass the primary body of work done for the project. Each section covers one Python script in turn (with a small section covering all batch scripts at the end), and can each be divided into two parts. The first, 'Overview', describes why we felt it necessary to add the script to the system, what problem it's supposed to solve, and it's broad operations. The second, 'How it works', describes the script generally as either a sequence of steps or describes its different functions in more detail, depending on which is deemed more appropriate. In conjunction with extensive commenting found in each script, the hope is that by using this chapter, any user could understand in detail the purpose and behaviour of each script of the system.

Below, we can see a diagram of all the scripts involved in the system for the project. It covers all the inputs that are needed for the project, how they are processed by the various scripts, and what types of outputs are produced by the system. Note that this is only a vague overview, without any details of how the scripts do work (this is discussed further below) nor details of the names and locations of the source files (which is covered extensively in 'The Local Directory' section of the 'Project and Local Directories' chapter), with the aim more to show the order things should be run in either by the user or by the batch scripts (not included below) and how the outputs relate to each other. Additionally, it is hoped that it will serve as a useful reference point to understanding the complete system and how each script fits into it.



9.2 'comp_stat_vals.py'

9.2.1 Overview

In an effort to experiment with both raw measurement values used in RNN models and also pre-computed features, we needed a separate script that takes in the body of raw measurements from source '.mat' files and computes various statistical values taken over the entire file and the majority of useful measurements. This is the primary purpose of this script: to take in a source '.mat' file of suit data for one or more subjects and, for each of them, produce an output '.csv' file containing rows of statistical value data of the source '.mat' file.

Each of the statistical analyses that are used are implemented by a distinct function that performs a bit of syntactical help (e.g. the function to calculate mean includes type changing and rounding of numbers). The statistical features that are computed include: 'mean', 'variance', 'mean absolute diff values', 'FFT', 'covariance components between axes', 'mean sum of values across axes', among others. The bulk of these features have been extracted 'intra-columns'. This is meant by the following: consider a 'JA' (joint angle) file; it's columns correspond to only 1 'measurement', the 'feature' itself ('feature' in this context meaning one of the 17 sensor labels, 22 joint labels, or 23 segment labels, the choice of which depends on which measurement we are referring to), and the 'dimension' of this feature (as we are dealing with 3D position data, this is 3D with each representing the x, y or z dimension). Many of the statistical features are thus computed on the values within each individual column; for example, the mean for a specific column is computed by averaging all the values for a specific measurement's specific feature's specific dimension (e.g. measurement 'joint angle's feature 'jRightWrist's dimension x-dimension'), of which

there are 22k total values for this corresponding measurement/feature/dimension for the 22k frames (or rows) in a file. However, there are also several statistical functions that are applied one-layer up; that is, rather than calculating over a single column representing a single dimension of a feature of a measurement, it calculates over 3 adjacent problems for ALL dimensions of a feature of a measurement. These mainly include operations that calculate features over a 2-dimensional array of data.

The final variation of running statistical functions are those that operate on single columns are then reapplied to the calculate the same statistical function over all newly-calculated values. For example, if we are concerned with the variance values for the 'position' measurement over 23 feature names over the x-dimension, then we take the variance of these calculated values to form a new value representing the 'position' measurement, the '(over all features)' feature, and the x-dimension. This process is repeated with statistical functions that operate over the other 2 axis dimensions.

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

- Mean
- Variance
- Absolute mean sample difference
- Fast Fourier transform's (1-dimension) largest value

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

- Mean sum of the values of each dimension.
- Mean sum of the absolute values of each dimension.
- First eigenvalue of the covariance matrix of the 3 columns.
- Second eigenvalue of the covariance matrix of the 3 columns.
- x- to y-axis covariance (i.e. row 1 col 2 value of the 3x3 covariance matrix).
- x- to z-axis covariance (i.e. row 1 col 3 value of the 3x3 covariance matrix).
- y- to z-axis covariance (i.e. row 2 col 3 value of the 3x3 covariance matrix).
- Fast Fourier transform (2-dimension) largest 3 values (as 3 separate calculations).
- Proportion of samples outside the mean zone in every dimension.

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

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

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

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

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

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

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

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

The result is then a single row of all of these values for a whole file with *n* columns in the row, with each column corresponding to an above label with associated value computed over the whole file. As we may have many measurements over which to calculate (e.g. 'position', 'velocity', 'angular acceleration', etc.), many features (e.g. 23, 22, or 17 depending on the measurement), 3 dimensions (or 1 dependent on which statistical function we are using), and 15 statistical functions to compute, a single row for an 'AD' (all data) file can be several thousand columns long. Note that for a 'JA' file this is significantly less as we are only concerned with 1 measurement (the 'jointAngle' measurement as this is the only one in the file). Again, it's important to note that these values are calculated across each of the samples (e.g. 22k) for each of the single columns or collection of 3 columns (depending on the statistical function in question).

The split files functionality

An obvious problem from the method described above is that, while we might have plenty of statistical information computed over the single file, it's only contained within a single row. To work around this, we added in the '–split_size' functionality. This essentially divides up the files into sections along time (i.e. a certain number of rows) before computing the statistical values over each of these sections rather than the whole file. For example, let's say that we originally have 22000 rows of a data in a source '.mat' file for a subject. If we provide '–split_size=1' to the 'comp_stat_vals.py' script, we interpret this as 'compute the statistical values over 1 second increments'. This involves dividing up the file into sections of 60 rows (as 60 rows of data correspond to 1 second's worth of data due to the sampling rate of the suit being 60Hz), followed by computing the statistical values over each of these blocks of rows, and finally vertically concatenating these lines to produce the output as a '.csv'.

Thus, rather than having an output of shape (1, 4000), we instead have an output of shape (366, 4000); the downside of course is that each of these rows now

contain computed statistical values calculated only over blocks of 60 rows, as opposed to the whole file. However, it was felt necessary to undertake this process in order to produce a somewhat comparable amount of data to be used alongside raw measurements. It should also be noted that a split size of '1' isn't set in stone, and is something we shall be experimenting with in later experimentation.

9.2.2 How it works

The basic operation of the 'comp_stat_vals.py' script can be summarized as follows:

1. Read in a certain .mat file (either a 'JA', 'AD', or 'DC' file) into Python as an object of either the 'JointAngleFile', 'AllDataFile', or 'DataCubeFile' class. Alternatively, if provided with the 'all' name in place of a file's name, complete this process over all available files in the specified data set.

2. Apply statistical analysis on each file's various measurements, features of measurements, and dimension of the features; alternatively, if the '–split_size' functionality is set, divides the file into sections before computing the statistical values. Note that this is optional if a joint angle is selected and called with the 'write_direct_csv' method, which just translates a joint angle .mat file to .csv format.

3. Write the computed statistical value to a .csv file with a name corresponding to the read in file; the aim with this is for it to be an easy-to-digest format for the next stage in the analytics pipeline (e.g. the 'ft_sel_red.py' script to reduce the dimensionality of the computed statistical value files).

9.3 'ft_sel_red.py'

9.3.1 Overview

One of the consequences of using the 'comp_stat_vals.py' script is that the number of features produced as columns of data for a single subject's 'AD' file balloons several fold: for a single subject with 620 columns (with each being one feature, of 56, of one measurement, of 11) and 22K rows (360s at 60Hz suit sampling rate), this then becomes 360 rows (given '–split_size'=1, i.e. 1 row for every 60 source rows) of approximately 4000 columns. Hence our data shape has been transformed from (22000, 620) to (360,4000) for a single file. This is completely impractical to use as training data for a given model for several reasons:

1. The curse of dimensionality means that the models struggle to train at all when dimensionality is this large for the amount of data samples ('360') that we have available.

2. Many of these computed statistical features may hold not that much useful information in them, or at least less useful information compared to other useful statistical features.

3. Even if we were to use all these features, it would take a much longer time to train models for most likely very little gain (with it most likely being worse off than smaller dimensioned data), making it even worse from a practical standpoint.

Hence, for the '_stat_features.csv' files that are created by the 'comp_stat_vals.py' script, its more-or-less necessary to reduce the dimensionality to something a lot smaller prior to using this as training data. Note that this isn't done for raw measurement data for three reasons:

1. The dimensionality of these data files is already at a level that is feasible for training (ranging from 51 from sensor measurements to 69 for segment measurements).

2. There are far more rows of data within each of these files; this is due to the fact that, with using 'comp_stat_vals.py' with '–split_size'=1, we computed stat values over each block of 60 rows and hence reduce the number of actual 'numbers of data' (i.e. numbers that appear in our data set) by 60-fold. This 60-fold comparable increase in data when using raw measurements makes using this data of column size 51-69 a lot more feasible in training models.

3. Even though we may be computing many redundant features in 'comp_stat_vals.py', we are much less likely to have features that are as redundant as these in the raw measurements data. This is because every feature corresponds to a single dimension for a sensor, angle, or segment, which is much more likely to hold important information that many of the computed statistical values, and thus there is more of a motivation to keep all of these.

9.3.2 How it works

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

1. For a given file name in 'dir', read in the file (e.g. 'AD_D4_stat_features.csv') as a DataFrame and divide it into its x and y components.

2. Normalize each dimension of the data if the relevant optional argument is set.

3. Set the number of features to extract from the data if the relevant optional argument is set. As standard, we use '30', as this generally encompasses a vast amount of the variance inherent to each data file while also being a feasible data width for our RNN models.

4. Based on the 'choice' argument given by the user, use a technique to reduce the dimensionality of the data. This can be done in an unsupervised feature dimensionality reduction manner (e.g. using principal component analysis or Gaussian random projection), unsupervised feature selection manner (e.g. variance thresholding or feature agglomeration), or in a supervised feature selection manner (e.g. by using a random forest for feature selection). 'PCA' has been used up until this point, though further experimentation with other feature selection/reduction techniques is a promising direction to take the project in.

5. With the newly-reduced data, call the 'add_nsaa_scores()' function to add the overall and single-act NSAA scores to each of the rows of reduced-dimensionality data, which is necessary for getting the relevant *y* labels by the 'rnn.py' script, which the output of this script feeds into. The information for these scores comes from the 'nsaa_6mw_info.xlsx' file, which contains the scores for every subject that has undertaken the NSAA assessment; hence, all that is required is to select the row in this .xlsx file that corresponds to the subject we are currently dealing with.

6. The newly-reduced data, with the NSAA scores appended at the beginning of each row, is then written to the same directory as it was sourced, with the exception that an "FR_" ("feature reduced") prefix is appended to each newly-written file name to differentiate it from the file from which it was sourced.

7. Repeat this process for every other file name in 'dir' that is required which, if 'fn'='all', results in all files in 'dir' having their dimensions reduced.

9.4 'mat_act_div.py'

9.4.1 Overview

Along with using the full data files of the suit as computed statistical values used in various models and with varying target outputs (e.g. 'dhc', 'overall', etc.), we also wish to extract the single activities of source '.mat' files from the NSAA directory. As standard, each source '.mat' file in the NSAA directory contains the suit data of one full assessment for a single subject (though on occasion this is divided into two files if the 'walk or 'run' activities happened at a later point). This means that each file usually contains the subject performing all 17 activities within the same file which are separated in time, sometimes by only a second or two in the case of the 'climb/descend box' activities and sometimes by up to a minute in the case of the 'get off the floor' activities. Hence, it would be advantageous for us to extract the data of the individual activities from within each file in order to use them for training for several different model predictions sets that are explored later on.

As the data is contained within a very large table and each row is a single time instance of data (collected at 60Hz from the suit, therefore each frame is 1/60th of

a second's worth of suit data), to create new single-activity files, all we need to do is the following:

1. Determine the start and end rows within the overall file of the activity in question (e.g. if we wished to extract the second activity data that we know starts at 13s and ends at 15s in the subject's assessment, we would need to extract rows 780 to 900 of the source '.mat' file).

2. Slice the relevant rows from the table and create a new '.mat'-friendly tree structure within the script.

3. Write this data to an 'act_files' subdirectory of the source directory as a new '.mat' file with a file name reflecting which activity it represents.

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

9.4.2 How it works

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

This Google annotations sheet can either be found within the 'documentation' directory. From here, once this is read in by the script, there are two functions that are executed:

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

- *'divide_mat_file()'*: This function then takes the above two lists and, depending on what 'fn' argument the user has selected for the subject ('all' if one wishes to divide up all the subjects in the 'NSAA' directory), the relevant row within the 'act_times' list is retrieved. From here, for each activity the subject has completed, each activity 'pair' (i.e. two numbers that are the start and end times in the table for each activity) is retrieved along with the name of the file that contains that activity for the subject (generally the same for all activities for a given subject, though there are exceptions). The complete '.mat' file is then then loaded, the table of data within the '.mat' file is extracted, and the table is sliced for that activity pair. These rows then 'replace' the rows of the 'whole' '.mat' file and the '.mat' file is then rewritten to a different file with a name reflecting the activity it is currently concerned with in the 'for' loop. This then repeats for each of the 17 activities for the given 'fn' subject(s).

9.5 'ext_raw_measures.py'

9.5.1 Overview

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

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

From here, we can then use this data to train an RNN on these raw measurement values with y-labels (i.e. target values) that are determined by the classification of file this '.csv' of data corresponds to (i.e. a 'D' or 'HC' subject), or the overall or single-act NSAA scores that correspond to the subject name of this .csv (e.g. 'D4') that can be found with 'nsaa_6mw_info.xlsx'. In doing this, we provide an alternative to the production of RNN-ready data by 'comp_stat_vals.py' and 'ft_sel_red.py' and are

able to compare how manually extracted features differ in RNN performance versus raw data (where the RNN does its own feature extraction). This is explored further within the discussion of results.

9.5.2 How it works

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

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

9.6 'rnn.py'

9.6.1 Overview

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

- We chose to use LSTM units instead of traditional neurons mainly due to their ability to learn better and the fact that they don't suffer the vanishing or exploding gradient problems.

- Other hyperparameters within the RNN itself (number of layers, size of LSTM units, learning rate, etc.) are kept as a constant throughout the experiments. These were found based on prior 'best practices' through prior research projects undertaken by others as well as rudimentary tuning to find 'good enough' parameters. However, further experimentation to find optimal settings could still be looked into; see the chapter on 'Further Improvements' for a discussion on this.

- The final layer can be either a single node for classification, a single node for regression, or 17 total nodes for single-act classifications; hence, the building of the RNN model depends on the arguments passed to the script.

- The performance of the models that are built here are generally viewed by two means: the console output at the end of the running of the 'rnn.py' script (which provides the info we need to fill in the 'RNN Results.xlsx', provided '–no_testset' is not set) or the 'model_predictor.py' script (which provides info for 'model_predictions.csv'). See the later section within this chapter for more information of how 'model_predictor.py' works.

9.6.2 How it works

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

1. Reads in all required arguments (e.g. source directory, file name(s), output type, etc.) and optional arguments (e.g. sequence length, sequence overlap, leave out file choice, etc.) and checks each for validity.

2. Preprocesses the data from the source directory and file name(s) chosen; this includes reading in all source '.csv' files, fetching the relevant *y* labels for the *x* data from the files, splitting the data into sequences, discarding a proportion of the sequences if necessary, splits into train/test components, etc.

3. Builds the 'rnn' object (instantiated from the 'RNN' class) with the necessary feature length, sequence length, size of LSTM units, number of hidden layers, and so on.

4. Train the RNN on the 'x_train' and 'y_train' components and tests on the 'x_test and 'y_test' components.

5. Prints out the performance on the test set to the console.

6. Write to a '.csv' unique to this model the results of the predictions, the arguments used to run the script, and the results that were printed to the console output. See 'The Local Directory: 'rnn.py' Outputs' section in the 'Project and Local Directories' chapter for more information.

It's also worth touching on a few of the optional arguments. The required arguments should be self-explanatory and in no further need of elaboration. Note that there are several other significant optional arguments that can be set that are not covered below (e.g. '–seq_len', '–seq_overlap', and '–discard_prop'), though the descriptions of what these do can be found where they are used within either the experiment set or model predictions set which specifically utilizes it. However, some of these optional arguments aren't covered in any further details in any experiment sets or model predictions sets and so are covered below:

- *'–write_settings'*: This gives the user the option to store the results of the RNN that are printed to the output to the 'RNN Results.xlsx' file, rather than the user having to manually copy-paste console results to the file in a new row. This is generally used when new experiments sets with different RNNs are being carried out to save time and minimize the chances of human error; however, we generally don't set this when a model predictions set is being carried out, as we wish for the outputs to instead be written to 'model_predictions.csv' by 'model_predictor.py', and thus it serves no purpose to write the results on the test set of 'rnn.py' to 'RNN Results.xlsx' when we don't reference them.

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

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

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

- *'–leave_out'*: This is the standard way to leave out a specific subject short name (e.g. 'D4') when training the model. This is the workaround instead of removing a subject from the source directory so the model is not exposed to the subject in the training process. This is primarily used in conjunction with

'model_predictor.py' to test on the left-out subject in question for those particular models in various model predictions sets. See 'model_predictions.csv' and its section in the results discussions for more information on using this argument.

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

9.7 'model_predictor.py'

9.7.1 Overview

While gaining insights into various types of model parameters, source data types, data preprocessing options, and so on are an important and useful output of the project, one of the primary aims is to be able to assess complete files on models built by 'rnn.py'; for example, we may wish to see how the model performs when tested with a subject it has never seen before and record the results in 'model_predictions.csv'. Alternatively, we may want to be using models in their 'production' form to help inform specialists about subjects based solely on model results. To do this, we need a separate script that not only preprocesses a single subject's file(s) for testing, but also loads the relevant models from a specified source.

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

9.7.2 How it works

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

1. Checks the validity of each passed in argument.

2. For a given file name, loads in the '.csv' files for each of the file types provided; for example, if 'fn'='D4' and 'ft'='AD,jointAngle,sensorMagneticField', then the 'FR_AD_D4_stat_features.csv', 'D4_jointAngle.csv' and 'D4_sensorMagneticField.csv' files are loaded in (the names of which might slightly vary in practice due to naming conventions).

3. Identify the directories that contain the models that we require to use for the files' assessment; note that these are all contained within the '<local directory>\output_files\rn directory (unless the '–final_models' optional argument is used which uses the '<project directory>\source\rnn_models_final' directory instead), and have names that reflect how the models were built and on what data. This is done for all three output types as well. For example, if 'dir'='NSAA' and 'ft'='AD' are used, then 'NSAA_AD_all_dhc_–seq_len=10_–seq_overlap=0.9_–epochs=300', 'NSAA_AD_all_acts_–seq_len=10_–seq_overlap=0.9_–epochs=300' and 'NSAA_position_all_dhc_–seq_len=600_–seq_overlap=0.9_–discard_prop=0.9' are loaded as the model directory names containing the models.

4. Preprocesses the data from the '.csv' files so that they will fit into the pre-trained models (e.g. by having the expected batch size and sequence length) along with fetching the requisite *y* labels for the data in the same way as is done for 'rnn.py'.

5. For each output type and for each of the '.csv' files of the data for the subject in question, put all the data through the model that corresponds to the '.csv's file type (e.g. 'jointAngle' or 'AD') and its output type in prediction mode and have the predictions collected.

6. For a given output type, average together all predictions made over every sequence prediction for every file type to get a prediction for that output type for the whole file. For example, for the NSAA overall score output type, we average the scores for every sequence from a given file input type's predictions, repeat this for the other input types, and finally average these scores to get a prediction of the overall score that takes into account all predictions made for every sequence of all the input types (i.e. measurements) we are assessing on.

7. Outputs these scores to the user and appends these results to a new line within the 'model_predictions.csv' file, along with the name of the subject in question as well as the file types used, the source directory, etc.

Special attention should be paid to some of the optional arguments. Some are used exclusively by other calling scripts (e.g. '–handle_dash' and '–file_num' are exclusively used by the 'test_altdirs.py' script) and others are fairly simple and self-explanatory (e.g. '–show_graph' shows the true and predicted overall NSAA scores made for the subject, while '–single_act' is used when the input to the models are single-act files); however, there are a few others that each require a brief explanation:

- *'–alt_dirs'*: Provide this with a name of a directory that is not the same as 'dir' to test files on models that haven't been trained on the same directory; for example, if dir='allmatfiles' and 'alt_dirs'='NSAA' then subject files will be loaded from the 'allmatfiles' directory but tested on models trained on files originating from the 'NSAA' directory. The motivation and results of this are explored in more depth in the results discussion and can be seen in MPS 1 and MPS 22.

- *'–use_seen'*: For a given file name (e.g. matching or deriving from a subject short name like 'D2-009'), the default behaviour of the script is to seek out model directories where the subject has been completely left out of the training and testing process; in other words, the subject who we're assessing is completely new to the models assessing it. This is done by specifically seeking model directories with names containing '–leave_out=¡file name¿' (along with the other required directory and file type arguments). This is extensively used in model predictions sets involving the assessing of models' generalization performance to new subjects. Sometimes, we may not want to do this specifically: for example, when we want to compare a subject being tested on a model familiar with the subject to one that isn't; see MPS 6 for an example of this.

- *'–use_balanced'*: In a similar way that '–use_seen' seeks out model directories that haven't got something in their names, this optional argument specifically seeks model directories to use that have got '–balanced'='¡up/down¿' in the name (depending on the value given to '–use_balanced') and therefore have been created with upsampled or downsampled data sets. Hence, this allows us to test complete files on models that have trained on an upsampled or downsampled data set. For more information on the data balancing process, consult the README for 'data_balancer.py' or, for more info on how well this performed on complete files, see MPS 10.

9.8 'test_altdirs.py'

9.8.1 Overview

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

Crucially, this script only allows 'model_predictor.py' to work on assessing models' performances on unseen files that also have aren't trained on the same type of data. This allows us to see the strength of the correlation between different types of assessment for subjects wearing the suits and also whether or not predicting the assessment scores by models trained on one type of assessment can be used to infer assessment scores of data in a form that the models haven't been trained on. In other words attempting to answer the question: "Can we have subjects just do natural movement activities and then use the models that have been trained on NSAA and/or 6-minute walk assessments to determine their D/HC classification, NSAA overall scores, etc., just as well as if they had instead done the NSAA and 6-minute walk assessments instead?" The results of this are explored later in the relevant results discussions in MPS 1 and 22.

9.8.2 How it works

The 'test_altdirs.py' script is executed as a series of steps does the following when run:

1. Reads in the name of a directory from which we wish to source the files that we wish to use for assessment, and also the names of the directories that will have been used to train certain models (for example, supplying 'NSAA_6minwalkmatfiles' here will ensure that each time 'model_predictor.py' is then called it retrieves the models that are trained on NSAA and 6-minute walk files).

2. Retrieves a list of '.mat' file names from within the source directory (i.e. if 'allmatfiles' was passed as the 'dir' argument then the names of all '.mat' files from within 'allmatfiles' are retrieved and stored in a list).

3. For every file name within this list of file names, create a unique string that corresponds to the input string to run the 'model_predictor.py' script with the required arguments. This string includes the short file name of the file in question, the file types that the models will have been trained on (for example, if 'allmatfiles' was chosen as 'dir', then this must be 'jointAngle' as this is the only type of information that can be extracted from this type of data), the assessment file directory, and the source file directories that were used to train the models.

4. From here, all functionality is passed on to 'model_predictor.py' for the give file, which runs once for every file with the source directory as specified by the 'dir' argument. For further information on how this runs and what it produces, refer to section on 'model_predictor.py'.

9.9 'graph_creator.py'

9.9.1 Overview

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

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

1. Many times where we are running the scripts, we don't want to see the immediate plotting results or, rather, we can't. For example, when we run the 'model_predictor.py' script once, it's only concerned with writing a single line to 'model_predictions.csv', in the same way that 'rnn.py' only writes one line to 'RNN Results.xlsx', so for these to plot any results over several lines, the scripts would need additional user arguments to tell the script which lines it wishes to use for plotting, which adds to the already-high complexity of the scripts. Additionally, we often run 'rnn.py' via a batch script with many slight differences (to easily create several models to test on) and 'model_predictor.py' via 'test_altdirs.py', so stopping to produce a graph for every line that is written to an output file would be very inconvenient and would slow down the process.

2. In separating the functionality, we keep a large degree of modularity amongst the scripts. In other words, the scripts that write the output to the output files ('model_predictions.csv', 'RNN Results.xlsx, etc.) have nothing to do with the actual plotting of results in graphs. This helps in debugging (i.e. a problem in displaying the data will usually be isolated to 'graph_creator.py') and also allows us to choose when we wish to do the plotting (i.e. after the data that we determine we need has been collected, not after a predetermined point in the running of each 'rnn.py' or 'model_predictor.py' run). Furthermore, this sort of setup opens up the possibility for an easier collaborative effort: if others were to contribute to the output files (e.g. by adding experiment results done on other types of data that is still written to the output files in the same format), then it's possible to use 'graph_creator.py' as a standalone script without the need to have previously run any of the other scripts.

9.9.2 How it works

The direction that 'graph_creator.py' takes in terms of running entirely depends on the first argument as 'choice'. Based on this, the script calls one of five functions that

processes the other given arguments in a certain way. Note that, as each function operates on the arguments given differently, some of them are given generic names such as 'arg_one' and 'arg_two'. Also note that, as each function requires different numbers of arguments, every argument other than the first one ('choice') is optional; hence, when 'choice' is set to 'model_preds_single_acts', it won't throw an error when we only give it values for 'arg_one' and not the other three positional arguments.

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

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

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

- *'plot_model_preds_trues_preds()'*: This is essentially the same as 'plot_trues_preds()' but operates on 'model_predictions.csv' rather than 'RNN Results.xlsx'. Hence, when we use this function and specify a start and end row with 'arg_one' and 'arg_two', we look for those rows within 'model_predictions.csv', find the true and predicted overall NSAA score columns, and plot them alongside each other on the x- and y-axes. See 'plot_trues_preds()' above for further information about these graphs produced.

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

- *'plot_rnn_results()'*: This is the function that analyses the 'RNN Results.xlsx' files and is responsible for the majority of graphs that show the performance of different RNN setups (e.g. sequence lengths, overlap proportions, number of features, types of raw measurements, etc.). The 'arg_one' and 'arg_two' arguments take the start and end experiment numbers of the file (once it has been loaded in as a DataFrame object) by looking at the 'Experiment Number' column to decide on which rows of the DataFrame object that we are concerned with. From here, for each row (which is associated with a model that has been created and tested upon) we extract the names of each measurement the model in question has used, the sequence length, and the results that it has produced. Then, based on the fourth provided argument ('xaxis_choice'), we decide on what to plot along the x-axis: if it's set to 'seq_length', then for each measurement (e.g. 'AD', 'jointAngle, etc.), we create a line and plot how well it performed at various sequence lengths with respect to different metrics (e.g. R^2, RMSE, etc.) based on the third provided argument 'out_type'. If instead it's 'ft' (file type), 'seq_over' (sequence overlap), or 'features' (number of features used), then a single line to plot is used instead over all the lines from DataFrame selected to plot the aspect of the data specified by the 'xaxis_choice' against the metric specified by 'out_type'. Numerous examples of these types of graphs can be seen in the sections of the report discussing specific experiment sets.

9.10 'data_balancer.py'

9.10.1 Overview

One of the inherent problems with the dataset is the lack of 'variance' within the subjects for their overall scores. This is mainly a feature of how the NSAA assess-

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

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

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

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

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

Upsampling has the advantage of it being less likely to discard any of the data that has been given to us; however, it also means that many samples are repeatedly used as 'new' samples, which may lead to unpredictable training results, along with an inflated data set may being more challenging to train on. Downsampling,

meanwhile, might give better generalization results than non-resampled data while being a smaller data set (thus making it quicker to train models that achieve better results), but the discarding of many data points might leave out important insights from the data out of the training process.

9.10.2 How it works

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

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

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

- *'ext_label_dist()'*: For each file that the 'rnn.py' model is training on, reads in the 'nsaa_6mw_info.xlsx' file, finds the relevant row in the table corresponding to the file name in question, and returns the overall NSAA score for this file name. This is then used as the label for each of the sequences that are extracted from the file in question, and the process is then repeated for every other file in the source directory, 'dir'.
- *'downsample()'*: If the '–balance' argument is set as 'down', then this function is called that takes in the 'x_data' and 'y_data' created from sequences (as 'rnn.py' would normally create) and the additional 'y_data_balance' that we have created additionally to use to balance the script, and from these downsamples the data and produces two new lists of 'new_x_data' and 'new_y_data' via the algorithm outlined above.

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

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

9.11 'file_renamer.py'

9.11.1 Overview

One of the primary problems with working with '.mat' files as part of this project is the lack of standardization of file names as they were collected. We have primarily been dealing with 5 source directories containing '.mat' files: 'NSAA' (containing NSAA assessments of subjects), '6minwalk-matfiles' and '6MW-matFiles' (containing the 6 minute walk assessments of subjects), and 'allmatfiles' and 'NMB' (containing the natural movement files of subjects wearing the suit either as solely joint angles or all raw measurements, respectively). Each directory had its own primary way of labelling files but, even within directories time, it wasn't necessarily consistent throughout the directory.

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

1. It was not particularly extensible to new files with new formats being added. If new files were added to one of the source file directories with a slightly different naming format, it would require going deep into several scripts in order to change how they extracted the subject name of each new file it's associated with, it's D/HC label, etc. This process ends up just adding more 'if...else' clauses to many already-cluttered parts of the scripts.

2. As a result of having to change numerous things in several scripts, the process was more prone to human error. For example, as a result of a small oversight and not correctly reading the 'D' part of a file name that corresponded to subjects with 'D' in their subject name (e.g. 'D5'), the script was incorrectly interpreting the D/HC label for many files as being 'HC' rather than 'D' like it

should have been; hence, the model was trained incorrectly due to labelling sequences incorrectly. In comparison, if we would have used 'file_renamer.py' from the beginning, we would have easily spotted any files that have been renamed incorrectly and correct them before other scripts had the chance to misinterpret their labels.

9.11.2 How it works

The basic operation of the 'file_renamer.py' script can be summarized as follows:

1. Reads in the name of a source directory of '.mat' files of which we wish to standardize the names.

2. Gets the names of all '.mat' files within the directory and divides them into one of two categories: 'files_kept' (i.e. the vast majority of files which we don't want to remove) and 'files_to_delete' (files which we want to remove from the directory). Note that this is only for certain files that have been previously determined to be too large, too small, or not 'relevant' files to either training or testing models; for example, files that contain 'AllTasks' in their name in the 'allmatfiles' source directory, as these contain the same information as the other files in the directory but concatenated together for a single subject, so there's no need to use these as well as the others.

3. Based on the source directory name, apply a set of regular expression ('regex') rules to each file name that are in 'files_kept'. These are unique to each directory, as there are some things that we need to check for in some directories but not in other. These regular expressions are a set of substitutions: they search the file name for a certain characteristic and, if it finds it, replaces it with another before using this new string as the basis for the next regex. These regexes include: replacing non-capitalized subject names to capitalized versions (e.g. changing 'd4-003.mat' to 'D4-003.mat'), replacing 'NSA' with 'NSAA when found in a file name, changing instances of '-6MW.mat' to '-6MinWalk.mat' (as the type of activities they contain is the same whether it was sourced from '6minwalk-matfiles' or 6MW-matFiles'), and so on.

4. With this new list of file names that we are to change 'files_kept' to, we first remove the files within the source directory based on the file names within 'files_to_delete' and then, for each name in 'files_kept' and its corresponding name in 'new_files_names', replace the name of the file in the former with the name in the latter. The result is that all of the files within the specified source directory are automatically changed based on the standard we predefined.

However, it's important to note that this script is not intended to be run more than once, and only at the beginning. Hence, it should be executed before any of the other scripts like 'comp_stat_vals.py' or 'ext_raw_measures.py' are used. This is because these scripts use the names of the files they are sourced from to create new files with names based on their source names; hence, for 'file_renamer.py' to be

useful, they should be used prior to other files being created that are based on the files that 'file_renamer.py' wishes to rename. Hence, 'file_renamer.py' is only needed to be used once. For this reason, it's also included within 'setup.cmd' as part of the setup process and is applied before any of the other scripts for the above reason.

9.12 'settings.py'

9.12.1 Overview

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

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

1. It reduces the amount of overall 'clutter' within the scripts, especially when we need to reference large variables such as those holding large lists of strings, which makes the scripts themselves both easier to debug and easier to maintain.

2. For variables that are supposed to remain static, it reduces the possibility of accidentally changing them to suit the script they are currently being referenced in. For example, we are less likely to accidentally change the name of one of the 'raw_measurements' when they are only accessed in other scripts and not modified and, if one is changed in 'settings.py', then this change is reflected out to all other scripts in the same way (e.g. preventing two scripts from each having their own versions of 'raw_measurements', which could cause conflict in manipulating output files).

3. If they are required to change for whatever reason (e.g. if a new user has their 'local_dir' in a different location to the default value, or if the batch size to be used across numerous scripts is modified to be something else), then it's much easier to do so in a single 'settings' script rather than tracking down and modifying each respective variable in each script.

To access these values, each of the scripts calls the necessary variables from 'settings.py' in the 'import' section of the script. The idea of scripts only importing the variables that it needs was that it enhances clarity (i.e. if 'from settings import *' was used, we wouldn't as easily be able to see that 'local_dir' comes from 'settings' as if we had used 'from settings import local_dir'). Additionally, it's also recommended that any user using this project and 'setup.cmd' for the first time should first examine

the relevant path names (such as 'local_dir', 'results_path', etc.) to ensure that the source '.mat' files are contained in the expected location, the scripts can access the necessary output .'xlsx' and '.csv' files, and so on.

9.13 'predictions_selector.py'

9.13.1 Overview

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

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

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

on single-act files, etc.). We can see this script being used in particular in MPS 23.

9.13.2 How it works

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

1. Loads in the 'model_predictions.csv' file as a DataFrame object.

2. Filters the rows of the table based on the 'sfn' argument, which removes all rows where the subject name doesn't match the value of 'sfn'; alternatively, if 'sfn'='all', keep all rows at this point.

3. Filters the rows of the table based on the 'sd' argument, which removes all rows whose source directory column is different from the value of 'sd'.

4. If the 'mtd' is given (i.e. if we're concerned with 'altdir' rows), filters the rows of the table based on this arg, which removes all rows whose 'altdir' column is different from the argument value. Note that the this argument is given as comma-separated values, which corresponds to the list values of the column in question.

5. Based on whether the optional '–best' or '–worst' arguments are given (or both), extracts the first part of the argument (s) as the number of '–best'/'–worst' lines in the table and the second part as the short name of the metric to use to determine which are the '–best'/'–worst' (i.e. by deciding which of the output columns of the table to use to order the rows).

6. For each of the remaining rows of the tables (i.e. after having been filtered by steps 1-4), we now filter the columns of the table: the first four columns are kept (the subject name, source directory, model trained directories, and measurements tested), followed by one of the output columns (the column in question is selected by the second part(s) of the '–best'/'–worst' argument). These values are additionally preprocessed: e.g. if 'overall' is selected, then the absolute value of the difference between the true and predicted values in their respective columns are selected, while if we're using the 'percentage of predicted correct sequences' metric, the relevant column for 'Percentage of predicted ¡D, HC¿ sequences' is used based on the true D/HC label for the row.

7. Creates a list of column names to create a new table of the top n results that include the aforementioned 4 beginning column names from 'model_predictions.csv', followed by column names of the output metrics with the names of the directory that the models that outputted this metric were trained using.

8. Finally, select the top or bottom (or both) n number of lines based on the selected column metric, depending on which of '–best' or '–worst' has been selected and the number of lines to extract from each of them, having reversed

them if needed for percentage metrics ('pacp' and 'ppcs'), before printing out the selected rows to the console as a DataFrame object.

9.14 'dis_3d_pos.py'

9.14.1 Overview

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

1. Visualize the subject within the data as doing certain activities in order to provide a reference (along with the console 'Plotting time...' output) as to what activities are taking place at which time; this is particularly useful as it helped in creating the Google annotations sheet.

2. In plotting this, it easily allow for anomalies within the data file to be detected; for example, if the subject suddenly 'jumps' position or the limbs appear extremely contorted, it might indicate corrupted data which might need to be 'cut out' of the file (or have the whole file discarded).

Though this functionality also exists within the 'comp_stat_vals.py' script, it was felt necessary to also provide the functionality as a separate script within the system; hence, a lot of the code that was required by the '–dis_3d_pos' optional argument within 'comp_stat_vals.py' is repeated for this script.

9.14.2 How it works

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

1. Loads in a '.mat' file corresponding to the 'dir' and 'fn' arguments provided to the script. This is read in as a DataFrame object and is returned from 'preprocessing()' and passed to 'display_3d_positions()'.

2. Extracts the values from the 'position' column and reads this in as a 'positions' matrix (of shape (# of samples, 69)), separates the columns of this new matrix into tuples of x, y, and z axes for each segment within positions for every sample, define connected segments via tuples of pairs of values, sets the boarders of the 3D plot (i.e. the x'y/z mins/maxes), plots the 3D figure from the first sample with connections between points defined by the tuples of pairs of values, and animates it by fetching a new sample to plot every '1/sampling_rate' sections so the figure is animated in real-time while outputting to console the current time-stamp of the figure in seconds.

3. After 5 seconds where the data is sourced, extracted, reconfigured to work in 3D, and animated, a new window will appear. This is the 3D plot that runs in real time. Note that one should also see as a console output the time stamp in seconds of where the plot currently is at (i.e. how far through the positions matrix it is). There is no current way to pause, slow down, or speed up the plotting, though one can change the viewing perspective by left clicking and dragging with the cursor or zoom in and out by right clicking and dragging with the cursor.

9.15 'file_mover.py'

9.15.1 Overview

To enable the working of certain batch scripts, it became a necessity to build into the batch files the ability to relocate files that are located anywhere on a user's PC to the proper sub directory of the local directory in order to have the data pipeline run properly. For example, if there was a source '.mat' file for 'D9V2' subject as an NSAA file (i.e. the second NSAA assessment done for subject 'D9') located somewhere on a user's PC, we wish to be able to copy it over to the ' <local directory>\NSAA\matfiles' subdirectory. However, while we are able to do this potentially in a batch file via the 'move' command, we also wish to be able to change the location of where to copy the file to depend on the type of file we are working with (e.g. if the file is an NMB file, it would be placed in a different location within the local directory than if it was an NSAA file); additionally, we also wish to make use of the 'local_dir' variable stored in 'settings.py' so one wouldn't have to modify a variable within a batch file if the local directory location was changed.

For the above reasons, it was evident that it was simply easier to implement the 'move file' functionality in its own separate Python script. The intention, however, is to only ever use this file as part of a batch file or the 'assess_nsaa_nmb_file.py' script as the first step in placing a file in the correct location to be used within the data pipeline.

9.15.2 How it works

As this is a short script with a singular purpose, it's worth outlining the simple steps, as the program runs in a procedural manner:

1. Takes in as arguments the name of the directory within the local directory to place the file within based on the type of file (e.g. 'NMB', 'NSAA', '6minwalkmatfiles', etc.) and the complete or local path (relative to the '<project directory>\source\batc directory) to the file we wish to move.

2. Checks for argument validity for the 'dir' argument and, given it is one of the allowed options, adds the strings to the 'local_dir' variable based on the 'dir'

argument so that 'local_dir' now points to the correct 'inner' directory to store the copied source '.mat' file in.

3. Attempts to copy the file given as the argument to the program (as a complete or relative file path) to the new value of 'local_dir', and throws an exception if it cannot locate the file by the path given.

9.16 'assess_nsaa_nmb_file.py'

9.16.1 Overview

As part of the finished deliverables for the project, we wanted to create a 'wrapper' Python script that was able to assess a single file (either an NSAA or NMB file) wherever it was located within a user's local system on the models that we have selected as our 'final' chosen models (i.e. those that are contained within '<project directory>\source\rnn_models_final'). The idea of this script is that it would act as the primary tool that someone would use who only wants to assess a single file on the models that we have built and chosen as the best possible models for the job.

While we could have implemented this functionality as a batch script, it was felt that it would be easier implemented as a Python script. This made things like conditional calling of other scripts, argument parsing, and so on simpler to program than if we were using a batch script. It also allowed for dynamic user interaction through inputs to the script, which meant that the script could be written in a more user-friendly way. In other words, for this script we do away with taking in arguments and instead ask for user input at points throughout the script execution. The hope is that it makes it easier to use for any user and can simply run it with only the project directory obtained and a '.mat' file somewhere on their system of which they wish to assess.

As the script is meant to not require the local directory, this posed a potential problem to calling the other scripts (such as 'comp_stat_vals.py') which require the files to be located within the local directory in order to operate them. To get around this, we make use of the 'file_mover.py' functionality within 'assess_nsaa_nmb_file.py' where, if it doesn't see a local directory where it's expecting (based on the 'local_dir' variable value in 'settings.py'), as would be the case where the user doesn't have the local directory, it instead creates a local directory with the same name, with the corresponding inner directories, and places the file from wherever the user specified into here. This then allows the subsequent scripts to operate upon this file as normal.

9.16.2 How it works

The primary operation of the scripts is to take in user input and, from the various inputs, create strings that are passed in turn to the 'os.system()' function to call each script in turn with the correct arguments. Again, as this is a fairly simple script in its

execution with no function calls or object creations, we can summarize the script as a series of several steps:

1. Gets from the user the user-specified path to the '.mat' file which the script shall be assessing (can be either an absolute path or a path relative to the '<project directory>\source' directory.
2. Gets from the user whether the file is of an NSAA assessment or a natural movement behaviour file.
3. Executes 'file_mover.py' to move the file to the required subdirectory of the local directory (based on the NSAA\NMB choice specified) or, if the directory doesn't exist, creates the required directory and subdirectories and then copies the file to the required subdirectory.
4. Executes 'file_renamer.py' to rename the now-copied file if required.
5. Gets from the user the comma-separated measurements (raw or computed statistical values) to use to assess the file.
6. Executes 'ext_raw_measures.py' if required to extract the raw measurements from the file.
7. Executes 'comp_stat_vals.py' and 'ft_sel_red.py' if required to extract the computed statistical values and reduce the dimensionality of the file's computed statistical values.
8. Gets from the user whether or not they wish to use models built on 'alt_dirs' and/or built solely on non-'V2' files.
9. Based on the inputs given by the user regarding 'alt_dirs' and 'V2' files, execute 'model_predictor.py' to assess the file's measurement data on the appropriate models, display the results to console, and write the results to 'model_predictions_newfiles.csv'.

9.17 Additional batch scripts

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

- *'setup.cmd'*: This script runs the necessary 'pip' package installation commands to setup all the external libraries needed for running the project. Specific versions of the packages are used to match the exact versions used as part of this project to avoid potential complications, although setting up the most recent versions of the packages would most likely work just as well. We also run the necessary system scripts on all setup source directories. This requires that the

user has setup the source directories ('NSAA', '6minwalk-matfiles', etc.) in a base directory that matches the name of the 'local_dir' global variable stored in 'settings.py'. Assuming that, the rest of 'setup' will extract the statistical values from each file in every directory, along with reducing the features of these, standardizing the names of the files, extracting all raw measurements from every 'AD' file, and dividing up files to extract single activities from 'AD' files.

- *'models_no_leftout.cmd'*: At the point where we have found the optional model parameters to assess left-out subjects, we then wished to build new models but with no subjects left-out of training. Hence, this script is essentially an extension of MPS 20 where, instead of building models with left-out subjects as done in MPS 20, we build them with all subjects included. These models were then copied over to 'rnn_models_final' within '<project directory>\source' so they could be used by the 'assess_nsaa_nmb.py' script.

9.17.1 Model prediction set scripts

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

The idea is that, for each model predictions set that we are running, all that is needed is therefore to just run the specified '.cmd' script. This will build the requisite models, though sometimes it won't build any new models but will instead rely on models built by previous '.cmd' scripts; hence, it's recommended that each model prediction set batch file be run in numerical ascending order. Once a given model predictions set's batch file has been run, with the necessary models built and file predictions made, the results will appear in 'model_predictions.csv' as the final rows in the table. It's also worth noting the time discrepancies between some of the '.cmd' files: some will only be calling 'model_predictor.py' multiple times, which is comparatively quite quick to execute. However, those that call 'rnn.py' many times will take a lot longer; for example, 'model_predictions_set_3.cmd' needs to build 60 separate RNN models, each of which may take 10-15 minutes to run (assuming the user is building them using a GPU), which could take 10-15 hours in total to execute the script.

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

these '.cmd' scripts must modify each instance of the Python script that is called by the batch script in question in order to correctly alter these chosen script parameters.

Part IV

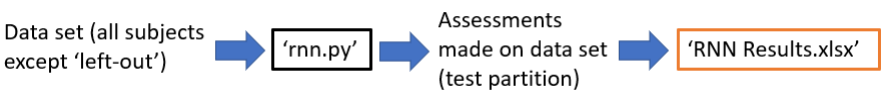
Experiments and Results Discussion

Chapter 10

Background

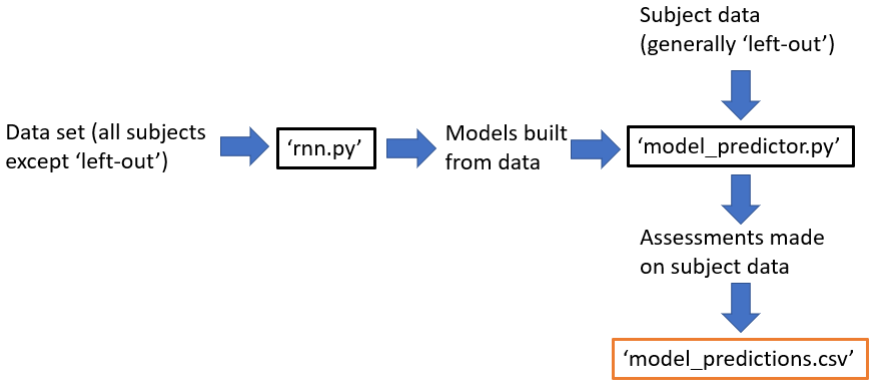
10.1 Experiment Sets and Model Predictions Sets: Overviews and Differences

Broadly speaking, there are two categories of experimentation that we carry out for the system. The first are what we call 'experiment sets'. These are generally speaking carried out using only the 'rnn.py' script and are done to ascertain the optimal model setups to be used later on in 'model predictions sets', the second category of experimentation. Experiment sets include determining the best raw measurements to use to build a system, the number of features from computed statistical values to use, the sequence length and sequence overlap, among other things. All of the results we obtain from the experiment sets are obtained only by the use of 'rnn.py' and the results are obtained by the output of the testing portion of the input data set. Therefore, the results that we obtain here are from testing data from files that it has seen before but data from those files that it hasn't. In other words, the testing data may reflect a more optimistic view of the real-world results on new subjects than would be the case in actuality, and therefore this is the primary cause of disparity between performance in many experiment sets and the results from many model predictions sets; for example, we tend to see a lot higher classification accuracies and more accurate predicted overall NSAA scores in experiment sets than in model predictions sets. However, even though the results might be less accurate than on left-out subjects, the experiment sets where we see improvements in the models by changing some aspect of the models' setup can be regarded to also similarly work on left-out subjects. The diagram below shows this relationship between the data that models are trained on, the script that builds the models, the final assessments made, and where we store the results:



The second category of experimentation are 'model predictions sets'. Unlike the experiment sets, these generally involve the use of two scripts: 'rnn.py' and 'model_predictor.py'. Furthermore, unlike examining the results from the testing data from built models,

we instead use a complete file from a subject left out of the data and do assessments on this subject using 'model_predictor.py'. These also make use of pre-built models, so many of the model predictions sets involve both the building of models with different settings set and also the loading of these recently-built models to assess a left-out subject on. These model predictions sets include testing the performance of different models built on different measurements, the impact of introducing noise to the data set in an effort to help generalisation, the leaving-out of anomalies from the data set, and so on. As these model prediction sets generally involve one or more subjects that have been left out of training completely, the results we obtain here are generally more indicative of real-world assessments of new subjects when the project has been completed and if the system is being used as an assessment tool. The diagram below shows the relationship between data sets, the scripts utilized, and where the results of model predictions sets are stored:



10.2 Experimentation Replication: How to Carry Out the Experiment Sets and Model Prediction Sets Outlined Below

Prior to outlining all parts of the experimentation undertaken for this project, it's preferable to outline for any prospective user the requisite steps needed to replicate the experimentation as best they can. This could be to validate the results that we claim to have achieved here, or to instead carry out further experimentation of their own and would thus benefit from a brief walk-through of the steps necessary to do so. Before we do any of this, however, we should note a few prerequisite steps that must be undertaken if they haven't already been by the user. This includes downloading the project directory, all parts of the local directory, setting up Python and PyCharm, the running of the 'setup.cmd' batch script, and enabling TensorFlow to use any available GPU. Further details of the steps necessary can be found within the 'System Setup' chapter of the report.

Once all the necessary setup steps have been done and with the project and local directories in place with the data sent through the data pipeline via the use of

'setup.cmd', we are ready to use the 'rnn.py' and 'model_predictor.py' scripts in order to undertake experimentation.

To replicate any of the experiment sets below, one must do the following:

1. For a given experiment set, locate the necessary experiments within 'RNN Results.xlsx' that were used as part of the experiment set. For example, for experiment set 3 outlined below, we can see that experiments 13 to 39 were used for this set.

2. Locate the necessary experiments within 'RNN Results.xlsx' (located at '<project directory>\documentation \RNN Results.xlsx'). In the above case, one would find the rows within the table that correspond to the column 'Experiment Number' ranging from 13 to 39. Each row represents a single experiment undertaken (e.g. a model built from a certain data set, with certain data preprocessing hyperparameters, etc.) and the script and argument combinations needed to run it in turn to obtain the results found in that row's 'Results' column. Also note that we store the hyperparameter settings to the right of the 'Results' columns so one can ensure that they have the same hyperparameter settings as the results obtained below.

3. For each of experiments in the relevant rows, ensure that each of the scripts and argument combinations are run. Note that they must be run in the order in which they are presented (i.e. the cell to the right of 'Experiment Number' must be run first, following by the next right cell, and so on). This is because many of the scripts rely on the results of the previous one in order to work (e.g. 'comp_stat_vals.py' must be run before 'ft_sel_red.py'). Also note that these are not required to be run before every experiment, but rather is a requirement to have been run beforehand. Hence, for experiments 13 to 39 we only need to run 'python ext_raw_measures.py NSAA all all' once, and not once per row (as this command just produces the raw measurement data and, while it is required to have been run for these experiments, it doesn't need to be repeatedly called). However, for several of the other commands (e.g. ones concerning 'rnn.py'), these must be run each time as they will very often differ at least slightly from the other experiments in the set and thus create different models (these differences are often what we are testing).

4. Once the user has ensured the requisite setup script combinations have been run followed by the 'rnn.py' variation we need, the user will be presented with a console output. Ensuring that the user has the hyperparameters setup that matches the hyperparameter settings outlined in the rightmost columns, the user can then directly compare the results for the model in question to the relevant row that has just been run and, specifically, compare the console outputs to the output displayed in the 'Results' column. Alternatively, if one wishes to instead carry out their own experiment results, simply disregard the final step about comparing to the previously obtained results for a given experiment and instead append the results of the model, the hyperparameter settings, etc.,

to the end of the table, with an adequate explanation of the purpose of the experiment in the 'Description' column.

To replicate any of the model predictions sets below, one must do the following:

1. As all model predictions sets have their 'rnn.py' and 'model_predictor.py' steps all automated via batch scripts then, assuming the user has successfully run the entirety of the 'setup.cmd' script, one must simply launch and run the relevant batch script for the set. For example, if the user wishes to replicate model predictions set 3, one would simply run 'model_predictions_set_3.cmd' which would run the required 'rnn.py' and 'model_predictor.py' variants to setup the models and test them with complete files.

2. The results obtained by the running of the 'model_predictor.py' scripts via the batch script will be outputted as rows to 'model_predictions.csv', with one row per subject prediction; these rows are written at the end of the table. To establish which of these final rows we are concerned with, note how many times the batch script has called 'model_predictor.py': this will correspond to the number of rows at the end of the table we are concerned with for this specific model predictions set that we are replicating.

3. Locate the previous rows within the table that signify the previous run of this model predictions set, the results of which we will discuss below. The user can do this by noting that the values in the 'Short file name' for the given rows will match those we just now obtained. Once the rows corresponding to the previously run iteration of the model predictions set has been found, we can directly compare the values held in each of the metrics columns (i.e. that include all columns except the first five). Doing so allows one to validate the results previously obtained and as discussed in detail below.

10.3 Experiment Sets Table of Results (Drawn from 'RNN Results.xlsx')

The results contained within 'RNN Results.xlsx' can be seen below in a column-shortened form. Note that the experiment number in the table below corresponds to the same experiment number in 'RNN Results.xlsx'. Hence, for more information about a single row in the table below (e.g. RNN model parameters used, the scripts and exact arguments used to obtain the results, etc.), please see the row in 'RNN Results' corresponding to the same experiment number. Note that 'Data Shape' is the total shape of the 'x' data that is used in the model and includes both the training and testing components. Also note the different parts of the shape: the first is the number of samples, the second number is the sequence length, and the third is the number of features of each sequence.

Exper Number	Source Dir	Measurement	Output Type	Subject List	Data Shape	Results
1	NSAA	Extracted statistical values	D/HC classify	1	(742, 10, 30)	Test Accuracy = 92.97%
2	NSAA	Extracted statistical values	Overall NSAA regress	1	(742, 10, 30)	MSE = 28.71, MAE = 2.90
3	6minutewalk-matfiles	Joint angles (from data cube)	D/HC classify	2	(8470, 60, 66)	Test Accuracy = 99.88%
4	6minutewalk-matfiles	Joint angles (from data cube)	Overall NSAA regress	2	(8470, 60, 66)	MSE = 0.476, MAE = 0.403
5	6minutewalk-matfiles	Joint angles (from joint angle files)	D/HC classify	3	(2143, 60, 66)	Test Accuracy = 100.0%
6	6minutewalk-matfiles	Joint angles (from joint angle files)	Overall NSAA regress	3	(2143, 60, 66)	MSE = 0.167, MAE = 0.292
7	6minutewalk-matfiles	Extracted statistical values	D/HC classify	4	(552, 10, 30)	Test Accuracy = 82.81%
8	6minutewalk-matfiles	Extracted statistical values	Overall NSAA regress	4	(552, 10, 30)	MSE = 29.41, MAE = 3.56
9	NSAA	Extracted statistical values	Single act classify	1	(742, 10, 30)	Ind Act Acc = 92.92%, All Act Acc = 79.69%
10	6minutewalk-matfiles	Joint angles (from data cube)	Single act classify	2	(8470, 60, 66)	Ind Act Acc = 99.77%, All Act Acc = 98.5%
11	6minutewalk-matfiles	Joint angles (from joint angle files)	Single act classify	3	(2143, 60, 66)	Ind Act Acc = 99.97%, All Act Acc = 99.74%
12	6minutewalk-matfiles	Extracted statistical values	Single act classify	4	(552, 10, 30)	Ind Act Acc = 85.48%, All Act Acc = 73.44%
13	NSAA	Position	D/HC classify	1	(9379, 60, 69)	Test Accuracy = 95.04%
14	NSAA	Position	Overall NSAA regress	1	(8034, 60, 69)	MSE = 13.15, MAE = 2.22
15	NSAA	Position	Single act classify	1	(8034, 60, 69)	Ind Act Acc = 94.4%, All Act Acc = 82.94%
16	NSAA	Velocity	D/HC classify	1	(9379, 60, 69)	Test Accuracy = 69.94%
17	NSAA	Velocity	Overall NSAA regress	1	(8034, 60, 69)	MSE = 38.11, MAE = 4.28, RMSE = 6.17, R^2 = -0.0004
18	NSAA	Velocity	Single act classify	1	(8034, 60, 69)	Ind Act Acc = 71.51%, All Act Acc = 4.75%
19	NSAA	Acceleration	D/HC classify	1	(9379, 60, 69)	Test Accuracy = 77.48%
20	NSAA	Acceleration	Overall NSAA regress	1	(8034, 60, 69)	MSE = 48.36, MAE = 4.62, RMSE = 6.95, R^2 = -0.4
21	NSAA	Acceleration	Single act classify	1	(8034, 60, 69)	Ind Act Acc = 74.98%, All Act Acc = 17.25%
22	NSAA	Angular Velocity	D/HC classify	1	(9379, 60, 69)	Test Accuracy = 71.01%

23	NSAA	Angular Velocity	Overall NSAA regress	1	(8034, 60, 69)	MSE = 37.64, MAE = 4.27, RMSE = 6.13, R^2 = -0.002
24	NSAA	Angular Velocity	Single act classify	1	(8034, 60, 69)	Ind Act Acc = 72.82%, All Act Acc = 19.12%
25	NSAA	Angular Acceleration	D/HC classify	1	X shape = (9379, 60, 69)	Test Accuracy = 71.17%
26	NSAA	Angular Acceleration	Overall NSAA regress	1	(8034, 60, 69)	MSE = 37.65, MAE= 4.42, RMSE = 6.14, R^2 = 0.03
27	NSAA	Angular Acceleration	Single act classify	1	(8034, 60, 69)	Ind Act Acc = 73.07%, All Act Acc = 21.81%
28	NSAA	Sensor Free Acceleration	D/HC classify	1	X shape = (9379, 60, 51)	Test Accuracy = 75.59%
29	NSAA	Sensor Free Acceleration	Overall NSAA regress	1	(8034, 60, 51)	MSE = 40.10, MAE = 4.38, RMSE = 6.33, R^2 = 0.002
30	NSAA	Sensor Free Acceleration	Single act classify	1	(8034, 60, 51)	Ind Act Acc = 74.87%, All Act Acc = 14.94%
31	NSAA	Sensor Magnetic Field	D/HC classify	1	(9379, 60, 51)	Test Accuracy = 99.57%
32	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(8034, 60, 51)	MSE = 4.49, MAE = 1.35, RMSE = 2.12, R^2 = 0.87
33	NSAA	Sensor Magnetic Field	Single act classify	1	(8034, 60, 51)	Ind Act Acc = 99.32%, All Act Acc = 96.5%
34	NSAA	Joint Angle	D/HC classify	1	(9379, 60, 66)	Test Accuracy = 98.11%
35	NSAA	Joint Angle	Overall NSAA regress	1	(8034, 60, 66)	MSE = 3.95, MAE= 1.23, RMSE = 1.99, R^2 = 0.9
36	NSAA	Joint Angle	Single act classify	1	(8034, 60, 66)	Ind Act Acc = 98.57%, All Act Acc = 93.38%
37	NSAA	Joint Angle XZY	D/HC classify	1	(9379, 60, 66)	Test Accuracy = 96.93%
38	NSAA	Joint Angle XZY	Overall NSAA regress	1	(8034, 60, 66)	MSE = 4.98, MAE = 1.36, RMSE = 2.23, R^2 = 0.88
39	NSAA	Joint Angle XZY	Single act classify	1	(8034, 60, 66)	Ind Act Acc = 96.98%, All Act Acc = 89%
40	NSAA	Extracted statistical values	Overall NSAA regress	1	(742, 10, 30)	MSE = 28.71, MAE = 2.9
41	NSAA	Extracted statistical values	Overall NSAA regress	1	(920, 10, 30)	MSE = 28.34, MAE = 2.26, RMSE = 5.32, R^2 = 0.39
42	NSAA	Extracted statistical values	Overall NSAA regress	1	(1227, 10, 30)	MSE = 13.22, MAE = 1.56, RMSE = 3.64, R^2 = 0.69
43	NSAA	Extracted statistical values	Overall NSAA regress	1	(1840, 10, 30)	MSE = 9.06, MAE = 1.13, RMSE = 3.01, R^2 = 0.79
44	NSAA	Extracted statistical values	Overall NSAA regress	1	(3710, 10, 30)	MSE = 1.19, MAE = 0.52, RMSE = 1.09, R^2 = 0.97
45	NSAA	Extracted statistical values	Overall NSAA regress	1	(7420, 10, 30)	MSE = 0.03, MAE = 0.16, RMSE = 0.18, R^2 = 0.99
46	NSAA	Position	Overall NSAA regress	1	(1013, 60, 69)	MSE = 0.13, MAE = 0.18, RMSE = 0.37, R^2 = 0.52
47	NSAA	Velocity	Overall NSAA regress	1	(1013, 60, 69)	MSE = 0.45, MAE = 0.44, RMSE = 0.67, R^2 = -0.38
48	NSAA	Acceleration	Overall NSAA regress	1	(1013, 60, 69)	MSE = 0.38, MAE = 0.42, RMSE = 0.62, R^2 = -0.43
49	NSAA	Angular Velocity	Overall NSAA regress	1	(1013, 60, 69)	MSE = 0.38, MAE = 0.38, RMSE = 0.61, R^2 = -0.24
50	NSAA	Angular Acceleration	Overall NSAA regress	1	(1013, 60, 69)	MSE = 0.27, MAE = 0.34, RMSE = 0.52, R^2 = 0.03
51	NSAA	Sensor Free Acceleration	Overall NSAA regress	1	(1013, 60, 51)	MSE = 0.33, MAE = 0.4, RMSE = 0.57, R^2 = 0.04

52	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(1013, 60, 51)	MSE = 0.13, MAE = 0.17, RMSE = 0.37, R^2 = 0.54
53	NSAA	Joint Angle	Overall NSAA regress	1	(1013, 60, 66)	MSE = 0.09, MAE = 0.12, RMSE = 0.3, R^2 = 0.65
54	NSAA	Joint Angle XZY	Overall NSAA regress	1	(1013, 60, 66)	MSE = 0.14, MAE = 0.18, RMSE = 0.37, R^2 = 0.55
55	NSAA	Position	Overall NSAA regress	1	(13365, 60, 69)	MSE = 16.27, MAE = 2.76, RMSE = 4.03, R^2 = 0.64
56	NSAA	Position	Overall NSAA regress	1	(13314, 90, 69)	MSE = 17.51, MAE = 2.6, RMSE = 4.18, R^2 = 0.6
57	NSAA	Position	Overall NSAA regress	1	(13315, 120, 69)	MSE = 17.89, MAE = 2.81, RMSE = 4.23, R^2 = 0.59
58	NSAA	Position	Overall NSAA regress	1	(13313, 180, 69)	MSE = 15.12, MAE = 2.44, RMSE = 3.89, R^2 = 0.67
59	NSAA	Joint Angle	Overall NSAA regress	1	(13365, 60, 66)	MSE = 4.46, MAE = 1.24, RMSE = 2.11, R^2 = 0.9
60	NSAA	Joint Angle	Overall NSAA regress	1	(13314, 90, 66)	MSE = 4.99, MAE = 1.27, RMSE = 2.23, R^2 = 0.89
61	NSAA	Joint Angle	Overall NSAA regress	1	(13315, 120, 66)	MSE = 4.21, MAE = 1.16, RMSE = 2.05, R^2 = 0.91
62	NSAA	Joint Angle	Overall NSAA regress	1	(13313, 180, 66)	MSE = 2.65, MAE = 0.99, RMSE = 1.63, R^2 = 0.94
63	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(13365, 60, 51)	MSE = 3.05, MAE = 1.13, RMSE = 1.75, R^2 = 0.93
64	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(13314, 90, 51)	MSE = 2.47, MAE = 0.89, RMSE = 1.57, R^2 = 0.94
65	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(13315, 120, 51)	MSE = 2.31, MAE = 0.84, RMSE = 1.52, R^2 = 0.95
66	NSAA	Sensor Magnetic Field	Overall NSAA regress	1	(13313, 180, 51)	MSE = 7.07, MAE = 1.76, RMSE = 2.66, R^2 = 0.84
67	NSAA	Joint Angle XZY	Overall NSAA regress	1	(13365, 60, 66)	MSE = 5.99, MAE = 1.44, RMSE = 2.45, R^2 = 0.86
68	NSAA	Joint Angle XZY	Overall NSAA regress	1	(13314, 90, 66)	MSE = 4.29, MAE = 1.28, RMSE = 2.07, R^2 = 0.9
69	NSAA	Joint Angle XZY	Overall NSAA regress	1	(13315, 120, 66)	MSE = 4.72, MAE = 1.24, RMSE = 2.17, R^2 = 0.9
70	NSAA	Joint Angle XZY	Overall NSAA regress	1	(13313, 180, 66)	MSE = 2.62, MAE = 1.01, RMSE = 1.62, R^2 = 0.94
71	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 10)	MSE = 0.03, MAE = 0.12, RMSE = 0.17, R^2 = 0.99
72	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 20)	MSE = 0.02, MAE = 0.09, RMSE = 0.14, R^2 = 0.99
73	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 30)	MSE = 0.01, MAE = 0.07, RMSE = 0.1, R^2 = 0.99
74	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 40)	MSE = 0.01, MAE = 0.09, RMSE= 0.12, R^2 = 0.99
75	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 50)	MSE = 0.02, MAE = 0.09, RMSE = 0.13, R^2 = 0.99
76	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 30)	MSE = 0.01, MAE = 0.07, RMSE = 0.1, R^2 = 0.99
77	NSAA	Extracted statistical values	Overall NSAA regress	1	(6550, 20, 30)	MSE = 1.13, MAE = 0.5, RMSE = 1.06, R^2 = 0.98
78	NSAA	Extracted statistical values	Overall NSAA regress	1	(3185, 40, 30)	MSE = 1.05, MAE = 0.67, RMSE = 1.02, R^2 = 0.98
79	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 30)	MSE = 0.01, MAE = 0.07, RMSE = 0.1, R^2 Score = 0.99

80	NSAA	Extracted statistical values	Overall NSAA regress	1	(12794, 20, 30)	MSE = 0.26, MAE = 0.26, RMSE = 0.51, R^2 = 0.99
81	NSAA	Extracted statistical values	Overall NSAA regress	1	(12195, 40, 30)	MSE = 0.53, MAE = 0.42, RMSE = 0.73, R^2 = 0.99
82	NSAA	Extracted statistical values	Overall NSAA regress	1	(13240, 10, 30)	MSE = 0.01, MAE = 0.07, RMSE = 0.1, R^2 = 0.99
83	NSAA	Extracted statistical values	Overall NSAA regress	1	(13265, 7, 30)	MSE = 0.05, MAE = 0.15, RME = 0.21, R^2 = 0.99
84	NSAA	Extracted statistical values	Overall NSAA regress	1	(13305, 5, 30)	MSE = 0.11, MAE = 0.19, RMSE = 0.33, R^2 = 0.99
85	NSAA	Extracted statistical values	Overall NSAA regress	1	(13453, 3, 30)	MSE = 0.12, MAE = 0.19, RMSE = 0.35, R^2 = 0.99
86	NSAA	Joint Angle	Overall NSAA regress	1	(13365, 60, 66)	MSE = 4.46, MAE = 1.24, RMSE = 2.11, R^2 = 0.9
87	NSAA	Joint Angle	Overall NSAA regress	1	(13314, 90, 66)	MSE = 4.99, MAE = 1.27, RMSE = 2.23, R^2 = 0.89
88	NSAA	Joint Angle	Overall NSAA regress	1	(13315, 120, 66)	MSE = 4.21, MAE = 1.16, RMSE = 2.05, R^2 = 0.91
89	NSAA	Joint Angle	Overall NSAA regress	1	(13313, 180, 66)	MSE = 2.65, MAE = 0.99, RMSE = 1.63, R^2 = 0.94
90	NSAA	Joint Angle	Overall NSAA regress	1	(13314, 60, 66)	MSE = 3.36, MAE = 1.21, RMSE = 1.83, R^2 = 0.93
91	NSAA	Joint Angle	Overall NSAA regress	1	(13315, 60, 66)	MSE = 3.43, MAE = 1.13, RMSE = 1.85, R^2 = 0.93
92	NSAA	Joint Angle	Overall NSAA regress	1	(13313, 60, 66)	MSE = 3.13, MAE = 1.07, RMSE = 1.77, R^2 = 0.93
93	NSAA	Joint Angle	Overall NSAA regress	1	(13240, 60, 66)	MSE = 2.4, MAE = 0.88, RMSE = 1.55, R^2 = 0.95
94	NSAA	Joint Angle	Overall NSAA regress	1	(13051, 60, 66)	MSE = 1.06, MAE = 0.6, RMSE = 1.03, R^2 = 0.98
95	NSAA	Joint Angle	Overall NSAA regress	1	(12436, 60, 66)	MSE = 1.48, MAE = 0.66, RMSE = 1.22, R^2 = 0.97
96	NSAA	Joint Angle	Overall NSAA regress	1	(11530, 60, 66)	MSE = 0.35, MAE = 0.41, RMSE = 0.59, R^2 = 0.99
97	NSAA	Joint Angle	Overall NSAA regress	1	(10130, 60, 66)	MSE = 0.35, MAE = 0.46, RMSE = 0.59, R^2 = 0.99
98	NSAA	Joint Angle	Overall NSAA regress	1	(7481, 60, 66)	MSE = 0.73, MAE = 0.7, RMSE = 0.86, R^2 = 0.98

= Experiment Set 1
= Experiment Set 2
= Experiment Set 3
= Experiment Set 4
= Experiment Set 5
= Experiment Set 6
= Experiment Set 7
= Experiment Set 8
= Experiment Set 9
= Experiment Set 10

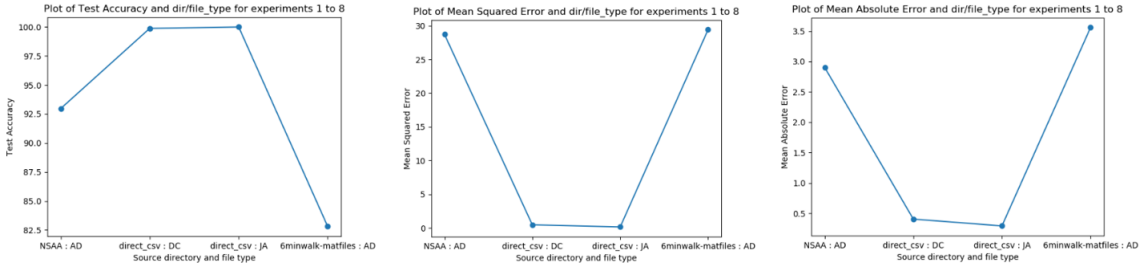
Subjects Lists

Subject List 1	D2, D3, D4, D4V2, D5, D6, D7, D9, D10, D11, D12, D14, D15, D16, D17, D18, D19, HC3, HC4, HC5, HC6, HC7, HC8, HC9, HC10
Subject List 2	D2, D3, D4, D5, D6, D7, D9, D10, D11, D12, D14, D15, D17, D18, D19, HC1, HC2, HC3, HC4, HC5, HC6, HC7, HC8, HC9, HC10
Subject List 3	D2, D3, D4, HC2, HC5, HC6
Subject List 4	D2, D3, D4, D5, D6, D7, HC1, HC2, HC3, HC4, HC5, HC7, HC8, HC9, HC10

Chapter 11

Experiment Sets

11.1 Experiment Set 1: Performance of RNNs on Different Source Data



The purpose of this experiment set is to determine whether and how well RNN models regress on different types of source data. By this, we mean data that is in the same format (i.e. source '.mat' files in the same organizational structure) but representing different measurements from different source directories. These are thus:

- *'NSAA : AD'* - Statistical values extracted from the 'all-data' ('AD') files by the 'comp_stat_vals.py' script, whose files are sourced from the 'NSAA' directory; hence these are stat values of the subjects performing the NSAA activities, which are then used to train model(s).
- *'direct_csv : DC'* - These are the joint angle values (i.e. raw measurements, not computed stat values) that are sourced from the data cube; this data cube contains the 6-minute walk data from various subjects, several of which aren't included in the standard joint angle files of subjects doing the 6-minute walk.
- *'direct_csv : JA'* - Similar to 'direct_csv : DC' as described above, these use the same source directory type and raw measurement, but contain joint angle files that aren't necessarily included within the data cube.
- *'6minwalk-matfiles : AD'* - Again, uses the files corresponding to subjects' 6-minute walk assessments, with the difference this time being that we aren't

using raw joint angle files either in the data cube format or as 'loose' files, but rather computing statistical values via 'comp_stat_vals.py' script; in this sense, it's the same as 'NSAA : AD' models but using different assessment data (6-minute walk rather than NSAA).

For the second and third diagrams above, with regards to the output type, all these file types and directory sources outlined above are used to train models to regress on the overall NSAA score for a given sequence from a file. This value is able to range between 0 and 34 (though based on how the assessment is done, it generally ranges between 15 and 34). Hence, for a given type of source data, if it has a MAE = 0.5, that means that, on test data of sequences from files of the given type of source data, the model predicts for each of them a score of between 0 and 34 on average '0.5' away from the true value for that sequence (the true value for a sequence being the overall NSAA score of the file that the sequence comes from). However, for the first diagram above, it tests the different files types on its ability to classify whether a sequence that come from a file are from a 'D' or 'HC' subject.

11.1.1 Results Discussion

In using just the raw joint angle values from 'DC' or 'JA' files, we achieve an approximate **99%** accuracy (i.e. for each sequence of 60 rows of 66 joint angle values, the model can predict with 99% 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%** for data that comes from the same assessment (6-minute walk). 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 **MAE = 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 = 3.56** from 6-minute walk 'AD' statistical value files. A further observation can be made about the experiments concerning the raw joint angle files 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% 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.

This large difference between raw measurements and computed statistical features 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. One possible reason for

raw joint angle models performing so well could be that there aren't that many subjects with these files that are provided to us in comparison with 'AD' files, so not as diverse a training set is used; this means that there would be a narrower range of overall NSAA scores to regress towards, making it easier for the model to approximate the true score with a smaller margin of error when just using the joint angle data. Another possible reason could be that the features that the RNN models extracts from its inputted data are more useful for it to train on and approximate an overall score than manually crafted features from 'comp_stat_vals.py': a prominent benefit of using neural networks is that they are traditionally noted to perform better with raw data than manually extracted features. Furthermore, when it trains on the stat value data from 'comp_stat_values.py' for the 'AD' file types, it still computes its own features within the RNN, and so this 'features from features' behaviour might have proved to be problematic for the network.

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 a shape of (552, 10, 30) due to how computing statistical values dramatically reduces the raw amount of available data. This decrease in the amount of available data also might help to account for the reduction in accuracy when using data sourced from 'AD' files.

Overall, given our limited data and without any way to currently increase it, we come to several conclusions:

1. RNN models can successfully regress and classify on data 6-minute walk and NSAA assessment data.

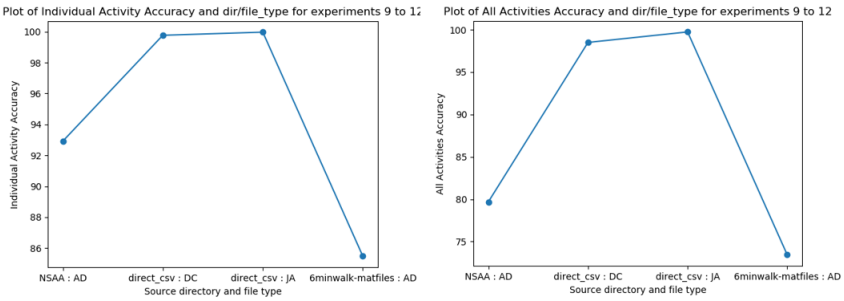
2. Not only this, but raw joint angles perform phenomenally well in both classification and regression tasks.

3. The raw data measurements might be better put through RNNs than extracted statistical features.

4. This is probably due to extracted statistical features being a lot smaller of a data set, RNNs better utilizing raw measurements over pre-computed features, and there being a small range of target values with files sourced as 'JA' or 'DC'.

We shall shortly be examining ways to improve the performance of models trained on computed statistical values by increasing sequence overlap, modifying sequence length, and so on.

11.2 Experiment Set 2: Performance of RNNs for Single Activity Classification



This experiment utilizes the same source data types as experiment set 1. Hence, we won't discuss what each of these data types represent in the x-axes of the graphs above, as these are exactly the same files as used previously. The difference with this experiment set, however, is that it is looking at a different output type: while experiment set 1 looked at performance for classification of D/HC labels and overall NSAA scores for given test sequences, this experiment set looks at predicting multiple classification values for a single sequence. Here, the models are trained to predict a sequence of 17 values of numbers of either 0, 1, or 2 (i.e. the 'acts' output type). These represent the single activity scores for a single sequence that correspond to the single activity scores of the file that the sequence is sourced from. It's worth noting that, as the overall NSAA score is the sum of these, a sequence will have an overall NSAA score that is equal to the sum of the same sequence's 17 single act scores.

Furthermore, when predicting single act scores, the RNN architecture will be different: for D/HC classification or overall NSAA score regression ('dhc' and 'overall' output types, respectively), there will be only 1 output neuron (though predicting different ranges of values for the two tasks), while for single acts there are 17 output neurons, 1 for each single act it is predicting. The overall aim of this experiment is thus to see if a model is able to predict, given a sequence of values from a file (that corresponds to a subject, e.g. 'D4'), what that file's single-act scores are. Note that the two metrics that we are using (that are computed by each RNN model over all its testing data) is individual activity accuracy and all activities accuracy: the former is the percentage of activities in the testing data set that were correctly predicted to be a 0, 1, or 2, while the latter is the percentage of whole activity sets (i.e. single RNN output of 17 values for a single test sequence) that were correctly predicted.

11.2.1 Results Discussion

In line with the results from experiment set 1, we can see here that raw joint angle data is much more useful in predicting the single act scores for test data of sequences than computed stat values from 'AD' files. This is most likely down to the same reasons as for the other output types; namely, comparatively small about of 'AD' data, neural networks better utilizing raw measurements, etc. Given that it

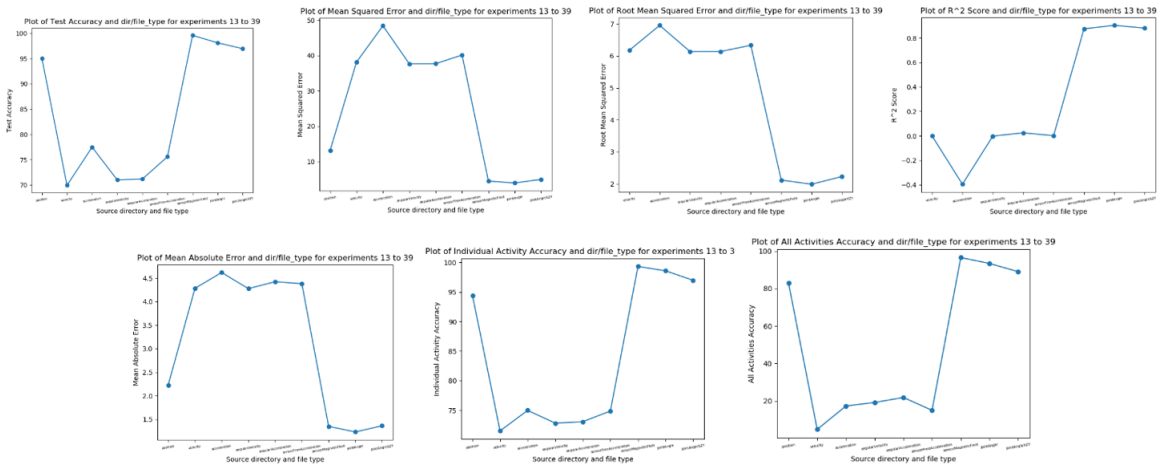
is consistent with the previous experiment set results in determining the comparative performance for different source data types, the results of this experiment set is therefore more about whether it is possible for these data types to train a RNN to predict sequences of values that correspond to activities that the sequence being tested might not be from (i.e. the sequence will be from part of a file that, at most, corresponds to one activity the subject is performing, and therefore can only try to assess what its likely other single activity scores are). The result is that it performs well, especially with joint angle data, predicting approximately **99%** of the individual activity scores and **98%** of the all activity sequence scores.

From this, we can draw several conclusions:

1. The different types of source data can be used to train a model to accurately predict single act values.

2. Like the previous two output types, raw joint angle measurements do this better than 'AD' stat values.

11.3 Experiment Set 3: Raw Measurements for All Output Types



With this experiment set, we now turn our attention to looking exclusively at raw measurements. This is primarily due to the much better performance of models trained on raw measurements compared with ones trained on computed statistical values stat values from 'AD' files, as seen in experiment sets 1 and 2. The question we thus ask is: can we show similarly high performance when we train models on types of raw measurements from the suit data other than just joint angles? To this end, we look at 9 raw measurements in total that are contained within the 'AD' files and that are recorded by the suit during use: 'position', 'velocity', 'acceleration', 'angularAcceleration', 'angularVelocity', 'sensorFreeAcceleration', 'sensorMagneticField', 'jointAngle', and 'jointAngleXZY'. Unlike the joint angle measurements, we don't have

the other raw measurements in separate, unique files. Therefore, we make extensive use of the 'ext_raw_measures.py' script in order to extract, for every file and for every raw measurement, the measurement data and store them in separate files as '.csv' files. From here, these are then able to train and test an RNN model in the same way.

For this experiment set, we are exclusively concerning ourselves now with NSAA assessment files rather than 6-minute walk files. This is more a choice based on the intended direction of the project to be more concerned with assessing and making predictions concerning the NSAA assessments and how that is connected to natural movement data (more on this later) than the 6-minute walk data. Thus, each of the entries along the x-axis of the graphs represents a single model trained on that particular raw measurement data for every NSAA file we have available. Currently, we are just looking at how the models performs on testing data from subjects it has already seen before, rather than complete subjects being left out of the training set, though this shall be explored later on with 'left out' subjects in trained models for various model predictions sets. Furthermore, though each of the models are trained with the same number of files and each with the same sequence length (due to the fact that at every time step the suit records all raw measurements), the feature size will vary; in other words, if the training shape to the RNN models are of shape (x, y, z), x and y will always be the same between raw measurements but z will vary: for raw measurements based on segment measurements of the suit z will be 69, while for angle-based measurements it will be 66 and for sensor readings it will be 51.

11.3.1 Results Discussion

A consistent finding in all of the above graphs is that there are three raw measurements that are far and away better than the others: 'jointAngle', 'jointAngleXZY', and 'sensorMagneticField'. These measurements can be seen to perform better for all three output types we are training towards: D/HC classification (graph 1), overall NSAA score (graphs 2 – 5), and individual activities classification (graphs 6 – 7). Both 'jointAngle' and 'jointAngleXZY' raw measurements were to be expected to perform this well, as this is consistent with results obtained in experiment sets 1 and 2 (note that this isn't using exactly the same files, as experiment sets 1 and 2 use joint angle exclusive files that were pre-extracted before this project's inception, while the models trained on 'jointAngle' here were instead measurements extracted from the 'AD' files and represent somewhat different subjects than the 'JA' or 'DC' files).

Position is another measurement that performs particularly well compared with the other 5 measurements, though for some metrics (e.g. MAE for overall NSAA score) it performs noticeably worse than 'jointAngle', 'jointAngleXZY', and 'sensorMagneticField'. We can observe that these 4 'useful' measurements predict on average **97%** of test sequences accurately for D/HC classification, compared with only **73%** for the other 5. Furthermore, when we look at models trained for overall NSAA score regression, we note an average MAE of **1.5** for the useful 4 measurements, while we see an average MAE of **4.3** for the other 5; and for single-activity classifi-

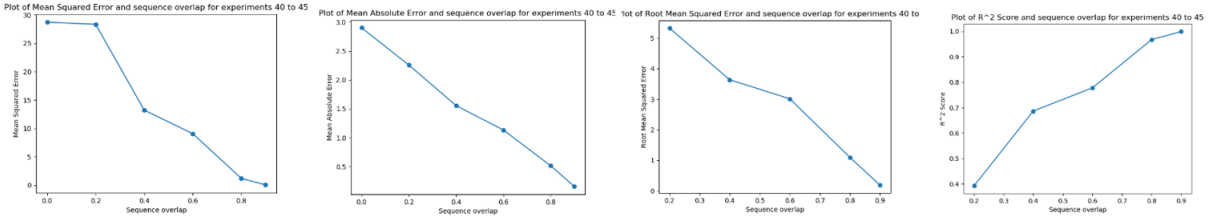
cation for the individual activity accuracy, we see **97%** for the useful 4 and **74%** for the other 5. With respect to the performance of the useful 4 measurements, while we expect to see this performance for 'jointAngle' and 'jointAngleXZY', the usefulness of 'sensorMagneticField' and 'position' is slightly more surprising. For position, we can speculate this to be as a result of subjects with more severe Duchenne being more inclined to have their limbs and torso in positions which are very indicative of their condition, compared with the healthy-control patients. The 'sensorMagneticField' measurement, however, is slightly more mystifying and does not have an obvious explanation at this point. What's more, our initial speculation was that the velocity and acceleration measurements would be fairly useful to distinguish between D and HC subjects and, moreover, help predict their overall NSAA scores, as we believed that, as these are key measurements of a subject's potential for movement, they would be useful for training an RNN model. We found these, however, to perform comparatively badly on test sequences, which leads us to believe that either velocities and acceleration of movement between subjects don't vary that much or that they aren't as indicative of movement ability as we thought.

Therefore, the conclusions we draw are as follows:

1. 'jointAngle', 'jointAngleXZY', 'position', and 'sensorMagneticField' are useful raw measurements for building RNN models on, while 'velocity', 'acceleration', 'angularVelocity', 'angularAcceleration', and 'sensorFreeAcceleration' are not.

2. sensorMagneticField' is surprisingly well performing (more or less just as good as the 'jointAngle' measurements), while 'position' is still strong but slightly worse, and 'velocity' and 'acceleration' aren't as useful as initially thought.

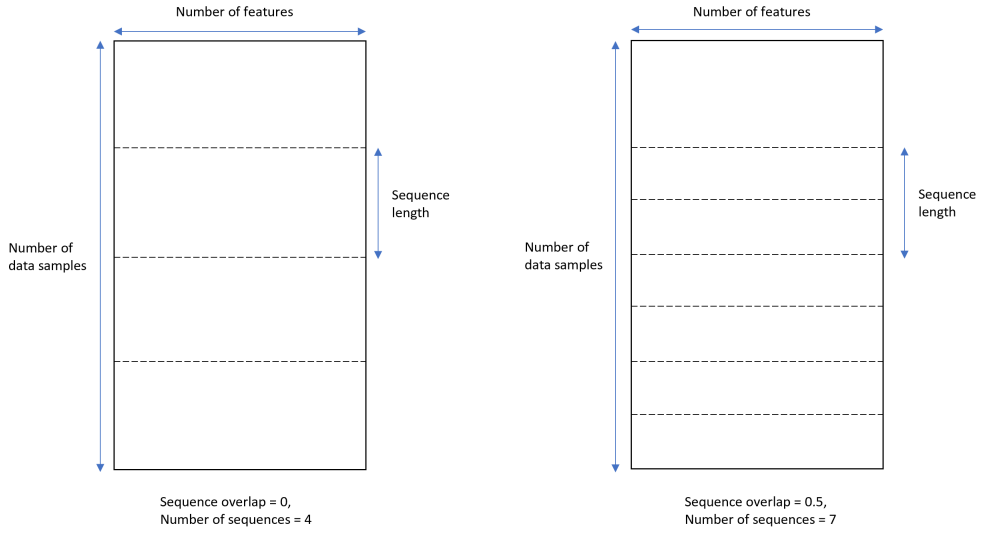
11.4 Experiment Set 4: Sequence Overlap for Stat Values from AD Files



One of the problems with stat values extracted from 'AD' files via the 'comp_stat_vals.py' script was that it dramatically reduced the amount of available data we have for training: when we are computing the stat values over intervals of 1 second (the standard measurement used for this project thus far), we are essentially doing calculations over 60 rows of raw measurement data (due to the sampling rate of the suit being 60Hz and we chose 1 second's worth of data) and producing as an output 1 row of data. While this theoretically contains much of the useful information of

the 60 rows simply condensed into 1 row, it doesn't change the fact that we have now reduced the amount of raw data that we feed into the RNN 60-fold. This is most likely a large factor in the comparatively-weak results of computed stat values seen in experiment sets 1 and 2. However, a way we chosen to get around this is by using a sequence overlap.

Sequence overlap is essentially used here as follows: consider a 2D block of data that we have available that is produced by the 'comp_stat_vals.py' script and 'ft_sel_red.py' (to reduce the dimensionality of the data). This block has a number of rows equal to the number of produced rows of statistical value data over all files that we are using (e.g. all NSAA files available in the directory) and a number of columns equal to the reduced dimensionality produced by 'ft_sel_red.py' (e.g. from 4000 from 'comp_stat_vals.py' to 30 to be read in by 'rnn.py'). Normally, we will take 'slices' of this block to produce smaller blocks with a number of rows now equal to 'sequence length' before moving on to the next block below it until all data is consumed (ignoring leftover data at the end of the block that won't fit into a smaller block). This is the case with the diagram below on the left.



However, if we consider a sequence overlap of 50% (i.e. overlap proportion of 0.5), we are instead able to capture 7 blocks of data rather than 4. Scaled up to smaller sequence lengths relative to the number of data samples, this is approximately 50% more data produced by 'comp_stat_vals.py'; if we scale up to an overlap of 0.9, we have 900% more data. Though we end up with a fair bit of redundancy with this approach (as the same vector of a data sample is used in numerous sequences), there are two primary benefits of doing this:

1. Much more available data, which is useful to train the models to become much more accurate.

2. With sequences that have no overlap, there is a high chance that one or more activities that occur in the NSAA assessment might be 'cut' along these lines

(see the dashed lines in the above left image); this would mean the activity isn't included in an entire sequence, while with a sufficient sequence length and sequence overlap, it's more likely to be captured in its complete form in at least one of the sequences.

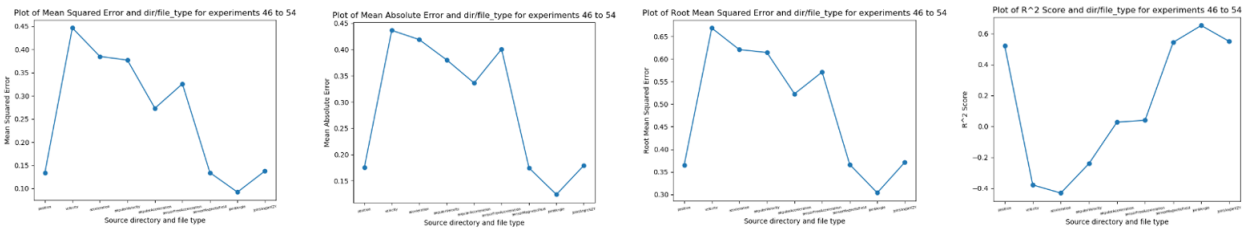
11.4.1 Results Discussion

Note that for this experiment set, we are only concerned with the overall NSAA score regression output type, as previous results show the performance of models on this output type are generally very consistent with results on other output types, hence there isn't expected to be any need to repeat these experiments with the other output types here. Unsurprisingly, we see a massive increase in the performance with a larger sequence overlap. This will be in part due to the massive increase in available data: while the total data for NSAA files from stat values produced by 'comp_stat_vals.py' for a sequence overlap of 0 is (742, 10, 30) and shows a **MAE = 2.9**, when the sequence overlap increases to 0.9 we have total data of shape (7420, 10, 30), ten times as many available sequences, which results in a **MAE = 0.1**. Note that these results are still just on test data, so it shows that the models can generalize much better to new, unseen sequences when the overlap is much higher.

We can therefore conclude two things:

1. A large sequence overlap leads to better performance for computed stat values from 'AD' files for NSAA files.
2. This is most likely due to the larger amount of available data and the increased window to capture complete activities.

11.5 Experiment Set 5: Sequence Overlap for Stat Values from AD Files



This is the first experiment set that makes use of 'mat_act_div.py' to use the source '.mat' files and the annotated Google sheet to experiment with what we call 'single-act' files. These are files that contain only a single NSAA assessment within it and no other activities. For example, if we take the 'D4' source '.mat' file, we then create 17 new files containing the 17 NSAA assessments, with each new file being a name

like 'D4_act1.mat', 'D4_act2.mat', etc. This process obviously contains files that are a lot smaller than the source file, as well as cutting out a lot of the bits in between the assessments in the original file. For example, the original file will contain a complete recording of the suit data for a subject; this includes the data where a subject isn't doing anything particularly relevant (e.g. standing around before beginning the next activity of the assessment or trying but failing to do one of the assessments). For more details on how this is done, see the section within the 'Script Ecosystem Overview' chapter on 'mat_act_div.py'.

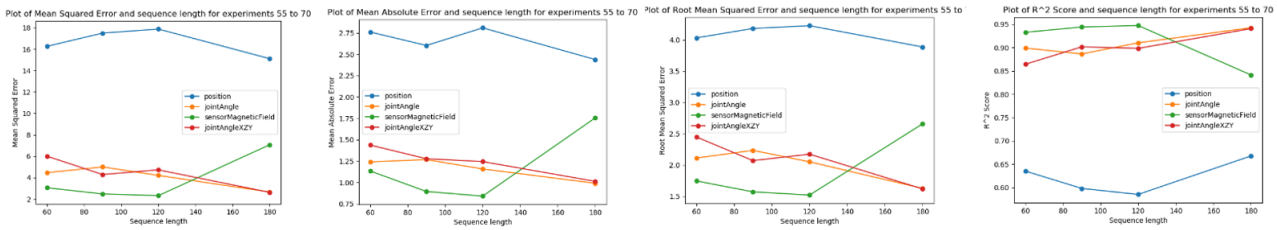
The aim of this experiment was to build models that are trained on one type of raw measurement. This is similar to experiment set 3 but with two differences: we're now assessing on the single activity score for the file in question (i.e. a value of 0, 1, or 2), and we're now using files that contain a single activity from a subject rather than all of the assessments (in most cases, though some smaller source files exist that contain only a handful of activities). We wanted to see whether or not the same raw measurements were as comparatively useful for single act source files as they were for the original files that contained all of the data; in a sense, we wanted to see if the measurements were as consistent when we removed a lot of the 'non-assessment' data from the source files (that is, data within the source files that don't have an activity being undertaken). If this was the case, then we can conclude fairly conclusively that these raw measurements are informative for NSAA assessments. Also, it's worth noting that the 'y' labels assigned to each sequence coming from a 'single-act' file is now based on not only the file name but also what activity it represents; these two features are used to lookup the relevant value from the annotated Google sheet to get the assessed value of the activity.

11.5.1 Results Discussion

As expected, the raw measurements that proved to be the 'best' in experiment set 3 ('jointAngle', 'jointAngleXZY', 'sensorMagneticField', and 'position') proved to be the best here as well, with the other 5 raw measurements heavily underperformed in comparison. This can be seen by the 'useful' 4 raw measurements obtaining a **MAE = 0.15** while the others obtaining a **MAE = 0.37**. Hence, this performs as we would expect, and the removal of a lot of the data between assessments from the source files does not affect the relative importance of some of the measurements. We can therefore conclude the following:

1. The relative importance of 'jointAngle', 'jointAngleXZY', 'sensorMagneticField', and 'position' as raw measurements compared to the other 5 is consistent with using just 'single-act' files rather than the complete source files.

11.6 Experiment Set 6: Different Sequence Lengths w/ Overlaps for Raw Measurements



For raw measurements, we have been (up until now) using a sequence length of '60' as default for all of the experiment sets thus far. This represents a time window for the sequence of 1 second, as the suit is sampling at 60Hz, which therefore means that there are 60 rows of data stored in the source '.mat' files every 1 second. Therefore, we wish to know whether or not this is a good time window to create each sequence with; however, with a given amount of source data, if we increase the sequence length, then we decrease the number of overall samples we can take from this finite block. Therefore, it was decided to use a scaling sequence overlap; that is, we increase the sequence overlap in proportion to the sequence length to keep the number of samples more-or-less constant (accounting for differing numbers of left-over data at the ends of every file that can't be made into a sequence). This is done by the following formula: $y = 1 - \frac{1}{x}$, where $y = sequence\,overlap$ and $x = proportion\,of\,original\,sequence\,length$.

For example, given our original sequence length of '60', if we wish to experiment with a sequence length of '180', then $x = 3$ (as 60*3=180) and therefore $y = 0.6667$; that is to say, we must set the sequence overlap proportion to 0.6667 to maintain a constant amount of samples (i.e. number of sequences) if we want a sequence length of '180'. The reason we do this is to hold the dataset set size as a constant, and therefore any changes in performance of differing sequence lengths would be due to sequence length alone. Also note that we again only assess the differing sequence length on the overall NSAA score output type (for reasons previously mentioned in the experiment set 5 discussion). Finally, we wish to explore differing sequence lengths on multiple different types of raw measurements and compared them side by side (hence why we plot them all together on the same graph); however, we exclude 5 of the less 'useful' measurements, as they've been determined in experiment sets 3 and 5 to be not as useful as the other 4; hence to save on time and graph readability, we don't consider them here.

11.6.1 Results Discussion

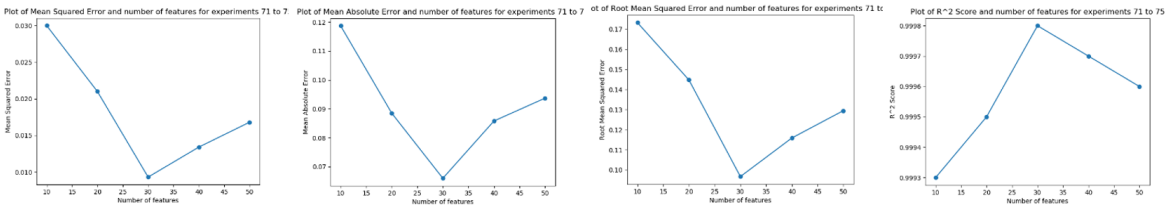
What we were looking for in the above graphs is a consistent pattern for a given graph among the four plotting lines of an increase or decrease in performance when we change the sequence length; in other words, whether performance increases or

decreases in the same way for all raw measurements for a given sequence length. This would give us a strong indication of how sequence length alters the performance of a certain metric. However, not only can we not see any inter-raw-measurements patterns of change (i.e. patterns of a line changing over time) but we can't see any discernible intra-raw-measurements pattern (i.e. can't see any whole single lines changing particularly strongly with respect to the output metric and consistently with the other single lines). For example, looking at the graph for the MAE metric, none of the lines for any of the raw measurements show any consistent improvement or worsening with increasing sequence length; each of them improves at some point on increasing the sequence length and also worsens at some point on increasing the sequence length. Additionally, the lines aren't consistent with each other: the improving or worsening performance with increasing the sequence length is often exclusive to one measurement and doesn't necessarily translate to the others. From this, we can conclude:

1. Increasing sequence length for data from raw measurements beyond 60 doesn't show a consistent increase or decrease of performance amongst different useful raw measurement types; this is most likely due to the capturing window of 1 second being long enough to correctly assess for a given output type.

2. To save on computational cost of training and testing on unnecessarily-longer sequences, we decide to keep the sequence length of 60 for the time being.

11.7 Experiment Set 7: Number of Features Needed for Stat Value Data



We now turn our attention back to stat value data from 'AD' files (i.e. the output of 'comp_stat_vals.py' and 'ft_sel_red.py' as opposed to raw measurements from files obtained by 'ext_raw_measures.py'). As a requirement of the pipeline, the data produced by 'comp_stat_vals.py' needs to have its dimensions dramatically reduced while keeping as much of the inherent variation of the data set still present. In the context of the data block diagram as seen in the section for experiment set 4, this is the 'width' of the block. As sequences are extracted from the source data 'block' by moving downwards, the number of features we chose to set is independent of the sequence length, sequence overlap, and discard proportion (more on this later), and so doesn't affect the number of samples used in any of the experiment sets, including this one. The intention here, therefore, is to experiment with the making of a different data shape to feed into the RNN models (e.g. from (7420, 10, 30) to (7420, 10,

50)) and see how the performance of the models tested on the test set differs from the initial data shape.

We have, up until now, been using '30' as the default number of features per frame and per sequence. This is due to 30 features being the number of features that generally included over 99% of the variance when using the 'variance threshold'; hence, this was a good default number of features to use for stat value files for any of the choices of feature reduction techniques. Also, it should be noted that the above results are for when the choice of feature reduction technique for 'ft_sel_red.py' was 'pca'; that is, every result above (and also for every previous experiment conducted) has used PCA (principal component analysis) to reduce the dimensions of outputs of 'comp_stat_vals.py' to a smaller value. Further exploration of different feature selection/reduction is possibly extension work for this project, though PCA was chosen here as the default feature selection/reduction technique based on success seen in similar studies.

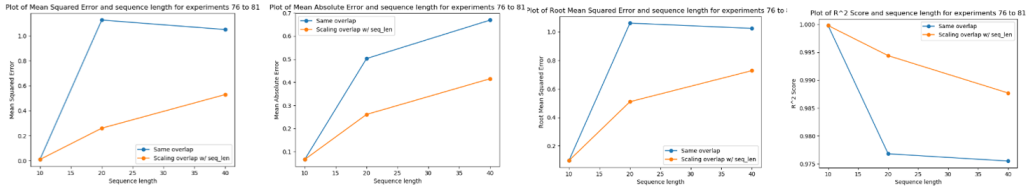
11.7.1 Results Discussion

Note that the above graphs are just for a single output type (the overall NSAA score) for reasons previously outlined and we are only interested in feature reduction for the statistical values from 'comp_stat_vals.py'. From observing the above graphs, we can see a dramatic reduction in performance when we reduce the number of features from 30 to 10: we go from **MAE = 0.1** to **MAE = 0.17**, an increase in error of approximately 70%. This is most likely due to 10 features being not enough for PCA to capture the vast amount of the variance within the data it is given; hence, when we feed this data into RNN models, it is not as able to make accurate predictions of overall NSAA scores because it doesn't have as complete a picture as if it's using 30 features. Alternatively, if we increase the number of features from 30 to 50, we can see a notable worsening of performance (from **MAE = 0.1** to **MAE = 0.13**); hence, even though by using 50 features and therefore capturing more of the inherent variance and therefore 'characteristics' of the original data from 'comp_stat_vals.py', this is evidently not enough to overcome the consequences of using higher-dimensional data for sequences to overcome the effects of the 'curse of dimensionality'. Therefore, the following conclusions can be drawn:

1. We shall continue with number of features to reduce to from data produced by 'comp_stat_vals.py' to 30.

2. This is the ideal 'middle ground' between sequences that don't capture enough of the variance (e.g. number of features = 10) and sequences that have too many features to effectively train on (e.g. number of features = 50).

11.8 Experiment Set 8: Larger Sequence Lengths w/ Overlaps for Stat Values from 'AD' Files



In the same way that we looked at increasing the sequence lengths used for raw measurements in experiment set 6, we now look at the same thing but from computed statistical values of source files by 'comp_stat_vals.py' and subsequently reduced to '30' features (as determined by experiment set 7). This time, however, we look at the effects of scaling the sequence overlap with sequence length (as done in experiment set 6) compared with not scaling the sequence overlap. We start with a sequence overlap of '0.9' as standard (the value of which was determined by experiment set 4) for the initial default sequence length of 10. Note that the initial sequence length of 10 was chosen due to the initial lack of data available to use before the use of sequence overlaps. Also, it's worth pointing out that this length of 10 corresponds to a sequence capture window of 10 seconds (this is due to every 'row' of stat value data representing 1 second's worth of data due to them being calculated over 60 rows of raw data sampled at 60Hz from the suit), unlike the raw data whose sequence length of 60 corresponds to a sequence capture window of 1 second. We also again assess only on the overall NSAA score output type for reasons previously discussed.

The primarily purpose of this experiment set is to see if the performance of the models improved with a longer sequence length for computed statistical values of files. However, the secondary purpose is to determine if a scaling overlap improves the performance of the model, as was previously assumed to do so for experiment set 6. This scaling overlap keeps the number of samples as a constant, as opposed to it decreasing due to the longer sequence lengths capturing more of the data per sequence. The experiments that have a scaling sequence overlap to keep the number of samples more-or-less constant are represented by the orange lines in the above graphs, while the ones with no scaling sequence overlap (and thus a progressively reducing number of overall samples) are represented by the blue lines.

11.8.1 Results Discussion

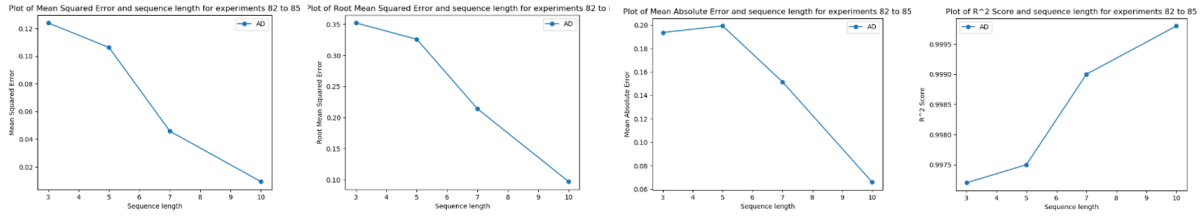
We can see that in both scenarios where we either have a scaling sequence overlap or don't, the performance of the model decreases with increasing sequence length. This is also significant because, for raw measurements, we can see that sequences are quite able to use longer sequence lengths and that '60' seems to be an ideal point for these measurements. Therefore, from a smaller sequence length being more ideal for computed statistical values, we can conclude that this is a result of each

row of data for computed statistical values containing more contextual data than a single row of raw measurement data, as opposed to it being down to the ideal data shape for an RNN to learn from (which is not the case as an RNN can learn quite well for a sequence length of '60', as seen in experiment set 6). This is most likely due to the data contextual window growing too large. For example, when we increase the sequence length to 50 here, that means that each sequence takes in 50 seconds worth of source data, which most likely isn't able to account for the minutia of details that differentiates sequences of different classifications, overall NSAA scores, etc. Additionally, our reasoning of using a scaling sequence overlap in here and in experiment set 6 is justified, as for every metric above, the increased sequence length performs better with a scaling sequence overlap, as opposed to having a static sequence overlap of '0.9', though in both cases it is still not as good as keeping the sequence length to 10. Therefore, we can conclude the following:

1. Increasing the sequence length beyond 10 is not ideal for computed statistical values, which shows that a contextual window of beyond 10 seconds makes learning for an RNN increasingly difficult.

2. The use of a scaling sequence overlap shows better results than not using a scaling sequence overlap.

11.9 Experiment Set 9: Smaller Sequence Lengths w/ Overlaps for Stat Values from 'AD' Files



Before moving onto the examining of larger sequence overlaps for raw measurements as opposed to just for extracted statistical values (as in experiment set 8), it's necessary to ensure that a sequence length of 10 is indeed ideal for our setup, as the previous experiment set only concurred that a sequence length of 10 or larger could be the ideal length. Here, we look at these smaller sequence lengths, going down to a sequence length of 3 (i.e. three sequences of vectors of 30 numbers are fed into the RNN which represents a context windows of 3 seconds). The expectation prior to undertaking these experiments was that a sequence length of 3 would be too short to draw time-contextual inferences from by the RNN when learning, while it was felt that 3 seconds might be too short to make accurate predictions for D/HC classifications, overall NSAA scores, etc., for many sequences. Note that everything else remains as it was in experiment set 8 (i.e. same source directory, same output type for the models, etc.), with the above graphs showing the results when we are using

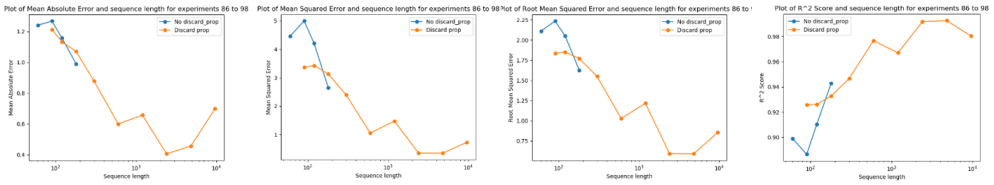
a scaling sequence overlap to keep the number of samples as a relative constant (i.e. decreasing the overlap as we decrease the sequence length), which corresponds to the orange lines in the graphs in experiment set 8.

11.9.1 Results Discussion

As we can see in the above graphs, the results evidently concur with our predictions: the performance for the overall NSAA score output type for models trained on extracted statistical values peak at a sequence length of around 10, with decreasing performance shown when we lower this value while keeping the number of samples relatively constant. As mentioned previously, this is most likely due to an RNN generally thriving on sequences of longer than 3 to get time-contextual information along with a context window of 3 seconds being not as useful as a 10 second window for NSAA scores. Furthermore, this knowledge that 10 seconds of context window for NSAA activity sequences as generally most useful will carry us over into the following experiment, where we shall attempt to use a similar context window with raw measurement data to see if we get comparable results. Therefore, we can conclude the following:

1. Smaller sequence lengths for models built on extracted statistical values show worse performance, and therefore a sequence length of 10 for these sequences is ideal.

11.10 Experiment Set 10: Very Large Sequence Lengths for Raw Measures w/ Discard Proportion



Having established that a 10 second contextual window was ideal for extracted statistical values, we now wish to see if this is something that is exclusive to computed statistical values from 'comp_stat_vals.py' or whether it exists for raw measurements as well; if it does, then the 10 second context window is more likely to be an inherent characteristic of sequences made from NSAA assessment files, regardless of what measurement types are drawn from these files. For raw measurements, each row of data that is going into the RNN represents 1/60th of a second, as was previously established due to the sampling rate of the suit producing the data being 60Hz. Therefore, to get 10 seconds worth of data from raw measurements, we would need 600 rows of data (i.e. a sequence length of 600) while computed statistical values only need a sequence length of 10. However, setting the sequence length to 600 for raw measurements presents us with two problems:

1. The increasing of the sequence length for raw measurement data from 60 (as we have used as the default up until this point) to 600 will in turn reduce the data 10-fold due to there being far more of the data needed for each of the sequences.

2. A sequence of several hundred has been previously established to be difficult for an RNN model to train on, along with being much more computational demanding and containing a lot of possibly redundant data.

The first problem is solved by using a sequence overlap of 0.9 (for increasing from a sequence length of 60 to 600); this keeps the number of samples at a relative constant as we increase the sequence size. This ensures that any changes that occur in performance will be down to just sequence length and not the number of samples used; see experiment set 4 for a more in-depth discussion of this. This allows us to change the data shape from (13365, 60, 66) to (13365, 600, 66). However, this does not solve the other problem of there being too-long sequences to feasibly train the RNN model on. What we want is a way for an RNN model to gain a longer contextual window (e.g. of 10 seconds, corresponding traditionally to a sequence length of 600) without lengthening the sequence length itself. In this sense, any increase in performance while increasing the context window will be entirely down to increasing the RNN's contextual information for a given sequence and not the actual sequence length. This is done by the using '–discard_prop' optional argument supplied to 'rnn.py'.

What this does is fairly simple: for each sequence, once it has been extracted from the source data block (also accounting for sequence overlap if appropriate), we keep only every 'nth' line of the sequence. For example, assume we have a sequence of length 600 and have set the '–discard_prop' argument to 0.9. This means that 90% of the data of the sequence is discarded, which results in keeping every 10th line in the sequence. This achieves two things:

1. Discards 540 rows and we are left with 60 rows of data which, as a result of taking the 'kept' rows at even increments along the sequence, means that we still have context covering the 600 rows.

2. By having even sampling of 'kept' rows of data, we also limit the possibility of missing any 'important' pieces of data that existing within the original 600 rows, as it's likely that 1 or more of the kept lines would have captured some of this information.

We therefore use this technique (along with a corresponding sequence overlap) to allow us to experiment with much larger sequence lengths (up to 9600) with corresponding discard proportions to keep the actual sequence lengths (i.e. the number of frames per sequence going through the model) as a constant; these are shown by the orange lines in the graphs above. We also look at not using the discard proportions for increasing sequence lengths, though we can only go up to a certain sequence

length before computational limits restrict us from continuing; for example, training models with data of a sequence length of 9600 with no discard proportion was expected to take between 3 and 4 days to train the model with the same number of epochs as the previous ones.

It's worth noting that the sequence length is scaling for all experiments done here so the number of samples is always kept relatively constant. This is true for both the blue and orange lines above, as in both scenarios with a sequence length of 120, the discard proportion is set to 0.5. Finally, we only use one raw measurement here (joint angles) to experiment with longer sequence lengths. This is mainly due to previous experiment sets showing us that the performance of the useful 4 raw measurements are almost always completely in-line with each other with respect to increases or decreases in performances with changing of independent variables. Therefore, to save on expensive experimentation time, we chose to evaluate the performance of models on only one raw measurement, safe in the knowledge that other raw measurements will most likely see the same change in performance.

11.10.1 Results Discussion

One thing we can see from the graphs above is the noticeable increase in performance (e.g. the lowering of MAE on the test set) up to a sequence length of approximately 600. This is the point on the graph of the MAE metric where the discard proportion line shows an upwards inflection afterwards (i.e. when going up to a sequence length of 1200, the MAE gets higher); furthermore, past this point in the graph, the improvements are comparatively minimal for the extra time it takes to pre-processing the data. With very large sequence lengths and corresponding discard proportions, even though the amount of data that is input into the RNN models is constant, it takes longer and longer to prepare the data. Therefore, a compromise was reached in the **sequence length of 600** being the best choice given the proportional discard proportion and sequence overlap for how long it takes the model to compute.

This is also a good length to have as it means the 'ideal' (in the sense of model performance and time needed to compute) context window for raw measurements with a high discard proportion and scaling sequence overlap is 10 seconds, which is the same as for computed statistical values. Another thing to note is that, for sequence lengths where we have results for both the blue and orange lines in the graphs, we see that the orange line (representing using a discard proportion) shows better results. This shows that, given the same sequence overlaps, when using a discard proportion to reduce the sequence length that is used to train the models we see better model performance; this is most likely due to RNN models not performing well for sequences of several hundred or longer. We can therefore conclude the following:

1. The performance of models increases with higher sequence length and scaling discard proportion, with a good compromise of performance and computing

cost being a sequence length of 600, a sequence overlap of 0.9, and a discard proportion of 0, which gives an 'actual' sequence length seen by the model of 60.

2. The ideal context window of 10 seconds for extracted statistical values is more-or-less replicated here, with the 10 second context window (sequence length of 600) being a high-performance parameter setting, and therefore 10 seconds is a good window for data from NSAA files in any measurement form.

3. Using discard proportion over not using it for identical sequence lengths shows better performance, which indicates that RNN models struggle to perform on sequences of too-high length.

Chapter 12

Model Predictions Sets

12.1 Model Predictions Set 1: Natural Movement Files on Models Built on NSAA and Walk Files

MAE between true and predicted NSAA scores over files (NSAA) = 4.66,

MAE between true and predicted NSAA scores over files (6minwalk-matfiles) = 5.03

Percentage of correct predicted file D/HC label (NSAA) = 75.92%

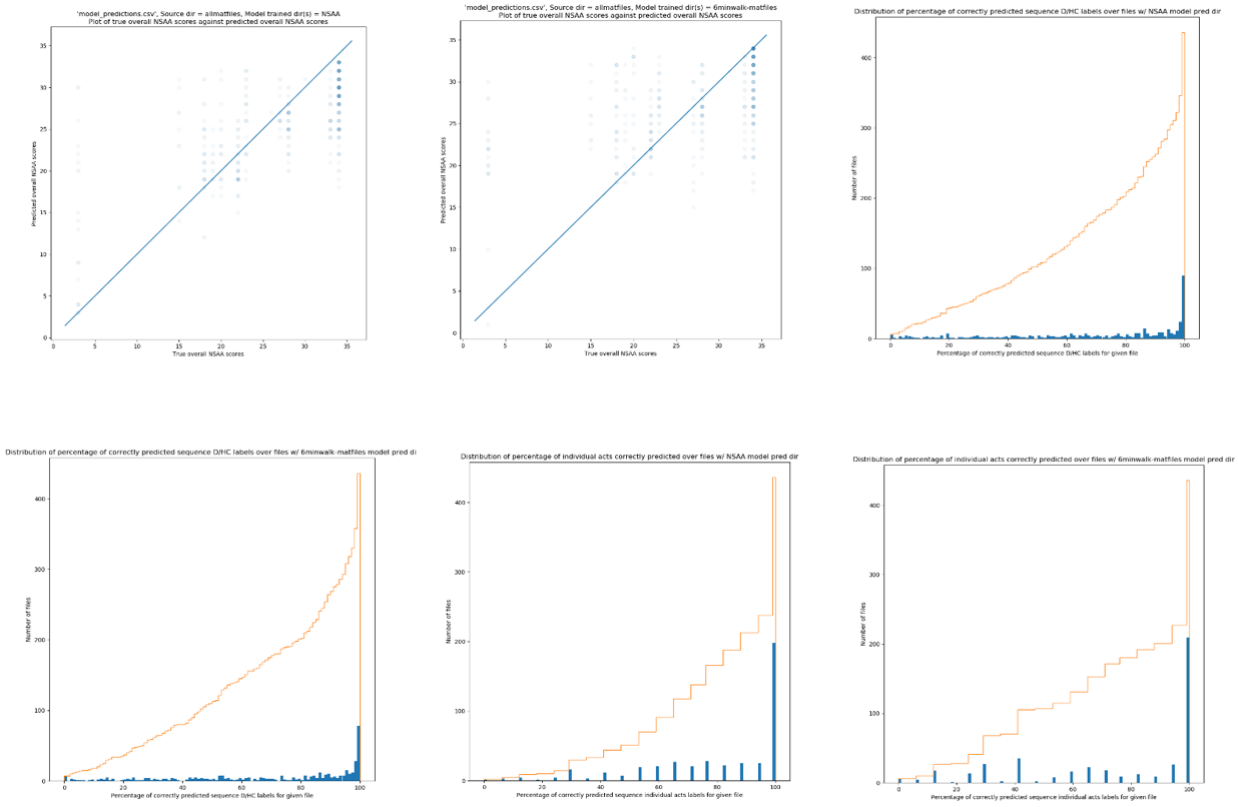
Percentage of correct predicted file D/HC label (6minwalk-matfiles) = 74.54%

MAE of percentage predicted wrong sequence D/HC classification over files (NSAA) = 28.72

MAE of percentage predicted wrong sequence D/HC classification over files (6minwalk-matfiles) = 29.28

Average percentage of single acts correctly predicted over files (NSAA) = 80.84%

Average percentage of single acts correctly predicted over files (6minwalk-matfiles) = 75.28%



With the models now built with many of the settings that we will most likely be sticking with for the duration of the project (i.e. with an ideal sequence length, overlap, discard proportion, the best raw measurements to use, etc.) ascertained by the previous experiment sets, we now move onto using these along with 'model_predictor.py' to predict on whole files. Previously, in all experiment sets, the data that was reported was testing sequences from a mixture of the source files used to build the data set. This meant that the assessment was done on a sequence-by-sequence basis, where the assessment of each sequence was to be independent of each other. With 'model_predictor.py', this is quite different: we instead provide a specific name for the file, which will use all of the data to do an assessment on (the difference being that for the previous experiment sets, there will be a mixture of target D/HC classifications, overall NSAA scores, etc. for the test sequences as they came from multiple different files, whereas with 'model_predictor.py' there is a D/HC classification, overall NSAA score, etc., that is common among all of the sequences within the source file.

For making predictions on whole files via 'model_predictor.py', the broad sequence of steps is as follows:

1. Split the source file into sequences, each with the same y labels (as the same file has the same y labels attached to it for each of its sequences).

2. Assess each of these sequences on each of the models that we wish to be using, with a model for each measurement type and each output type (e.g. if we are

assessing on 4 measurement types and 3 output types, there will be 12 models that each sequence is assessed on).

3. For each of the output types, average the response over all sequences for a given measurement and then average these over all measurements to get a single prediction for the given output type.

Each assessment of a file made by 'model_predictor.py' is then stored as a single row of results within 'model_predictions.csv'. This is what's used for the following model predictions sets.

For the first set, we are concerned with every file that we currently have of subjects as the 'allmatfiles' data set involved in what we call 'natural movement behaviour'; that is, data captured by the suit of the subject doing activities that aren't NSAA or 6-minute walk assessments, such as playing or eating. This amounts to over 400 files, with as many as 30 files for a single subject captured. Hence, for each of these files, we run it through 'model_predictor.py' to make various predictions for different output types. This is helped by the use of 'test_altdirs.py', which automates a lot of the process and doesn't require the user to manually run 'model_predictor.py' $>$400 times.

While we shall get to assessing NSAA files on models built on NSAA files in the next model predictions set, here we wish to look at how well models that are built and tuned on NSAA files perform when presented with natural movement behaviour files. Obviously, the models have never seen any part of the files before (as the assessing files are from a different data set as those used to train the models), and the data it is now assessing on does not necessarily have characteristics of the original NSAA assessment files; for instance, the models will have been trained on sequences from files that do specific activities (i.e. one of the 17 NSAA assessment activities) or contain the subject walking. However, the natural movement behaviour will most likely be of movement characteristics that it's never seen, and so the models are required to generalize their knowledge to other types of movement.

12.1.1 Results Discussion

Console Output

The following covers the model predictions of rows 1 to 436 of 'model_predictions.csv; for information on a prediction-by-prediction basis, see these rows. Instead of showing the results contained in these rows individually, however, we instead computed several statistics from these rows for certain columns with the aim to provide an insight into how well models build on NSAA and 6-minute walk files predict on natural behaviour files. It's also worth noting that, at present, the natural movement behaviour files only exist as raw joint angle files. Hence, the only raw measurement type of model we can use are the ones built on joint angle data (unlike predictions on NSAA files we shall discuss later, which uses 5 total input types for models).

The first two console outputs as seen above takes the average of the difference between each file's true value for overall NSAA score and predicted value. This is done by, for each relevant row of 'model_predictions.csv', finding the absolute difference between the true and predicted overall NSAA value columns; this gives a measure of how well the model predicted the overall NSAA score for that file, with this difference being 0 if it predicted the correct score; this is then averaged over all the relevant rows. What we can see is that models built on NSAA files that predict natural behaviour files predict a file on average within **4.66** of its true value, whereas models built on 6-minute walk files predict within **5.03** of its true value. These are reasonably strong initial results, though are possibly slightly skewed given the prevalence of files within the 25 to 34 score range. This is also including every single natural behaviour file available, and so the scores might be negatively impacted by natural movement files that are more-or-less not possible to infer an exact score (e.g. a file of a subject sitting very still for long periods).

The second two console outputs are the percentage of the total number of file assessments made that have the correct D/HC predicted classification. For each of the files, 'model_predictor.py' takes the most common 'D' or 'HC' label for the sequences for the D/HC output type and makes that the prediction for the file; for example, for a file that contains 100 sequences, if 60 of them are predicted as being from a 'D' file and 40 are predicted as being from an 'HC' file, then the overall prediction for the file is a 'D' label. We also have a 'true' D/HC classification for the file; this is just based on which of the two the file's name begins with. Hence, the percentage of correctly predicted D/HC labels is the percentage of rows in 'model_predictions.csv' that we are concerned with that have a predicted D/HC label that matches the true D/HC label. The results of **75.92%** and **74.54%** shown that in a majority of cases, natural movement files assessed on models built on either NSAA or 6-minute walk files get a correct classification; this is a notable good result that indicates, with further refinement and tuning, that these models would identify quite accurately what type of classification the subject is by simply observing natural movement data, even if the models were trained instead on NSAA files.

With the third set of console outputs, we are still concerned with analysing the D/HC classification predictions. Unlike the second set, for each file we aren't concerned with the single estimated classification of the file by the models; rather, we look at it on a sequence by sequence basis in that we want to see the percentage of correctly classified sequences for the file being tested on. We then compute the 'percentage of predicted wrong sequences'; this is simply the difference of the percentage of correctly classified sequences and 100%. For example, for 'allmatfiles\D10-001', we have 77.81% sequences predicted as being of 'D' label and 22.19% predicted as being of a 'HC' label. Since they are all supposed to be 'D' sequences (since they all came from a 'D' file), it therefore got 22.19% of the sequences wrong: this is the 'percentage predicted wrong sequences'. We then repeat this over all other files from the natural movement behaviour data set and find the MAE of this set. This results in an error percentage of **28.72%** on models built from NSAA files and **29.28%**

on models built from 6-minute walk files. This is fairly similar to the second set of console outputs in that it shows the models predict correct classes the majority of the time, though ideally this will approach 0% for both model types (NSAA and 6-minute walk) as we continue to refine and improve the system.

The final set of console outputs are more straightforward. For every assessed file (i.e. row we are concerned with in 'model_predictions.csv'), we get the value contained within the 'Percent of acts correctly predicted'. This looks at the single act predictions for that file (i.e. a list of 17 values between 0 and 2 that the model believes are the correct single act scores for that file) and sees how many of those it got correct with respect to their true values; this then manifests as this 'percent of acts correctly predicted' score. We then repeat this over all the natural movement behaviour files and average these values to find the scores that appear on the fourth set of console outputs. The values of **80.84%** and **75.28%** are quite impressive: this means that on average the models that are trained on NSAA files predict 13.7 of the 17 activities with the correct score, while the models trained on 6-minute walk files on average predict 12.8 of the 17 activities. It's also noted that particularly accurate scores for certain rows generally correspond with an overall NSAA prediction that is quite close to its true value, which makes sense as they are both involved with ascertaining NSAA scores; it's just that the latter is predicting cumulative scores.

Graph Output

Along with the console output that is produced by 'graph_creator.py', as discussed above, we also have several graphs that have been produced. However, in comparison to previous experiment sets, it is somewhat harder to ascertain specific results from these graphs, as these contain 400 points and therefore must rely on the more evident 'trends' shown within the graphs. The first two graphs show the true overall NSAA scores for each natural movement behaviour file along the x-axis and the files' corresponding predicted overall NSAA scores along the y-axis by the NSAA models and 6-minute walk models. The closer these points are to the $y = x$ line projected through the middle of the plot the better, as this means they are closer to their true predicted values. From these graphs, we can see a tendency for points to hover around this line; however, there is still a great deal of variation around these areas. We can also see the model particularly struggles when presented with files that have an overall score of '3'; this is most likely due to these files being from an 'outlier' subject (due to the subject not completing many of the activities, which resulted in a lower score than they most likely should have). Additionally, we can see that files that have an overall score of '34' (i.e. being from an 'HC' subject) are often assigned a score a fair bit lower of between 25 and 30. These observations from the two graphs for both types of models therefore give us good guidance of where to focus on improving the models.

The next two graphs show the distribution of the percentage of correctly predicted sequence D/HC labels for every natural movement behaviour file we tested on. This is essentially looking at either the 'Percentage of predicted 'D' sequences' or 'Percent-

age of predicted 'HC' sequences' for each file (depending on whether the file is a 'D' or 'HC' file). We then plot the distribution of these percentages for each of the file predictions made for both types of models (NSAA or 6-minute walk), along with plotting the cumulative distribution lines. We can see from these graphs that there is high number of files where 100% of its sequences were predicted with the correct label, which is doubly impressive given the models these files are testing on have never seen natural movement behaviour before. It also shows us that, in both model types cases, there is a wide distribution of scores, with a not insignificant number of files having fewer than 50% of their sequences correctly classified (100 files in both cases out of 400 total). These graphs highlight that, while we have seen fairly positive results for the classification as shown in the second set of console outputs, there is still a lot of room for improvement.

The final two graphs again show a distribution of files. This time, however, we are looking at the distribution of percentages of correctly predicted sequence individual acts labels (which corresponds to the distribution of scores used to compute the fourth set of console outputs). Here, we can see a noticeable difference between the two types of models: while the models built on NSAA files have 50 files that have ¡50% of the correctly predicted sequence individual acts labels, models build on 6-minute walk files have 100 files that have ¡50% of the correctly predicted sequence individual acts labels. This is particularly impressive with regards to the NSAA models and shows the models proficiency to learn these individual scores. The disparity between the two model types also makes sense: the 6-minute walks don't contain any NSAA activities within them (only the walk activity) and so we are asking the model to make predictions for the individual activities for a subject when the model itself never sees any of these NSAA activities in the files; it's only presented with sequences from subjects with corresponding individual activities. Because it can't as easily draw any inferences about walk data to correspond to other activity scores, it makes sense that it doesn't perform as well compared with a model that does see these activities.

12.2 Model Predictions Set 2: Model Performance on Left-Out vs Non-Left-Out Files

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3 (already seen)	AD, Joint Angle, Joint Angle XZY, Position, Sensor Magnetic Field	100%	D	100%	15	18
D3	AD, Joint Angle, Joint Angle XZY, Position, Sensor Magnetic Field	23.53%	D	80%	15	26
D3	AD	5.88%	D	54.69%	15	28
D3	Joint Angle	29.41%	D	97.1%	15	27
D3	Joint Angle XZY	23.53%	D	81.7%	15	26
D3	Position	41.18%	D	85.94%	15	25
D3	Sensor Magnetic Field	47.06%	HC	35.49%	15	22

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D11 (already seen)	AD, Joint Angle, Joint Angle XZY, Position, Sensor Magnetic Field	100%	D	100%	27	27
D11	AD, Joint Angle, Joint Angle XZY, Position, Sensor Magnetic Field	76.47%	D	100%	27	27
D11	AD	70.59%	D	71.88%	27	28
D11	Joint Angle	82.35%	D	65.78%	27	26
D11	Joint Angle XZY	82.35%	D	74.53%	27	25
D11	Position	52.94%	D	84.84%	27	28
D11	Sensor Magnetic Field	47.06%	D	99.69%	27	26

In this model predictions set, we look at 2 subjects ('D3' and 'D11') and look at how well models perform on predicting upon them when they have seen the files in training vs when they haven't. These two subjects were chosen as their overall NSAA scores were mid-range (i.e. weren't close to a perfect '34' and weren't outliers like '3'), though we intend to repeat this for numerous other subjects in the near-future, and each table represents the results concerning a single one of these two subjects. It should be noted that the models these files are tested upon are models built on NSAA files, along with the testing files being NSAA files themselves, so the models should be familiar with this sort of data.

12.2.1 Results Discussion

We'll first examine the results of the 'D3' subject. The first row is based on models that have already seen the 'D3' file in training and not only is it familiar with the subject, but it is also familiar in the particular data it is being tested on. This is only useful to us as a 'baseline' of how well the model could theoretically perform, as this is using some of the data that has been used for training for testing purposes. This therefore holds no practical use as a metric, as files that we would present to the models in a real-world setting would obviously not have been seen by the models before. Note that this uses all the measurements that we have decided upon using (extracted statistical values and the 4 'useful' raw measurements) to make its estimations upon. Unsurprisingly, it predicts all of its sequences with the correct D/HC classification, along with all of all of the single act labels. However, it is still out on its estimation of the overall NSAA score by 3, which is somewhat surprising as not only should it be closer because the models are familiar with the file's data, but the single-act scores are completely accurate; if these single act scores are correct, then should also have an overall score of a correct value as this is simply the accumulation of these values. This discrepancy of performance of different output types is possibly due to the single-act models being better trained to deal with sequences from this file; the cause of this is prompt for further investigation and shall be a proposed solution as seen in MPS 18.

For the second line of the table, we then observe how well models predict on files that have been left-out completely of the training and testing process. This more accurately simulates what it would be like for the model in 'production'; that is, assessing on new files in its intended capacity. This also uses the same input types to the models as the first line in the table and thus aggregates the results from the different input types to make a single prediction for the whole file for a specific output type. We can see a noticeably steep drop in performance of the models: it only gets

23.53% of the single act predictions correctly, the number of correct sequence classifications drops 20% (though it still correctly determines the file to be a 'D' file) and the predicted overall NSAA score drops to being 11 away from the real score, down from 3. This is particularly poor generalization performance for this newly-seen file for the models and the **results are heavy motivation to continue with generalization techniques to generalise the models' learning inferences to new files**. It's worth noting that this was a particularly low score for 'D' files (with most falling in the +20 range), so this might contribute somewhat to its poor performance, though further investigation into other 'left out' files will be conducted later to confirm this.

The next 5 lines of the table again look at models that have had the 'D3' subject 'left out' of the model training (done by setting the 'rnn.py' optional argument to '–leave_out=D3'). However, instead of using all 5 measurements for the three output types and aggregating their predictions, we instead look at individual measurements in turn; that is, a measurement is extracted from 'D3' (e.g. joint angle) and is tested on the three that correspond to this measurement (with 'D3' being left out of all of them) for each output type. The aim with this is to determine whether or not any particular measurements are more useful for generalizing to new, unseen files. At this point, the results from these rows are fairly inconsistent: while models built from the position and sensor magnetic field measurements perform better with respect to the single-act scores output, the joint angle and joint angle XZY measurements perform better with respect to the D/HC classification output type. Additionally, there is not a noticeable disparity of improvement for the overall NSAA score among the measurements (with the exception of sensor magnetic field). Hence, **at this stage, this is not enough information to draw any conclusions regarding the most ideal measurements for left-out subjects when using 'model_predictor.py' and more of these 'left out' models for other subjects are thus needed**.

Many of the same conclusions can be drawn for the 'D11' subject table. For the models that are trained on all measurements when seeing 'D11' as a left-out file (i.e. the 2nd row of the table), however, the models are a lot better at predicting more accurate outputs: there is a much higher percent of acts correctly predicted, higher percent of correct predicted sequences, and a closer true overall NSAA score compared with that of 'D3'; here, the model predicts the 'D11' overall NSAA score correctly, rather than in the case of 'D3' where it is off by 11. This is particularly impressive for 'D11', as these models obviously have never seen this subject before. A likely cause for this increase in accuracy, however, is that 'D11' is closer to the mean overall NSAA score amongst the subject groups: **'D11' is more representative of the 'average' subject with Duchenne, hence it is easier to generalise to the true score of the subject than it would be for an outlier**. Again, however, when we look at the left-out models but only single-measurements (rows 3 to 7 of the table), we again cannot drawn any particular conclusions with respect to one measurement or another being more useful to model generalization of unseen data; again, more of these left-out subjects (rather than just 'D3' and 'D11') are most likely needed to draw any conclusions in this regard.

12.3 Model Predictions Set 3: Model Performance over the Chosen 5 Left-Out Subjects

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100%	19	27
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100%	22	25
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	27	28
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50%	33	28
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75%	34	31

With a majority of the programming work and options added to each of the scripts, we now move onto among the key stages of the experimentation: the testing of left-out subjects to assess and improve upon generalization performance of models to assess, unseen new subjects. To this end, we chose 5 subjects ('D3', 'D9', 'D11', 'D17', and 'HC6'). This is because these subjects show a great variety of overall NSAA scores and cover most of the spectrum of various scores we see in real-world subjects. The reason why we chose 5 subjects and not the totality of the subjects available (which amounts to approximately 30 subjects) is the disproportionate amount of time to build and assess the models for a limited amount of improvement. For example, if we chose to assess on all 30 subjects left-out, that would mean this model predictions set would take at least 6 times as long to complete for not much of a better an idea of generalisation performance, as the 5 chosen subjects to leave out of models cover the majority of the spectrum of the true overall NSAA scores.

For this set, we built a total of 60 models. This is due to each model corresponding to each output type (D/HC classification, overall NSAA score, individual act scores), for each measurement we are concerned with (position, sensor magnetic field, joint angle, and computed statistical values), and each set of these built for a different subject that has been left out of training (5 subjects, as outlined above). In other words, the about assessments have been done on models that specifically have not seen the subject in question, but have seen the other 4. Hence, we have $3\,output\,types\,x\,4\,measurements\,x\,5\,subjects = 60\,models$ for this experiment set.

It should be noted that there are a few changes made between the previous model predictions set, as evidenced by the differing values for 'D3' and 'D11'. The first change is that the true overall score for subjects 'D2' and 'D3' were found to be mistakenly labelled by the assessors, and so we've been working with slightly-incorrect data since then for these two subjects; the reference file which the scripts use to draw the NSAA labels has now been changed to reflect these new values. The second change is that we no longer use the 'Joint Angle XZY' measurement. This is due

to learning that this is simply a different way of displaying the joint angle data within the suit data files and provides no more useful information to the user. What's more, by including it within the measurements, it's essentially 'diluting' the impact of the other measurements when they combine their assessments.

12.3.1 Results Discussion

From the results seen above, we can see a propensity for the overall NSAA score predictions to tend towards the median overall value among all the subjects (28); however, we can still see that the lower true value overall NSAA scores for patients tend to have predicted scores lower than this median value. This implies that **there is still some generalisation ability to unseen subjects amongst the models, though not to the degree for which we are aiming**. This will be attempted to be rectified by various methods in the coming model predictions sets. It should also be noted that we will be paying special attention to generalising to outside of the median overall NSAA value; as we can see above for a subject with a true value of 19, the model struggles to generalise to this degree, which is something that needs to be looked into. It should also be noted that this model predictions set will serve as the 'standard' way of assessing these 5 left-out subjects, the performance of which shall be referenced in many of the upcoming model predictions sets when a new technique is tried to help the models generalise to unseen subjects better.

12.4 Model Predictions Set 4: Model Performance for 5 Subjects for Specific Measurements

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100%	19	27
D3	Position	58.82%	D	85.27%	19	25
D3	Sensor Magnetic Field	52.94%	D	63.84%	19	24
D3	Joint Angle	47.06%	D	90.4%	19	27
D3	AD	23.53%	D	51.56%	19	32
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100%	22	25
D9	Position	52.94%	D	96.56%	22	24
D9	Sensor Magnetic Field	52.94%	D	99.06%	22	26
D9	Joint Angle	58.82%	D	85.62%	22	25
D9	AD	47.06%	D	87.5%	22	24
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	27	28
D11	Position	41.18%	D	85.62%	27	31
D11	Sensor Magnetic Field	70.59%	D	99.06%	27	24
D11	Joint Angle	58.82%	D	72.97%	27	27
D11	AD	58.82%	D	65.62%	27	29
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50%	33	28
D17	Position	94.12%	HC	12.5%	33	28
D17	Sensor Magnetic Field	94.12%	HC	12.5%	33	28
D17	Joint Angle	76.47%	D	89.84%	33	27
D17	AD	94.12%	D	93.75%	33	28
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75%	34	31
HC6	Position	100.0%	HC	64.45%	34	32
HC6	Sensor Magnetic Field	100.0%	HC	100.0%	34	33
HC6	Joint Angle	100.0%	HC	61.72%	34	29
HC6	AD	100.0%	D	40.62%	34	30

This model predictions set is an extension of the work that was done in the previous set. It looks at the predictions for each of the subjects for their associated 'left-out' models but, along with using the aggregate predictions made by the models trained on the four measurements (now that we have decided to discount 'jointAngleXZY'), we now look at how each of the measurements predict in turn. That is to say, each subject has one measurement type taken from it and loads the appropriate three models (one for each output type) for that measurement type and that 'left-out' subject, as opposed to using all measurements as the basis of the models. The aim therefore is to see if there is any particular relationship between the measurement type and its prediction ability.

One benefit from this measurement set is that it doesn't require us to build any more models than have already been build, as it simply uses all the models built for model predictions set 3 but we simply use fewer of the models per each of the rows in the above table (disregarding the included rows from the table in model prediction set 3). The rows copied over from model prediction set 3 also serve to show how well the aggregation ability of those rows work with respect to the true values for each given subject.

12.4.1 Results Discussion

The main takeaway from this experiment set is that **there is no obvious measurement type that is 'dragging down' the aggregate predictions seen in model predictions set 3 at this stage**. For the overall NSAA score predictions, each measurement generally predicts within 1 or 2 of each other and no measurement type is generally closer to the true overall NSAA value than any of the others. The same can be said for the percentage of correctly predicted sequences (for output type 'dhc'): there are no measurements that consistently perform better than the others to predict sequence D/HC labels. We can note two things from this metric, however: the aggregation effect for the metric for output type 'dhc' works in our favor, as for 4 of the 5 subjects it increases the average percentage from its predictions using single measurements to something higher with the aggregate prediction (for example, for subject D11, the average of the 4 single-measurement predictions was (85.62% + 99.06% + 72.97% + 65.62%) / 4 = 80.82%, while the aggregate prediction over all 4 measurements was 100%). This suggests to us that **the aggregation effect of all measurements to cover outlying predictions is useful at least for output type 'dhc' at this point in the experimentation**. This, however, is not observed in the same way for the percentage of acts correctly predicted (for output type 'acts'), though it also does not show an average decrease when compared to predictions made by single measurements. Furthermore, for this metric there is again no particular measurement types that stand out as being notably and consistently better or worse than the others in being useful to estimate a higher percentage of correctly predicted single-act scores.

Hence, at this point in the process with the models struggling to generalize to sub-

jects that they haven't previously seen, **we see no improvements by simplifying predictions by only considering one measurement at a time for a given subject over using an aggregation of the measurements**. Rather, for the percent of correctly predicted sequences output metric, we see an improvement in performance over 4 out of the 5 subjects when using this aggregation. Therefore, for upcoming model prediction sets, **we will continue to use measurement aggregations to make predictions for subjects left-out of the training process**.

12.5 Model Predictions Set 5: Comparable Performance of 'FR_' vs. 'FRC_' Files for 'AD' Models

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	AD	23.53%	D	51.56%	19	32
D3 (FRC)	AD	23.53%	D	100.0%	19	12
D9	AD	47.06%	D	87.5%	22	24
D9 (FRC)	AD	58.82%	D	100.0%	22	27
D11	AD	58.82%	D	65.62%	27	29
D11 (FRC)	AD	35.29%	D	100.0%	27	12
D17	AD	94.12%	D	93.75%	33	28
D17 (FRC)	AD	70.59%	D	100.0%	33	26
HC6	AD	100.0%	D	40.62%	34	30
HC6 (FRC)	AD	64.71%	D	0.0%	34	26

One of the oversights of the feature reduction we perform on the computed statistical values (i.e. the 'ft_sel_red.py' script operating on the 'AD' output of 'comp_stat_vals.py') is that we have been reducing the file's dimensionality on a file-by-file basis: as single files of computed statistical values come into the 'ft_sel_red.py' script, they are projected to a lower-dimensional space one file at a time. This mainly due to a programming oversight and the fact that we computed statistical values one file at a time, so it felt natural to reduce the dimensionality on a file-by-file basis as well. However, this does not guarantee that each file will be projected to the same dimensioned subspace via PCA (which is the feature reduction technique that we have been using thus far in 'ft_sel_red.py') and, moreover, it is extremely unlikely that each file will be on exactly the same feature subspace as any other; hence, they can be considered to contain different features than each other, even if they each contain the same number of features. Hence, we wanted to investigate whether or not this will have been an issue for the models by building models from computed statistical values with files having been reduced to the same lower-dimensional space; if these performed better than the previous ones, then the models' learning potentials benefit from having computed statistical values sharing the same feature space.

To carry out this prediction set, we first ran 'ft_sel_red.py' with the '–combine_files' optional argument set. This combines all the files vertically (i.e. stacks all the rows of all computed statistical values from all files available in the data set) on top of each other before performing dimensionality reduction with PCA before separating them back into their original files (i.e. with the same number of rows of data as before but with far fewer columns). These take the exact same form as their normally-reduced

'.csv' output counterparts in terms of shape, but each of these 'newly' reduced files share the same reduced dimensionality space. These files are placed in the same directory as the other files but with a 'FRC_' (for feature-reduced via concatenation) at the beginning of the file name as opposed to 'FR_' of the others. To get models specifically trained on these 'FRC_' files, we build them in 'rnn.py' with the '–use_frc' optional argument to select these files (note: this is only done for models build on 'AD' rather than raw measurements as raw measurements don't have their features reduced) and 'model_predictor.py' then uses a similar '–use_frc' optional argument to select these models.

12.5.1 Results Discussion

As we can see above, each of the 5 left-out subjects are tested on two groups models for the 'AD' (computed statistical values) measurement only for each of the three output types: one group contains models trained on 'normally' reduced computed statistical values (the 'FR_' files), while the other contains the 'concatenated' version of the files (the 'FRC_' files) which we described above. For the percent of acts correctly predicted metric (for the 'single-acts' output type), we see that the 'FR_' files perform better for 3 of the subjects, while the 'FRC_' performs better for 1 subject and they predict the same percentage for 1 subject. However, we see an improvement for 4 of the 5 subjects for the percentage of correctly predicted sequences (for the 'dhc' output type), suggesting that generally **models trained on computed statistical values to assess D/HC classification perform better if the stat values share the same feature subspace over all files**.

However, we see that for the predicted overall score metric, we see that, much like for the metric for the 'single-acts' output type, the original version of the files (i.e. the 'FR_' files) are consistently better to use to train the models to accurately assess for 4 out of the 5 subjects. As a result of these findings, even though the new 'FRC_' files build models that perform better on left-out subjects for D/HC classification, this isn't the main focus of these models, which places the predictions of accurate single-act scores and overall NSAA scores ahead of that of basic classification (as this is considered to be more important with respect to the project deliverables). As a result of these priorities, we therefore decide to continue to use the traditional way of reducing the dimensionality of files (i.e. on a file-by-file basis) and hence use the 'FR_' files with which to build models.

12.6 Model Predictions Set 6: Chosen Subjects on Familiar vs Non-Familiar Models

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100%	19	27
D3 (already seen)	Position, Sensor Magnetic Field, Joint Angle, AD	100%	D	100%	19	22
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100%	22	25
D9 (already seen)	Position, Sensor Magnetic Field, Joint Angle, AD	100%	D	100%	22	22
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	27	28
D11 (already seen)	Position, Sensor Magnetic Field, Joint Angle, AD	100%	D	100%	27	27
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50%	33	28
D17 (already seen)	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	D	100%	33	31
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75%	34	31
HC6 (already seen)	Position, Sensor Magnetic Field, Joint Angle, AD	100%	HC	75%	34	32

It's worth clarifying at this point what we mean by 'familiar' vs 'non-familiar' models. We consider the models that we have been using in the past several model predictions sets (i.e. where one subject was left out of training for any given model and we then use this model to assess using 'model_predictor.py' the subject that was left-out of training) to be 'non-familiar' with respect to the left-out subject, as the model is non-familiar with all parts of the subject when it comes to assessing it. However, we also want a reference point with respect to how well the models could potentially do when it is said to have 'understood' the subject; at this point, this is done by seeing a lot of the subject's data during the training process. Note that on average, the models that are familiar with the subject will have been trained on 80% of its data, as on average 20% per subject will have been placed in the testing set by 'rnn.py' and so won't have been used for testing.

An important thing to note about these 'familiar' models that we will have built: while they show much better results than the standard non-familiar models, it is not a good indicator of the ability of the models to 'generalize' to new subjects (i.e. properly function in 'production' where we wish to assess new subjects, which is what the 'non-familiar' models aim to replicate). Thus, the 'familiar' models must be taken for what they are: an assessment of the ability of RNNs to learn from the subject data from computed statistical values and raw measurements, and not necessarily its ability to generalize to new subjects. This also provides us with essentially a 'gold standard': a target the models should aim for in their generalization capabilities.

12.6.1 Results Discussion

It should be noted that, in the table above, the rows with a file name containing '(already seen)' were the subject files tested on models that were trained on data that included the subject in question (and therefore no '–leave_out=<subject name>' value was set), while the ones not containing it were the same files tested on models that have not seen data from those subjects before (as done in model predictions set 3). Hence, these pairs of rows are exactly the same data, just assessed on a different set of 15 models (15 models being all combinations of each of the 5 measurements and for each of the 3 output types). Additionally, all rows that are '(already seen)' file assessments are in fact trained on the same 15 models, as these 15 are just the models but with no subjects left out, so there is no reason (or way) to create models that both trained on all subjects and also exclusive to certain subjects during the assessment via 'model_predictor.py'.

As we expected, we see an improvement for each of the subjects when assessed on models that have already seen some of the data of that subject before (again, due to the train/test split ratio of the models being 0.2, it's likely that these models will have seen 80% of the subject files' data during training). Notably, **for 3 of the 5 subjects, there is a significant increase in the percentage of acts correctly predicted when assessed on models familiar with the subjects**, while for the other 2 it is already very high and there being little to improve upon. The increase in D/HC classification potential is also highlighted in this set, with the 'D17' subject (which was previously misclassified when assessed by models not familiar with 'D17') now correctly classified as 'D'. Finally, for all 5 of the subjects, **the overall NSAA scores much more closely approximate the true overall score when using models that have seen the subjects before**: in particular, 'D3' (which has a very low score relative to the average of all subjects) is much more closely approximated when using a model familiar with the subject, while for 2 of the 5 subjects they now achieve an exact match in scores when using 'familiar' models.

Again, it's necessary to temper the magnitude of the takeaways of this experiment set, as this is looking at models that we won't really be using further along in the project; rather, these just help serve as a baseline for the kind of results we should be aiming to achieve with these left-out subjects when we further look at generalization techniques in upcoming model prediction sets. However, one thing that we can definitely take away is that **the model architecture shows an ability to learn D/HC classifications and individual/overall NSAA scores for a variety of severities of subjects with Duchenne, even if at this point it struggles to generalize to unseen patients**.

12.7 Model Predictions Set 7: Assessing Subjects Using Single-Act Files

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	91.74%	19	26
D3 (act 1)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	79.69%	19	27
D3 (act 2)	Position, Sensor Magnetic Field, Joint Angle	35.29%	D	100.0%	19	26
D3 (act 3)	Position, Sensor Magnetic Field, Joint Angle	29.41%	D	100.0%	19	20
D3 (act 4)	Position, Sensor Magnetic Field, Joint Angle	35.29%	D	100.0%	19	22
D3 (act 5)	Position, Sensor Magnetic Field, Joint Angle	35.29%	D	100.0%	19	26
D3 (act 6)	Position, Sensor Magnetic Field, Joint Angle	70.59%	D	100.0%	19	24
D3 (act 7)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	100.0%	19	27
D3 (act 8)	Position, Sensor Magnetic Field, Joint Angle	41.18%	D	100.0%	19	24
D3 (act 9)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	19	28
D3 (act 10)	Position, Sensor Magnetic Field, Joint Angle	41.18%	D	100.0%	19	27
D3 (act 11)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	81.25%	19	27
D3 (act 12)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	19	27
D3 (act 13)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	19	26
D3 (act 14)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	19	25
D3 (act 15)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	19	24
D3 (act 16)	Position, Sensor Magnetic Field, Joint Angle	35.29%	D	100.0%	19	25
D3 (act 17)	Position, Sensor Magnetic Field, Joint Angle	52.94%	HC	42.19%	19	23
D9	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	22	26
D9 (act 1)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	22	24
D9 (act 2)	Position, Sensor Magnetic Field, Joint Angle	82.35%	D	100.0%	22	25
D9 (act 4)	Position, Sensor Magnetic Field, Joint Angle	70.59%	D	100.0%	22	26
D9 (act 5)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	22	25
D9 (act 6)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	22	25
D9 (act 7)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	27
D9 (act 8)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	100.0%	22	24
D9 (act 9)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	100.0%	22	24
D9 (act 10)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	22	24
D9 (act 11)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	22	26
D9 (act 12)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	26
D9 (act 13)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	22	29
D9 (act 14)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	30
D9 (act 15)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	22
D9 (act 16)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	20
D9 (act 17)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	22	28
D11	Position, Sensor Magnetic Field, Joint Angle	82.35%	D	94.84%	27	27
D11 (act 1)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	27	29
D11 (act 3)	Position, Sensor Magnetic Field, Joint Angle	70.59%	D	85.94%	27	27
D11 (act 4)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	27	28
D11 (act 5)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	27	30
D11 (act 6)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	27	22
D11 (act 7)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	100.0%	27	27
D11 (act 8)	Position, Sensor Magnetic Field, Joint Angle	88.24%	D	100.0%	27	22
D11 (act 9)	Position, Sensor Magnetic Field, Joint Angle	76.47%	D	100.0%	27	24
D11 (act 10)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	27	22
D11 (act 11)	Position, Sensor Magnetic Field, Joint Angle	76.47%	D	90.62%	27	26
D11 (act 12)	Position, Sensor Magnetic Field, Joint Angle	35.29%	D	84.38%	27	27

D11 (act 13)	Position, Sensor Magnetic Field, Joint Angle	58.82%	D	100.0%	27	26
D11 (act 14)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	100.0%	27	23
D11 (act 15)	Position, Sensor Magnetic Field, Joint Angle	47.06%	D	100.0%	27	23
D11 (act 16)	Position, Sensor Magnetic Field, Joint Angle	82.35%	D	100.0%	27	29
D11 (act 17)	Position, Sensor Magnetic Field, Joint Angle	76.47%	D	100.0%	27	29
D17	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	22.66%	33	28
D17 (act 1)	Position, Sensor Magnetic Field, Joint Angle	76.47%	D	100.0%	33	29
D17 (act 2)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	90.62%	33	28
D17 (act 3)	Position, Sensor Magnetic Field, Joint Angle	88.24%	HC	42.19%	33	22
D17 (act 4)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	30
D17 (act 5)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	20
D17 (act 6)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	33
D17 (act 7)	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	33	30
D17 (act 8)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	31
D17 (act 9)	Position, Sensor Magnetic Field, Joint Angle	70.59%	D	100.0%	33	31
D17 (act 10)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	29
D17 (act 11)	Position, Sensor Magnetic Field, Joint Angle	100.0%	HC	25.0%	33	31
D17 (act 12)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	20.31%	33	33
D17 (act 13)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	29
D17 (act 14)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	26
D17 (act 15)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	26
D17 (act 16)	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	0.0%	33	27
D17 (act 17)	Position, Sensor Magnetic Field, Joint Angle	70.59%	HC	50.0%	33	28

One of the things we were curious to know was how the performance on certain NSAA activities were reflected in the overall performance of the subject: we wanted to know if there were certain single activities of the 17 undertaken by each subject that can be shown to be more 'impactful' to the overall assessment of the subject. Therefore, we wanted to see whether specific activities undertaken by subjects are indicative of their overall NSAA score, more so than other activities. This leads us to a hypothesis that motivates this model prediction set: **single-activity data that is consistent across multiple subjects at being able to approximate overall NSAA scores and D/HC classification would show these to be the more useful activities at assessing a subject**. The idea is that, if we found several activities of the 17 that are far better at predicting overall scores and classification than others, this could motivate assessors to pay special attention to the scoring of these activities during assessment or, alternatively or perhaps in addition, could result in the discarding of some or many of the 17 activities if they prove to have little or no correlation to the overall assessment.

Conveniently, we have the single acts data files already in place and in the same 'format' as the source '.mat' files for each subject (i.e. the files containing all 17 activities with 'filler' in between). For each of the 5 subjects, we then compare the results of these 17 activities against the assessment of the subjects done with their complete files (as done in model predictions set 3). If certain specific activities prove to be better at assessing the subject for all, or most, of the 5 left-out subjects, then we know that these activities are more useful to the assessment. It should be noted that we are testing these single-act files for each of the subjects (17 activities x 5 subjects = 85 file assessments) on the same models as was built for model predictions set 3; in other words, the 'single-activity' files are being tested on the same models used to predict the 'all-activities' files.

It should be noted that we are not including 'HC6' in this model predictions set. This is more out necessity than anything else: at the present time, the reference sheet that we have for subjects does not contain entries for most of the activities for subject 'HC6' for their activities. Hence, rather than including only several of these activities, we omitted this subject from this model predictions set entirely. Conveniently, having an 'HC' subject for this set isn't as necessary, as we are only comparing the performance of single act files to their corresponding all-act files, rather than seeing in general how the models perform in classification and regression statistics. It should also be noted that there are several activities missing from a few of the subjects; this is mainly due to the subjects having not performed those specific activities in their assessments. Rather than excluding these subjects completely, as it is only a few activities we instead decide to keep the subjects in this model predictions set and work around these missing values.

Finally, an important thing to note here is the absence of using the 'AD' measurement (i.e. the computed statistical values). This is due to the inability to use 'ft_sel_red.py' for single-act 'AD' files. This is because many single activities that are extracted from their base files are very short in length (often around 1 second for activities such as 'step up R'). Consequently, when we compute the statistical values over 1 second intervals, the result is that the file containing the computed statistical values for a given subject and a certain activity (e.g. 'AD_D3_act7_stats_features.csv') may contain only several rows of data; hence, these might have a shape of something like (3, 4000). As in many cases the number of rows are fewer than the target number of reduced columns (e.g. target of 30 columns), we can't reduce the dimensions of these files, so we can't use them in our RNN models (as we can't viably setup a model with 4000 columns). Therefore, as we are limited to the number of possible single-act files for the computed stat value measurement for activity files of $>$30 seconds in length (which doesn't cover many of the activities), we decide to discard this measurement and instead compute models just based on position, joint angle, and sensor magnetic field.

12.7.1 Results Discussion

As this table is fairly challenging to interpret at a glance, considering it has 70 entries total, it was felt necessary to make use of the 'predictions_selector.py' script in order to filter the lines based on certain metrics. We therefore ran the script three times: once to find the best 15 rows according to metric measuring the difference between the true and predicted overall NSAA values (for the 'overall' output type), the percentage of acts correctly predicted (for the 'acts' output type), and the percentage of predicted correct sequences (for the 'dhc' output type). The results can be seen below:

Best performing 15 rows by Diff true/pred overall NSAA score...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Diff true/pred overall NSAA score
42	D11 (act 7)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
2	D11	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
46	D11 (act 11)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
68	D17 (act 16)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
51	D11 (act 16)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
62	D17 (act 10)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	0
43	D11 (act 8)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
40	D11 (act 5)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
50	D11 (act 15)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
52	D11 (act 17)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
24	D9 (act 5)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
10	D3 (act 7)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	1
31	D9 (act 12)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	2
25	D9 (act 6)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	2
32	D9 (act 13)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	2

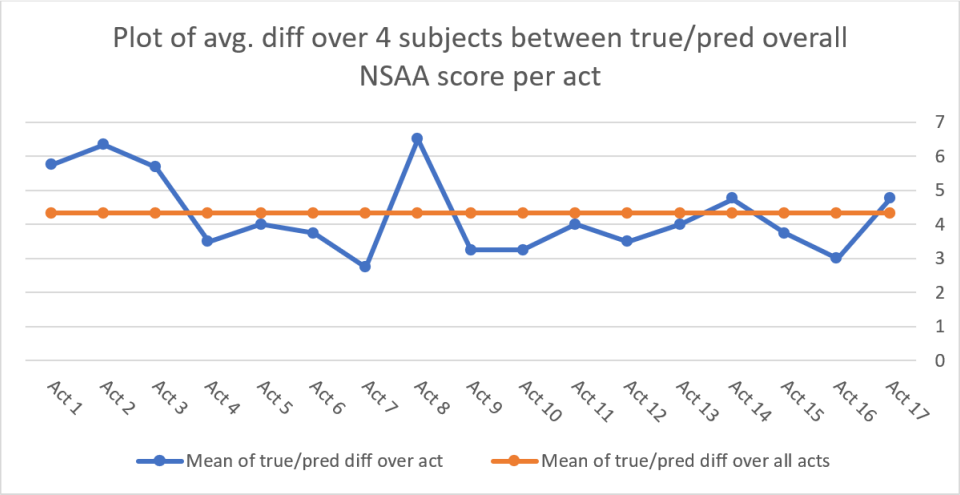
Best performing 15 rows by Percentage of acts correctly predicted...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Percentage of acts correctly predicted
63	D17 (act 11)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.00
56	D17 (act 4)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
65	D17 (act 13)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
57	D17 (act 5)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
58	D17 (act 6)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
60	D17 (act 8)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
64	D17 (act 12)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
62	D17 (act 10)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
66	D17 (act 14)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
67	D17 (act 15)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
68	D17 (act 16)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
3	D17	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	94.12
43	D11 (act 8)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	88.24
55	D17 (act 3)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	88.24
22	D9 (act 2)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	82.35

Best performing 15 rows by Percentage of predicted correct sequences...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Percentage of predicted correct sequences
35	D9 (act 16)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
48	D11 (act 13)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
26	D9 (act 7)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
27	D9 (act 8)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
28	D9 (act 9)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
29	D9 (act 10)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
30	D9 (act 11)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
31	D9 (act 12)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
32	D9 (act 13)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
33	D9 (act 14)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
34	D9 (act 15)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
1	D9	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
36	D9 (act 17)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
37	D11 (act 1)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0
51	D11 (act 16)	NSAA	NaN	['position', 'sensorMagneticField', 'jointAngle']	100.0

As we are primarily focused on the regression capabilities of the model, our focus mainly lies with the metric shown in the first and third images. However, for the third image seen above (for the 'acts' output type), there are numerous lines that perform very well according to that metric; hence, we decide to at this point focus exclusively on the first image that shows the best activities for each subject ranked according to how closely they predict the overall NSAA score (with perfectly predicting resulting in a '0' value, i.e. 0 difference between the true and predicted overall NSAA score). We saw a greater number of acts between 5 and 7 in this better performing region (i.e. files of act 5, act 6, or act 7 of each subject tended to better approximate the true overall NSAA score on models that hadn't seen the subject before). However, we wanted to view this in a bit more detail and find out which activities tended to perform better or worse and see if there any trends.



The graph above shows, for each activity number, the average difference between the true and predicted overall NSAA scores over the 4 subjects as plotted by the blue line, while the orange line is the average over both all subjects and all activities. Therefore, activities significantly below this line would indicate that they are more likely to be useful in predicting the overall NSAA score (and thus be a more useful activity to the assessment), and vice versa. One thing we can note is that activities 1, 2, 3, and 8 all perform significantly worse than the average. **These are the 'stand', 'walk', 'stand up from chair', and 'descent box right', and are the activities that show much lower correlation with the overall NSAA assessment for the subject, and hence are less likely to be useful in the assessment**. Conversely, we can see particularly good performance from activities 7, 9, and 10, which are the 'climb box left', 'descend box left' and 'get to sitting'; these are **more likely to be activities that the assessor should pay special attention to according to the above results**.

It should be noted before making any conclusions that these are far from complete recommendations for specialists due to:

- Only considering the chosen 5 subjects (reduced to 4 due to not having the single-act data for 'HC6').
- Models struggling to generalize at this point, giving worse scores than if we were using models that generalized better.
- Cross-confirmation of single-act times entered into the reference table is needed; as multiple users entered certain times into the table, it's advisable for everyone involved in creating it to confer and agree upon exact times for each subject's individual activities in order to ensure accurate activity times from the files.

While the above results give us an indication of what activities are the most and least useful, this does prove to potentially be fertile ground to draw conclusions about the NSAA assessment at a later point with more improved models and techniques, which we shall explore further in MPS 23.

12.8 Model Predictions Set 8: Model Performance over the 5 Subjects w/ Outlier Excluded

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100%	19	27
D3 (other_lo=D10)	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	HC	50%	19	28
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100%	22	25
D9 (other_lo=D10)	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	22	26
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	27	28
D11 (other_lo=D10)	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100%	27	27
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50%	33	28
D17 (other_lo=D10)	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50%	33	29
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75%	34	31
HC6 (other_lo=D10)	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75%	34	28

One of the features of this data set is that it contains one subject ('D10') that is an outlier; not for medical reasons (i.e. not because he suffers much worse with DMD in comparison with the other subjects), but rather as a consequence of him not undertaking the complete assessment. He instead only undertook 3 of the 17 activities as part of the NSAA assessment for what are believed to be personal reasons. Hence, a score of 3 for the overall NSAA score was recorded for that subject. This is significantly lower than the next lowest score of '15' for subject 'D2', and does not reflect his 'true' overall NSAA score, which should be at least in the mid-teens. We believed that this may affect the models' ability to generalise to new subjects, as it is forced to learn a wider range of scores for subjects: with the 'D10' included, the model has to learn scores of between '3' and '34', while with it not included it only has to learn how to predict overall scores of between '15' and '34'. The idea here was that, as these particularly-low scores aren't a true reflection of the subject in question (as the subject only achieved this score due to real-world circumstances), by discarding this subject from the training set, it will make it easier for the models to learn scores within this narrower range, while excluding an incorrectly assessed subject and hopefully better generalise.

To achieve this, we simply modified the 'rnn.py' script to take in several subject names for the '–leave_out' argument and pass in the 'D10' subject as the second part of this argument. For example, for the 'D3' row above we built it with models from the '–leave_out=D3' argument, while for the 'D3 (other_lo=D10)', we built it with the '–leave_out=D3,D10' argument. We then specified the 'model_predictor.py' script to search for these models with multiple subjects left-out via the '–other_lo' optional

argument.

12.8.1 Results Discussion

The findings from this set of model predictions weren't particularly promising. Focusing on the performance for the 'overall' output type, the predicted overall NSAA scores did not seem to improve with leaving the 'D10' subject out of training completely; rather, for 3 of the 5 subjects they got worse with the 'HC6' subject getting worse by being 3 further away from the true score. And while 2 of the 5 subjects showed an improvement in approximating the overall score better, **in leaving out the 'D10' subject we saw an increase in cumulative difference between the true and predicted scores from 20 to 23**. And while we saw performance increase for the percent of acts correctly predicted metric (for the 'acts' output type), we also saw a diminishing performance with respect to the 'dhc' output type: **when having 'D10' left out of the training set, only 3 of the 5 subjects have the correct label predicted, while including this subject we see 4 of the 5 subjects with the correct label**.

Rather than helping the model to more easily learn the scores within what we would consider the 'normal' range (i.e. not including particularly low scores like '3') and help with the generalization ability, in removing the 'D10' subject the models tend to perform worse. This can be most likely attributed to the 'D10' subject adding some much-needed noise to the data set. It's been well documented that machine learning models often need noise in the data to better perform on unseen data and thus to be said to be truly 'learning'. **Hence, including this subject with a particularly low score seems to help the models in predicting other subjects within the 'normal' range by adding a 'noisy' sample**. This also presents a good argument to be experimenting with adding noise to the data set, which we shall explore further in MPS 15.

12.9 Model Predictions Set 9: Model Performance for 5 Subjects on Single-Act-Concat Models

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100.0%	19	27
D3 (src sac)	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	75.0%	19	29
D3 (src normal)	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	75.0%	19	28
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100.0%	22	25
D9 (src sac)	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	84.38%	22	28
D9 (src normal)	Position, Sensor Magnetic Field, Joint Angle, AD	47.06%	D	75.0%	22	28
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100.0%	27	28
D11 (src sac)	Position, Sensor Magnetic Field, Joint Angle, AD	64.71%	D	75.0%	27	24
D11 (src normal)	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	75.0%	27	24
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50.0%	33	28
D17 (src sac)	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	39.06%	33	27
D17 (src normal)	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50.0%	33	28
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75.0%	34	31
HC6 (src normal)	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	50.0%	34	29

A feature of the NSAA assessment data files that we get as source data is that it contains the full recording of the subject's NSAA assessment, though these are sometimes broken down into 2 separate files. While these usually contain all 17 of the activities performed by the subject, the files often contain data that is not quite as useful; for example, the subjects standing around before they undertake the next activity (possibly listening to instructions from the assessor about what activity to do at the time). Additionally, the start and end times of the activities that were recorded in the Google annotation sheet only include the start and end times for the activities that were completed and that could only include one of each activity within the time window. Hence, all data that was captured by the suit where the subject either failed to complete the activity or did the activity twice was also captured in these files. However, when 'mat_act_div.py' is used on these source files to divide up the files, we selected only the parts of the file where the activities were completed; hence, much of this data was discarded.

An option that was created as part of the 'mat_act_div.py' script was the ability to 'recombine' the single-act '.mat' files into a single file per subject again. Much like the source '.mat' file, this recombined file (which we refer to as a 'single-act-concat' file), contains all the activities for the assessment done by the subject in question. However, where it differs from the original source file is that it has a lot of data cut

out of it, data that includes the subject just standing around or trying but failing to complete activities one or multiple times. A lot of this data was therefore not considered to be particularly 'useful' for training the model, at least in comparison to the completed activities. Moreover, it was thought that some of this 'less-useful' data (e.g. subjects standing around for extended periods of time) would possibly bias the model away from learning important insights from the 'real' activities within the data, as the models would have to account for both the useful and less-useful data in training. To this end, the 'mat_act_div.py' script was run to divide up and recombine the files, with the 'comp_stat_vals.py' and 'ext_raw_measures.py' scripts being then used to obtain the computed stat values and raw measurements in the usual manner, respectively, followed by 'rnn.py' being trained with the '–use_sac' argument to use build models from these 'single-act-concat' versions of the computed statatistical values and raw measurements files, and finally 'model_predictor.py' being run on the left-out subjects assessed on the models built with '–use_sac'.

12.9.1 Results Discussion

There are three types of 'left-out' subjects that we consider in 'model_predictions.csv' for each subject. The first of the three is the 'baseline'; i.e. the results obtained in model predictions set 3 for subjects that were assessed on models that weren't built on 'single-act-concat' files, but rather files that didn't have single-activities extracted from them. The second type were for 'single-act-concat' subject files that were assessed using models trained on these 'single-act-concat' files; hence, to predict on these subjects, one would first need to run 'mat_act_div.py' along with 'comp_stat_vals.py' and 'ext_raw_measures.py' before assessing these subjects. The third type was for standard subject files (i.e. the same as those used the first time as done for model predictions set 3 with no single activities extracted from them) but assessing on models built on 'single-act-concat' files; the idea of looking at both varieties of assessing file on models built on 'single-act-concat' files came from the desire to see whether models were any better or worse at assessing the subject when presented with files that contained this 'less-useful' data.

However, we can very clearly see from the above tables that **both varieties of assessed files (single-act-concat files or normal) assessed on models built on single-act-concat files performed worse than their counterparts which were assessed on models built from the normal data files as in MPS 3**. This was the case over all 3 metrics with no improvements anywhere, which was unlike any other model prediction set we've seen thus far where even with changes that produced overall worse models, they still showed some signs of improvement for some metric(s) and some subject(s). While disappointing that this new approach didn't lead to any improvement in results, it isn't entirely surprising. For one, we are excluding a lot of the data: in the process of extracting small pieces of the source files and knitting them back together, we reduce the amount of raw data we have available by a factor of between 5 and 10, depending on the subject (generally higher if the subject has a lower overall NSAA score as the subject generally takes more attempts

at completing each activity). Even though we consider a lot of this data that we excluded 'less-useful', **the removal of not-as-useful data is not enough to overcome to loss in performance as a consequence of having much less overall data**. Furthermore, in the same way as in model predictions set 8 that removing a subject most likely showed worse results due to a comparable lack of 'noise' now present in the data set, there is a good chance that the same thing happened here, where the data of the subjects 'standing around' and other minor activities (that weren't necessarily a part of the NSAA assessment such as raising one's arm) may have added some much-needed 'robustness' to the data set.

Additionally, another theory as to why models trained on 'single-act-concat' files performed not-as-well as models built on 'normal' files could be due to the fact that the capturing of activities that weren't properly completed by the subject (which aren't included as part of the 'single-act-concat' files) gave a fair bit of information about the subjects to the models. This is evidenced by the fact that for the lowest-scoring 2 of the 5 subjects, the models trained on 'single-act-concat' files predict a higher overall NSAA score than the models trained on the 'normal' files, where **the inclusion of the 'failed' activities most likely gave the models a better sense for the lower-scoring subjects that they should have a corresponding lower score**. Therefore, the main takeaway we have of this model predictions set is that **it is worthwhile using all of the available data captured by the bodysuit as opposed to only using the data showing the completed activities**.

12.10 Model Predictions Set 10: Model Performance 5 Subjects on Downsampled Files

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100.0%	19	27
D3 (downsampled)	Position, Sensor Magnetic Field, Joint Angle, AD	41.18%	D	100.0%	19	25
D9	Position, Sensor Magnetic Field, Joint Angle, AD	52.94%	D	100.0%	22	25
D9 (downsampled)	Position, Sensor Magnetic Field, Joint Angle, AD	47.06%	D	75.0%	22	23
D11	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	D	100.0%	27	28
D11 (downsampled)	Position, Sensor Magnetic Field, Joint Angle, AD	58.82%	D	100.0%	27	24
D17	Position, Sensor Magnetic Field, Joint Angle, AD	94.12%	HC	50.0%	33	28
D17 (downsampled)	Position, Sensor Magnetic Field, Joint Angle, AD	70.59%	HC	50.0%	33	25
HC6	Position, Sensor Magnetic Field, Joint Angle, AD	100.0%	HC	75.0%	34	31
HC6 (downsampled)	Position, Sensor Magnetic Field, Joint Angle, AD	58.82%	D	25.0%	34	23

Another feature of the data set that makes it less-than-ideal for machine learn-

ing model training is the distribution of data with different overall and individual NSAA scores and D/HC classifications. In particular, with regards to predicted overall score, we have a lot of data with an overall score of 34 (as this covers every 'HC' file), while only one subject with a score of '3' and one with '15'. Hence, we have a very uneven distribution of classification and regression scores that we are expected to train on which, while it reflects the real-world nature of the dataset and the real distribution of subjects with DMD, it does not help with the ability of models to learn generalized knowledge of a variety of classifications and regression scores that is able to generalize to new subjects. To overcome this, we decided to experiment with a rebalancing approach to the data so that we have the exact same number of sequences for every possible overall NSAA score in the original data set that we are presented with which are used to train models.

To do this, we consider a downsampling approach. This is primarily due to time and computing restrictions at this point: with the upsampling approach, each model would take approximately 30-35 minutes to build which, given we have 12 models per left-out subject to build (3 output types x 4 measurement types), it would take between 30-35 hours to build the requisite models; meanwhile, downsampling the data and building models from this data takes a fraction of the time as each training epoch takes a much shorter amount of time in comparison. To downsample the data, the 'rnn.py' is specified with the '–balance=down' argument; this then results in the requisite functions from the 'data_balancer.py' script being called. From here, for any overall NSAA score for sequences that aren't the 'least popular' overall score, the sequences are instead randomly sampled from the pool of sequences all corresponding to a specific overall NSAA score. This random sampling is repeated until we have a number of samples equal to that of the 'least popular' overall score (note that this is sampling with replacement as the same sequence can be selected multiple times by the random sampler). This is then repeated for all possible overall NSAA scores that appear in the data set: this results in a much smaller data set but with an equal number of sequences for each overall score.

12.10.1 Results Discussion

With regards to the two types of output lines in the table above, the first is the 'standard' left-out subject results as seen in model predictions set 3, while the second is the same subject file but assessed on models built on a balanced data set by way of down sampling. While we can see an improvement for 2 of the 5 subjects with respect to the difference between predicted and true overall NSAA scores for the 'overall' output type, we can see a significant worsening for the other 3 subjects. Indeed, **by using down sampling the overall accumulative difference error over the 5 subjects for the overall NSAA score increases from 20 to 29, an almost 50% increase**. Additionally, we see no consistent 'pattern' of the predicted scores: we would expect them to increase vaguely in line with the true overall value, even if they aren't that close to the true overall NSAA values, as we can somewhat see with the standard 'left-out' subject results. However, the down sampled models show no

sense of predicting when one sequence is higher or lower than the other with respect to the true overall NSAA score, which the standard models can somewhat do.

A clue for what might be motivating this behaviour can be found in the distribution of these overall scores: the predicted values are limited to between 23 and 25. This is near the median value for the data set **which implies that the models are trying to minimize its own loss by simply guessing new sequences from unseen files to be somewhat close to the middle of the expected range of overall values**; in this sense, the models would be said not to be truly learning from the data but rather adopting a strategy to try to minimize its own loss by other means. A possible reason for this poor performance might be the massive reduction in data samples: while the standard 'left-out' subjects setup has a data set of roughly 13000 sequences, the models built from downsampled data only has approximately 2700 sequences to work with. This is due to there being so few sequences for the overall NSAA score of '3' that we have to discard many of the most common sequences with corresponding scores (such as '30' or '34') so that we have the same number of each overall NSAA score for the sequences. In a similar vein to model predictions set 8 and 9 where some of the data is discarded, we can conclude here that **the rebalancing of the data set to have an even distribution of scores is not enough to overcome to loss in performance due to comparable lack of data**.

12.11 Model Predictions Set 11: Model Performance for 5 Subjects on Feature Concat Models

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	91.74%	19	25
D3 (feature concat = 60)	Position, Sensor Magnetic Field, Joint Angle	23.53%	D	77.23%	19	28
D3 (feature concat = all)	Position, Sensor Magnetic Field, Joint Angle	41.18%	D	87.95%	19	29
D9	Position, Sensor Magnetic Field, Joint Angle	64.71%	D	100.0%	22	25
D9 (feature concat = 60)	Position, Sensor Magnetic Field, Joint Angle	41.18%	D	80.94%	22	28
D9 (feature concat = all)	Position, Sensor Magnetic Field, Joint Angle	41.18%	D	73.75%	22	25
D11	Position, Sensor Magnetic Field, Joint Angle	82.35%	D	94.84%	27	27
D11 (feature concat = 60)	Position, Sensor Magnetic Field, Joint Angle	82.35%	D	69.53%	27	26
D11 (feature concat = all)	Position, Sensor Magnetic Field, Joint Angle	70.59%	D	63.12%	27	27
D17	Position, Sensor Magnetic Field, Joint Angle	94.12%	HC	22.66%	33	27
D17 (feature concat = 60)	Position, Sensor Magnetic Field, Joint Angle	94.12%	D	79.69%	33	26
D17 (feature concat = all)	Position, Sensor Magnetic Field, Joint Angle	88.24%	D	81.25%	33	26
HC6	Position, Sensor Magnetic Field, Joint Angle	100.0%	HC	83.2%	34	31
HC6 (feature concat = 60)	Position, Sensor Magnetic Field, Joint Angle	52.94%	D	14.06%	34	26
HC6 (feature concat = all)	Position, Sensor Magnetic Field, Joint Angle	100.0%	HC	86.72%	34	32

Up until this point, we have been building separate models for each of the data types (i.e. input types) that are used to build models. For example, for a given subject to be left out of the training set and for a given output type that the models are to produce (e.g. 'overall'), if we are using 4 measurement types total to assess the subject (e.g. position, sensor magnetic field, joint angle, and AD), we need to build 4 separate models, each built on a different measurement type that is from the same data from the source '.mat' files. However, we wanted to look at whether it was possible to combine this data prior to feeding it through a model; that way, we would need only 1 model per subject file and output type combination. This would take the total numbers needed for a typical model predictions set (i.e. for 5 different left-out subjects for 4 input types and 3 output types) from 60 down to 15. This would hopefully hold two advantages:

- Reduce overall training time as, while each model will take longer to train, the dramatic reduction in numbers of models to create and train would hopefully offset this to be an overall time save.
- The requirement of models to train and assess on multiple measurement types for each model may encourage it to learn more general trends from the data and hopefully help them generalise better.

To do this, it was necessary to add the options in 'rnn.py' and 'model_predictor.py' to concatenate the data types together. Specifically, the data was to be added together along the 'features' dimension. For example, for a given left-out subject and output type, let's say that we had (13026, 60, 66) as the shape for the 'jointAngle' data, (13026, 60, 51) as the shape for 'sensorMagneticField' data, and (13026, 60, 69) as the shape for the 'position' data. Concatenating all these together prior to feeding the data into the model gives a shape of (13026, 60, 186). Hence, all the raw measurements are fed into the same model rather than three separate models and then concatenating predictions made on the three separate models. It should be noted that we are excluding the computed statistical values from this model prediction set. This is due to that data having a shape that differs along more than the 3rd dimension in comparison to the other raw measurement data: the 'AD' data will generally have fewer rows and a shorter sequence length (which is at this time set to '10' as opposed to '60' for the raw measurement types). Hence, this data can't be concatenated together with the raw measurement data and so we don't consider this measurement here. Therefore, to have a fair comparison with the traditional method of creating models (one model per measurement type), we also don't consider the 'AD' type in these rows of the above table. In other words, they are similar to how we predicted measurements in model predictions set 3, only not including 'AD' as part of the assessment process (hence why the corresponding lines in the above table are different to those in MPS 3).

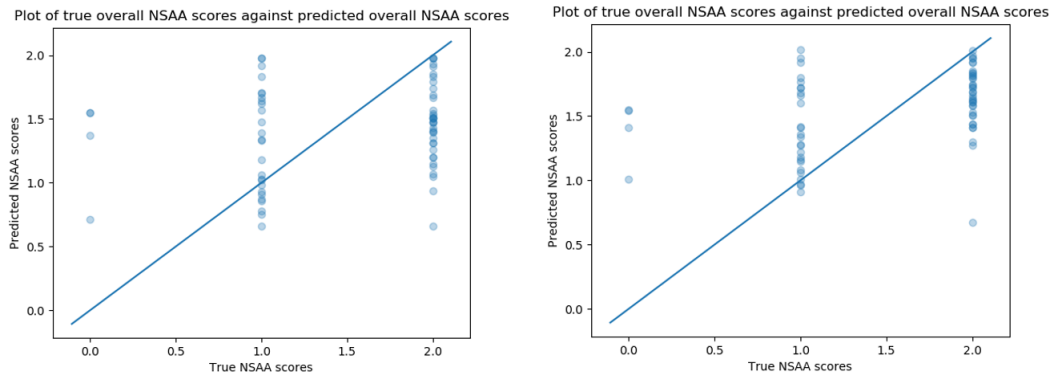
12.11.1 Results Discussion

We consider two different types of concatenation of features (along with the standard method which is shown by the first line in each group in the table). The first uses PCA to reduce the dimensionality from 186 features down to 60. The reason this is done is so investigate whether feature reduction would be effective for raw measurements once they have been concatenated together (as we have up until this point only done so for computed statistical values). The reason that '60' was chosen to be the number of features to be reduced to was that we have effectively seen that models can be built from sequences with between 51-69 features with them, so '60' was chosen to be an ideal size that also held a majority of the inherent variance within the original 186 features. The other type of model that was built and assessed on was those using all of the concatenated features (i.e. not reducing the dimensionality of the concatenated raw measurements data); this is shown by the 'feature concat = all' lines. These are the lines that show the results of left-out subjects built on models that are built in line with the explanation given above (i.e. having a data shape of (13026, 60, 186)). Also note that that the data for the subject we are using to test each of the models are transformed into the way the models are expecting: for example, for the models build with 'feature concat = all', the subject file that we are assessing on has all of its raw measurements concatenated together in the same way as was the data used to train the model.

Considering the difference between the true and predicted overall scores for vari-

ous left-out subjects (for the 'overall' output type), we can see that in every case the feature reduction of concatenated raw measurement features to 60 shows a worse performance with respect to the 'standard' method (i.e. the setup with no concatenation of features). We can thus infer from this that **the concatenation and subsequent feature reduction of raw measurements is inferior to using the separate models for each measurement type**. This is most likely due to the comparative loss of information: while we keep the majority of the variance when we reduce the dimensions down to 60, it's still a significant loss when compared with using different models trained on individual measurement types. Additionally, while we don't see such extreme results when we use the non-feature-reduced data in 'feature concat = all', we still do see slightly worse results: when looking at the total accumulation of the difference between true and predicted overall NSAA scores for all subjects, the standard approach yields a total of '18' while the new approach has a total of '22'. This is notably worse in comparison, particularly when looking at the 'D3' subject which has a particularly low score. This difference in performance is more pronounced when looking at the percentage of acts correctly predicted, where the standard approach to model building is still superior. We can predict that the reason for this loss of performance is the comparative difficulty of learning from sequences with '186' dimensions as opposed to '60' due to the curse of dimensionality, as we otherwise have all the same information available as if we had instead built three separate models per left-out subject and per output type instead of the one with all three measurements used to train it. Hence, we can say that **concatenating the raw measurement features to train models is an inferior approach to training separate models with each raw measurement type used to train a different model**.

12.12 Model Predictions Set 12: Model Performance for 5 Subjects (Single-act files) for the 'indiv' Output Type



Up until this point, we have not really considered the fourth 'output type' that the RNN models can be built to target. This is the 'indiv' output type that ensures that

the RNN models regress on a single continuous value on a single output neuron. In this sense, it is much like for the 'overall' output type except, while that output type regressed for the overall NSAA score for the given subject data, this output type regresses for the single-act score for a given subject **and a given activity number**; hence, while the 'overall' output will range between 0 and 34, the 'indiv' output type will range between 0 and 2. However, while models trained for the 'overall' output type can receive any type of file, be it a complete source file or a single-act source file, models trained for the 'indiv' output type are only expected to receive files representing single-activities. This makes intuitive sense, as it makes no real sense to predict the single-act score of between 0 and 2 for a file that would contain up to 17 activities, as the models wouldn't know which activity of which the user wants the score. Hence, to assess the performance for the 'indiv' output type, we only use 'single-act' files.

These single-act files are the extracted single-activities from the source '.mat' files for a given subject that are divided up based on the annotated Google sheet and by using the 'mat_act_div.py' script. From here, the raw measurements are extracted as normal, so for a given subject that we divide up into 17 activities and extract the position, sensor magnetic field, and joint angle raw measurements from, we would have 51 total '.csv' source files. It's important to note that we don't compute the statistical values (i.e. the 'AD' measurement type) for single-act files. This is because, for single-activities, there might be comparatively very few rows in the source '.mat' file's data table and so, when we compute the statistical values for these, we may end up with files that we need to reduce the dimensions of with a shape of something like (3, 4000) (as computing the statistical values reduces the number of rows 60-fold in our current setup); hence reducing the dimensions to from '4000' to '30' becomes impossible with that few rows, and therefore it is simpler just to exclude the computed values when we work with single-act files.

We again look at the 5 subjects that are left-out of training in their corresponding models. The models themselves will have been trained on single-activity files to only target the 'indiv' output type, though for each left-out subject we would compute 3 models, one for each raw measurement type we're interested in. However, instead of assessing the subjects' 'complete' files on the models via 'model_predictor.py' that have been trained on single-act files for the 'indiv' output type, we instead assess each of the left-out subject's single-act files in turn on the aforementioned models; hence, we end up with 17 assessments for each of the 5 left-out subjects in total, or 85 assessments total. In fact, we don't have quite this many assessments made (i.e. lines in 'model_predictions.csv'), as the Google annotation sheet doesn't contain every single activity start and end times for the 5 subjects we assess on, due to some of the activities not appearing in the corresponding source file or the subject not performing the activity for other reasons. Therefore, some of the activities for some of the subjects are missing.

Each of these assessments (e.g. act 13 for subject 'D9') has a 'true' score for the cor-

responding subject and activity number that's determined by the 'nsaa_6mw_info.xlsx' reference sheet (for the activities that have been performed). For each of the 85 assessments made within this model predictions set, this true value is written to the column in the table (i.e. being one of 0, 1, or 2) and the aggregated predicted value made by the three models (one for each of the measurement types) is written as a value between 0 and 2. Note that this predicted value is a continuous value rather than a discrete value, as it was felt that a continuous value would be more useful for the above plots at showing how close or far away predictions were, due to the comparatively smaller range of true values when compared to plotting the true and predicted values for overall NSAA scores. The above left plot therefore shows each of the 85 assessments on a 2D plot, with its x-position representing it's true single-act NSAA value of either 0, 1, or 2 and its y-position representing it's predicted value of between 0 and 2.

Finally, it's worth noting that the right image is the similar to the left image in that it shows the distribution of true and predicted single-act scores for single-activity files of the 5 left-out subjects. However, the right image shows the same exact files that are assessed except that these are assessed on models that are already familiar with the subjects from training (in a similar way as was done for MPS 6). Therefore, we would expect these to be more accurate (i.e. the points should be closer to the $y = x$ line) than as seen in the left image; in this sense, this is analogous to MPS 6 in that we are trying to establish a baseline with which to compare the generalisation ability of the models.

12.12.1 Results Discussion

As we can see in the above left graph, the results for this setup are overall not particularly promising. For assessing single-act files scores for left-out subjects on models trained on single-act files to predict between 0 and 2, the predicted values are not particularly indicative of the true scores for the files. If they were highly correlated, then a large number of the points would follow the $y = x$ line in the graph closely; however this is not the case, as we can see in the aforementioned graph and, for activities and subjects with true scores of either 1 or 2, the predicted scores for those subject/activity combinations are not particularly indicative of the true scores and the predictions often have a large distribution of predicted values, ranging between 0.5 and 2 and, while for subject/activity combinations with large true values have somewhat higher predicted scores than their smaller true valued counterparts (evidenced by the median being higher for subject/activity combinations of true values equalling 2 than those of true values equalling 1), this is not large enough to conclude that the models can differentiate between subject/activity combinations with different true single-act scores.

What's also interesting is that, even with the subject being familiar to the models (i.e. the 5 left-out subjects instead being used to train the model along with the other subjects), as seen in the above right graph, the models are still not particularly

good at approximating the true single-act NSAA score. Hence, **the problem with this output type is less to do with its generalisation ability and more to do with the models not being able to learn the differentiating factors between subject/activity combinations of different scores**. This might be down to several factors, the most notable of which might be the drastic reduction in the amount of data when using only single-activity files to train the model (as this process involves 'cutting out' a lot of the data between the activities performed in the source file; see conclusions drawn in MPS 7). The results, or lack thereof, of this model predictions set might seem even more perplexing when compared with the results of previous model predictions sets when assessing for the 'acts' output type (i.e. 17 output nodes in the model predicting all the single-activity scores for complete files). However, this could be down to regressing towards a value between 0 and 2 in the case of 'indiv', while for 'acts' we are classifying for 17 output nodes as being either a 0, 1, or 2. This might be due to **regression possibly not be as reliable for regressing towards 1 of 3 values (i.e. between 0 and 2) in comparison with classification as the limited range of true values for single-act files don't easily lend themselves to regression**. Hence, an alternative method could examine the same 'indiv' output type but instead retraining the models as a classification model for three possible classes (0, 1, or 2) that might achieve better results than we see here.

12.13 Model Predictions Set 13: Model Performance for 5 Subjects w/ Included 6-minute Walk Data

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	['position', 'sensorMagneticField', 'jointAngle', 'AD']	41.18%	D	100.0%	19	27
D3 (add dir = 6minwalk-matfiles,6MW-matFiles)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	35.29%	D	75.0%	19	27
D3	['jointAngle']	47.06%	D	90.4%	19	27
D3 (add dir = 6minwalk-matfiles,6MW-matFiles)	['jointAngle']	47.06%	D	92.63%	19	26
D9	['position', 'sensorMagneticField', 'jointAngle', 'AD']	52.94%	D	100.0%	22	25
D9 (add dir = 6minwalk-matfiles,6MW-matFiles)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	82.35%	D	75.0%	22	24
D9	['jointAngle']	58.82%	D	85.62%	22	25
D9 (add dir = 6minwalk-matfiles,6MW-matFiles)	['jointAngle']	70.59%	D	82.19%	22	25
D11	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	27	28
D11 (add dir = 6minwalk-matfiles,6MW-matFiles)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	27	23
D11	['jointAngle']	58.82%	D	72.97%	27	27
D11 (add dir = 6minwalk-matfiles,6MW-matFiles)	['jointAngle']	76.47%	D	80.16%	27	26
D17	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	50.0%	33	28
D17 (add dir = 6minwalk-matfiles,6MW-matFiles)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	88.24%	HC	50.0%	33	22
D17	['jointAngle']	76.47%	D	89.84%	33	27
D17 (add dir = 6minwalk-matfiles,6MW-matFiles)	['jointAngle']	88.24%	D	85.16%	33	25
HC6	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	75.0%	34	31
HC6 (add dir = 6minwalk-matfiles,6MW-matFiles)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	100.0%	34	29
HC6	['jointAngle']	100.0%	HC	61.72%	34	29
HC6 (add dir = 6minwalk-matfiles,6MW-matFiles)	['jointAngle']	100.0%	HC	73.44%	34	30

Up until this point, we have been working exclusively with files taken from the 'NSAA' data set. This is primarily due to our main focus being on the analysis of NSAA files to make assessments upon the NSAA data from new subjects. However, this is not the only directory we can use to train models; specifically, we have both the '6minwalk-matfiles' and '6MW-matFiles' data sets (both containing the 6-minute walk assessments of the subjects) and 'allmatfiles' (which contains the natural movement behaviour of the subjects). It should be noted that these are from the same subjects as in the NSAA directory, so when a 6-minute walk file references subject 'D4', it refers to the same 'D4' subject as appears in the NSAA directory, just a different type of assessment. What we want to look at here, however, is whether or not our models' generalisation ability improves with the addition of new files from other directories containing different types of data (here meaning data of differ-

ent assessments). When comparing to previous model prediction sets, performance has always dropped when we reduced the amount of data in the set (e.g. through the removal of outliers, downsampling the most popular classes, concatenating raw measurement features and then reducing the dimensions, etc.) so we wanted to investigate whether increasing the amount of available data (albeit with walking data rather than NSAA assessment data) would help the model generalise to new subjects' NSAA assessments, as in the previous cases. On the one hand, we are providing more data to the models, which should help it learn better and generalise easier, while on the other hand the new data is being used to train with might not be very useful in helping a model to generalise to NSAA activities due to this new data predominantly showing different movements than appears in the assessed NSAA data.

To enable this to happen, we modify the 'rnn.py' script so that the 'dir' argument can now take more than one directory argument. In the cases where we wish to use more than one directory as the source of the data files, we provide these to the same 'dir' argument but separated by commas. This causes the preprocessing step to repeat over all the directories and, with the data from each directory name in the 'preprocessing()' function, concatenates this 'vertically' (i.e. concatenating each frame, not concatenating the features as done previously), and then splits the total data into training and testing components. Once these models have been built, we then modify 'model_predictor.py' in a way such that, if the '–add_dir' optional argument is provided with additional directories to have been used within the model directory name, ensures that the models that are loaded are those trained on all the directories provided. For example, if 'dir=NSAA' and '–add_dir=6minwalk-matfiles,6MW-matFiles' are both provided to 'model_predictor.py', it ensures that only models containing 'NSAA,6minwalk-matfiles,6MW-matFiles' in the model subdirectory names are used. This ensures that we can use 'model_predictor.py' in conjunction with 'rnn.py' when it has built models from multiple source directories.

12.13.1 Results Discussion

There are several things to note before continuing to an in-depth results discussion. For one, the only rows that we are focused on here are the rows with all the rows reflecting subjects assessed using 4 measurements rather than just joint angles. This is so that the results we see here can be more easily compared with results from previous model predictions sets, which predominantly make use of all 4 measurements. The reason we include the predictions made with just using the joint angles, however, is so we can potentially compare the results with models trained on NSAA and natural movement behaviour data sets: the natural movement behaviour data set ('allmatfiles') currently only exists as joint angles (at the time of carrying out this model predictions set), and so we would only be able to build and assess models built on these data sets using the joint angle measurement; hence, it's necessary to have models here that also have only been trained on joint angles to make a fair comparison between using additional 6-minute walk or natural movement files.

Another thing to note (that should be fairly evident) is the difference between the two types of lines. For each subject, we have either than 'standard' setup (i.e. as used in model predictions set 3, where the subject is not seen by the models before, uses the 4 measurement types of data to do the assessment, but is otherwise not modified), while the second type of line (i.e. with the subject name and '(add dir =...)') are assessing the subject on models that have been trained on files in the NSAA source directory and the directories described by 'add dir='. However, the subject that is being assessed, however, is still source from the NSAA directory. This is because we still want to see how well the model generalises to left-out subjects from the NSAA directory, just also wanting to see the effect of using additional files from other directories.

From the table above, the results are not particularly promising. When comparing the performance of models on left-out subjects that have been trained on either just NSAA files or NSAA files along with files from '6minwalk-matfiles' and '6MW-matFiles', **we see an overall drop in performance when using these additional directories: the cumulative difference between true and predicted NSAA scores rises from 20 to 30**. This is a dramatic loss in performance, though the performance for the 'percent of acts correctly predicted' and 'percent of correct predicted sequence' metrics remain relatively stable (however, these aren't as significant to assessing the models as the diff true/predicted overall score metric). This loss in performance is most likely due to the 6-minute walks between subjects being not nearly as distinctive between subjects as their corresponding NSAA assessments, as while the NSAA assessments test for many activities that could help models distinguish between subjects, the walk assessments aren't as comparatively distinctive. Hence, in forcing the models to train on the walk data as well as the NSAA assessments, we are essentially 'blurring the lines' between subjects for the models due to them having to accommodate this additional 'not-as-useful' walk data. This therefore reduces its ability to generalise on new subjects, as certain bands of overall NSAA scores are less associated with specific patterns of joint angles, positions, and so on than if it had only used the NSAA files. Therefore, **to assess left-out subjects from the NSAA directory we should not also train the corresponding models on the 6-minute walking data**.

12.14 Model Predictions Set 14: Model Performance for 5 Subjects w/ Measurement Feature Reduction via PCA

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	['position', 'sensorMagneticField', 'jointAngle', 'AD']	41.18%	D	100.0%	19	27
D3 (feature reduced = 20)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	41.18%	D	100.0%	19	27
D9	['position', 'sensorMagneticField', 'jointAngle', 'AD']	52.94%	D	100.0%	22	25
D9 (feature reduced = 20)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	75.0%	22	27
D11	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	27	28
D11 (feature reduced = 20)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	76.47%	D	100.0%	27	27
D17	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	50.0%	33	28
D17 (feature reduced = 20)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	88.24%	D	75.0%	33	27
HC6	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	75.0%	34	31
HC6 (feature reduced = 20)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	58.82%	D	0.0%	34	26

Referring back to MPS 11, we can see the results of concatenating the different measurement types that are used to train the models (excluding the 'AD' type) and subsequently reducing the dimensions down from 186 to 60 (as shown in the lines of the table containing 'feature concat = 60'). This model prediction set therefore makes use of feature concatenation as well as subsequent feature reduction capabilities of the scripts. However, we now wish to observe the results of excluding the feature concatenation step of the above process; in other words, here we're looking at reducing the dimensionality of each of the measurement types (69, 66, 51, and 30 for position, joint angle, sensor magnetic field, and AD, respectively) going into the models. The difference between this and model prediction set 11 (apart from reducing to a different dimensionality) is that here we are only building models from one measurement type at a time (as is the standard way currently that's been used thus far everywhere except MPS 11) but still reducing the dimensions of the measurements.

The aim here is therefore to see if the models are able to generalise to the 5 chosen left-out subjects in turn when the models have been trained on feature reduced measurements and the subjects that are being assessed via 'model_predictor.py' are also reduced to the same dimensionality for their features (so that they 'fit' into the models they are being assessed on). The hope is that, with choosing a lower dimensionality for the data, the models may be able to keep a majority of the inherent variance of the training data whilst also generalizing better to new subjects due to the comparative ease of models learning from lower dimensioned data as opposed to higher dimensioned data. For this model prediction set, we have chosen 20 as

the target lower dimensionality of each of the measurements; that is, models that are trained on position data will be reduced from having an input shape of (x, y, 69) to (x, y, 20), models trained on joint angles will be reduced from (x, y, 66), to (x, y, 20), and so on (excluding the 4th dimension based on batch size). This was chosen due to it being sufficiently small to hopefully make it easier to train the models whilst also capturing most of the inherent variance of the training data sets: generally, it captured between 80-95% of the variance of the data. It should also be noted that we apply the same dimensionality reduction technique to the 'AD' data which will have been noted to already have been reduced in dimensionality from approximately 4000 to 30 via 'ft_sel_red.py'. This additional application of feature reduction is more to keep the process homogenous with respect to all measurement types so we could say that all models would always expect a feature dimensionality of 20, irrespective of the input measurement data.

12.14.1 Results Discussion

As we can see from the above table, we again see no real improvement in modifying away from the standard approach setup in MPS 3. For the percent of acts correctly predicted metric (corresponding to the 'acts' output type), we see an improvement for 2 of the subjects when assessing on models that have been trained on feature reduced data (with the subjects in question also being reduced to the same dimensionality as the models) but a worsening for 2 subjects. Likewise, for the percent of correct predicted sequences (for the 'dhc' output type), we see that while reducing the feature dimensionality to 20 improves for subject 'D17', it worsens for subjects 'D9' and 'HC6'. Most significantly, however, is the performance of the model with respect to the difference between the true and predicted overall NSAA scores (corresponding to the 'overall' output type): **the sum of the differences between the true and predicted overall NSAA scores for the 5 subjects goes up from 20 to 27**, which is noticeably much worse than that of the 'standard' approach of MPS 3 (i.e. not reducing the dimensionality of each of the measurements any further within 'rnn.py' or 'model_predictor.py'). We can therefore say, as reducing the dimensionality to 20 shows no real improvement for the three output types, that **reducing the dimensionality of each measurement, including raw measurements and further reducing computed statistical values, has a negative effect on the ability of the models to generalise to unseen subjects**.

While these models would most likely have found it easier to train with this lower-dimensioned data, it is shown from the results above that this is not enough to offset the results of losing a portion of the inherent variance of the data. Even though this is generally not much variance that is lost when compressing the data to a lower dimensioned space via PCA, we can see that it is enough to reduce the performance of the models on new subject data as the comparative lack of information from the data is enough to make distinguishing between sequences of different overall NSAA scores, individual NSAA scores, and D/HC labels that much more difficult. As a result, **to help the models generalise towards new subjects it's necessary to**

keep all the dimensions of the measurement data that comes into the 'rnn.py' and 'model_predictor.py' scripts.

12.15 Model Predictions Set 15: Model Performance for 5 Subjects w/ Added Gaussian Noise

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	['position', 'sensorMagneticField', 'jointAngle', 'AD']	41.18%	D	100.0%	19	27
D3 (noise = 0.0001)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	47.06%	D	75.0%	19	26
D9	['position', 'sensorMagneticField', 'jointAngle', 'AD']	52.94%	D	100.0%	22	25
D9 (noise = 0.0001)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	58.82%	D	100.0%	22	27
D11	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	27	28
D11 (noise = 0.0001)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	64.71%	D	100.0%	27	26
D17	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	50.0%	33	28
D17 (noise = 0.0001)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	50.0%	33	28
HC6	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	75.0%	34	31
HC6 (noise = 0.0001)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	75.0%	34	30

A common technique that has been used in order to help with overfitting in machine learning problems (and therefore potentially help with the generalization problems we have encountered) is adding noise to the data that the model is trained on. In doing this, it generally forces the model to learn more general features of the data in order to approximate a more general function that models the training set, as opposed to be more easily being able to learn specific minutia of the training samples that wouldn't carry over onto the left-out set. Therefore, the theory in going into this model prediction set was that in adding noise, it might force our models to learn general features about movement data from position, joint angle, sensor magnetic field, and computed statistical value measurements and how they are connected with the true overall NSAA score, the D/HC classification, and the single-act NSAA scores; in other words, it was hoped that in adding noise to the training set it would help it better approximate to new, unseen subjects.

To carry this out, we first needed to compute the distribution of which we wished to draw from in order to add noise to our data set. With the data nearly fully preprocessed in 'rnn.py' (just prior to splitting into training and testing components), we add a certain amount of Gaussian-distributed noise to each sample. The reason we chose Gaussian for our noise distribution is that it's generally the typical distribution used for noise in most machine learning models due to it generally modelling most real-world distributions the best. We first flatten the data from it's form as separated

sequences to these 'stacked' sequences: we go from a x data shape of (a, b, c) to ($a * b$, c). We then calculate the standard deviations for each of these columns so we have c standard deviations total. Then, for each sample of this 'stack' of sequences (i.e. a vector of length c) and for each feature within this, we draw a random sample from the Gaussian distribution of $N(0, std(j) * k)$ where $std(j)$ is the standard deviation of the column that corresponds to the current feature and k is a constant factor. Hence, each value within the ($a * b$, c) matrix has a Gaussian distributed value added to it that is proportional to the constant factor k and the standard deviation of the column the value is contained within.

Regarding the constant factor, this is essentially a 'scaling' factor of how much noise to add to the data set. If this is set to 0, then no noise is added, while if we set it to 1 then each value within the data set has an amount added to it that is drawn directly from the Gaussian distribution with mean 0 and standard deviation set to the standard deviation of each column of the data set, which in most cases is far too large. Indeed, in trying to find a good value for the constant factor k we experimented with 1 first; however, this outright prevented the models from learning for a variety of different measurements as input types and for different output types, as we wouldn't see any drop in the training loss in training epochs. Indeed, we continued decreasing this factor and $k = 0.0001$ became the first value we encountered that enabled the training of a model to actually decrease with respect to the loss during training (i.e. showing that the models are able to now learn from the data). Therefore, the aim for this model predictions set is to directly compare the results of the 'standard' model setup to that of models with a small amount of added noise that is proportional to the standard deviations of each of the data's features.

12.15.1 Results Discussion

One thing we can say immediately about the addition of Gaussian noise to the training set is that **only a small amount of Gaussian noise is able to be added; otherwise the models won't train**. Therefore, the data can be said as being quite sensitive to the addition of noise, which implies that **the data doesn't contain any easily-recognizable patterns or trends that the models would be able to pick up on as, if there were, the models would probably be more resilient to noise than they are**. An alternative to this could be due to the interdependency of many of the features on each other. For example, the position values for several features (e.g. those representing the x, y, and z coordinates of the left-ankle) would be proportional to the position values of other features (e.g. those representing the coordinates of the left-leg) and so added random noise to each of these features, of which each addition of noise is independent of the others, would possibly distort this relationship. Therefore, this could possibly be avoided by instead adding noise per frame (i.e. over a vector of c values) rather than per value so as to not distort this relationship too much and therefore possibly allow us to work with higher levels of noise.

As for the results themselves from this addition of noise to the data set, alternative models are rebuilt with this addition of noise with $k = 0.0001$ and are used to evaluate left-out subjects (which don't have added noise to them as adding noise to the evaluation set in machine learning is generally avoided). These are then compared with the 'standard' setup results which have still proved to be the best at generalization to left-out subjects. We can see in the table above that the difference between the true and predicted overall NSAA scores only worsened when using models built using this 'noise-added' data, as the accumulation of these differences between the overall NSAA scores rises from 20 to 22. While performance with respect to the metrics relevant to the 'dhc' output type (i.e. the percent of correct predicted sequences and the predicted D/HC label) remains the same and the performance with respect to the 'acts' output type (i.e. percent of acts correctly predicted) marginally improves for 2 subjects while decreasing for only 1, as a result of the worsened results for the 'overall' output type **we consider these noise-added models marginally worse than the standard setup at generalizing to left-out subjects**. This is most likely due to there not being enough noise added to the data set to help it generalize properly, while adding the requisite noise in practice distorts the data enough to make it impossible to train on.

12.16 Model Predictions Set 16: Model Performance for 5 Subjects w/ Included Joint Angle Data of Natural Movement Behaviour

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	['jointAngle']	47.06%	D	90.4%	19	27
D3 (additional dir = allmatfiles)	['jointAngle']	47.06%	HC	48.44%	19	27
D9	['jointAngle']	58.82%	D	85.62%	22	25
D9 (additional dir = allmatfiles)	['jointAngle']	58.82%	HC	34.38%	22	24
D11	['jointAngle']	58.82%	D	72.97%	27	27
D11 (additional dir = allmatfiles)	['jointAngle']	82.35%	D	52.66%	27	27
D17	['jointAngle']	76.47%	D	89.84%	33	27
D17 (additional dir = allmatfiles)	['jointAngle']	82.35%	D	67.19%	33	27
HC6	['jointAngle']	100.0%	HC	61.72%	34	29
HC6 (additional dir = allmatfiles)	['jointAngle']	100.0%	HC	84.77%	34	31

From all the model prediction sets we have undertaken thus far, there is a common theme running through many of them: the more data that we have, the better results we get on left-out subjects. This includes including any outliers, not downsampling the data, not reducing the dimensionality of the raw measurements, and so on. However, this doesn't necessarily apply to including data from the 6-minute walk assessments: as we see in MPS 13, adding 6-minute walk data to the data set along with NSAA files doesn't result in any better assessments made on new subjects'

NSAA files via 'model_predictor.py'. It was noted, though, that the type of movement behaviour expressed in these 6-minute walk files are predominantly different than the behaviour expressed in the NSAA files, as these files only contain a fraction of the amount of movement that's similar to walking while the majority of the other data within the file includes the subject climbing boxes, getting off the floor, hopping on one leg, etc. This variety of movement expressed in the NSAA files and thus what we required the model to learn so we can assess a new subject's NSAA file is therefore not helped by adding in walking data. What might help, however, is adding in natural movement behaviour data of the subjects: the larger variety of movement of the subjects in natural movement behaviour files is more likely to show a variety of movement contortions that would be more useful to models in order to learn generalized correlations between NSAA scores and subjects' movements than adding in extra 6-minute walk data that doesn't provide any additional insight to the data that we already have.

To this end, we intend to repeat MPS 13 but with using the 'allmatfiles' directory as the additional directory (along with the 'NSAA' directory) instead of '6minwalkmatfiles' and '6MW-matFiles'. However, there is a problem with this direct approach: there exists far more natural movement behaviour data than in the NSAA or either of the two 6-minute walk directories. It was felt that including the entirety of the 'allmatfiles' data set would cause the models that are training from both NSAA and 'allmatfiles' files to prioritize learning from the natural movement behaviour files more so than the NSAA files as it would be far more conducive in the training process to focus on optimizing on the 'allmatfiles' data to reduce overall training loss. By way of comparison, each RNN model is trained on approximately 13,000 sequences from the NSAA data set, while we have >100,000 sequences to draw from in 'allmatfiles'. Furthermore, the data from 'allmatfiles' is less likely to be as useful to models that needs to make inferences about new subjects' NSAA scores than data from NSAA files due to the natural movement files not containing actual NSAA activities.

To reduce the size of the data set of the 'allmatfiles' directory, we add an optional argument, '–balance_allmatfiles', to 'rnn.py' that takes in an integer value. This value specifies the maximum number of files per subject that can be drawn from the 'allmatfiles' directory when we're loading files into 'rnn.py'. For example, setting '–balance_allmatfiles=3' results in a maximum of 3 files per short subject name to be drawn from the 'allmatfiles' data set. To maintain a variety of different type of movement files used, these 3 files per subject are chosen at random from the possible subjects. As a result, instead of drawing data from 452 files we instead use 62 from 'allmatfiles' and, therefore, we go from using >100,000 sequences from 'allmatfiles' to just using approximately 14,000, which is much more comparable to the size of the NSAA data set. We hope that with using only this amount of natural movement behaviour data that the data will complement the use of data from the NSAA data set for the models learning how to assess new subjects' NSAA files, as opposed to overwhelming the model with natural movement behaviour data and possibly impacting its ability to generalize to these new subjects' NSAA assessments.

12.16.1 Results Discussion

One thing to be reminded of at this point is that we currently only have the 'allmatfiles' data set in joint angle form. In other words, we have source '.mat' files for the 'allmatfiles' data but these source files currently only contain the joint angle values, as opposed to also containing data for the other >10 raw measurements. Note that this is down to, at the time of writing, all the data directly from the suits in '.mvnx' format having not been converted to '.mat' format and is something we will hope to obtain in the near future. Hence, to make a fair comparison, we compare models that have been built solely on the NSAA directory from the joint angle measurement to those built on files from NSAA and 'allmatfiles', also only on joint angles. This is also why the results why the rows of the table corresponding to the 'standard' setup (i.e. that don't contain 'additional dir' in the name) differ in results to those seen in MPS 3.

An immediate observation that we can make from the above table is the apparent disconnect between the performance for the output types concerning NSAA scores ('acts' and 'overall') and the performance for the D/HC classification output type ('dhc'). In using the additional data from the 'allmatfiles' data set with '–balance_allmatfiles=3', we see that models build using this argument end up predicting two subjects as the incorrect classification (assessing both 'D3' and 'D9' as 'HC'). Additionally, the two 'D' subjects that it gets correct ('D11' and 'D17') have a much lower percentage of correctly predicted sequences with the correct classification when using these models as opposed to models with the 'standard' setup (i.e. only using the 'NSAA' directory). Therefore, this implies that **using additional data in the form of 'allmatfiles' data results in models predicting new subjects with less accurate D/HC classifications than if we only used the 'NSAA' directory**. One reason for this might be due to natural movement behaviour blurring the lines between what is considered 'healthy' movement ability, which may come from using NMB files where the subjects aren't performing tasks that don't discriminate particularly well between subjects with DMD ('D') and those without ('HC'); for example, natural movement behaviour files where the subject is sitting, talking, or eating won't be as useful for models to help classify the subject compared with activities that are specifically designed for the task (i.e. that can be found in files from the 'NSAA' directory).

However, this is not consistent with the results we see with respect to the metrics concerned with the NSAA scores: here, we see that the percentage of acts correct predicted increases for two subjects while decreasing for none when using models trained on NSAA and NMB files compared to just having trained on NSAA files. We can also see that the accumulation of the differences between the true and predicted overall NSAA scores decreases from 22 when using models trained only on NSAA to 19 when using models trained on both NSAA and NMB files. Hence, **the fact that we see improvement for the metrics concerning the NSAA scores implies that using additional natural movement behaviour files along with NSAA files helps with assessing the NSAA scores (both single and overall) of new subjects even**

though the D/HC classification is worse. One reason for this could be that, while the NMB data blurs the line between what's considered healthy or DMD behaviour due to having more 'unclassifiable' activities (e.g. sitting and eating), it also would provide more activities that are in 'allmatfiles' that contain characteristics specific to certain overall scores or sequences of single-act scores. So while this requires further experimentation when we have all measurements available (as opposed to just using models built from joint angles as done here), we can tentatively state that **the use of supplementary data from the NMB directory 'allmatfiles' is preferable to only using data from the 'NSAA' directory when we are more concerned with performance for the 'overall' and 'acts' output type than 'dhc'**. We shall further investigate this in MPS 20.

12.17 Model Predictions Set 17: Model Performance for 5 Subjects using a Single Sequence from Each Subject

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score
D3	['position', 'sensorMagneticField', 'jointAngle', 'AD']	41.18%	D	100.0%	19	27
D3 (single sequence)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	19	28
D9	['position', 'sensorMagneticField', 'jointAngle', 'AD']	52.94%	D	100.0%	22	25
D9 (single sequence)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	52.94%	D	100.0%	22	27
D11	['position', 'sensorMagneticField', 'jointAngle', 'AD']	70.59%	D	100.0%	27	28
D11 (single sequence)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	58.82%	D	100.0%	27	28
D17	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	50.0%	33	28
D17 (single sequence)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	94.12%	HC	0.0%	33	30
HC6	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	HC	75.0%	34	31
HC6 (single sequence)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	100.0%	D	0.0%	34	32

As each of the models are trained on sequences rather that complete files, it stands to reason that we are only able to test subjects on prebuilt models by using sequences from the assessing file(s). To do this, the assessed file is split into sequences that are each used as input to the models that are fed to the models in batches. The results from each of these sequences from the assessed file obtained by 'model_predictor.py' are then averaged together to provide assessment for the subject for a given input type and output type; in other words, the assessments for various metrics for a single model (e.g. joint angles of the subject used to predict the overall NSAA score) are in fact the average of the results of each of the sequences from the subject fed through the model. This process repeats for each measurement we are using and each output type we are targeting to then make accumulated predictions over all input and

output types.

Here, however, we wish to look at the possibility of drawing a single sequence from the assessed file rather than dividing the source file into numerous sequences and finding the average. For example, let's say we are assessing subject 'D9' on models that haven't seen 'D9' during training, and let's say we are specifically concerned with the model trained on joint angle data for the 'overall' output type. Traditionally, we would take corresponding subject file(s) for 'D9' and divide it up into sequences: if the joint angle information for subject 'D9' (as the file 'D9_jointAngle.csv') is of shape (3000, 66), we would divide this into 50 sequences so the new data shape would be (50, 60, 66), feed these 50 sequences into the relevant model, and average the 50 predictions to obtain the predicted overall NSAA score for that subject for the joint angle input type and 'overall' output type as a single prediction. For this model prediction set, however, we take a different approach: from an input data shape of (3000, 66), we instead draw a single sequence that is sampled throughout the data so that every 50 rows of data we sample a row ('frame') of data, repeating throughout the file. Hence, from data of a shape (3000, 66) we get (1, 60, 66).

The idea behind this was to investigate whether or not there are obvious issues with our initial assumption of being able to assess a complete file by assessing on its individual sequences and combining the results of these sequences. Therefore, if the 'single sequence' method is evidently superior, then we can infer that we cannot realistically assess on a complete subject file in 'model_predictor.py' by splitting it into sequences. However, we do not expect this to be the case due to a single reason: the creation of a single sequence where each frame is temporally separated by 59 other frames in the source '.mat' file would most likely result in each frame being temporally unrelated and so would impact the RNN's ability to make use of its 'memory' within the LSTM units, as for a given raw measurement the states of the hidden units would modify the next frame of a sequence going through the network based on the assumption that the previously seen frame haven't existed '1 frame in time' prior to the current one, while in actually it would exist '59 frames in time' prior to the current one. Hence this expected incorrect assumption that results in the models' LSTM units incorrect use of its own memory would most likely lead it to draw incorrect conclusions, so we would expect this approach to not work as well.

12.17.1 Results Discussion

It should be noted first and foremost that the assessments made for a given subject are using the exact same models for a given 'left-out' subject for both using a single sequence and using the 'standard' approach of multiple sequences. For example, for the two rows concerning the 'D3' left-out subject based on the 'standard' and 'single-sequence' approach, they both use the same models to predict on, and the only difference between them is how the 'D3' files for the different measurements are preprocessed by 'model_predictor.py' (i.e. whether they are divided into a single sequence or not).

The results concerning the 'overall' and 'acts' output types are surprisingly similar to that of the 'standard' setup as used in MPS 3: in using the 'single-act' approach over the 'standard' approach, we see for the accumulation of the differences between the true and predicted overall NSAA scores decrease from 20 to 19. However, as this is a very small decrease and where the models improve on are for the higher-scored subjects like 'D17' and 'HC6' (where improvements aren't as important as lower-scored subjects like 'D3' due to requiring our models to improve mainly in this area due to them being particularly underperforming for low-scored subjects), we can't see this as an overall improvement. For the percent of acts correctly predicted metric (for the 'acts' output type), we see a similar case as the percent improves for one subject but worsens for another ('D3' and 'D11', respectively). Therefore, **these results surprisingly indicate that the models do not assess any worse with respect to the 'acts' and 'overall' output types in using a single-sequence from the assessing file over using all available sequences, though also don't really perform better**.

However, while the results for the 'overall' and 'acts' output type metrics are comparable, the results for the D/HC classification are notably worse: using single-sequences results in 2 out of 5 of the subjects being misclassified, as opposed to only 1 out of 5 when using the standard approach. Additionally, for the assessments made with single sequences it 'completely' misclassifies them: 0% of the sequences from 'D17' and 'HC6' are correctly classified as being 'D' or 'HC' sequences, respectively. This is far worse than their equivalents in using the standard approach, where even in the case of incorrectly classifying 'D17' as being 'HC' it still managed to get half of the sequences classified as correct (note that 'model_predictor.py' was coded in a way that requires more than 50% of the sequences to be classified as 'D' to get an overall file label of 'D'). Hence, **the use of the single-sequence assessments of subjects show that it is an inferior approach to using all the sequences available from the subject for D/HC classification**. Therefore, we can conclude that as a result of not improving with respect to two of the output types and noticeably worsening for the third, **we shall remain with the approach of drawing the full amount of sequences from the given assessed subject's file(s)**.

12.18 Model Predictions Set 18: Model Performance for 5 Subjects w/ Aggregated Assessments for Overall NSAA Score

File Name	Measurements tested	True Overall Score	Predicted Overall Score	Aggregated Predicted Overall Score
D3 (aggregate overall)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	19	27	27
D9 (aggregate overall)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	22	25	24
D11 (aggregate overall)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	27	28	28
D17 (aggregate overall)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	33	28	32
HC6 (aggregate overall)	['position', 'sensorMagneticField', 'jointAngle', 'AD']	34	31	33

One thing that we have emphasized in many of the model predictions sets thus far is the relative importance of the 'overall' NSAA score compared to the other output types. This is primarily due to two reasons: it is the easiest of the three to evaluate generalization performance of left-out subjects, and it will most likely be the most important output of the system to specialists in order to give us information about the subject. Therefore, we wanted to add an option to 'model_predictor.py' to be able to somehow combine the assessments made for 'acts', 'dhc', and 'overall' into a more accurate 'overall' prediction for the given subject. The idea behind this is similar to the reasoning behind the combining of models trained on different measurements: we are expecting to get more accurate results from combining different models for different output types than models on their own, as we can see having been done regarding input types for MPS 4. We have seen numerous cases where, for example, subject 'HC6' has been correctly classified as an 'HC' subject with 100% of its acts correctly predicted but not having its overall NSAA score being predicted as 34. In other words, two out of the three output types have been correctly determined by models. In this case, the overall score can be inferred by the other two: if two other output types believe the subject is a 'HC' subject, then it stands to reason that they would believe it has an overall NSAA score of 34, which is not what the models for the 'overall' output type predicts. Therefore, in cases like these it would behove us to use all three types of model output to correctly assess the overall NSAA score.

To that end, an additional function was added, 'combine_preds()', to 'model_predictor.py'. Thus function takes in the strings that are to be outputted to console and written to 'model_predictions.csv' and does the following:

1. From the relevant strings (i.e. the output strings corresponding to predictions made by the models, not the true corresponding values), extract the predictions made for the predicted 'acts' value (i.e. the list of 17 values of 0, 1, or 2), the predicted 'D' or 'HC' classification, the percentage of sequences correctly classified, and the predicted overall NSAA scores.

2. For each of the three output types, compute an approximated overall NSAA score. For the 'overall' output type, this is easy: it is the value predicted by

the models for the 'output type'. For the 'acts' output type, again this is fairly simple: the sum of the 17 values made by the models aggregated over all input types would be the overall NSAA score predicted by models trained for the 'acts' output type. For the 'dhc' output type, however, this is slightly more complex: if the predicted D/HC label is 'HC', then the corresponding overall NSAA score by 'dhc' models would be 34. However, if it predicted 'D', then we find the 'median' value of all overall scores for the 'D' subjects in the 'nsaa_17subtasks_matfiles.csv' table and assign that value: therefore, the "correct" overall NSAA score for models built for the 'dhc' output type would be this median value.

3. Take the average of the overall NSAA scores predicted by the three types of model output to get the aggregated overall NSAA score over the three output types. It's worth noting, however, that the value for the 'dhc' output type is averaged with the others in proportion to the percentage of correct predicted sequences. This is because the D/HC classification has a much narrow band of values to be added aggregated (i.e. either the median of 'D' subjects or 34, depending on the predicted D/HC label). For example, if we were assessing for the left-out subject 'HC6' and the models predicted a sequence of all 2's for the 'acts' models, an overall NSAA score of 30, and a D/HC label of 'HC' with 75% of sequences correctly predicted, the aggregated prediction would be computed as:

$$aggregate\,overall\,NSAA = \frac{\sum[2,2,2,...,2]+30+(30*0.75)}{2+0.75} = 32.55$$

This is then rounded up to an aggregated score of '33', which is closer to the true overall NSAA value of 34 (as it's an 'HC' subject) than by just using the 'overall' models, which would have predicted a '30', due to taking advantage of all three output types.

12.18.1 Results Discussion

As we are only concerned with aggregating predictions made here, for the table above we run 'model_predictor.py' only once per subject (and on the 'standard' models seen in MPS 3), as opposed to most of MPS's where it is run once with the 'standard' setup and another time with some other option(s) set. Instead, we are here comparing two types of model output to each other directly: when the '–combine_preds' argument is set, both the predicted overall NSAA score that is predicted by 'overall' models and the aggregated predicted overall NSAA score (as discussed above) are written to file. We can see that this approach sees a noticeable improvement: the accumulative difference between the true and predicted overall NSAA score decreases from 20 when using the predicted values from the 'Predicted Overall Score' column to 13 when using the 'Aggregated Predicted Overall Score'. Therefore, in the case of the 5 left-out subjects **there is a noticeable improvement of the predictable ability of the models with respect to the overall NSAA score**

predictions when aggregating the predictions made by all three output types.

In particular, we don't see any real improvement for the subjects with a low or 'medium' overall NSAA score. However, when we apply this to subjects that have a higher score (e.g. 'D17' or 'HC6'), the results show much more accurate overall NSAA scores. This is due to the performance of the 'acts' and 'dhc' models for these subjects being particularly good, and so helps compensate and 'bring up' the scores predicted by 'overall' models that are lower than they should be. Additionally, another important point to draw is that we don't see any worsening of the predictions made for the overall NSAA scores for lower scoring subjects; rather, we simply see a non-improved value. Therefore we can say that based on the above results that **the aggregation of the output types to estimate the overall NSAA score is a good approach to use for subjects with a variety of degrees of severity of their condition**.

12.19 Model Predictions Set 19: Model Performance for 5 Subjects w/ All Available Data Used for Training

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score	Aggregated Predicted Overall Score
D3 (additional dir = allmatfiles) (aggregate overall)	['jointAngle']	47.06%	HC	48.44%	19	27	30
D3 (additional dir = allmatfiles) (aggregate overall) (no testset)	['jointAngle']	47.06%	D	55.13%	19	27	28
D9 (additional dir = allmatfiles) (aggregate overall)	['jointAngle']	58.82%	HC	34.38%	22	24	29
D9 (additional dir = allmatfiles) (aggregate overall) (no testset)	['jointAngle']	82.35%	HC	50.0%	22	26	27
D11 (additional dir = allmatfiles) (aggregate overall)	['jointAngle']	82.35%	D	52.66%	27	27	28
D11 (additional dir = allmatfiles) (aggregate overall) (no testset)	['jointAngle']	82.35%	D	51.25%	27	26	27
D17 (additional dir = allmatfiles) (aggregate overall)	['jointAngle']	82.35%	D	67.19%	33	27	27
D17 (additional dir = allmatfiles) (aggregate overall) (no testset)	['jointAngle']	94.12%	D	68.75%	33	28	29
HC6 (additional dir = allmatfiles) (aggregate overall)	['jointAngle']	100.0%	HC	84.77%	34	31	33
HC6 (additional dir = allmatfiles) (aggregate overall) (no testset)	['jointAngle']	100.0%	HC	83.59%	34	32	33

Having established that providing additional data to the models trained on NSAA in the form of a selection of files from the natural movement behaviour 'allmatfiles'

data set along with the preference to use the 'aggregation' option to make an aggregate overall NSAA score prediction, we now turn our attention to what has been an oversight amongst most of the model predictions sets undertaken thus far. The default 'train-test-split' ratio for training the models up until now has been 0.2, which means that 80% of the data that is provided to the models as training data and 20% is used as test data. This test data was what was used during the experiment sets to assess the models' performance for various measurement types, sequence lengths, and so on. However, for many of the model predictions set we also leave out certain subjects' data for certain models, and thus this is the data that we assess on, not the 20% of the data that is left out of the data set after the 'left-out' subject's data is removed from the data set. Therefore, all that leaving out this 20% from the training set does is deprive us of 20% of the data that could be used to help train the model, which would hopefully achieve better generalization performance while not impacting our ability to test on 'left-out' subjects, as these subjects would still not be familiar to the models they are assessed on even with the inclusion of the models' test sets (which won't contain any of the 'left-out' subject data as it had been partitioned after the removal of the 'left-out' subject had taken place).

To achieve this, we simply added an optional argument to 'rnn.py', '–no_testset', that sets the train-test ratio used by the 'train_test_split()' function to 0. Upon setting this, all of the data available (that is not covering the 'left-out' subject in question) is used when the RNN object calls its 'train()' method. As a result of having no test data, the 'predict()' method is not called, along with no results being written to the 'RNN outputs' directory (as since there is no test data there can be no predictions made of which to compare the data): when the 'train()' method is complete, the program that created this model finishes. Additionally, the 'model_predictor.py' script is modified to be able to call models that have had the '–no_testset' optional argument set.

12.19.1 Results Discussion

Having noted in MPS 18 previously that the aggregate overall NSAA score that can be calculated is the most useful output metric to us (as thus is the target we're most interested in optimizing, along with being a good reflection of the performance of all three output types), here we exclusively focus on the models' performance with respect to this metric. It's worth noting that both types of models for a given 'left-out' subject are trained on both 'NSAA' and the 'allmatfiles' directories; hence, we can only use the joint angle measurement as this is the only form that 'allmatfiles' currently takes (natural movement behaviour data with more than just joint angles will be discussed later when the data is obtained). The only difference between the two types of rows within the table is that the second row in each group of two are trained on models that have the full 100% of the data used, rather than just 80% partitioned for the training set. It should also be noted that, in both cases, the subject is being assessed by 'model_predictor.py' in exactly the same way, only using either models trained on 80% of the available data or models trained on 100% of the available data.

Unsurprisingly, **we see more promising results with respect to the aggregated predicted overall score for models trained on 100% of the data** (i.e. when option '–no_testset' is used). There is an overall decrease in the cumulative difference between the true and aggregate predicted overall NSAA scores from 26 to 19, which is a fairly significant boost to performance. This is most likely down to the usefulness of this additional 20% of the data used for training, where we see significant improvements with respect to the 'acts' and 'overall' output types through the percent of acts correctly predicted and percent of correctly predicted sequences metrics. This additional data therefore most likely helps the models to generalize to new, unseen subjects better due to being exposed to new sequences with their corresponding labels. Hence, **we therefore shall set the models to use all available data when training models to assess on unseen subjects via 'model_predictor.py'**.

12.20 Model Predictions Set 20: Model Performance for 5 Subjects w/ All Chosen Measurements from Natural Movement Behaviour Data

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score	Aggregated Predicted Overall Score
D3 (additional dir = NMB) (aggregate overall) (no testset)	['position']	52.94%	D	78.57%	19	28	27
D3 (additional dir = NMB) (aggregate overall) (no testset)	['sensorMagnetic Field']	52.94%	D	73.44%	19	23	24
D3 (additional dir = NMB) (aggregate overall) (no testset)	['jointAngle']	47.06%	D	94.87%	19	28	27
D3 (additional dir = NMB) (aggregate overall) (no testset)	['AD']	58.82%	D	68.75%	19	28	26
D9 (additional dir = NMB) (aggregate overall) (no testset)	['position']	41.18%	HC	0.94%	22	27	32
D9 (additional dir = NMB) (aggregate overall) (no testset)	['sensorMagnetic Field']	52.94%	D	99.69%	22	25	26
D9 (additional dir = NMB) (aggregate overall) (no testset)	['jointAngle']	52.94%	D	95.94%	22	26	27
D9 (additional dir = NMB) (aggregate overall) (no testset)	['AD']	41.18%	D	90.62%	22	26	28
D11 (additional dir = NMB) (aggregate overall) (no testset)	['position']	47.06%	D	64.53%	27	25	25
D11 (additional dir = NMB) (aggregate overall) (no testset)	['sensorMagnetic Field']	70.59%	D	99.69%	27	25	25
D11 (additional dir = NMB) (aggregate overall) (no testset)	['jointAngle']	82.35%	D	76.88%	27	25	27
D11 (additional dir = NMB) (aggregate overall) (no testset)	['AD']	58.82%	D	95.31%	27	29	29
D17 (additional dir = NMB) (aggregate overall) (no testset)	['position']	88.24%	HC	35.16%	33	28	31
D17 (additional dir = NMB) (aggregate overall) (no testset)	['sensorMagnetic Field']	94.12%	HC	7.03%	33	33	34
D17 (additional dir = NMB) (aggregate overall) (no testset)	['jointAngle']	88.24%	D	96.09%	33	27	27
D17 (additional dir = NMB) (aggregate overall) (no testset)	['AD']	94.12%	D	90.62%	33	29	29
HC6 (additional dir = NMB) (aggregate overall) (no testset)	['position']	100.0%	HC	100.0%	34	32	33
HC6 (additional dir = NMB) (aggregate overall) (no testset)	['sensorMagnetic Field']	100.0%	HC	97.27%	34	34	34
HC6 (additional dir = NMB) (aggregate overall) (no testset)	['jointAngle']	100.0%	HC	52.34%	34	30	32
HC6 (additional dir = NMB) (aggregate overall) (no testset)	['AD']	100.0%	D	12.5%	34	26	28

At this point in the project, we have obtained the natural movement behaviour in their 'AD' form: in other words, we have the all captured measurements from the suits of the natural movements of the subjects in the form of the 'NMB' data set. We have also made several conclusions based on the previous model predictions sets that modify the way we build our models that include:

- Using a number of natural behaviour files in the joint-angle-only form (i.e. from the 'allmatfiles' data set) proved to be more useful than not including any natural movement files, so we therefore concluded that adding in natural movement behaviour would be beneficial to the generalisation ability.
- Computing the overall NSAA score using an aggregation of all output types proved to be a better approximator of the true value, hence it was decided to compute this value in addition to the other metrics.
- Adding in all available data to the training set of the models is preferably to keeping 20% of it aside for testing purposes, as we are evaluating the models on 'left-out' subjects and so the 20% for testing the model by 'rnn.py' isn't necessary or even used in this evaluation process.

Therefore, we wish to apply the three conclusions made outlined above to making use of the all chosen measurements-form of the natural movement behaviour, which is contained within the 'NMB' data set. To do so, we first downloaded the complete data set (approximately 106GB of data), followed by pushing all this data through our data pipeline (i.e. 'comp_stat_vals.py', 'ft_sel_red.py', and 'ext_raw_measures.py', though not 'mat_act_div.py' as since these aren't NSAA files then there are no 'single activities' to extract from them). Note that the 'setup.cmd' batch script has now been modified to carry out the above steps for this data set as well as for the other data sets.

Once this was completed, the necessary models were built. These were built in a similar manner to most MPS's having done so before, with one model built per input type (measurement) and output type pairing and, as there are 4 input types we are concerned with (position, joint angle, sensor magnetic field, and computed statistical values) and 3 output types ('overall', 'acts', and 'dhc'), there are 12 models built per subject left-out of the data set, which gives us 60 models in total. Unlike other MPS's, however, we decide to set the 'dir' argument to 'NSAA,NMB' so we use both data sets, while the optional argument '–balance_nmb=3' is set to use only 3 files per subject from the 'NMB' as outlined in MPS 16. We also set the '–no_testset' argument for all models so all available data is used for training and, when we do predictions using 'model_predictor.py', we set the '–combine_preds' argument so we get the aggregate predicted overall NSAA score as well as outputs for the other metrics.

12.20.1 Results Discussion

With the models built as described above, we next turned our attention to how these models should be evaluated. It was decided that, as models have been built

for each measurement type and each left out subject, it would be advantageous to revisit different input types used to build models and how the results differ for the left-out subjects. In essence, we are revisiting MPS 4, only with additional natural movement behaviour data used along with additional optional arguments set in 'rnn.py' and 'model_predictor.py' to use all available data as training and to compute the aggregate overall NSAA score. In particular, at this point we were quite keen to see stronger generalisation properties of the models to left-out subjects, and so were motivated towards revisiting using stand-alone measurements, as we have been using them all together in most model predictions sets throughout the experimentation process. Therefore we evaluated each of the 5 left-out subjects on their respective models (i.e. the models that have not seen any part of the left-out subject during training) and on models trained on one particular input type, using the matching input type from the left-out subject on each. For example, for the first row of the table, we take the position data from 'D3' and evaluate it on the three models (one for each output type) that were trained on the position data of all subjects except 'D3'.

For the reasons outlined in MPS 18, we turn the focus of our evaluation to the aggregated overall NSAA score. We then use this to compute, for each of the measurements type (i.e. the type of model within the 'Measurements tested' column), the total over all 5 left-out subjects for the absolute difference between the true overall NSAA score and the aggregate predicted overall NSAA score. For instance, for position the total would be:

$$abs(19, 27) + abs(22, 32) + abs(27, 25) + abs(33, 31) + abs(34, 33) = 8 + 10 + 2 + 2 + 1 = 23$$

If we then repeat this for all measurements we get: **position = 23, sensorMagneticField = 12, jointAngle = 21, and AD = 25**. The most obvious thing of course for the above results is the relative performance for the sensor magnetic field measurement type in comparison to the others: **over all 5 left-out subjects, we see when we use the sensor magnetic field data from the subjects on models trained on sensor magnetic field data that the three models used for the aggregated overall NSAA score prediction are on average 2.4 away from the true value**. Relative to the range of true overall NSAA score values and the performance of other measurements (including that of combining the results over all measurements as done for many model predictions sets), this is an incredibly promising performance. Why this is so much more useful than the other measurements is at this point unknown. It should also be noted that the result of '12' for sensor magnetic field data is a large improvement from '18' previously observed for the sensor magnetic field measurement in MPS 4; this is further evidence that **modifying the models with extra natural movement behaviour data, using all available non-subject-left-out data, and using the overall aggregated score technique is a superior way of constructing the models than using none of these**.

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score	Aggregated Predicted Overall Score
D3 (additional dir = NMB) (aggregate overall) (no testset)	['position', 'sensorMagnetic Field', 'jointAngle', 'AD']	52.94%	D	78.57%	19	27	26
D9 (additional dir = NMB) (aggregate overall) (no testset)	['position', 'sensorMagnetic Field', 'jointAngle', 'AD']	41.18%	D	0.94%	22	26	28
D11 (additional dir = NMB) (aggregate overall) (no testset)	['position', 'sensorMagnetic Field', 'jointAngle', 'AD']	58.82%	D	64.53%	27	26	26
D17 (additional dir = NMB) (aggregate overall) (no testset)	['position', 'sensorMagnetic Field', 'jointAngle', 'AD']	94.12%	HC	35.16%	33	29	32
HC6 (additional dir = NMB) (aggregate overall) (no testset)	['position', 'sensorMagnetic Field', 'jointAngle', 'AD']	100.0%	HC	100.0%	34	30	33

To validate the claim that better results would not be obtained by combining predictions made over several measurement types, we repeated the above evaluations using 'model_predictor.py' but, this time, used all four measurement types of the subject in question at once to assess the subject, rather than using one at a time. The resulting total of the differences between the aggregated predicted overall NSAA score and the true overall NSAA score over all the left-out subjects is **16 when using all the measurements to predict on**. While these results are much better than solely using position, joint angle, or computed statistical values, this method of measurement aggregation is still inferior to that of using solely sensor magnetic field values, and thus **our focus shall be on using solely sensor magnetic field results to assess subjects**. However for a future MPS, if the sole use of sensor magnetic fields when used across all subjects doesn't perform as well as hoped for, then we still have a reasonable alternative in using the measurement aggregation approach as shown in the table above.

12.21 Model Predictions Set 21: Model Performance for 'V2' Files Left Out

Short file name	Measurements tested	Predict D/HC label (short file name)	True overall NSAA score (other version)	Predict overall NSAA Score (short file name)	True single act scores (other version)	Predict single acts scores (short file name)
D4V2-005	['sensorMagneticField']	D	20	23	[1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1]	[2, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
D4V2-005	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	20	26	[1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 1, 1, 1]
D4V2-010	['sensorMagneticField']	D	20	20	[1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1]	[1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1]
D4V2-010	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	20	25	[1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1]	[2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 2, 0, 2, 1, 2, 1]
D5V2-010	['sensorMagneticField']	D	34	27	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 1, 1]
D5V2-010	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	34	27	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 1, 1, 1]
D5V2-011	['sensorMagneticField']	D	34	27	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0, 2, 2, 1, 1, 1]
D5V2-011	['position', 'sensorMagneticField', 'jointAngle', 'AD']	HC	34	32	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2]
D6V2-007	['sensorMagneticField']	D	30	23	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1, 1]	[2, 1, 1, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
D6V2-007	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	30	29	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1, 1]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
D6V2-008	['sensorMagneticField']	D	30	26	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1, 1]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 1, 1, 1]
D6V2-008	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	30	30	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1, 1]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1]
D7V2-002	['sensorMagneticField']	D	26	23	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[2, 2, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 0, 2, 1, 1, 1]
D7V2-002	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	26	27	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 1, 1, 1]
D7V2-004	['sensorMagneticField']	D	26	21	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[2, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1, 1]
D7V2-004	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	26	27	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 1]
D7V2-03	['sensorMagneticField']	D	26	12	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[1, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
D7V2-03	['position', 'sensorMagneticField', 'jointAngle', 'AD']	D	26	26	[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2]	[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 1, 1]

At this point in the process, we've established to a reasonable degree of satisfaction the optimal setup for the models and the different combinations of input data as well as ways to preprocess and postprocess the data through many experiment sets and model predictions sets, and in particular to help generalise towards subjects that have been left-out of training. Now, we turn our attention to one of the fundamental aims of the project: to help assess subjects' symptoms of DMD as they progress over time. The data sets that we have (including the NSAA data, the natural movement behaviour, and the 6-minute walk assessment data sets) contain, for some subjects, subject names with 'V2' attached to them. For example, we have both 'D4' and 'D4V2' files within the NSAA data set, which for the purposes of this model predictions set we consider as two separate subjects. In actuality, however, they are the same subject but having done the assessment approximately 6-months after the first.

The primary goal of this model predictions set is to build models on all available data for a given 'version' (i.e. in this case, files for subjects which do not have 'V2' in their name and are therefore 'V1' files) and then generalizing these models to subject

files with 'V2' in their name and assessing their performance. Hence, while the models will be familiar with the subject itself, it may not be familiar with that subject's assessment version: for instance, we train models using 'D4' data but not 'D4V2' data, so when the models come to assess 'D4V2' data, the models will have seen the 'D4' subject before in training but no data from the subject's second assessment. The hope is that this will allow us to observe how well the models can be used for monitoring the progress of DMD within subjects. If proved successful, it would provide a tool for specialists in aiding with subjects in the study coming in for further assessments. It should be noted though that this is in contrast to many of the previous model predictions sets, where our primary focus was on building models that generalised to completely new subjects, while here we look at generalising to the same subjects but different versions of their files. Also, it's worth noting here that we have no corresponding entries for any 'V2' assessments within 'nsaa_6mw_info.xlsx': that is to say, we don't have information about the true overall NSAA and true single activity scores for any of the 'V2' subjects. Hence, when we go to predict on them, we don't have any corresponding true values to compare them to in order to know whether or not the model is generalizing well. However, we can use a rudimentary understanding of DMD progression to assume that in most cases, the performance for a 'V2' subject (e.g. 'D4V2') would be on average show lower overall and single act NSAA scores than a non-'V2' subject (e.g. 'D4').

Taking the lack of true labelling for 'V2' files into account, we slightly modify our approach with 'model_predictor.py': rather than comparing predicted values to true values for a given subject as we have done for many of the previous model predictions sets, we instead here compare the predicted results for the 'V2' file to the true results for the non-'V2' file (e.g. when we predict the overall NSAA score for 'D4V2', we also look at the true overall NSAA score for 'D4' as we don't have the true overall score for 'D4V2'). While we aren't expecting the results to be similar (as in most real-world cases the performance of the subject for the NSAA assessments will change over a 6-month period between assessments), we would assume it to be pointing in the correct direction if it predicts lower scores on average for the 'V2' subject than the non-'V2' subjects.

To undertake this MPS, we modified the 'rnn.py' script to use a '–leave_out_version' optional argument, which takes a string (e.g. 'V2') as a value and allows us to filter out all file names containing the string (in the previous case, 'V2'); hence the models are able to be built exclusively on non-'V2' files, which enables us to assess on this with 'model_predictor.py' in the knowledge that the models being used to assess have seen no 'V2' data before. It should be noted at this point that 'V2' removes all the files for a specific version. Hence, we aren't just leaving one subject's files out of the data set, as was done for many previous model predictions sets using the '–leave_out' argument (which isn't used here). We therefore only need one 'set' of models to be built for this MPS, meaning we only need 12 models to be built (one for each input and output type combination), as opposed to 60 models for when we look at 5 left-out subjects. This is because we want to consider the models here as

built only on non-'V2' files, and not with using some 'V2' files that we aren't then going to be assessing on (if this were the case we could use 'leave_out' as, for example, '–leave_out=D4V2'), which lets us say that these models are based only on one 'round' of NSAA subject assessments and can therefore generalise to any of the subjects in the study who have new assessments (e.g. 'V2' subject names).

With the models built by 'rnn.py' with the '–leave_out_version=V2' argument set, we then turn to assessing the subject file names with 'model_predictor.py'. An option '–leave_out_version' was also added to 'model_predictor.py' to enable us to seek out models that have been built by 'rnn.py' with the same argument. We also make sure 'model_predictor.py' seeks out the models that have been built with optional arguments that were decided to be used by previous the results from previous model predictions sets, such as using the additional directory 'NMB' to supplement the 'NSAA' data set, the aggregation of output types for a more accurate overall NSAA score, and using no test set within 'rnn.py'. We also make sure to set the '–new_subject' argument, as the 'V2' subjects are considered 'new subjects' to 'model_predictor.py' as it doesn't have any information to label the data with true values from the 'nsaa_6mw_info.xlsx' table. Finally, we modify what the '–new_subject' argument actually does. As we do not have any 'true' information about the new subject from 'nsaa_6mw_info.xlsx', we cannot write the necessary information to 'model_predictions.csv' as it wouldn't be in the correct shape for the table; hence, we previously skipped the writing to 'model_predictions.csv' step with '–new_subject'. However, it was decided that we would instead write the results if the '–new_subjects' argument is set to a different file called 'model_predictions_newfiles.csv'. Essentially, it's the same as 'model_predictions.csv' except that it's only for files where we don't have the true information of the file's overall and single-act NSAA scores, as is the case for file assessments in 'model_predictions.csv'.

12.21.1 Results Discussion

It should first be noted that the 'Short file name's for each of the rows in the table above also have the "(additional dir = NMB) (aggregate overall) (no testset) (leave out version = V2)" appended at the end of them; however, we have excluded these in the above table for the sake of readability and the fact that they are synonymous with every line in the table, unlike in many previous model predictions sets, and so there's no need to include them in every single line. Additionally, following on from the conclusions drawn in MPS 20, we used two ways of analysing a file: one using just the sensor magnetic field data and the other using all 4 chosen measurement types. The hope is that one of the two approaches would show more 'sensible' values than the other (i.e. a small to moderate drop on average for a given subject between their 'V1' assessment and their 'V2' assessment). Furthermore, all the files that were assessed and presented in the table above are sourced from the NSAA data set, which are assessed on models trained on both NSAA and NMB data sets (using the '–add_dir=NMB' optional argument). Finally, as we chose to use the aggregated overall NSAA score with 'model_predictor.py', the values in the 'Predicted overall

NSAA score...' column are in fact these aggregated scores, and we do not consider the sole 'overall' output as we have done so in many previous model predictions sets.

Much like previous experiment sets, we focus our attention for the predictions on the predictions made for the overall NSAA scores over the single-act or D/HC predictions. When we consider only the sensor magnetic field data, we see that for 7 out of 9 of the 'V2' files it predicted a drop in overall NSAA score for an average over all 9 files of a decrease of 4.89 between versions; in other words, the **models that were trained on 'V1' files using only the sensor magnetic field data would on average predict a 'V2' file with a 4.89 lower overall NSAA score than the overall NSAA score of the corresponding 'V1' file used to train the model**. If, instead, we use all 5 types of measurement data and the corresponding models, we see that for only 3 out of 9 of the 'V2' files it predicted a drop in the overall NSAA score for an average over all 9 files of an increase of 0.33 between versions. Hence, **using all 4 available measurements to predict the aggregated overall NSAA scores for new assessments of familiar subjects shows on average a marginal increase in score**.

While it is impossible to discount whether or not this is the correct approach for the models to be taking (again, as we don't have the true label information in the 'nsaa_6mw_info.xlsx' table), it's safe to assume that, based on knowledge of DMD progression, subjects are expected to get progressively worse between the assessments, as 6-months between the assessments is a large enough time for a subject's DMD conditions to get much worse; therefore **the use of all 4 measurements to generalise to new versions of familiar subjects is most likely not a good approach**. However, an average of 4.89 when solely using sensor magnetic field data is more feasible: this corresponds to getting '1' score worse on 5 of the 17 activities during the 6 months between assessments, which again (based on the difficulty of some of the activities for those with DMD) is somewhat more plausible. Therefore, we can conclude that **sensor magnetic field on its own, as previously shown in MPS 20, is superior to using all 4 measurements to generalise to new version files of familiar subjects and demonstrates predictions that seem in line with the progression one would expect of subjects in the given timespan**.

12.22 Model Predictions Set 22: NMB Files on Models Built from NSAA Files and Vice Versa

File assess dir	Model trained dir	Measurements tested	MAE between true and predicted NSAA scores over files (source dir)	Percentage of correct predicted file D/HC label (source dir)	MAE of percentage predicted wrong sequence D/HC classification over files (source dir)	Average percentage of single acts correctly predicted over files (source dir)
NMB	NSAA	['sensorMagneticField']	3.05	90.67%	12.69	87.14%
NMB	NSAA	['position', 'sensorMagneticField', 'jointAngle', 'AD']	4.99	80.04%	30.21	73.76%
NSAA	NMB	['sensorMagneticField']	2.33	92.86%	8.9	90.9%
NSAA	NMB	['position', 'sensorMagneticField', 'jointAngle', 'AD']	4.79	80.95%	24.78	77.45%

Now that we have access to the natural movement behaviour as a data set with all the measurements available (i.e. the 'NMB' data set) as opposed to solely the joint angles (i.e. the 'allmatfiles' data set), we intend to extend the experimentation of assessing files from one data set on models built using another data set. While in MPS 1 we only looked at 'allmatfiles' files' joint angle data assessed on models built on the joint angle data of NSAA files, here we look at four 'variations':

- 'NMB' files' sensor magnetic field data assessed on models built on sensor magnetic field data of 'NSAA' files.
- 'NMB' files' all four measurements assessed on models built on all four measurements data of 'NSAA' files.
- 'NSAA' files' sensor magnetic field data assessed on models built on sensor magnetic field data of 'NMB files.
- 'NSAA' files' all four measurements assessed on models built on all four measurements data of 'NMB' files.

The models are built and assessed in the same way, however, in that they rely on 'test_altdirs.py' to repeatedly call the 'model_predictor.py' script on all available files in a given data set. This way, we are able to make a model prediction for each of the files in each of the data sets on models built from the other data set. As there are 650 files within the NMB data set and 50 files in the NSAA data set, this means there are approximately approximately 1400 total file predictions, which manifest as approximately 1400 rows of data within 'model_predictions.csv'. As it's obviously impractical to manually draw inferences over this large data set, we added in additional functionality in 'graph_creator.py'. The 'test_altdirs.py' script is called once for each of the above 4 variations and subsequently we run 'graph_creator.py' for each variation to create graphs that show the relative performance of assessing certain measurements of one data set on models trained using the other data set. While these graphs provide useful information as to the performance of a 'variation', it is not as easy to compare graphs for different variations side by side. Hence, we make use of the additional functionality from the 'graph_creator.py' script, which produces 4 statistical outputs that are computed over all the rows of 'model_predictions.csv' that are produced per each of the 'variations'; one can find the corresponding 3 graphs created for each of the 4 variations above within the '<project directory>\documentation\Graphs\' subdirectory.

We also chose to compare results of using solely the sensor magnetic field data primarily due to the relative performance shown in MPS 20 and 21. Our expectation was thus that the preference for only using this raw measurement would extend to the testing of files on 'alternative directories' (i.e. where the assessing file is from a different directory than the files used to train the models). Furthermore, this could prove to be advantageous to the future assessment of subjects, as it could negate the requirement to compute the feature-reduced computed statistical values from 'comp_stat_vals.py' and 'ft_sel_red.py', while minimizing the use of

'ext_raw_measures.py'. An important note is that here, when we build models using the data set described in 'Model trained dir', we do so using exclusively this data set, and don't use several files as supplement from another data set (as seen in MPS 20). Another thing to note is that, while the models will have never seen any part of the assessing files before (as they are from a different directory), the data will often be representing similar things as the data set used for model training; for example, certain NMB files may bear a strong resemblance to certain NSAA activities, and hence model performance would be expected to be particularly strong for these due be used to this sort of activity occurring within the data. On the other hand, however, the assessing data might well be movement characteristics that it's never seen before (for example, assessing a subject's natural activity of sitting and eating on models that have been trained on NSAA files), which might require the models to generalize their knowledge to other types of movement. This is essentially the root of what we are trying to find out with this MPS: whether models can generalize new files of familiar subjects performing different activities to achieve good results for NSAA score regression and D/HC classification over most of the assessed files.

12.22.1 Results Discussion

One thing that is made very clear is the comparative performance of models trained and assessed only with sensor magnetic field data for both variations. While we saw how they were more useful for assessing left-out subjects in MPS 20 and generalizing towards new versions of seen subjects in MPS 21, this was no guarantee of performance for using them in 'alt dirs' settings. However, **this MPS shows that they are the superior choice over using an aggregation of measurements in all possible applications of the system, be it for new subject generalization, new version generalization, or inferences made on 'alt dir' models**.

In particular, we see that they have a significantly lower MAE difference between the true and predicted overall NSAA scores than those achieved using all 4 measurements. For example, if we look at row three in the table, this tells us that (over the 50 files to be assessed in the NSAA data set), on average the predicted overall NSAA score for each file assessed using 'model_predictor.py' differs from the true overall NSAA score by only 2.33. These are very comparable scores to MPS 20, where the average difference over the 5 left-out subjects was 2.4; hence **assessing NSAA files using only the sensor magnetic files on models built exclusively on NMB data is comparable, if slightly better, than assessing the same files on models built on the same data set as the assessed files**. What's more, as the assessing and model trained data sets differed, we didn't need to create models exclusive to one subject being left out (e.g. traditionally 5 left-out subjects for 4 input measurements and 3 output types resulted in 60 models being constructed), which meant that not only was it a lot quicker to build the models (requiring only 12 models to be created for the train-on-NSAA/assess-on-NMB variation pairs and 12 for the train-on-NMB/assess-on-NSAA variation pairs) but we also averaged over all available subject files in the NSAA files and having done so much more time- and compute-efficiently

than the 'left-out method' seen previously. Hence, the **results displayed here are more indicative of true performance due to having effectively 'left out' all of the subjects from the model trained data set, while also having to create fewer models in the process**.

While we're primarily focused on the MAE of the difference between the true and predicted overall NSAA score metric, it's also worth pointing out the relative performance of using only the sensor magnetic field data compared to the aggregation of the other 4 measurements. For example, the percentage decreases from approximately 20% incorrectly classified files when using all 4 measurements to only about 10% when using solely sensor magnetic field data. This means that, **in either case where we are assessing either NSAA or NMB files on models trained on the other data set, we are only misclassifying the files as being from 'D' or 'HC' subjects in 10% of the cases, which is classification performance not matched in either MPS 20 or 21 (even though they are using a slightly different 'inter'-file classification metric while here we use an 'intra'-file classification metric)**. Furthermore, looking at the third metric, we can see that models trained on just the sensor magnetic field data has 10% of each of the files' sequences misclassified, while using all 4 measurements raises this to 27%. Finally, this improvement can also be seen with the models built towards the 'acts' output type, as the average percentage of single acts classified correctly is on average 13% higher when using solely sensor magnetic field data than using all 4 measurements.

The main takeaway from this MPS set is therefore that **it is very possible, and possibly even ideal, to assess models with files from a different data set than was used to train the models**. This opens up interesting applications to the assessments of subjects; for example, by using these pretrained models for future needed NSAA assessments, we would only need the subject to perform natural movement behaviour data and would then only be interested in capturing the sensor magnetic field data from the suit (train-on-NSAA, test-on-NMB). Alternatively, in cases where in the first version of the assessments we only captured NMB of the subject, we (upon capturing the NSAA assessments in the second version) still use the models to help assess the subject even without having seen the subject's initial NSAA assessment (train-on-NMB, test-on-NSAA).

12.23 Model Predictions Set 23: Assessing Subjects Using Single-Act Files w/ New Models and Only Using Sensor Magnetic Field Data

File Name	Measurements tested	Percent of acts corrected predicted	Predicted D/HC Label	Percent of correct predicted sequences	True Overall Score	Predicted Overall Score	Aggregated Predicted Overall Score
D3	['sensorMagneticField']	52.94%	D	73.44%	19	23	24
D3 (act 1)	['sensorMagneticField']	70.59%	HC	20.31%	19	18	23
D3 (act 2)	['sensorMagneticField']	52.94%	D	100.0%	19	25	25
D3 (act 3)	['sensorMagneticField']	41.18%	HC	0.0%	19	28	31
D3 (act 4)	['sensorMagneticField']	41.18%	D	100.0%	19	17	23
D3 (act 5)	['sensorMagneticField']	64.71%	D	100.0%	19	17	21
D3 (act 6)	['sensorMagneticField']	70.59%	D	57.81%	19	26	23
D3 (act 7)	['sensorMagneticField']	70.59%	D	100.0%	19	21	22
D3 (act 8)	['sensorMagneticField']	23.53%	D	90.62%	19	32	30
D3 (act 9)	['sensorMagneticField']	41.18%	D	100.0%	19	27	26
D3 (act 10)	['sensorMagneticField']	41.18%	D	100.0%	19	27	27
D3 (act 11)	['sensorMagneticField']	41.18%	D	100.0%	19	27	27
D3 (act 12)	['sensorMagneticField']	70.59%	D	82.81%	19	24	23
D3 (act 13)	['sensorMagneticField']	64.71%	D	100.0%	19	24	24
D3 (act 14)	['sensorMagneticField']	52.94%	D	64.06%	19	19	22
D3 (act 15)	['sensorMagneticField']	52.94%	D	100.0%	19	16	21
D3 (act 16)	['sensorMagneticField']	41.18%	D	100.0%	19	18	23
D3 (act 17)	['sensorMagneticField']	64.71%	D	85.94%	19	31	26
D9	['sensorMagneticField']	52.94%	D	99.69%	22	25	26
D9 (act 1)	['sensorMagneticField']	52.94%	D	100.0%	22	22	25
D9 (act 2)	['sensorMagneticField']	52.94%	D	100.0%	22	27	27
D9 (act 4)	['sensorMagneticField']	88.24%	D	100.0%	22	24	23
D9 (act 5)	['sensorMagneticField']	47.06%	D	100.0%	22	26	26
D9 (act 6)	['sensorMagneticField']	52.94%	D	100.0%	22	27	27
D9 (act 7)	['sensorMagneticField']	52.94%	D	100.0%	22	25	26
D9 (act 8)	['sensorMagneticField']	52.94%	D	100.0%	22	28	27
D9 (act 9)	['sensorMagneticField']	52.94%	D	100.0%	22	27	27
D9 (act 10)	['sensorMagneticField']	52.94%	D	100.0%	22	27	27
D9 (act 11)	['sensorMagneticField']	64.71%	D	100.0%	22	27	26
D9 (act 12)	['sensorMagneticField']	58.82%	D	96.88%	22	25	26
D9 (act 13)	['sensorMagneticField']	88.24%	D	100.0%	22	25	23
D9 (act 14)	['sensorMagneticField']	52.94%	D	100.0%	22	26	27
D9 (act 15)	['sensorMagneticField']	52.94%	D	100.0%	22	25	26
D9 (act 16)	['sensorMagneticField']	52.94%	D	100.0%	22	23	25
D9 (act 17)	['sensorMagneticField']	64.71%	D	100.0%	22	23	25
D11	['sensorMagneticField']	70.59%	D	99.69%	27	25	25
D11 (act 1)	['sensorMagneticField']	70.59%	D	100.0%	27	28	26
D11 (act 3)	['sensorMagneticField']	47.06%	D	100.0%	27	23	23
D11 (act 4)	['sensorMagneticField']	35.29%	D	100.0%	27	25	23
D11 (act 5)	['sensorMagneticField']	47.06%	D	100.0%	27	24	23
D11 (act 6)	['sensorMagneticField']	47.06%	D	100.0%	27	29	25
D11 (act 7)	['sensorMagneticField']	82.35%	D	100.0%	27	27	26
D11 (act 8)	['sensorMagneticField']	82.35%	D	100.0%	27	24	25
D11 (act 9)	['sensorMagneticField']	82.35%	D	100.0%	27	24	25
D11 (act 10)	['sensorMagneticField']	47.06%	D	100.0%	27	26	25
D11 (act 11)	['sensorMagneticField']	82.35%	D	93.75%	27	21	25

D11 (act 12)	['sensorMagneticField']	47.06%	D	100.0%	27	24	23
D11 (act 13)	['sensorMagneticField']	41.18%	D	100.0%	27	28	24
D11 (act 14)	['sensorMagneticField']	47.06%	D	100.0%	27	28	25
D11 (act 15)	['sensorMagneticField']	35.29%	D	100.0%	27	22	22
D11 (act 16)	['sensorMagneticField']	70.59%	D	100.0%	27	26	25
D11 (act 17)	['sensorMagneticField']	35.29%	D	100.0%	27	28	24
D17	['sensorMagneticField']	94.12%	HC	7.03%	33	33	34
D17 (act 1)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 2)	['sensorMagneticField']	94.12%	HC	34.38%	33	33	34
D17 (act 3)	['sensorMagneticField']	94.12%	HC	28.12%	33	31	33
D17 (act 4)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 5)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 6)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 7)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 8)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 9)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 10)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 11)	['sensorMagneticField']	94.12%	HC	0.0%	33	29	32
D17 (act 12)	['sensorMagneticField']	94.12%	HC	20.31%	33	32	33
D17 (act 13)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 14)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 15)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 16)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34
D17 (act 17)	['sensorMagneticField']	94.12%	HC	0.0%	33	34	34

While we managed to achieve some promising results in MPS 7 with regards to determining whether certain single-act files of left-out subjects correspond more highly to overall NSAA scores, we also must recognize that drawing conclusions from this MPS at this point isn't particularly advisable. This is for two reasons:

1. We are using all the measurements which, as the previous three model predictions sets have now shown, isn't preferable to simply using the sensor magnetic field data.

2. The models used to assess those single-act files are not trained with the optional arguments that enable the model to use the supplementary data from the NMB data set and to not partition the data into a training and testing set. Additionally, the 'model_predictor.py' script is not set to compute the aggregated overall NSAA score.

Hence, in this MPS we are essentially replicating the process that we undertook for MPS 7 with modifying the models and how we evaluate them on the files by fixing the issues outlined above. We again make use of the 'predictions_selector.py' script here to draw information from all the predictions made within this MPS by 'model_predictor.py' (which is approximately 68 total predictions). We are however, still interested in the same thing as in MPS 7 though: whether there were certain single activities of the 17 undertaken by each subject that can be showed to be more 'impactful' to the overall assessment of the subject (i.e. more indicative of their overall NSAA score or correct classification) than other activities. We also use the same hypothesis from MPS 7: that certain single-activities whose data is consistent across multiple subjects at being able to approximate overall NSAA scores and D/HC classification would show themselves to be the more useful activities at assessing a subject. Finding these 'more useful activities' could motivate specialists to pay special attention to the scoring of these activities during assessment or, alternatively or

perhaps in addition, could result in the discarding of some or many of the 17 activities if they prove to have little or no correlation to the overall assessment.

The models that we need to use for this model predictions set were already created by MPS 20; we are using the same models as created beforehand (i.e. using the supplementary NMB data and the '–no_testset' option) but now are testing them with single-act files for the left-out subjects as opposed to the complete-act files as was done in MPS 20. Additionally, the models are still considered here to be 'unfamiliar' with the subjects: testing the activity 6 file for subject 'D9' would be tested on models trained on complete-act files for all subjects except 'D9'. Finally, it should be noted that the same reasons for not including 'HC6' in MPS 7 here hold for this set too, as the Google annotations sheet that we have for the subjects does not contain entries for most of the activities for subject 'HC6' for their activities and hence, rather than including only several of these activities, we omitted this subject from the model predictions set entirely.

12.23.1 Results Discussion

Note that, while the entries for the 'File Name' column in the table above contain only the subject name and activity number, as the corresponding rows appear in 'model_predictions.csv' they also include '(additional dir = NMB) (aggregate overall) (no testset)' within the 'Short file name' column cell; we've omitted this extra string needed for each cell in the table above so as to not unnecessarily 'clutter' the table (since they all use it then it doesn't provide any additional information about any row in particular).

As this table is fairly challenging to interpret at a glance, considering it has 70 entries total, it was felt necessary to make use of the 'predictions_selector.py' script in order to filter the corresponding lines in 'model_predictions.csv' based on certain metrics. We therefore ran the script three times: once each to find the best 15 rows according to the metric measuring the difference between the true and predicted overall NSAA values (for the 'overall' output type), the percentage of acts correctly predicted (for the 'acts' output type), and the percentage of predicted correct sequences (for the 'dhc' output type). The results can be seen below:

Best performing 15 rows by Diff true/pred overall NSAA score...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Diff true/pred overall NSAA score
17	D3 (additional dir = NMB) (act 14) (aggregate ...	NSAA	NaN	['sensorMagneticField']	0
21	D9 (additional dir = NMB) (act 1) (aggregate o...	NSAA	NaN	['sensorMagneticField']	0
54	D17 (additional dir = NMB) (act 2) (aggregate ...	NSAA	NaN	['sensorMagneticField']	0
3	D17 (additional dir = NMB) (aggregate overall)...	NSAA	NaN	['sensorMagneticField']	0
42	D11 (additional dir = NMB) (act 7) (aggregate ...	NSAA	NaN	['sensorMagneticField']	0
53	D17 (additional dir = NMB) (act 1) (aggregate ...	NSAA	NaN	['sensorMagneticField']	1
52	D11 (additional dir = NMB) (act 17) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
51	D11 (additional dir = NMB) (act 16) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
49	D11 (additional dir = NMB) (act 14) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
69	D17 (additional dir = NMB) (act 17) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
45	D11 (additional dir = NMB) (act 10) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
68	D17 (additional dir = NMB) (act 16) (aggregate...	NSAA	NaN	['sensorMagneticField']	1
35	D9 (additional dir = NMB) (act 16) (aggregate ...	NSAA	NaN	['sensorMagneticField']	1
36	D9 (additional dir = NMB) (act 17) (aggregate ...	NSAA	NaN	['sensorMagneticField']	1
37	D11 (additional dir = NMB) (act 1) (aggregate ...	NSAA	NaN	['sensorMagneticField']	1

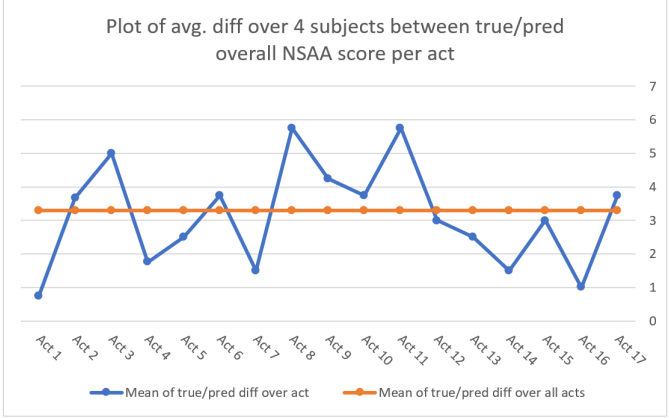
Best performing 15 rows by Percentage of acts correctly predicted...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Percentage of acts correctly predicted
69	D17 (additional dir = NMB) (act 17) (aggregate...	NSAA	NaN	['sensorMagneticField']	94.12
64	D17 (additional dir = NMB) (act 12) (aggregate...	NSAA	NaN	['sensorMagneticField']	94.12
53	D17 (additional dir = NMB) (act 1) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
54	D17 (additional dir = NMB) (act 2) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
55	D17 (additional dir = NMB) (act 3) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
56	D17 (additional dir = NMB) (act 4) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
57	D17 (additional dir = NMB) (act 5) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
58	D17 (additional dir = NMB) (act 6) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
59	D17 (additional dir = NMB) (act 7) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
61	D17 (additional dir = NMB) (act 9) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
62	D17 (additional dir = NMB) (act 10) (aggregate...	NSAA	NaN	['sensorMagneticField']	94.12
63	D17 (additional dir = NMB) (act 11) (aggregate...	NSAA	NaN	['sensorMagneticField']	94.12
60	D17 (additional dir = NMB) (act 8) (aggregate ...	NSAA	NaN	['sensorMagneticField']	94.12
65	D17 (additional dir = NMB) (act 13) (aggregate...	NSAA	NaN	['sensorMagneticField']	94.12
3	D17 (additional dir = NMB) (aggregate overall)...	NSAA	NaN	['sensorMagneticField']	94.12

Best performing 15 rows by Percentage of predicted correct sequences...

	Short file name	Source dir	Model trained dir(s)	Measurements tested	Percentage of predicted correct sequences
35	D9 (additional dir = NMB) (act 16) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
47	D11 (additional dir = NMB) (act 12) (aggregate...	NSAA	NaN	['sensorMagneticField']	100.0
25	D9 (additional dir = NMB) (act 6) (aggregate o...	NSAA	NaN	['sensorMagneticField']	100.0
26	D9 (additional dir = NMB) (act 7) (aggregate o...	NSAA	NaN	['sensorMagneticField']	100.0
27	D9 (additional dir = NMB) (act 8) (aggregate o...	NSAA	NaN	['sensorMagneticField']	100.0
28	D9 (additional dir = NMB) (act 9) (aggregate o...	NSAA	NaN	['sensorMagneticField']	100.0
29	D9 (additional dir = NMB) (act 10) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
30	D9 (additional dir = NMB) (act 11) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
32	D9 (additional dir = NMB) (act 13) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
33	D9 (additional dir = NMB) (act 14) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
34	D9 (additional dir = NMB) (act 15) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
36	D9 (additional dir = NMB) (act 17) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
37	D11 (additional dir = NMB) (act 1) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
38	D11 (additional dir = NMB) (act 3) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0
39	D11 (additional dir = NMB) (act 4) (aggregate ...	NSAA	NaN	['sensorMagneticField']	100.0

As we are primarily focused on the regression capabilities of the model, our focus mainly lies with the metric shown in the first and third images. However, for the third image seen above (for the 'acts' output type), there are numerous lines that perform very well according to that metric; hence, we decided to at this point focus exclusively on the first image that shows the best activities for each subject ranked according to how closely they predict the overall NSAA score (with perfectly prediction for a file resulting in a '0' value for this metric). We saw a greater number of acts between 14 and 17 in this better performing region (i.e. files of act 14, act 15, act 16, or act 17 of each subject tended to better approximate the true overall NSAA score on models that hadn't seen the subject before). However, we wanted to view this in a bit more detail and find out which activities tended to perform better or worse and see if there any trends.



The graph above shows, for each activity number, the average difference between the true and predicted overall NSAA scores over the 4 subjects as plotted by the blue line, while the orange line is the average over both all subjects and all activities. Therefore, activities significantly below this line would indicate that they are more likely to be useful in predicting the overall NSAA score (and thus be a more useful activity to the assessment) than the activities above the line, and vice versa.

One thing that we can note is the climb and descend box activities that use the right leg (acts 6 and 8, respectively) are noticeably worse at being used to predict the overall score for the subject it corresponds to in comparison to the same activities performed by the subjects using their left leg. This might be due to a natural tendency for subjects to generally prefer using their right to complete activities (e.g. climbing a step) if possible; hence, the performance for the generally-less-dominant leg would most likely correspond more accurately to how severe the subject's DMD is. We can also see this with activities 15 and 16 (hopping on right and left leg, respectively), where the performance of the subject on the 'hopping on left leg' task is much more useful to the assessment than when they complete the activity using their right leg. Therefore, we can conclude that in general **activities using the left leg in more muscular-demanding activities are generally more useful to the assessment**.

One thing we can note is that activities 8 to 11 all perform worse than the average, with 2 of them performing significant worse; these are the 'descend box right', 'descent box left', 'gets to sitting', and 'rise from floor' activities. These are the activities that **show much lower correlation between the predicted and true overall NSAA assessment for the subject, and hence are less likely to be useful in the assessment so removing them from the assessment would most likely not impact the overall subject's assessment of DMD severity as severely as if we had removed other activities**. Conversely, we can see particularly good performance from activities in the range of 13 to 17, which are the 'stand on heels', 'jump', 'hop right leg', 'hop left leg', and 'run'. Additionally, the difference for 3 of the 4 subjects for activity 17 was 1, with only the other 1 subject's difference being 12 meant that the average was pulled above the orange line. If we chose to discard this result as being a (most likely) anomaly, we get an even more consistent performance of activities 13 to 17 as being more useful than others at predicting the overall NSAA score. This is particularly interesting, as these are all activities which are performed when the subject is stood up; conversely, the group of activities that show worse performance (8 to 11) have several activities where the subject completes the activity from the floor. We can therefore infer from this that **activities that are performed by the subject standing up (either stationary or in motion) are more indicative of accurate assessment of a subject's DMD severity than activities that are performed from the floor**. A potential takeaway from these conclusions for assessors could be to have subjects undertake more activities where they are upright and, preferably, using their less-dominant leg for activities, as opposed to activities where they start from the floor.

Part V

Evaluation and Conclusions

Chapter 13

Project Timeline and Gantt Chart

In this chapter, we shall give a brief overview of the project's timeline between it's beginning with background research at the start of April until it's conclusion with final report construction at the beginning of September. During the 5-months of undertaking this project, an ongoing lists of tasks completed and tasks needed to be completed have been compiled so others could keep track of the project's progress if necessary and also to provide good immediate- and long-term directions to take the project in. This can be found in the base directory of the project directory as a '.ods' file entitled 'MSc Project Plan'. However as opposed to simply referencing this file, as simply a list of tasks ordered by completion date, we've repurposed this as a Gantt chart, which is displayed further below within this section. It should be noted that, as this covers 5-months' worth of work and in many cases several tasks were completed per day, there are in excess of 200 individual tasks completed, which requires us to display the chart over multiple images.

We've also divided the project into 'Stages', with 9 total stages of the project in turn covering background research, data pipeline construction, the RNN and predictor scripts construction and modification, running of experimentation sets, running of model prediction sets, and writing of the final report, among other things. This should help give the reader a broad overview of how the project progressed from its inception. Additionally, the numbered columns correspond to the week number the tasks were completed in and range from '1' (week 1; 1st – 7th April) to '23' (week 23; 2nd – 6th September). Furthermore, tasks completed are given different colours corresponding to different stages of the project; however, in many cases clear distinctions cannot be made between different stages of the project. For example, further system modifications tasks that would more intuitively take place in stage 3 would often instead take place in stage 6 as these modifications to the setup of the RNN model would only be decided upon at this stage of the report as a necessity to carry out specific experimentation.

Stage	Task	1	2	3	4	5	6	7	8
1: Background Research and Setup	Research + summarise from 7 papers								
	Presentation of initial background research								
	Find + make notes from 5 papers on RNNs								
	Northstar background research and note taking								
	Type up notes thus far								
	Find further 5 papers relating to RNNs and action/motion analysis								
	Create research papers folder and file previous papers								
	Setup laptop for CUDA / TF GPU								
	Read and complete programs in 'Python ML', ch 16								
	Make RNN / CRF presentation (2 slides each, 1 description, 1 usefulness)								
	Paper review: "Deep, Convolutional, and Recurrent Models for HAR using Wearables"								
	Do RNN presentation								
	Type up and make import points from: "Deep, Conv, and Recurrent Models for HAR using Wearables"								
	Paper review: "Deep Recurrent NN for Mobile HAR with High Throughput"								
	Paper review: "Hierarchical Recurrent Neural Network for Skeleton Based Action Recognition"								
2: Programming of Data Pipeline	Download/setup suit data and create diffs plot function								
	Add function to extract statistical features of joint files + print results to separate .csv								
	Create extract/dynamic visualisation functions for all-data files								
	Setup classes for different file types, comment all code, and commit								
	Create AD function for accel/magnet/position statistical analysis								
	Add to DC init function (inc extract table to .csv and read in, and passing to JA objects)								
	Add Fourier analysis functionality to statistical extraction for AD class (veloc+accel)								
	Make presentation of 2 weeks inc: suit data format, .py script, next steps								
	Swap order of col label to be: measure, feature, axis, stat								
	Create dict mapping from measure name to seg/joint/sens names								
	Add command line functionality and arguments								
	Fix crashes-on-close problem with pyplot windows								
	Add name of file to output col (i.e. one col for if HC or D file)								
	Add option to 'split up the input JA/AD file w/ 'HC'/'D' labels into numerous files w/ same label								
	Add function that checks for behaviour abnormality over all features for each file written								
	Implement 'extract_stat_features' function to be called by write_stats by both classes including all stat functions								
	Implement function to extract JA .mat files to .csv								
	Add other stat features (see paper for full list of features to additionally include)								
	Make display 3D pos plot run in real-time								

3: Setup of RNN model and predictor (+ continuation of data pipeline modification)	Implement basic RNN model with .csv output file								
	Create README file of 'matfile_analysis'								
	Edit RNN layout and suppress warnings on output								
	Add optional argument to automatically delete the created output .csv files at end of command								
	Eliminate need for 'file name' when 'file type' given as 'dc'								
	Ensure different file types are written to different subdirectories of 'output_files' (e.g. for AD, JA, DC)								
	Add alternative to 'split files' in 'matfiles_analysis' that specifies a preferred file length rather than num of splits								
	Modify 'matfiles_analysis' to work with diff subdirs								
	Download additional necessary dataset files								
	Comment both scripts								
	Add script with various feature reduction techniques								
	Add option to read in NSS's into df as new column								
	Implement NSS as labels in RNN script								
	Adapt RNN to predict north star scores as alternative								
	Add functionality to extract act scores from .xlxs into df								
Stage	**Task**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
3: Setup of RNN model and predictor (+ continuation of data pipeline modification)	Annotate D11-D20 NSAA files and edit google sheet								
	Fix issue with 'JA/DC all' split_size=1 only showing '19s' on 'y-true'								
	Add function in RNN to write real and predicted scores to .csv								
	Create 'mat_act_div' script to divide up .mat files and rewrite to new dir								
	Fix 3D image pos not ending and exiting issue								
	Fix 3D image pos distortion issue								
	Use 'mat_an' to setup 'NSAA\AD', '6minwalk-matfiles\AD', 'direct_csv\DC', and 'direct_csv\JA'								
	Modify 'ft_red' to work with 'fn=all' argument and encapsulate feature selection/reduction								
	Modify RNN to work with both stat vals and raw joint angles for both DMD/HC class and NSAA regress								
	Create .csv file with results of RNN setups (DMD/HC class and NSAA regress for both AD and DC raw joint files)								
	Change names of all scripts and put them in 'source' directory (alter output dirs and batch as needed)								

4: Initial experiments w/ models (+ continuation of overall system modification)	Add to 'RNN results' for '6minwalk-matfiles\AD' and 'direct_csv\JA' for both choice='dhc' and 'overall'								
	Modify RNN to predict 17 output nodes for each activity w/ AD files from NSAA								
	Add optional argument to 'rnn' to write results and settings to relevant columns in 'RNN Results.ods'								
	Add optional argument to 'comp_stat_vals' to write all files to one 'all' file								
	Add 'ext_raw_measures' script to extract specific raw measurement values from given AD .mat file(s)								
	Add R^2 and RMSE as metrics in 'RNN.py' for regression								
	Implement confusion matrix output in 'RNN.py' for D/HC classification								
	Experiment for all input options for choice='acts'								
	Add optional argument to 'RNN.py' to specify proportion of sequence overlap								
	Add 'all' option for 'measurements' in 'ext_raw_measures.py' and run script with 'measurements=all'								
	Modify 'RNN.py' to work with raw measurements from local drive								
	Modify 'add_nsaa_scores' in both scripts to read NSAA scores from 'KineDMD data updates Feb 2019'								
	Change 'model_dir' to reflect script arguments								
	Add 'RNN.py' results output to 'write_to_csv'								
	Add optional 'RNN.py' argument to set num_epochs as argument								
	Experiment with different raw data measurements from AD files for various RNN outputs								
	Experiment with various sequence overlap proportion for NSAA\AD stat vals for choice='overall'								
	Enable 'comp_stat_vals' to extract features from 'single-act' matfiles								
	Enable 'ext_raw_measures' to work with 'single-act' matfiles and extract all measures from 'single-act' files								
	Modify 'RNN.py' to work with single-act .mat files to predict single-act NSAA scores								
	Add a README for 'How to use 3D visualization function'								
	Comment 'mat_act_div.py' and 'ext_raw_measures.py' scripts								
	Download missing files from 'NSAA\matfiles\ directory								
	Create .csv file with all subject info from 3 prev files								
	Experiment with single-act raw measurements files to predict single-act score (0, 1, or 2)								
	Move 'output_files' off dropbox into 'msc_project_files' and have all scripts point to there								
	Write script that loads a trained RNN model and tests on a complete file								
	Comment 'rnn.py' and 'ft_sel_red.py' scripts								

5: Further experimentation w/ models and writing of suplementary scripts	Experiment w/ useful raw measurements with longer sequences and larger sequence overlap
	Modify pipeline to ensure ALL available files are processed
	Change both 'add_nsaa_scores' to only use 'nsaa_6mw_matfiles'
	Add more experiments with longer seq length and more raw measurements
	Download the additional 'NSAA\matfiles'
	Write script to create matplotlib graphs from lines in 'RNN Results'
	Rerun NSAA stat vales pipeline with new files
	Ensure 'model_predictor' works with stat vals output
	Modify 'rnn' and 'model_predictor' to use .xlsx file for storing seq_len
	Ensure 'graph_creator' withs with all expers and choices, and rerun
	Run experiments w/ 'NSAA\AD' diff features and plot graphs
	Run experiments w/ 'NSAA\AD' different seq len and overlaps and plot graphs
	Modify 'ft_sel_red' to always delete previous 'FT_' files in directory
	Make slides of purpose of project, diagram of all scripts, experiments + results thus far, and next steps
	Do the video annotations required and fill out the Google sheet
	Add option in 'model_predictor' to create ensemble of various raw measurements and stat values
	Fix 'rnn.py' to write to 'model_shapes'
	Fix 'model_predictor' to work w/ AD files
	Add option to 'rnn.py' to create graph of real vs predicted overall NSAA scores
	Make 'write_to_csv' write to unique files based on args/epochs and a unique index number
	Modify 'ext_raw_measures' to extract 'allmatfiles' files and run
	Experiment w/ AD files w/ shorter seq_lens for AD and graph results
	Add '--discard_prop' option to 'rnn' to optionally discard every nth element to reduce size of sequence length
	Experiment w/ longer seq_lens and discard_prop for jointAngle and graph results
	Add option to 'graph_creator.py' to create graph of trues vs preds overall NSAA scores from single file name arg
	Modify 'model_predictor.py' to round overall NSAA scores after averaging across models
	Write batch script to work with a single 'allmatfiles' or AD file and feed through to 'model_predictor'
	Modify 'model_predictor.py' to write and display trues/preds graph
	Write batch script to do all package setup and run 'comp_stat_vals', 'ft_sel_red', ext_raw_measures'. 'mat_act_div'
	Modify 'file_predictor.cmd' to work with AD source files through pipeline

	Add optional 'rnn.py' argument to use additional source dir along with current 'dir' to combine 6minwalk and 6MW
	Add optional 'rnn.py' argument to leave a file out
	Run 'rnn.py' w/ NSAA top-4 raw measurements and AD for all output types with most ideal seq_len, seq_overlap, etc.
	Reorganize batch scripts into subdir of 'source'
	Run 'rnn.py' w/ combined 6minwalk and 6MW w/ same data types, output types, etc. as with NSAA
	Change RNN_outputs .csv name to be based on sys.argsv
	Add optional 'alt_dirs' argument to 'model_predictor' to use different directory(s) to test the file on
	Create 'file_predictor_altdirs.cmd' to load an 'allmatfile' file and test it on the NSAA\jointAngle and 6m\jointAngle models
6: Using the model predictor script for model predictions sets and continuation of system modifications	Modify 'model_predictor.py' to write 'output_strs' to a single test file (each model = 1 row, each string = 1 col of row)
	Write 'test_altdirs' script that automates the testing of each file in a dir on different dirs
	Add option to 'graph_creator.py' to graph the results in 'model_predictions.csv'
	Experiment w/ all 'allmatfiles' files using 'test_altdirs.py' w/ models trained on both NSAA and 6MW and graph results
	Write 'settings' script that stores all global variables used across files ('local_dir', 'source_dir', 'file_types', etc.)
	Write 'file_renamer' script to standardize names of 'mat' files in source directories and add to 'setup.cmd'
	Write batch script to train RNNs with given file left out then run model predictor on that file
	Add option to 'model_predictor.py' to use models that were specifically trained on for given 'fn'
	Experiment w/ 'left out' batch script with left-out 'D3' files (i.e. having retrained RNN w/ leaving out 'D3' files each time)
	Modify 'comp_stat_vals.py' so files aren't overwritten w/ diff sub-files
	Download all '6minwalk-matfiles', '6MW-matFiles', and 'NSAA' files and run 'setup.cmd'
	Modify 'model_predictor.py' to work with single-act files
	Fix issues w/ 'model_predictions' not being written correctly and rerun 'allmatfiles on NSAA,6minwalk' and 'D3'
	Modify 'graph_creator.py' that reads in 'model_predictions' and computes statistical values

7: Documentation of system and experiment results thus far and continuation of system modifications	Add script READMEs for 'settings', 'file_renamer', and 'test_altdirs'
	Add script READMEs for 'mat_act_div' and 'ft_sel_red'
	Add script README for 'ext_raw_measures'
	Write 'data_balancer.py' script to balance datasets and have before and after counts of y-labels printed to output
	Modify 'model_predictor' to ensure that non-balanced and non-left out models are used when appropriate
	Refactor 'graph_creator with everything in functions and more appropriate arguments
	Add function to 'graph_creator' to, for a given file and plot results of single-act predictions
	Experiment w/ single-act 'D11' files tested on 'D11-left-out' models to approximate overall NSAA score
	Add script README for 'graph_creator'
	Experiment w/ up and down balanced datasets for left-out 'D3' file on models for all outputs and record the results
	Experiment w/ left-out 'D11' natural movement files to train the 3 models
Stage	**Task**
7: Documentation of system and experiment results thus far and continuation of system modifications	Add general README to GitHub repo outlining the project and how the repo is laid out
	Add script README for 'data_balancer'
	Update powerpoint diagram and add to script ecosystem overview and GitHub 'source' 'README.md'
	Write results discussion of experiment sets 1 - 2
	Write results discussion of experiment sets 3 - 4
	Add script README for 'rnn'
	Download additional data files, modify pipeline for new 'left-out' dir, and use it on each of these
	Add script README for 'model_predictor'
	Add script README covering all batch files
	Write script discussion w/ copy-paste of script READMEs and include updated powerpoint diagram
	Write results discussion of experiment sets 5 – 8
	Add table of details for experiments 1 – 8
	Write results discussion for experiments 82 - 98
	Write results discussion of 'model_predictions' for 'altdirs'
	Write results discussion of 'model_predictions' for 'left-out' models

8: Experimentation with model predictions sets for MPS 3 - 11 and continuous system modifications to enable MPS's	Download new Google sheet and modify 'mat_act_div.py' to use this instead of previous one and run
	Add optional arg to 'mat_act_div.py' to create source files with single acts but concatenated together
	Add optional arg to 'ft_sel_red' to reduce dimensions on whole dataset, rather than on file by file basis and run
	Modify and run 'comp_stat_vals' and 'ext_raw_measures' on single_act concat and non-concat files
	Modify 'rnn.py' and 'model_predictor.py' to use 'single_acts_concat' files and differently feature-reduced dataset
	Write 'predictions_selector' script that reads 'model_predictions.csv' with given arguments
	Add 'cnn_preprocessing' function to 'rnn.py'
	Build CNN models on 'D3', 'D9', 'D11', 'D17', 'HC6' left-out subjects, and provide 'rnn_outputs' results
	Modify reference table w/ correct 'D2'/'D3' entries and re-run all setup lines that make use of this file
	Add README script for 'predictions_selector', add it to script explanations, and modify pipeline diagrams
	Add optional args to 'rnn.py' to standardize all features of data and add Gaussian noise to the dataset
	Restructure 'plans_and_presentations' and 'project_files' dirs into outer dir
	Report: write summary of use of Python, IDE, and TensorFlow
	Build standard 'left-out' models w/ <chosen 5> ('D3', 'D9', 'D11', 'D17', 'HC6') left-out subjects
	MPS 4: Make predictions for <chosen 5> for individual measurements tested and record results
	MPS 5: Experiment w/ '<chosen 5> (left out)' on (AD) models built from FRC_ files
	Add optional arg to 'rnn.py' to horizontally concatenate all measurements data together
	Write unique batch files for each model prediction set to launch, including previous and future MPS's
	Add to batch scripts README for model prediction sets and add to script explanations
	Sort 'rnn_models' into 'rnn_models_old' as required
	MPS 6: Make predictions for '<chosen 5> (left out)' on models w/ subject LO vs models w/ subject not-LO
	MPS 7: Make predictions for '<chosen 5> (left out) (act #)' on models built from standard LO subjects
	Modify 'left-out' arg in 'rnn' to account for specific file name (e.g. 'D2-009' rather than just 'D2')
	MPS 8: Experiment w/ '<chosen 5> (left out)' but with the 'D10' outlier subject left out
	MPS 9: Experiment w/ '<chosen 5> (left out)' on models built from single acts (concat) files
	MPS 10: Experiment w/ '<chosen 5> (left out)' on models built from 'downsampling'
	MPS 11: Experiment w/ '<chosen 5> (left out)' on models built from measurement-concatenated data
	Modify 'rnn' and 'model_predictor' to use multiple 'dir's and setup batch file

9: Creation of final report and finishing the execution of MPS's 12 - 22, and making of necessary system modifications	Report: write full system setup, including running of 'settings.py' and required system specs (RAM, GPU)
	Report: write data setup and data types section (borrowing heavily from initial report), w/ emphasis on output types
	MPS 13: Experiment w/ '<chosen 5> (left out)' on models built both NSAA and 6minwalk directories
	Add function to 'graph_creator' to plot true and preds for given row number range and create graphs for MPS 12
	MPS 12: Experiment w/ '<chosen 5> (left out)' on models built from single-act files only
	Separate the 'dis_3d_pos' functionality from 'comp_stat_vals.py' into its own script, update script README's
	MPS 14: Experiment w/ '<chosen 5> (left out)' on models built from measurements that have had features reduced to '20'
	Report: add section on documents' significance explanation
	Move Google sheet annotations file to 'documentation' and make all document paths inherit from 'settings.py'
	MPS 15: Experiment w/ '<chosen 5> (left out)' on models built from noise-added data (small amount of noise) and record results
	Report: add glossary of terms (review other sections for terms to add)
	Report: add project aims and objectives section (borrowing heavily from initial report)
	Add ability for 'rnn.py', when loading 'add_dir=allmatfiles', to only sample a certain amount of data for each of the subjects
	Add option to 'model_predictor.py' to draw a single sequence from the subject file
	Add function to 'model_predictor.py' to combine all final predictions to make a more-accurate prediction for overall NSAA
	Report: add to 'experiments and results discussions' a description of how experiment sets and MPS's are run
	MPS 16: Experiment w/ '<chosen 5> (left out)' on models built both NSAA and allmatfiles directories (w/ downsampled allmatfiles)
	MPS 17: Experiment w/ '<chosen 5> (left out)' but with the assessed file having a single sequence drawn from the total file
	MPS 18: Experiment w/ '<chosen 5> (left out)' but using the aggregated predictions to compare to non-aggregated predictions
	Add option to 'rnn.py' to use entire data set with 'train_test_ratio = 0.0' and modify 'model_predictor.py'
	Modify 'model_predictor' to work with subjects never seen before
	Download new NMB data set, set it up in 'msc_project_files', and modify scripts to use this
	Report: write bibliography of all papers used and will use, along with textbooks and webpages used
	MPS 19: Experiment w/ '<chosen 5> (left out)' but with entire data set (except LO files)
	Report: write section of explaining all directories, including data directories in 'MSc_project_files'
	Report: write section on explaining how to replicate the experiment sets and MPSs, etc.
	Clone project directory to GitLab
	Report: write RNN review w/ focus on LSTMs and use TensorFlow with RNNs
	Report: write summary of Duchenne, w/ references to previous studies
	Copy local directory to OneDrive (Imperial) and modify 'Directories Overview' to reflect this

	MPS 20: Repeat MPS 16 but with new NMB data set using all 5 measurements
	Modify 'rnn.py' and 'model_predictor.py' to use certain versions of files (e.g. 'V2' or not files)
	Write batch script that allows assess of source 'V2' mat file; add to README and script explanations
	Write 'file_mover.py' script; add README and add to script explanations and diagram
	MPS 21: Experiment w/ 'V2' files left out and assessed on all available 'V2' files
	Report: write project timeline and Gantt chart
	Add necessary 2 top-level READMEs as required by email
	Report: write section on further improvements and project directions to undertake
	Report (LaTeX): put together report parts 1.1 – 1.3 in LaTeX, including editing and spell check
	MPS 22: Experiment w/ 'alt_dirs' with files trained on NSAA and predicting NMB files (AND vice versa)
	Report (LaTeX): put together report parts 2.1 – 2.2 in LaTeX, including editing and spell check
	MPS 23: Repeat single-act MPS but only using sensor magnetic field data and new model options
	Report (LaTeX): put together report parts 3.1 – 3.2 in LaTeX, including editing and spell check
	Move final chosen models to directory within project directory, modify 'model_pred' to use these, and add to 'Final evaluation results'
	Write 'assess_nsaa_file.py' to run on final models (and extract necessary measurements) for given path and measurements
	Remove and/or repurpose any excess batch scripts within 'batch_files'
	Add 'assess_nsaa_file.py' README, update script explanations for this and for deleted batch scripts, and update diagram
	Report (LaTeX): put together report parts 3.3 in LaTeX, including editing and spell check
	Report (LaTeX): put together report parts 3.4 in LaTeX, including editing and spell check
	Report (LaTeX): put together report parts 4.1 in LaTeX, including editing and spell check
	Finish adding to 'Final Improvements'
	Update OneDrive link for 'msc_project_files' with the final versions of all files
	Report (LaTeX): put together report parts 5.2-5.3 in LaTeX, including editing and spell check
	Report (LaTeX): add code to appendix
	Report: add to 'Final evaluation results' section the final models used and the 'assess_' script
	Report: add to 'Final evaluation results' conclusions from all experiments sets and MPS's (in particular 20-23)
	Report (LaTeX): write 'Abstract', edit and include within LaTeX
	Report (LaTeX): add 'Literature Review' to report, including reviewing and editing
	Report (LaTeX): add 'Conclusions' section of the report

Chapter 14

Further Improvements and Additional Project Adaptations

14.1 Background

While an extensive amount of refinements and improvements have been undertaken during the course of this project, there still remains a great deal of improvements that can be made to the system and different ways we can adapt the system to either different project domains or to the same domain but with using different technologies. Many of these involve tasks and modifications that have not been made due to time constraints or modifications that exist outside the scope of the project but that could be added if the scope was slightly modified. In the sections below, we shall outline a few of each of these with the hope that one could use these as a starting point to continue with the project to either improve performance, adapt it to conditions other than DMD, work with different types of source files, and so on.

14.2 Immediate Improvements

14.2.1 Further modifications of RNN hyperparameters

- While there was an extensive amount of system modification tested through experiment sets and model predictions sets to ascertain the optimal combination of measurement data to be used, the best sequence hyperparameters (sequence overlap, discard proportion, etc.), there were many hyperparameters that were simply untested and we ended up using the same values throughout the project.

- This includes the number of hidden layers used in the models (2), the learning rate of the optimizer (0.001), the algorithm used as the optimizer ('adam'), among other hyperparameters. The values chosen for these were generally chosen as they were either good 'rules of thumb' with respect to the hyperparameter (e.g. the learning rate of 0.001) or we had previously seen good results in other projects using the setting (e.g. in using the 'adam' optimizer).

- These, of course, are no guarantee of ideal performance for the problems we are trying to solve in this project, and the primarily reason why alternative values for these hyperparameters weren't explored is due to time limitations. Furthermore, it's felt that exploring other system settings in our experiments would give settings that would also give an insight about the data itself; for example, finding out during our experiments that the joint angle measurement was much more useful for building models with than the velocity or acceleration measurements leads us to more insightful conclusions about movement data than finding out that a learning rate of 0.001 is more optimal than either 0.01 or 0.0001.

14.2.2 Use of greater compute resources to run the data pipeline and build models faster

- While the local PC used for the majority of the work of the project has proved adequate for our purposes, it also showed limitations in some of the more demanding tasks. For example, to run the whole 'setup.cmd' script that creates all the necessary intermediate data that is used to create the models, running the script with a high-end processor (e.g. 'Intel i7-8750H') still requires upwards of 10 hours to fully complete the script. Additionally, for each model created with a moderate-to-high-end GPU (e.g. 'GeForce GTX 1050ti'), for the typically-sized data set used to train a model it takes between 5-10 minutes to create: in the cases of model prediction sets where we need up to 60 models to be created, many hours might be needed per MPS.

- An alternative to this would be to make use of the Condor computing framework made available by Imperial College London's Department of Computing in order to submit these as batch jobs. This would enable to user to complete the setup and model construction phases of the project fairly quickly, which should make collaborative future work on the project more feasible for users who do not have many hours available to do the necessary setup and model construction.

14.2.3 Exploration of alternatives in sequence modelling other than RNNs

- For this project, we have focused solely on using RNNs with LSTM units as our sequence modeller of choice with which to build models. However, there are several alternatives that were considered but were not explored any further to keep the project scope focused on a single adaptation to a solve a specific problem.

- These include hidden Markov models (HMMs) and conditional random fields (CRFs), which can both be implemented within Python and could theoretically work with the sequences that we create in the system to predict either classification or regression targets.

- Therefore, a feasible next step in the project's lifecycle could be the replacement of the 'rnn.py' script with a script that builds from the same data a CRF-based script to predict on the same output types as 'rnn.py'. From this, one might find out that a different class of sequence modelling produces better models for the data sets than what we have been using with RNNs.

14.2.4 Experiment with feature selection/reduction techniques other than 'PCA'

- For the duration of the project, we have been using 'PCA' as the feature selection/reduction method of choice and, even though we have looked at using different numbers of features, we have not experimented with any other techniques only than 'PCA'.

- Another possible model predictions set to undertake could involve the experimenting with using different feature selection/reduction techniques within 'ft_sel_red.py' (such as Gaussian random projection or random forest feature selector), training models with this data, and comparing the performance on left-out subjects.

14.3 Additional Project Adaptations

14.3.1 Verification of source '.mat' file data integrity through the use of data anomaly detection techniques and/or use of the 'dis_3d_pos.py' script

- To the best of our knowledge, the majority of the source '.mat' files have not been checked for data integrity or checked for any anomalies within the data that may be the fault of the suit itself or the software involved in transforming the data from '.mvnx' to '.mat' format.

- It's advisable to ensure that all the data in each of the data sets used in this project is checked for anomalies within the data (i.e. parts of the data that is not a true reflection of the subject's behaviour); this would help to ensure that the models learn from 'correct' data and thus would hopefully help them to generalise better when used in production.

- This could possibly be done by creating a script that analyses all of the data within each of the source '.mat' files and checks for sudden changes in values for the various frames. From here, we could have the file manually inspected to see if these seem to be anomalies; alternatively, we could use the 'dis_3d_pos.py' to see if the file shows strange behaviour at the time flagged.

- We could possibly 'cut' out parts of the source '.mat' file that are distorted to avoid having to discard the whole file.

14.3.2 Given an annotation sheet of natural movement behaviour file names and what activity is contained within each, draw conclusions about the type of natural behaviour that is better able to predict a subject's NSAA assessments

- Ideally, one would be able to obtain a sheet corresponding to each of the natural movement behaviour files and what activities are contained within each; failing that, one could analyse each file in either 'dis_3d_pos.py' or by watching the '.mov' video files that correspond to each source '.mat' file to assess which natural movement behaviours are exhibited by the subject in the given file.

- If this is done, we would be able to run a new model predictions set with the aim to test each of the NMB data set files in turn and try to see whether or not specific natural activities and corresponding NSAA scores are more closely linked to the true overall NSAA score; in other words, we would do the same thing for different types of natural movement behaviours as we had done in MPS 23 for different single activity files.

- In doing this, we could find out the most useful natural activities that subjects could do to assess their overall NSAA scores the most accurately. Coupled with the results from MPS 22 where we saw NMB files being assessed on NSAA models fairly accurately, this could lead us to a small set of natural movements that a subject needs to undertake for the system to assess them accurately via models trained on the full set of NSAA files from the NSAA data set, as opposed to having the subject carry out numerous NSAA activities as their assessment.

14.3.3 Validate the correctness of the original NSAA scores obtained in 'nsaa_6mw_info.xlsx' and the single-act start/end times within the Google annotations sheet

- One of the main drawbacks of using this type of data is that the true values for subjects' overall and single-act NSAA assessments (as contained in 'nsaa_6mw_info.xlsx') is that the scores were obtained by possibly only one assessor and only viewing the activity undertaken by the subject once.

- To ensure that the 'correct' scores have been entered into the table, it would be advisable for one or more of those within the wider research initiative to independently go through the available video footage of the NSAA assessments for the available subjects and validate that the subjects were given the correct scores.

- If it's widely and obviously determined that one or more of the activities for a certain subject was mislabelled, then modifying the 'nsaa_6mw_info.xlsx' and rebuilding the models with the correct labels may help lead the models towards performing better for tasks such as generalising to new subjects or new versions of existing subjects.

14.3.4 Working with source '.mat' files of subjects with different movement conditions other than DMD

- We could very easily modify the system to work with data of subjects with a different type of condition other than DMD; this could possibly be another form of muscular dystrophy or any other condition where subjects' movements differ from what would be considered 'healthy' movement.

- If this condition is also captured from the subjects by the Xsens body suit and if the data is then converted from '.mvnx' to '.mat' format, it will contain the data in the same structure as we are used to; hence it would be processed the same way and would be used to train models and assess specific files in the same way as we've done for DMD data. Alternatively, any movement capturing software that produces '.mat' files with data in the same format as the source '.mat' files used here would be acceptable for the data pipeline.

- The only other necessary information needed for the system would be an annotation file in the same format as 'nsaa_6mw_info.xlsx' to contain the requisite labels for various subjects. Additional modification of the 'preprocessing()' functions used by 'rnn.py' and 'model_predictor.py' scripts would be necessary to build models that train towards these different 'y' target labels. For example, if the new condition (other than DMD) that we have source '.mat' files of and that are used to train models needs to be classified as being of several classes (i.e. an extension of binary classification as we used for the 'dhc' output type), the 'preprocessing()' functions would need to be modified to reflect this.

- With the new condition data obtained (with the data sets in the local directory and modifying the 'sub_dirs' variable in 'settings.py' to point to this), the annotation file setup as a replacement to 'nsaa_6mw_info.xlsx' (modifying the 'nsaa_6mw_path' variable in 'settings.py' to reflect this change), and the necessary preprocessing functions modified, we could feasibly build models on this data which we could then use to assess new subjects with the condition in the same way as we have done with the DMD subjects.

- In this sense, the framework for the system would be nearly identical, with only minor modifications needed to adapt the system to work with this new data. Therefore, it's fairly trivial to set this up as continuation work to get the system to work with new forms of data.

14.3.5 Working with sequence-based biomedical data of non-'.mat' data sets

- Taking the project at its highest level, we essentially have a system that takes in source '.mat' files, extracts sequences from them either from raw or feature extracted measurements, trains models on this data, and assesses new files using these models.

- While the above section provides an outline for data for different conditions that exists in a similar or identical format as what we are used to, we could feasibly adapt the system to work with different sorts of source data files (e.g. '.csv' files) and of non-human-movement based conditions. In this case, the complete system created by this project would serve as a 'roadmap' for one way to build a complete data preprocessing, model training, and file assessment system, which has proved to be a success for this project. Heavy modifications, however, would have to be made to the scripts outlined here in order for them to work with this drastically-different data set.

- The only real requirements would be that the data can be loaded and manipulated using Python, that the data can be sensibly divided into sequences (e.g. not image data, as the images themselves are independent of subsequent images in most cases), and we have information that allows us to create true labels of the data we are given (i.e. an equivalent to 'nsaa_6mw_info.xlsx' to provide classification labels or regression values that correspond with the created sequences). In this sense, we could use the skeleton of the system outlined in this project to provide the basis of a completely-new project (e.g. sequence modelling of medical records to predict patient risk of various medical conditions).

Appendix A

Source code

A.1 'assess_nsaa_nmb_file.py'

```
import os

file_path = input("\nPlease enter the path to the '.mat' file to assess (inc '.mat' extension): ")

nsaa_nmb = ""
while nsaa_nmb != "NSAA" and nsaa_nmb != "NMB":
    nsaa_nmb = input("\nPlease specify either 'NSAA' or 'NMB' for the type of activities contained in the file:↘
        ")


print("\n---------- Running 'file_mover.py...' ----------")
#Runs the 'file_mover.py' script to move the file to the 'NSAA' directory for
os.system("python file_mover.py " + nsaa_nmb + " " + file_path)

print("\n---------- Running 'file_renamer.py...' ----------")
#Runs the 'file_renamer.py' script to rename the file that has now been moved to the 'NSAA' directory
os.system("python file_renamer.py " + nsaa_nmb)


measures = input("\nPlease enter the measurements (separated by commas) to use to assess the file: ")


#Uses the full path name to the file to get the name of the file itself, the subject name within the file, and ↘
    a
#list of measurements that we wish to extract as a list from the 'measures' arg
file_name = file_path.split("\\")[-1]
subject_name = file_name.split("-")[0]
measurements = measures.split(",")


print("\n---------- Running 'ext_raw_measures.py...' ----------")
#Extracts the raw measurements using all measurements specified apart from 'AD' (if it was included)
measures_no_ad = ",".join([m for m in measurements if m != "AD"])
os.system("python ext_raw_measures.py " + nsaa_nmb + " " + file_name + " " + measures_no_ad)

#Runs the 'comp_stat_vals.py' and 'ft_sel_red.py' scripts if 'AD' was one of the measurements given
if "AD" in measurements:
    print("\n---------- Running 'comp_stat_vals.py...' ----------")
    os.system("python comp_stat_vals.py " + nsaa_nmb + " AD " + file_name + " --split_size=1")
    print("\n---------- Running 'ft_sel_red.py...' ----------")
    os.system("python ft_sel_red.py " + nsaa_nmb + " AD " + subject_name +
              " pca --num_features=30 --no_normalize --new_subject --batch")


use_altdir = ""
while use_altdir != "y" and use_altdir != "n":
    use_altdir = input("\nDo you wish to assess the file on an 'alt dir' (e.g. if file is an 'NSAA' file, would↘
        "
                       "assess the file on models built only on NMB) ('y'/'n')?: ")


print("\n---------- Running 'model_predictor.py...' ----------")
#Assess the models on the pre-built models contained within 'rnn_models_final' in the project directory
altdir = "NSAA" if nsaa_nmb == "NMB" else "NMB"
if use_altdir == "n":
    use_leaveoutV2 = ""
    while use_leaveoutV2 != "y" and use_leaveoutV2 != "n":
        use_leaveoutV2 = input("\nDo you wish to assess the file on models built that specifically haven't been↘
            "
                               "trained on any 'V2' files? ('y'/'n'): ")
    if use_leaveoutV2 == "n":
        os.system("python model_predictor.py " + nsaa_nmb + " " + measures + " " + subject_name + " --add_dir="↘
```

```
                +
59                  altdir + " --combine_preds --no_testset --use_seen --new_subject --final_models --↘
                        no_nsaa_flag")
60      else:
61          os.system("python model_predictor.py " + nsaa_nmb + " " + measures + " " + subject_name + " --add_dir="↘
                +
62                  altdir + " --combine_preds --no_testset --use_seen --new_subject --final_models --↘
                        no_nsaa_flag " +
63                  "--leave_out_version=V2")
64  else:
65      os.system("python model_predictor.py " + nsaa_nmb + " " + measures + " " + subject_name + "--alt_dirs=" +
66                altdir + " --combine_preds --no_testset --use_seen --new_subject --final_models --no_nsaa_flag")
```

A.2 'comp_stat_vals.py'

```
1   import sys
2   sys.path.append("..")
3   import scipy.io as sio
4   import numpy as np
5   from numpy import linalg as la
6   import pandas as pd
7   from matplotlib import pyplot as plt
8   from mpl_toolkits.mplot3d import Axes3D
9   import mpl_toolkits.mplot3d.axes3d as p3
10  import matplotlib.animation as animation
11  from tkinter import*
12  import os
13  from os.path import isfile
14  import argparse
15  from settings import sampling_rate, local_dir, sub_dirs, source_dir, short_file_types, axis_labels, ↘
        segment_labels, \
16      joint_labels, sensor_labels, measure_to_len_map, seg_join_sens_map
17
18
19  #Reassigns the name of 'source_dir' to 'output_dir' in this script, as what is used as the source for several
20  #other scripts is what is used as the output directory in this script
21  output_dir = source_dir
22  file_types = short_file_types
23
24
25  """
26  Section below covers a few general-purpose mathematical functions used for several file object types
27  for statistical analysis
28  """
29  def mean_round(nums):
30      """
31          :param list of values of which we wish to find the mean/average:
32          :return a float value rounded to 2 decimal places of the mean of 'nums':
33      """
34      return round(float(np.mean(nums)), 4)
35
36  def variance_round(nums):
37      """
38          :param list of values of which we wish to find the variance:
39          :return a float value rounded to 2 decimal places of the variance of 'nums':
40      """
41      return round(float(np.var(nums)), 4)
42
43  def compute_diffs(nums):
44      """
45          :param list of values of which we wish to find the diffs between each successive value:
46          :return list of diff values of len = len(nums)-1, where each value is difference between value
47          in the 'nums' list and the next value in the list:
48      """
49      return [(nums[i+1]-nums[i]) for i in range(len(nums)-1)]
50
51  def mean_diff_round(nums):
52      """
53          :param list of values of which we wish to find the mean diff values:
54          :return mean value of the absolute values of the diffs list (see 'compute_diffs'):
55      """
56      return round(float(np.mean(np.absolute(compute_diffs(nums)))), 4)
57
58  def fft_1d_round(nums):
59      """
60          :param list of values of which we want to find the 1-dimensional FFT for 3 values
61          :return largest FFT value from list of 3 values rounded to 4 decimal place
62      """
63      fft_1d = np.fft.rfft(nums, n=3)
64      fft_1d.sort()
65      return round(np.real(np.flip(fft_1d)[0]), 4)
66
67  def covariance_round(nums):
68      """
69          :param list of values of 2D array of numbers to find covariance of (shape = # of dimensions of data (e.↘
                g. 3 for
70          x,y,z data) x # of samples):
71          :return covariance values of rounded float values of shape = # of dimensions of data x # of dimensions ↘
                of data,
72          for 'top' right triangle at 1D list:
```

```
73      """
74      return [[round(float(n), 8) for n in num] for num in np.cov(nums.tolist())]
75
76  def fft_2d_round(nums):
77      """
78          :param list of 2D values of which we want to find the 2-dimensional FFT for 3 values:
79          :return list of 3 values returned from the 2D FFT operation, in descending order and each rounded
80          to 4 decimal places
81      """
82      fft_2d = np.fft.fft2(nums, s=(1, 3))
83      fft_2d.sort()
84      fft_2d = np.flip(fft_2d)
85      return [round(np.real(fft_2d[0][i]), 4) for i in range(len(fft_2d[0]))]
86
87  def mean_sum_vals(nums):
88      """
89          :param list of values of 2D array of numbers of which we wish to find the mean of the sums of each ↘
                dimension
90          :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
91          values), rounded to 4 decimal places
92      """
93      return round(float(np.mean([np.sum(nums[i]) for i in range(len(nums))])), 4)
94
95  def mean_sum_abs_vals(nums):
96      """
97          :param list of values of 2D array of numbers of which we wish to find the mean of the sums of each ↘
                dimension
98          :return same as 'mean_sum_vals', with difference that each value in 2D input array is 'absoluted' first
99      """
100     return round(float(np.mean([np.sum(np.abs(nums[i])) for i in range(len(nums))])), 4)
101
102 def cov_eigenvals(nums, i):
103     """
104         :param list of values of 2D array of numbers of which we wish to find the eigenvals of covariance ↘
                matrix, and an
105         index to indicate which of the eigenvals we wish to return
106         :return one of the two largest (rounded to 4 decimal places) eigenvalues of the covariance matrix of ↘
                the
107         2D array of numbers (i.e. the top 2 of 3 eigenvals), depending on the index 'i'
108     """
109     vals = la.eig(covariance_round(nums))[0].tolist()
110     vals = list(vals)
111     #Ensures that we only use the 'real' components here, as sorting based on complex numbers isn't possible ↘
            here
112     try:
113         vals.sort(reverse=True)
114     except TypeError:
115         vals = list(np.real(vals))
116         vals.sort(reverse=True)
117     top_vals = vals[:2]
118     return round(top_vals[i], 4)
119
120
121
122 def prop_outside_mean_zone(nums, percen=0.1):
123     """
124         :param array of 2D numbers of which we wish to find the proportion of the samples in the array that are↘
                outside
125         'percen' boundaries around the mean zone (e.g. if mean zone is 5 for one dim, the sample would have to ↘
                have
126         a value not between 4.5 and 5.5 for that dimension's value to be considered 'outside'; the same just ↘
                also
127         be true for its other 2 dimensions)
128         :return the proportion of the samples within 'nums' that are outside the mean zone (i.e. ALL 3 of its
129         lie beyond their respective mean zones
130     """
131     #Mean values of the array of nums for each of its x, y, and z dimensions; used to calculate if a sample is
132     #outside the mean zone
133     x_mean, y_mean, z_mean = np.mean(nums[0]), np.mean(nums[1]), np.mean(nums[2])
134     nums_outside = 0
135     nums = np.swapaxes(nums, 0, 1)
136     for num in nums:
137         #Each sample must have all 3 of its dimensions within the 'mean zone' to have 1 added to
138         if not x_mean*(1-percen) < num[0] < x_mean*(1+percen):
139             if not y_mean*(1-percen) < num[1] < y_mean*(1+percen):
140                 if not z_mean*(1-percen) < num[2] < z_mean*(1+percen):
141                     nums_outside += 1
142     return round(nums_outside/len(nums), 4)
143
144
145
146 def extract_stats_features(f_index, file_part, split_file, file_name, measurement_names, is_recursive_call=↘
        False):
147     """
148
149     :param 'f_index' for a number the file_part represents of the complete file, 'file_part' being the data ↘
            itself that
150     we will be extracting the statistical features of, 'split_file' being the total number of parts within the ↘
            complete
151     file that 'file_part' comes from, 'file_name' being the name of the file that 'file_part' comes from,
152     'measurement_names' being a list of measurements (e.g. angularAcceleration, position, jointAngles, etc.) we↘
            wish to
153     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
        try:
            frame_data = [[elem[0] if len(elem[0]) != 1 else elem[0][0] for elem in row] for row in frame_data]
        except IndexError:
            #Accounts for missing 'contact' values in certain rows of some '.mat' files by ignoring the 'contact' column
            new_frame_data = []
            for m, row in enumerate(frame_data):
                #Ignore rows that don't have 'normal' as their 'type' cell
                if row[3][0] != "normal":
                    continue
                row_data = []
                for i in range(len(row)):
                    if i == len(row) - 1:
                        row_data.append(["", ""])
                    elif len(row[i][0]) != 1:
                        row_data.append(row[i][0])
                    else:
                        row_data.append(row[i][0][0])
                new_frame_data.append(row_data)
            frame_data = new_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])
```

```
323             ax.set_zlim(mins[2], maxs[2])
324
325             xy_ratio = (maxs[1]-mins[1])/(maxs[0]-mins[0])
326             xz_ratio = (maxs[2]-mins[2])/(maxs[0]-mins[0])
327             ax.get_proj = lambda: np.dot(Axes3D.get_proj(ax), np.diag([1, xy_ratio, xz_ratio, 0.25]))
328
329             colors = plt.cm.jet(np.linspace(0, 1, len(xyz_pos[0][0])))
330             lines = sum([ax.plot([], [], [], '-', c=c) for c in colors], [])
331             pts = sum([ax.plot([], [], [], 'o', c=c) for c in colors], [])
332
333             stick_lines = [ax.plot([], [], [], 'k-')[0] for _ in stick_defines]
334
335             #Defines the 'init' function object to setup the points and lines defining a person in 3d space
336             def init():
337                 for line, pt in zip(lines, pts):
338                     line.set_data([], [])
339                     line.set_3d_properties([])
340                     pt.set_data([], [])
341                     pt.set_3d_properties([])
342                 return lines + pts + stick_lines
343
344             #Function object that updates the plot based on the next sample of 3D coordinates for each feature (i.e
                .
345             #each 'point' on the walking figure in the plot)
346             def animate(i):
347                 if i >= len(xyz_pos[0]):
348                     exit()
349                 if (i*5)%sampling_rate == 0:
350                     print("Plotting time: " + str(int((i*5)/sampling_rate)) + "s")
351                 i = (5 * i) % xyz_pos_t.shape[1]
352                 for line, pt, xi in zip(lines, pts, xyz_pos_t):
353                     x, y, z = xi[:i].T
354                     pt.set_data(x[-1:], y[-1:])
355                     pt.set_3d_properties(z[-1:])
356                 for stick_line, (sp, ep) in zip(stick_lines, stick_defines):
357                     stick_line._verts3d = xyz_pos_t[[sp, ep], i, :].T.tolist()
358                 fig.canvas.draw()
359                 if (i+5) >= int(len(xyz_pos[0])):
360                     plt.close()
361                     print("Animation ended...")
362                 return lines + pts + stick_lines
363
364             #Plot the figure in 3D space, with an update interval (defined in miliseconds) to be in real time
365             anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len(xyz_pos[0]),
366                                            interval=1000 / sampling_rate, blit=False, repeat=False)
367             plt.show()
368
369
370
371     def file_info(self):
372         """
373             :param string name of the unique identifier of an 'AD-' data matlab file (e.g. 'HC3'):
374             :return no return, but sets up the DataFrame 'df' attribute from the given matlab file: name
375         """
376         print("\nAd " + self.ad_file_name + " keys:", ", ".join([("'" + key + "'") for key in self.df]), "\n")
377         for k in list(self.df.keys())[:3]:
378             print(k, ":", (self.df[k]))
379
380
381     def write_statistical_features(self, measurements_to_extract=None, output_name=None, split_file=1,
          split_size=None):
382         """
383             :param 'measurements_to_extract' to be an optional list of measurement names from the AD file that
                  we wish to
384             apply statistical analysis (otherwise, defaults to 'measurement_names'), along with 'output_name'
                  that
385             defaults to the short name of the AD file, which writes to an individual file for the object (i.e.
386             a single line .csv for the object); specify shared 'output_name' to instead append to an existing .
                  csv;
387             'split_file' and 'split_size' optionally given as 2 different ways to break up a file into parts
388             :return no return, but writes to an output .csv file the statistical values of the certain
389             measurements to an output .csv file
390         """
391         #Sets the default measurements to extract from the AD file object if none are specified as args
392         measurement_names = measurements_to_extract if measurements_to_extract else [
393             "position", "sensorFreeAcceleration", "sensorMagneticField", "velocity", "angularVelocity",
394             "acceleration", "angularAcceleration"]
395
396         # Extracts values of the various measurements in different layout so 'f_d_s_data' now has shape:
397         # (# measurements (e.g. 3 for position, accel, and megnet field) x # of features (e.g. 23 for segments)
398         # x # of dimensions (e.g. 3 for x,y,z position values) x # of samples (~22K))
399         if args.single_act or args.single_act_concat:
400             extract_data = self.df.loc[:, measurement_names].values
401             f_d_s_data = np.zeros((len(measurement_names),
402                                    max(len(segment_labels), len(joint_labels), len(sensor_labels)), 3,
403                                    len(self.df.loc[:])))
404             for i in range(len(measurement_names)):
405                 for j in range(int(len(self.df.loc[:, measurement_names[i]].values[0])/3)):
406                     for k in range(len(axis_labels)):
407                         for m in range(len(self.df.loc[:])):
408                             f_d_s_data[i, j, k, m] = extract_data[m, i][(j*3)+k]
409         else:
410             extract_data = self.df.loc[3:, measurement_names].values
411             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 i, f in enumerate(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 args.single_act:
                ad_output_dir += "act_files\\"
            elif args.single_act_concat:
                ad_output_dir += "act_files_concat\\"
            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
            af = "_" + self.ad_file_name.split(".")[0].split("_")[-1] if args.single_act else ""
            if not output_name:
                if not args.single_act_concat:
                    output_complete_name = ad_output_dir + "AD_" + self.ad_short_file_name + af + "_stats_features.csv"
                else:
                    output_complete_name = ad_output_dir + "AD_" + self.ad_short_file_name.split(".")[0] + af + "_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 + af + "_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

```

```
496             try:
497                 self.dc_table = pd.read_csv(local_dir + "data_cube_table.csv")
498                 self.dc_short_file_names = self.dc_table.values[:, 1]
499                 self.dc_file_names = self.dc_table.values[:, 2]
500             except FileNotFoundError:
501                 print("Couldn't find the 'data_cube_table.csv file. Make sure to run the "
502                       "'writetable(excel_table, \"data_cube_table.csv\") in matlab with data_cube.mat file open")
503                 sys.exit()
504             self.dc = sio.loadmat(local_dir + "data_cube_6mw", matlab_compatible=True)["data_cube"][0][0][2][0]
505
506             self.ja_objs = []
507             for i in range(len(self.dc_short_file_names)):
508                 print("Extracting data from data cube file (" + str(i+1) + "/" + str(len(self.dc_short_file_names))↘
                        +
509                       "): " + self.dc_short_file_names[i] + "...")
510                 self.ja_objs.append(JointAngleFile(ja_file_name=self.dc_file_names[i], sub_dir=self.dc_sub_dir,
511                                                    dc_object_index=i, dc_short_file_name=self.dc_short_file_names[i↘
                                                        ]))
512
513             if dis_data_cube:
514                 self.display_info()
515
516
517
518         def display_info(self):
519             """
520                 :param no params
521                 :return no return, but displays some basic info about the Data Cube attribute
522             """
523             print("\nData cube keys:", ", ".join([("'" + key + "'") for key in self.dc]), "\n")
524             print("'__header__': ", self.dc["__header__"])
525             print("'__version__': ", self.dc["__version__"])
526             print("'__globals__': ", self.dc["__globals__"])
527             print(self.dc["data_cube"])
528
529
530         def write_statistical_features(self, output_name="all", split_file=1, split_size=None):
531             """
532             :param 'output_name', which defaults to 'All', which is the short name of the output .csv stats file ↘
                      that the
533             objects will share...specify a different name if desired; 'split_file' and 'split_size' optionally ↘
                      given as 2
534             different ways to break up a file into parts
535             :return no return, but creates a single .csv output file that contains the statistical analysis of each↘
                       joint
536             angle file contained within the data cube via calls to 'JointAngleFile.write_statistical_features()' ↘
                      method
537             for each of the joint angle objects extracted at initialization
538             """
539             written_names = []
540             for i in range(len(self.ja_objs)):
541                 written_names += self.ja_objs[i].write_statistical_features(output_name=output_name, split_file=↘
                                      split_file,
542                                                                             split_size=split_size)
543             return written_names
544
545
546         def write_direct_csv(self, output_name="all"):
547             """
548             :param no params
549             :return for each JointAngleFile object within the datacube, call their 'write_direct_csv' method with ↘
                      their
550             'output_name' set to their respective short name
551             """
552             written_names = []
553             for i in range(len(self.ja_objs)):
554                 written_names += self.ja_objs[i].write_direct_csv()
555             return written_names
556
557
558     class JointAngleFile(object):
559
560         def __init__(self, ja_file_name, sub_dir, dc_object_index=-1, dc_short_file_name=None):
561             """
562                 :param 'ja_file_name' being the string name of the complete name of the source file (i.e. full file
563                 path name), with 'short_fn' being the string name of the unique identifier of an 'joint angle' data
564                 matlab file (e.g. 'D2'), and 'sub_dir' being the name of the sub-directory within 'output_dir' to ↘
                        place
565                 the extracted statistical values when 'write_statistical_features' is called, and 'dc_object' which↘
                         is
566                 set to non-zero if object is created as part of a DataCubeFile:
567                 :return no return, but sets up the DataFrame 'df' attribute from the given matlab file
568             """
569             self.ja_file_name = ja_file_name
570             split_name = ja_file_name.split("\\")[-1].split("-")
571             splits = [[s for s in split_name if "HC" in s], [s.upper() for s in split_name if "D" in s.upper()]]
572             if not dc_short_file_name:
573                 self.ja_short_file_name = splits[1][0][splits[1][0].index("D"):] \
574                     if not splits[0] else splits[0][0][splits[0][0].index("HC"):]
575             else:
576                 self.ja_short_file_name = dc_short_file_name
577             self.ja_sub_dir = sub_dir
578             self.is_dc_object = True if dc_object_index != -1 else False
579             #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(local_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
```

```
666         #Angles has shape: (# of dimensions of features (3 for x,y, and z) x # features (22 here) x # samples (~22K),
667         #hence angles[i][j] selects list of sample values for dimension 'i' for feature 'j'
668         angles = np.swapaxes(np.asarray([x_angles.T, y_angles.T, z_angles.T]), 0, 1)
669 
670         #If 'split_size' is given as command line argument and 'split_file' isn't, set 'split_s' to the given number
671         #multiplied by sampling rate (i.e. if given num is 5 and 'sampling_rate' is 60, 'split_s' is 300, i.e. the
672         #number of rows in each 'split_file') and 'split_file' is set to the number of these sizes that fit in the
673         #original file (i.e. how many parts we can get out of a file given a 'split_s')
674         if split_file == 1 and split_size:
675             split_s = int(sampling_rate * split_size)
676             split_file = int(len(angles[0, 0])/split_s)
677         else:
678             split_s = int(len(angles[0, 0])/split_file)
679 
680         written_names = []
681         for i, f in enumerate(range(split_file)):
682             ang = angles[:, :, (split_s * f):(split_s * (f + 1))]
683             #Creates a dictionary of extracted statistical values for a given file part
684             df = extract_stats_features(f_index=f, split_file=split_file, file_name=self.ja_short_file_name,
685                                         measurement_names=["jointAngle"], file_part=[ang])
686             file_prefix = "DC" if self.is_dc_object else "JA"
687 
688             #Creates the relevant sub-directories within 'output_files' to store the created .csv
689             dc_ja_output_dir = output_dir + self.ja_sub_dir + "\\"
690             if not os.path.exists(dc_ja_output_dir):
691                 os.mkdir(dc_ja_output_dir)
692             dc_ja_output_dir += file_prefix + "\\"
693             if not os.path.exists(dc_ja_output_dir):
694                 os.mkdir(dc_ja_output_dir)
695 
696             #Sets output name of file (and related print statement) to whether or not the user is storing it in
697             #an 'all' .csv or a .csv determined by the name of the writing file
698             if not output_name:
699                 output_complete_name = dc_ja_output_dir + file_prefix + "_" + self.ja_short_file_name + "_stats_features.csv"
700                 print("Writing", file_prefix, self.ja_file_name, "(", (f+1), "/", split_file,
701                       ") statistial info to", output_complete_name)
702             else:
703                 output_complete_name = dc_ja_output_dir + file_prefix + "_" + output_name + "_stats_features.csv"
704                 print("Writing", file_prefix, self.ja_file_name, "(", (f+1), "/",
705                       split_file, " ) statistial info to", output_complete_name)
706             #Writes just the data to file if the file already exists, or both the data and headers if the file
707             #doesn't exist yet
708             if isfile(output_complete_name):
709                 with open(output_complete_name, 'a', newline='') as file:
710                     df.to_csv(file, header=False, line_terminator="")
711             else:
712                 with open(output_complete_name, 'w', newline='') as file:
713                     df.to_csv(file, header=True, line_terminator="")
714             written_names.append(output_complete_name)
715         #Returns the complete names of the file(s) that have been written or appended to a .csv
716         return written_names
717 
718 
719     def write_direct_csv(self, output_name=None):
720         """
721             :param 'output_name', which, if specified, ensures the file is written to a specified file name rather
722             than a name dependent on the source file name (e.g. 'D2')
723             :return: no return, but instead directly writes a JointAngleFile object from a .mat file to a .csv
724             without extracting any of the statistical features
725         """
726         df = pd.DataFrame(self.ja_data, index=[self.ja_short_file_name for i in range(len(self.ja_data))])
727         headers = [("(" + joint_labels[i] + ") : (" + axis_labels[j] + "-axis)")
728                    for i in range(len(joint_labels)) for j in range(len(axis_labels))]
729         file_prefix = "DC" if self.is_dc_object else "JA"
730 
731         # Creates the relevant sub-directories within 'output_files' to store the created .csv
732         dc_ja_output_dir = output_dir + "direct_csv\\"
733         if not os.path.exists(dc_ja_output_dir):
734             os.mkdir(dc_ja_output_dir)
735         dc_ja_output_dir += file_prefix + "\\"
736         if not os.path.exists(dc_ja_output_dir):
737             os.mkdir(dc_ja_output_dir)
738 
739         if not output_name:
740             output_complete_name = dc_ja_output_dir + file_prefix + "_" + self.ja_short_file_name + ".csv"
741             print("Writing", file_prefix, self.ja_file_name, "to", output_complete_name)
742         else:
743             output_complete_name = dc_ja_output_dir + file_prefix + "_" + output_name + ".csv"
744             print("Writing", file_prefix, self.ja_file_name, "to", output_complete_name)
745         if isfile(output_complete_name):
746             with open(output_complete_name, 'a', newline='') as file:
747                 df.to_csv(file, header=False)
748         else:
749             with open(output_complete_name, 'w', newline='') as file:
750                 df.to_csv(file, header=headers)
751         return output_complete_name
752 
```

```
753
754
755
756 def class_selector(ft, fn, fns, sub_dir, is_all, split_files, is_extract_csv=False, split_size=None):
757     """
758         :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
759         (i.e. the full-directory path of the file) if 'class_selector' is dealing with a single file, 'fns' is ↘
                the
760         full file names if 'class_selector' is dealing with multiple files(i.e. the 'all'
761         command line argument is given for 'fn'), 'sub_dir' is the sub-directory to write the file(s)
762         in question in their extracted stats .csv format, 'is_all' is set to true if 'all is given as 'fn' ↘
                command line
763         argument (else false), 'split_files' is set at the value given by '--split_files' command line argument↘
                (else 1),
764         'is_extract_csv' is true if calling 'write_direct_csv' on JointAngleFile object (else false), and '↘
                split_size'
765         is the value set by the '--split_size' command line argument (else None)
766         :return list of names that have already been written to output .csv; however, the more important ↘
                operation is
767         the creation of various file objects and the calling on their respective 'write_statistical_features' ↘
                methods
768         with arguments governed by the passed in arguments to 'class_selector'
769     """
770     names = []
771     if ft == "AD":
772         if is_all:
773             for f in fns:
774                 names += AllDataFile(f, sub_dir).write_statistical_features(split_file=split_files, split_size=↘
                        split_size)
775         else:
776             names.append(AllDataFile(fn, sub_dir).write_statistical_features(
777                 split_file=split_files, split_size=split_size))
778     elif ft == "JA":
779         if not is_extract_csv:
780             if is_all:
781                 fns = [fn for fn in fns if "jointangle" in fn]
782                 for f in fns:
783                     names += JointAngleFile(f, sub_dir).write_statistical_features(
784                         output_name="all", split_file=split_files, split_size=split_size)
785             else:
786                 names += JointAngleFile(fn, sub_dir).write_statistical_features(
787                     split_file=split_files, split_size=split_size)
788         else:
789             if is_all:
790                 fns = [fn for fn in fns if "jointangle" in fn]
791                 for f in fns:
792                     names += JointAngleFile(f, sub_dir).write_direct_csv()
793             else:
794                 names += JointAngleFile(fn, sub_dir).write_direct_csv()
795     else:
796         if not is_extract_csv:
797             names += DataCubeFile(sub_dir).write_statistical_features(split_file=split_files, split_size=↘
                    split_size)
798         else:
799             names += DataCubeFile(sub_dir).write_direct_csv()
800     return list(dict.fromkeys(np.ravel(names)))
801
802
803
804 def del_files(names):
805     for name in names:
806         try:
807             os.remove(name)
808             print("'" + str(name) + "' successfully deleted")
809         except FileNotFoundError:
810             print("'" + str(name) + "' doesn't exist, unable to delete...")
811
812
813
814 """Section below encompases all the command line arguments that can be supplied to the program, with those ↘
        beginning
815 with '--' being optional arguments and the others being required arguments (with the exception of 'fn' argument↘
         not
816 being necessary if 'ft' is 'DC')"""
817 parser = argparse.ArgumentParser()
818 parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained ↘
        within "
819                                 "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles', "
820                                 "'NMB', or 'NSAA'.")
821 parser.add_argument("ft", help="Specify type of file we wish to read from, being one of 'JA' (joint angle), "
822                                "'AD' (all data), or 'DC' (data cube).")
823 parser.add_argument("fn", nargs="?", default="DC",
824                     help="Specify the short file name to load; e.g. for file 'All-HC2-6MinWalk.mat' or "
825                          "jointangleHC2-6MinWalk.mat, enter 'HC2'. Specify 'all' for all the files available "
826                          "in the default directory of the specified file type. Optional for 'DC' file type.")
827 parser.add_argument("--dis_3d_pos", type=bool, nargs="?", const=True,
828                     help="Plots the dynamic positions of an AllDataFile object over time. "
829                          "Only works with 'ft' set as an 'AD' file name.")
830 parser.add_argument("--dis_diff_plot", type=bool, nargs="?", const=True,
831                     help="Plots the diff plots of all features and axes of a JointAngleFile object over time."
832                          "Only works with 'ft' set as a 'JA' file name.")
833 parser.add_argument("--dis_3d_angs", type=bool, nargs="?", const=True,
834                     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'")
parser.add_argument("--single_act", type=bool, nargs="?", const=True,
                    help="Specify if the files to operate on are 'single act' files.")
parser.add_argument("--single_act_concat", type=bool, nargs="?", const=True,
                    help="Specify if the files to operate on are 'single act concat' files.")
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 'local_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:
    local_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', 'NSAA', 'direct_csv', 'allmatfiles', 'NMB', or 'left-out'.")
    sys.exit()
file_names = []
if args.dir == "6minwalk-matfiles":
    if args.ft.upper() == "AD":
        local_dir += "all_data_mat_files\\"
        file_names = os.listdir(local_dir)
    elif args.ft.upper() == "JA":
        local_dir += "joint_angles_only_matfiles\\"
        file_names = os.listdir(local_dir)
    elif args.ft.upper() == "DC":
        local_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(local_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()
elif args.dir == "left-out" or args.dir == "NMB":
    file_names = [f for f in os.listdir(local_dir)]
else:
    if args.ft.upper() == "AD":
        # Only 'matfiles' subdirectory of 'NSAA' applicable for analysis with this script
        local_dir += "matfiles\\"
        if args.single_act:
            if args.dir == "NSAA":
                local_dir += "act_files\\"
            else:
                print("Must be using 'NSAA\\AD' files when using '--single_act' optional argument...")
                sys.exit()
        elif args.single_act_concat:
            if args.dir == "NSAA":
                local_dir += "act_files_concat\\"
            else:
                print("Must be using 'NSAA\\AD' files when using '--single_act_concat' optional argument...")
                sys.exit()
        file_names = [f for f in os.listdir(local_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 = local_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 = [local_dir + fn for fn in file_names if fn.endswith(".mat")]
                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 = local_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 = local_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 = local_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()
```

A.3 'data_balancer.py'

```
from collections import OrderedDict
import pandas as pd
from random import shuffle
from random import randint
from settings import nsaa_6mw_path


def ext_label_dist(file_name, batch):
    """
        :param the name of the file name (not inc directory path) from which we wish to get the overall NSAA score
        :return the 'overall NSAA score' that corresponds to the file name
    """
    #Gets the short file name (e.g. 'D4') from the fill name
    if file_name.startswith("FR"):
        short_file_name = file_name.split("_")[2]
    elif file_name.startswith("AD"):
        short_file_name = file_name.split("_")[1]
    else:
        short_file_name = file_name.split("-")[0]

    #Loads the table of short-file-names-to-overall-NSAA-score, gets the relevant columns, and creates a dictionary
    #from these columns
    global nsaa_6mw_path
    nsaa_6mw_path = "..\\" + nsaa_6mw_path if batch else nsaa_6mw_path
    nsaa_6mw_tab = pd.read_excel(nsaa_6mw_path)
    nsaa_6mw_cols = nsaa_6mw_tab[["ID", "NSAA"]]
    nsaa_overall_dict = dict(pd.Series(nsaa_6mw_cols.NSAA.values, index=nsaa_6mw_cols.ID).to_dict())

    #Ensures that short file names ending with 'V2' get a label (e.g. 'D6V2' uses the score for for 'D6') and, with
    #this, gets the overall NSAA score from the dictionary for the given short file name
    if short_file_name in nsaa_overall_dict:
        y_label_balance = nsaa_overall_dict[short_file_name]
    else:
        y_label_balance = nsaa_overall_dict[short_file_name[:-2]]
    return y_label_balance



def downsample(x_data, y_data, y_data_balance):
    """
        :param 'x_data' (the 'x' component of the data that we wish to balance), 'y_data' (the 'y' component of the
        data that we wish to balance), and 'y_data_balance' (corresponding overall NSAA scores for each sample of 'x'
        and 'y' that are used to determine which x and y samples to remove or keep; not this is the same as 'y_data' if
        'choice' arg from 'rnn.py' is 'overall')
        :return the downsampled x and y components of data as 'new_x' and 'new_y', with strings to print later on in
        'rnn.py' (at the end of training and testing) returned here as 'out_strs'
    """

    out_strs = []
    #Creates a dictionary of overall NSAA scores and their frequency within 'x' and 'y' components in ascending
    #frequency order
    label_dict = {l: y_data_balance.count(l) for l in set(y_data_balance)}
    label_count = sorted(label_dict.items(), key=lambda x: x[1])
    out_strs.append("Overall NSAA labels for sequences and freq in Y before downsampling: " +
                    str(dict(OrderedDict(sorted(label_dict.items(), key=lambda t: t[0])))))
    #Shuffles the zipped 'x' and 'y' data so they both get shuffled in the same way so the samples that we remove are
    #completely randomised within the sample
    Z = list(zip(x_data, y_data, y_data_balance))
    shuffle(Z)
    x_data, y_data, y_data_balance = zip(*Z)

    #Gets the lowest frequency of a given overall NSAA score; this is the frequency of each overall NSAA scores'
    #corresponding 'x' and 'y' samples should be reduced to
    tar_len = label_count[0][1]
    dict_count = {}
    new_x, new_y = [], []

    #For each zipped x sample, y sample, and corresponding overall NSAA score, add 1 to their corresponding overall NSAA
    #score in the 'dict_count' dictionary and, if it's below the lowest frequency of scores, add the x and y samples to
    #their respective lists; this ensures that, for a given overall NSAA score, there are never more than 'tar_len'
    #x and y samples added to the new data lists
    for x, y, z in zip(x_data, y_data, y_data_balance):
        if z in dict_count:
            if dict_count[z] < tar_len:
                dict_count[z] += 1
                new_x.append(x)
                new_y.append(y)
        else:
```

```
79              dict_count[z] = 1
80              new_x.append(x)
81              new_y.append(y)
82
83      out_strs.append("Overall NSAA labels for sequences and freq in Y after downsampling: " +
84                      str(dict(OrderedDict(sorted(dict_count.items(), key=lambda t: t[0])))))
85      return new_x, new_y, out_strs
86
87
88
89  def upsample(x_data, y_data, y_data_balance):
90      """
91          :param 'x_data' (the 'x' component of the data that we wish to balance), 'y_data' (the 'y' component of the
92          data that we wish to balance), and 'y_data_balance' (corresponding overall NSAA scores for each sample of 'x'
93          and 'y' that are used to determine which x and y samples to remove or keep; not this is the same as 'y_data' if
94          'choice' arg from 'rnn.py' is 'overall')
95          :return the upsampled x and y components of data as 'new_x' and 'new_y', with strings to print later on in
96          'rnn.py' (at the end of training and testing) returned here as 'out_strs'
97      """
98      out_strs = []
99      #Creates a dictionary of overall NSAA scores and their frequency within 'x' and 'y' components in ascending
100     #frequency order
101     label_dict = {l: y_data_balance.count(l) for l in set(y_data_balance)}
102     label_count = sorted(label_dict.items(), key=lambda x: x[1])
103     out_strs.append("Overall NSAA labels for sequences and freq in Y before upsampling: " +
104                     str(dict(OrderedDict(sorted(label_dict.items(), key=lambda t: t[0])))))
105
106     #Gets the highest frequency of a given overall NSAA score; this is the frequency of each overall NSAA scores'
107     #corresponding 'x' and 'y' samples should be increased to by random sampling
108     tar_len = label_count[-1][1]
109
110     #Dictionary contains keys as 'y_data_balance' values (i.e. overall NSAA values), with values being a list of
111     #2-element lists, with 1st elem being an x-val and 2nd being a y-val, each pair corresponding to a value
112     #in y_data_balance
113     class_dict = {}
114     for x, y, z in zip(x_data, y_data, y_data_balance):
115         if z in class_dict:
116             class_dict[z].append([x, y])
117         else:
118             class_dict[z] = [[x, y]]
119
120     dict_count = {k: 0 for k in class_dict}
121     new_x, new_y = [], []
122     #For each overall NSAA score, we sample from it's list of x/y sample pairs at a random point within the entirety
123     #of it's list length ('rand_pos') and add each part to their respective 'new_x' and 'new_y' lists (note that as we
124     #are not removing it from the original list when adding to 'new_x', this is sampling with replacement)
125     for key in class_dict:
126         for i in range(tar_len):
127             rand_pos = randint(0, len(class_dict[key])-1)
128             new_x.append(class_dict[key][rand_pos][0])
129             new_y.append(class_dict[key][rand_pos][1])
130             dict_count[key] += 1
131
132     out_strs.append("Overall NSAA labels for sequences and freq in Y after downsampling: " +
133                     str(dict(OrderedDict(sorted(dict_count.items(), key=lambda t: t[0])))))
134
135     return new_x, new_y, out_strs
```

A.4 'dis_3d_pos.py'

```
1   import argparse
2   import pandas as pd
3   from settings import sub_dirs, local_dir, sampling_rate
4   import sys
5   import os
6   import scipy.io as sio
7   import numpy as np
8   from matplotlib import pyplot as plt
9   from mpl_toolkits.mplot3d import Axes3D
10  import mpl_toolkits.mplot3d.axes3d as p3
11  import matplotlib.animation as animation
12  from tkinter import*
13
14
15  parser = argparse.ArgumentParser()
16  parser.add_argument("dir", help="Specifies which source directory to use so as to retrieve the file contained within "
17                                  "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles' or 'NSAA'.")
18  parser.add_argument("fn", type=str, help="Specify the short file name to load; e.g. for file 'All-HC2-6MinWalk.mat' or "
19                                           "jointangleHC2-6MinWalk.mat, enter 'HC2'.")
```

```
20  args = parser.parse_args()
21
22
23
24  def preprocessing():
25      """
26          :return: Given the 'dir' and 'fn' arguments, checks for argument validity, loads the file from '.mat' ↘
                form,
27          and returns the complete file as a DataFrame
28      """
29
30      #Checks the 'dir' argument for validity (note: 'allmatfiles' and 'direct_csv' can't be use as these ↘
            currently
31      #only contain joint angle files and not the position data needed here)
32      if args.dir + "\\" in sub_dirs and args.dir != "allmatfiles" and args.dir != "direct_csv":
33          dir = args.dir
34      else:
35          print("First arg ('dir') not a valid directory. Must be within 'settings.sub_dirs'...")
36          sys.exit()
37
38      #Gets the file names of the '.mat' files for the corresponding 'dir' argument
39
40      #Gets the correct path to the necessary directory as specified by the 'dir' argument
41      if dir == "6minwalk-matfiles":
42          dir_path = local_dir + dir + "\\all_data_mat_files\\"
43      elif dir == "NSAA":
44          dir_path = local_dir + dir + "\\matfiles\\"
45      else:
46          dir_path = local_dir + dir + "\\"
47
48      #Gets the full name of the file within the 'dir' directory based on the 'fn' argument; exits the script if ↘
            there
49      #is no matching file containing 'fn'
50      try:
51          full_file_name = [dir_path + fn for fn in os.listdir(dir_path) if fn.endswith(".mat") and args.fn in fn↘
                ][0]
52      except IndexError:
53          print("Second arg ('fn') must be the short name of the file to load within 'dir'...")
54          sys.exit()
55
56      #Loads the data from the given file name and extracts the root 'tree' from the file
57      ad_data = sio.loadmat(full_file_name)
58      #Corresponds to location in matlabfile at: 'tree.subjects.frames.frame'
59      tree = ad_data["tree"]
60
61      #Try-except clause here to catch when 'D6' doesn't have a 'sensorCount' category within 'tree.subject.↘
            frames'
62      print("Extracting data from AD file " + full_file_name + "....")
63      try:
64          frame_data = tree[0][0][6][0][0][10][0][0][3][0]
65      except IndexError:
66          frame_data = tree[0][0][6][0][0][10][0][0][2][0]
67
68      #Gets the types in each column of the matrix (i.e. dtypes of submatrices)
69      col_names = frame_data.dtype.names
70
71      #Extract single outer-list wrapping for vectors and double outer-list values for single values
72      try:
73          frame_data = [[elem[0] if len(elem[0]) != 1 else elem[0][0] for elem in row] for row in frame_data]
74      except IndexError:
75          # Accounts for missing 'contact' values in certain rows of some '.mat' files by ignoring the 'contact' ↘
                column
76          new_frame_data = []
77          for m, row in enumerate(frame_data):
78              # Ignore rows that don't have 'normal' as their 'type' cell
79              if row[3][0] != "normal":
80                  continue
81              row_data = []
82              for i in range(len(row)):
83                  if i == len(row) - 1:
84                      row_data.append(["", ""])
85                  elif len(row[i][0]) != 1:
86                      row_data.append(row[i][0])
87                  else:
88                      row_data.append(row[i][0][0])
89              new_frame_data.append(row_data)
90          frame_data = new_frame_data
91
92      #With the newly extracted data from their wrappers, returns the data as a DataFrame
93      return pd.DataFrame(frame_data, columns=col_names)
94
95
96
97  def display_3d_positions(df):
98      """
99          :param takes in the DataFrame object, 'df', of the 'dir' and 'fn' of the corresponding subject
100         :return no return, but plots position values for the given file object on a 3D dynamic plot,
101         with the necessary components connected, and outputs a dynamic accumulated time in seconds to the ↘
                console:
102     """
103     #Disregard first 3 samples (usually types 'identity', 'tpose' or 'tpose-isb')
104     positions = df.loc[:, "position"].values[3:]
105
106     #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()


#Preprocesses the data to extract the data from the '.mat' file in question and passes it to have its
#position values plotted
display_3d_positions(preprocessing())
```

A.5 'ext_raw_measures.py'

```
import argparse
import sys
import os
import scipy.io as sio
import pandas as pd
from settings import local_dir, sub_dirs, raw_measurements, axis_labels, segment_labels, joint_labels, \
    sensor_labels, measure_to_len_map, seg_join_sens_map


"""Section below encompasses all the required arguments for the script to run. Note that the default behaviour of the
script is to operate on complete files, rather than 'single-act' files produced by the 'mat_act_div.py' script; hence,
the optional '--single_act' argument must be specified if it wants to operate on those files to ensure the correct
```

```
13  files are retrieved."""
14  parser = argparse.ArgumentParser()
15  parser.add_argument("dir", help="Specifies which source directory to use so as to process the files contained within "
16                                  "them accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles', "
17                                  "'NSAA', 'NMB', or 'allmatfiles'.")
18  parser.add_argument("fn", help="Specifies the short name (e.g. 'D11') of the file that we wish to extract the specified "
19                                 "raw measurements. Specify 'all' for all the files available in the 'local_dir'.")
20  parser.add_argument("measurements", help="Specifies the measurements to extract from the source .mat file. Separate "
21                                           "each measurement to extract by a comma, or provide 'all' for all measurements.")
22  parser.add_argument("--single_act", type=bool, nargs="?", const=True,
23                      help="Specify if the files to operate on are 'single act' files.")
24  parser.add_argument("--single_act_concat", type=bool, nargs="?", const=True,
25                      help="Specify if the files to operate on are 'single act concat' files.")
26  args = parser.parse_args()
27
28
29  #Sets 'local_dir' to the correct directory name, based on the argument passed in for 'dir' and whether or not the
30  #optional arguments '--single_act' or '--single_act_concat' were set or not.
31  if args.dir + "\\" in sub_dirs:
32      if args.dir == "6minwalk-matfiles":
33          local_dir += args.dir + "\\all_data_mat_files\\"
34      elif args.dir == "6MW-matFiles" or args.dir == "allmatfiles" or args.dir == "left-out" or args.dir == "NMB":
35          local_dir += args.dir + "\\"
36      else:
37          local_dir += args.dir + "\\matfiles\\"
38          if args.single_act:
39              local_dir += "act_files\\"
40          elif args.single_act_concat:
41              local_dir += "act_files_concat\\"
42  else:
43      print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
44            "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'NMB', or 'allmatfiles'.")
45      sys.exit()
46
47  #Gets the names of all the files within the 'local_dir' directory and filters them to a list of one element if
48  #the 'fn' argument corresponds to a file within this, or a list of all '.mat' files within this directory if the 'fn'
49  #argument is 'all'
50  file_names = os.listdir(local_dir)
51  if any(args.fn in fn for fn in file_names):
52      full_file_names = [local_dir + [fn for fn in file_names if args.fn in fn][0]]
53  elif args.fn == "all":
54      full_file_names = [local_dir + file_name for file_name in file_names if file_name.endswith(".mat")]
55      #Filters out the 'AllTasks' files from 'allmatfiles' for space reasons
56      if args.dir == "allmatfiles":
57          full_file_names = [ffn for ffn in full_file_names if "AllTasks" not in ffn and "ttest" not in ffn]
58  else:
59      print("Second arg ('fn') must be the short name of a file (e.g. 'D2' or 'all') within", local_dir)
60      sys.exit()
61
62
63  #Sets measures to all possible measurement names if the argument given is 'all', or get all parts of the 'measurements'
64  #argument, splits it up by commas, and adds the measurement names to a list. E.g., if the script was run as
65  #'python ext_raw_measures.py NSAA all position,acceleration,jointAngle', then measures would now contain:
66  #['position', 'acceleration', 'jointAngle']
67  if args.dir == "allmatfiles" and args.measurements != "jointAngle":
68      print("Third arg ('measurements') must be 'jointAngle when 'dir' argument is \'allmatfiles\'")
69      sys.exit()
70  measures = []
71  if args.measurements == "all":
72      measures = raw_measurements
73  else:
74      for measure in args.measurements.split(","):
75          if measure in raw_measurements:
76              measures.append(measure)
77          else:
78              print("'" + measure + "' not a valid 'measurement' name. Must be 'all' or one of:", raw_measurements)
79              sys.exit()
80
81  #For each of the measurements to extract from the source file(s), create a unique subdirectory within 'local_dir'
82  #with a name equal to the measurement name
83  for measure in measures:
84      if not os.path.exists(local_dir + measure):
85          os.mkdir(local_dir + measure)
86
87  #For each of the files that we wish to extract the raw measurements of (given as a short file name in 'fn' or all
88  #available filenames if 'fn' is set as 'all'...
89  for full_file_name in full_file_names:
90      print("\nExtracting", measures, "from '" + full_file_name + "'...")
91      #Loads the file given the file name and extracts the table data from within the '.mat' structure
92      mat_file = sio.loadmat(full_file_name)
93      if not args.dir == "allmatfiles":
94          tree = mat_file["tree"]
```

```
95          try:
96              frame_data = tree[0][0][6][0][0][10][0][0][3][0]
97          except IndexError:
98              try:
99                  frame_data = tree[0][0][6][0][0][10][0][0][2][0]
100             except IndexError:
101                 frame_data = tree[0][0][6][0][0][9][0][0][2][0]
102         #Gets the names of each of the columns within
103         col_names = frame_data.dtype.names
104         # Extract single outer-list wrapping for vectors and double outer-list values for single values
105         try:
106             frame_data = [[elem[0] if len(elem[0]) != 1 else elem[0][0] for elem in row] for row in frame_data]
107         except IndexError:
108             # Accounts for missing 'contact' values in certain rows of some '.mat' files by ignoring the '↘
                  contact' column
109             new_frame_data = []
110             for m, row in enumerate(frame_data):
111                 # Ignore rows that don't have 'normal' as their 'type' cell
112                 if row[3][0] != "normal":
113                     continue
114                 row_data = []
115                 for i in range(len(row)):
116                     if i == len(row) - 1:
117                         row_data.append(["", ""])
118                     elif len(row[i][0]) != 1:
119                         row_data.append(row[i][0])
120                     else:
121                         row_data.append(row[i][0][0])
122                 new_frame_data.append(row_data)
123             frame_data = new_frame_data
124             import numpy as np
125             print(np.shape(frame_data))
126             sys.exit()
127     else:
128         frame_data = mat_file["jointangle"]
129         col_names = None
130
131     #Creates a DataFrame from the data extracted from the source '.mat' file in question, skipping the first 3 ↘
          rows
132     #if it's not a 'single-act' file (as these just correspond to 'setup' rows)
133     if args.single_act or args.single_act_concat:
134         df = pd.DataFrame(frame_data, columns=col_names).iloc[:]
135     else:
136         df = pd.DataFrame(frame_data, columns=col_names).iloc[3:]
137
138     #For each measurement to extract from the file, gets a list of names of features for that measurement (23 ↘
          segments
139     #labels, #22 joint labels, or 17 sensor labels), create a list of column names for each column of extracted
140     #data from the file, gets the necessary columns from the DataFrame corresponding to the measurement in ↘
          question,
141     #creates a new DataFrame from this with the index names being the short-file name (e.g. 'D11'), and writes ↘
          this
142     #to a .csv file within 'local_dir' with a name corresponding to its source file name and extracted ↘
          measurement
143     for measure in measures:
144         measurement_names = seg_join_sens_map[measure_to_len_map[measure]]
145         headers = ["(" + measurement_name + ") : (" + axis + "-axis)"
146                    for measurement_name in measurement_names for axis in axis_labels]
147         if args.dir == "allmatfiles":
148             measure_data = frame_data
149         else:
150             try:
151                 measure_data = [list(data) for data in df.loc[:, measure].values]
152             #Accounts for files where we expect to see more measurements than their exists within the '.mat' ↘
                  file (i.e.
153             #in an 'AD' file where we expect all measurements but instead only contains 'jointAngle' and
154             #'jointAngleXZY' measurements
155             except KeyError:
156                 print("Measurement '" + measure + "' that we expect to see within '" + full_file_name +
157                       "' are not present, skipping this measurement...")
158                 continue
159         if full_file_name.split("\\")[-1].startswith("All"):
160             short_file_name = full_file_name.split("\\")[-1].split("-")[1]
161         else:
162             short_file_name = full_file_name.split("\\")[-1].split("-")[0]
163         measure_df = pd.DataFrame(measure_data, index=[short_file_name for i in range(len(measure_data))])
164
165         new_file_name = local_dir + measure + "\\" + full_file_name.split("\\")[-1].split(".mat")[0] + \
166                         "_" + measure + ".csv"
167         #If file already exists, remove it in preparation for new file being written
168         if os.path.exists(new_file_name):
169             os.remove(new_file_name)
170         print("Writing '" + new_file_name + "' to '" + local_dir + measure + "\\'")
171         measure_df.to_csv(new_file_name, header=headers)
```

A.6 'file_mover.py'

```
1  from settings import local_dir, sub_dirs
2  import argparse
3  import shutil
```

```
import sys
import os


parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specify this as the directory within 'local_dir' in which to place the file.")
parser.add_argument("file_path", help="The path to the file in question which the user wishes to move to 'local_dir'. "
                                      "Either specify the local path from 'source' or the complete path.")
args = parser.parse_args()

#Ensures that the 'dir' argument is one of the allowed values
if args.dir + "\\" in sub_dirs:
    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', 'NMB', or 'direct_csv'.")
    sys.exit()

#Modifies the local copy of 'local_dir' to point to the correct subdirectory of the selected directory within
#'local_dir', dependent on the directory itself
if dir == "6minwalk-matfiles":
    local_dir_copy = local_dir + "6minwalk-matfiles\\all_data_mat_files\\"
elif dir == "NSAA":
    local_dir_copy = local_dir + "NSAA\\matfiles\\"
else:
    local_dir_copy = local_dir + dir

#If the user does not have the 'local directory' setup, creates one for them to contain this sole file so it can be
#correctly accessed by the data pipeline
if not os.path.exists(local_dir_copy):
    print("Creating the '" + local_dir_copy + "' directory to contain '" + args.file_path + "'...")
    os.makedirs(local_dir_copy)
if not os.path.exists(local_dir + "output_files"):
    print("Creating the output directory within '" + local_dir + "'...")
    os.makedirs(local_dir + "output_files")


#Attempts to copy the chosen file (given it's complete or relative path) to the chosen subdirectory of 'local_dir'
try:
    shutil.copy(args.file_path, local_dir_copy)
except FileNotFoundError:
    print("'" + args.file_path + "' not found as a source file...")
    sys.exit()
```

A.7 'file_renamer.py'

```
import argparse
import os
import sys
from re import sub
from settings import local_dir, sub_dirs


parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies the directory that we wish to use to source the file names that we wish to "
                                "rename. Must be one of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', "
                                "NMB, or 'allmatfiles'.")
args = parser.parse_args()

dir = args.dir



#Checks that the 'dir' argument is one of the allowed values and exits if it isn't
if dir + "\\" not in sub_dirs:
    print("First arg ('dir') must the name of the source directory from which we wish to transform the file names and "
          "must be one of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'allmatfiles', or 'NMB'.")
    sys.exit()

#Gets the directory within 'dir' that contains the files of which we wish to change the name
if dir == "NSAA":
    local_dir += dir + "\\matfiles\\"
elif dir == "allmatfiles" or dir == "6MW-matFiles" or dir == "NMB":
    local_dir += dir + "\\"
else:
    local_dir += dir + "\\all_data_mat_files\\"

#Filters out the non-mat files from the file names (i.e. so we only change the names of .mat files in 'local_dir)
file_names = [f for f in os.listdir(local_dir) if f.endswith(".mat")]

#Removes certain files within 'allmatfiles' that we don't want to keep, including any containing 'AllTasks' or 'Session'
files_to_delete, files_kept = [], []
if dir == "allmatfiles":
```

```
38      for f in file_names:
39          if "jointangle" not in f or "AllTasks" in f or "Session" in f:
40              files_to_delete.append(f)
41          else:
42              files_kept.append(f)
43  else:
44      files_kept = file_names
45
46  #Creates new 'target' file names for each of the old file names based on sets of regexes for each 'dir'
47  new_file_names = []
48  if dir == "NSAA":
49      for fn in files_kept:
50          #Capitalizes files that start with 'd' to start with 'D'
51          f = sub("^d", "D", fn)
52          #Capitalizes the 'NSAA' part of the file
53          f = sub("(nsaa|nsaa1|NSA1|NSA|NSAA|NSAAX)\.mat", "NSAA1.mat", f)
54          #Captures the 'nsaa1a', 'nsaa1b', 'nsaa2a', and 'nsaa2b' cases
55          f = sub("nsaa", "NSAA", f)
56          #Gets rid of any 'xxx' within a 'NSAA' sequence
57          f = sub("xxx", "1", f)
58          #Handles the misspelled 'HC7' and 'HC3' files
59          f = sub("7l", "7", f)
60          f = sub("03", "3", f)
61          f = sub("(nsaa2|NSA2)", "NSAA2", f)
62          # Capitalizes any 'version' nums (e.g. 'Dv2' goes to 'DV2')
63          f = sub("v", "V", f)
64          new_file_names.append(f)
65  elif dir == "6minwalk-matfiles":
66      for fn in files_kept:
67          f = sub("All-", "", fn)
68          f = sub("^d", "D", f)
69          # Removes extra string parts after '6MinWalk' or '6MW' (e.g. '6MinWalk-SensorDropped.mat' becomes '6↘
                MinWalk.mat')
70          f = sub("6MinWalk.*\.mat", "6MinWalk.mat", f)
71          new_file_names.append(f)
72  elif dir =="6MW-matFiles":
73      for fn in files_kept:
74          f = sub("All-", "", fn)
75          f = sub("^d", "D", f)
76          #Removes extra string parts after '6MinWalk' or '6MW' (e.g. '6MinWalk-SensorDropped.mat' becomes '6↘
                MinWalk.mat')
77          f = sub("6MinWalk.*\.mat", "6MinWalk.mat", f)
78          f = sub("6MWT.*\.mat", "6MW.mat", f)
79          f = sub("6MW.mat", "6MinWalk.mat", f)
80          new_file_names.append(f)
81  elif dir == "NMB":
82      for fn in files_kept:
83          f = sub("^d", "D", fn)
84          f = sub("^hc", "HC", f)
85          f = sub("^HC ", "HC", f)
86          f = sub("^HC0", "HC", f)
87          f = sub("7l", "7", f)
88          f = sub("v2", "V2", f)
89          f = sub("\)\.mat", ".mat", f)
90          f = sub("- \(", "-", f)
91          f = sub(" \(", "-", f)
92          f = f.split("-")[0] + "-" + '{0}'.format(f.split("-")[1].split(".")[0]).zfill(3) + ".mat"
93          new_file_names.append(f)
94  else:
95      for fn in files_kept:
96          #File names containing 'jointangle' has the beginning of each chopped off if it doesn't start with '↘
                jointangle'
97          f = sub(".*jointangle", "jointangle", fn)
98          #Capitalize the beginning of short file names within the file name
99          f = sub("jointangled", "jointangleD", f)
100         #If the file is a 6 min walk file, make sure it ends with '6MinWalk.mat' and not '6MW.mat'
101         f = sub("6MW\.mat", "6MinWalk.mat", f)
102         #Replaces spaces with dashes (e.g. 'HC10 (15)' becomes 'HC10-(15)'
103         f = sub(" ", "-", f)
104         #Removes enclosing parenthesis around numbers (e.g. 'HC10-(15)' becomes 'HC10-15')
105         f = sub("[()]", "", f)
106         #Removes dashes after 'HC' and before the number of the short file name (e.g. 'HC-6' becomes 'HC6')
107         f = sub("HC-", "HC", f)
108         #Adds '.mat' to any files that are missing it
109         f = sub("k$", "k.mat", f)
110         new_file_names.append(f)
111
112
113 #Removes any of the files within 'allmatfiles' that we don't want to keep (i.e. discards 'AllTasks' or 'New-↘
      Session' files)
114 if files_to_delete:
115     print(files_to_delete)
116     choice = input("Delete all above files?")
117     if choice == "y":
118         for file in files_to_delete:
119             os.remove(local_dir + file)
120
121
122 #Takes the new file names that have been created in the predefined format and replaces the original file names ↘
      with them
123 for nfn, fn in zip(new_file_names, files_kept):
124     try:
125         os.rename(local_dir + fn, local_dir + nfn)
126     except FileExistsError:
```

```
127         os.remove(local_dir + fn)
```

A.8 'ft_sel_red.py'

```
1   import pandas as pd
2   import numpy as np
3   import os
4   import sys
5   import argparse
6   from sklearn.preprocessing import normalize
7   from sklearn.decomposition import PCA
8   from sklearn.random_projection import GaussianRandomProjection
9   from sklearn.cluster import FeatureAgglomeration
10  from sklearn.feature_selection import VarianceThreshold
11  from sklearn.ensemble import RandomForestClassifier
12  from sklearn.feature_selection import SelectFromModel
13  from settings import local_dir, source_dir, sub_dirs, sub_sub_dirs, nsaa_6mw_path
14
15
16
17  def ft_red_select(x, y, choice, no_normalize, dis_kept_features, num_features=30):
18      """
19      :param 'full_file_name', which is the full path name to the file in question that we wish to do ↘
            dimensionality
20      reduction on
21      :return: the new reduced 'x' and 'y' components of the file to be later written to a new file
22      """
23
24      #Normalize the data
25      if not no_normalize:
26          x = normalize(x)
27
28      #Given the argument choice of feature selection/reduction, creates the relevant object, fits the 'x' data ↘
            to it,
29      #and reduces/transforms it to a lower dimensionality
30      new_x = []
31      print("Original 'x' shape:", np.shape(x))
32      if choice == "pca":
33          pca = PCA(n_components=num_features)
34          new_x = pca.fit_transform(x)
35          print("Explained variance = " + str(round(sum(pca.explained_variance_)*100, 2)) + "%" )
36      elif choice == "grp":
37          grp = GaussianRandomProjection(n_components=num_features)
38          new_x = grp.fit_transform(x)
39      elif choice == "agglom":
40          agg = FeatureAgglomeration(n_clusters=num_features)
41          new_x = agg.fit_transform(x)
42      elif choice == "thresh":
43          #Below threshold gives ~26 components upon application
44          vt = VarianceThreshold(threshold=0.00015)
45          new_x = vt.fit_transform(x)
46          print("Explained variance = " + str(round(sum(vt.variances_)*100, 2)) + "%")
47          kept_features = list(vt.get_support(indices=True))
48          if dis_kept_features:
49              print("Kept features: ")
50              for i in kept_features:
51                  print(col_names[i])
52      elif choice == "rf":
53          y_labels = [1 if s == "D" else 0 for s in y[:, 1]]
54          clf = RandomForestClassifier(n_estimators=10000, random_state=0, n_jobs=-1)
55          print("Fitting RF model....")
56          clf.fit(x, y_labels)
57          sfm = SelectFromModel(clf, threshold=-np.inf, max_features=num_features)
58          print("Selecting best features from model...")
59          sfm.fit(x, y_labels)
60          kept_features = list(sfm.get_support(indices=True))
61          if dis_kept_features:
62              print("Kept features: ")
63              for i in kept_features:
64                  print(col_names[i])
65          new_x = x[:, kept_features]
66
67      print("Reduced 'x' shape:", np.shape(new_x))
68      return new_x, y
69
70
71
72  def add_nsaa_scores(file_df, batch):
73      """
74      :param 'file_df', which contains the values in a 2D numpy array, to have the values NSAA scores appended on↘
            each
75      of its rows
76      :return: the same data as before, but with the overall and individual NSAA scores appended at the beginning↘
            of
77      each row of the data
78      """
79      #To make sure that accepted parameter is as a DataFrame
80      file_df = pd.DataFrame(file_df)
81
82      #For the table of data that we have on the subjects, load in the table, find the columns with ID and
```

```
    #overall NSAA scores, and create a dictionary of matching values, e.g. {'D4': 15, 'D11: 28,...}, with all values
    #from each table

    new_nsaa_6mw_path = "..\\" + nsaa_6mw_path if batch else nsaa_6mw_path
    nsaa_6mw_tab = pd.read_excel(new_nsaa_6mw_path)
    nsaa_6mw_cols = nsaa_6mw_tab[["ID", "NSAA"]]
    nsaa_overall_dict = dict(pd.Series(nsaa_6mw_cols.NSAA.values, index=nsaa_6mw_cols.ID).to_dict())

    # If the first column's names begins with "jointangle", remove this part
    if file_df.iloc[0, 0].startswith("jointangle"):
        for i, row in file_df.iterrows():
            row[0] = row[0].split("jointangle")[1]

    #Adds column of overall NSAA scores at position 0 of every row of the data values, with the NSAA score being
    #appended determined by the short file name of the data as found at the beginning of each row of the data
    nss = [nsaa_overall_dict[i.upper()[:-2] if i.upper().endswith("V2") else 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)

    #Loads the data that contains information about single act NSAA scores from the .xlsx file, extracts the
    #file names and single-acts columns, and creates a list of label names (i.e. the names of the activities) and a
    #dictionary that maps the label names to a list of single-act scores
    nsaa_single_dict = {}
    for name, acts in zip(nsaa_6mw_tab.loc[:, "ID"].values, nsaa_6mw_tab.iloc[:, 5:].values):
        if not any(np.isnan(acts)):
            nsaa_single_dict[name] = acts
    nsaa_act_labels = nsaa_6mw_tab.columns.values[5:]

    #For each label name and for every row, adds the score that is found in the single-acts dictionary for the relevant
    #activity for a given short file name (if it isn't found in the dictionary, add a '2' as we're assuming it's a
    #healthy control patient), add these together, and insert each new row of values at the beginning of the old rows
    #so each now have the additional single-act scores and overall NSAA scores at the beginning of each row and return it
    label_sample_map = []
    for i in range(len(nsaa_act_labels)):
        inner = []
        for j in range(len(file_df.index)):
            fn = file_df.iloc[j, 1].split("_")[0].upper()
            fn = fn[:-2] if fn.endswith("V2") else fn
            if fn in nsaa_single_dict:
                inner.append(nsaa_single_dict[fn][i])
            elif fn.startswith("HC"):
                inner.append(2)
            else:
                # If patient isn't found in the table (and thus we don't have info on the individual NSAA scores),
                # don't continue with the file and move onto the next one
                raise KeyError
        label_sample_map.append(inner)
    for i in range(len(nsaa_act_labels)):
        file_df.insert(loc=(i + 1), column=nsaa_act_labels[i], value=label_sample_map[i])
    return file_df




def main():
    """Section below encompasses all the arguments that are required to setup and train the model. This includes the name
    of the directory to use to load the file(s) to reduce the dimensions of, the type of file that we are expected to
    be dealing with, the short name of the file to reduce the dimensions of", and the choice of feature selection/reduction
    technique to use to process the file. The script also normalizes the data by default, so specify '--no_normalize' if
    this isn't desired, along with '--dis_kept_features' to print the features selected and '--num_features' to display
    the number of features chosen by the script."""
    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', "
                                    "'NMB', 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.")
```

```
158     parser.add_argument("--num_features", type=int,
159                         help="Specify an 'int' here to define the features to reduce the data to.")
160     parser.add_argument("--dis_kept_features", type=bool, nargs="?", const=True, default=False,
161                         help="Specify this if user wishes to print to console the kept features after a feature selection "
162                              "choice is made. Has no impact if an unsupervised feature reduction choice is made (e.g. "
163                              "'pca', 'grp', or 'agglom').")
164     parser.add_argument("--combine_files", type=bool, nargs="?", const=True, default=False,
165                         help="Specify this to concatenate all the files before reducing them, to then reduce the features "
166                              "on this combined set before separating them to write them to output files as standard. "
167                              "Ensures that all files are reduced in the same way.")
168     parser.add_argument("--single_act", type=bool, nargs="?", const=True, default=False,
169                         help="Specify this if the files to reduce the dimensions of are single-act files (located in a "
170                              "different sub-directory for the given 'dir').")
171     parser.add_argument("--single_act_concat", type=bool, nargs="?", const=True, default=False,
172                         help="Specify this if the files to reduce the dimensions of are single-act concat files (located "
173                              "in a different sub-directory for the given 'dir'.")
174     parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
175                         help="Option that is only set if the script is run from a batch file to access the external files "
176                              "in a correct way.")
177     parser.add_argument("--new_subject", type=bool, nargs="?", const=True, default=False,
178                         help="Specify this if the we wish to treat the subject, 'fn', as one with no 'true' y-labels. "
179                              "Hence, specify this if the subject does not have 'true' values recorded by specialists that "
180                              "is stored in the 'nsaa_6mw_info.xlsx' file (e.g. if the subject is a new subject with no "
181                              "true labels or if we otherwise wish to treat them as such).")
182     args = parser.parse_args()
183
184     choices = ["pca", "grp", "agglom", "thresh", "rf"]
185     global source_dir
186     #Appends the sub_dir name to 'source_dir' if it's one of the allowed names
187     if args.dir + "\\" in sub_dirs:
188         source_dir += args.dir + "\\"
189     else:
190         print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
191               "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'NMB', or 'direct_csv'.")
192         sys.exit()
193
194     #Appends the sub_sub_dir name to 'source_dir' if it's an allowed name, along with appending 'act_files\\' if it's 'NSAA'
195     if args.ft.upper() + "\\" in sub_sub_dirs:
196         source_dir += args.ft + "\\"
197     else:
198         print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
199               "\'JA', or \'DC\'.")
200         sys.exit()
201
202     batch = True if args.batch else False
203
204
205     #Find the matching full file name in 'source_dir' given the 'fn' argument, otherwise use all file names in 'source_dir'
206     #if 'fn' is 'all'
207     match_fns = [s for s in os.listdir(source_dir) if s.split(".")[0].split("_")[1].upper() == args.fn.upper()]
208     if match_fns:
209         full_file_names = [source_dir + match_fns[0]]
210     else:
211         if args.fn == "all":
212             if args.single_act:
213                 source_dir += "act_files\\"
214             elif args.single_act_concat:
215                 source_dir += "act_files_concat\\"
216             match_fns = [s for s in os.listdir(source_dir)]
217             full_file_names = [source_dir + match_fns[i] for i in range(len(match_fns)) if "FR_" not in match_fns[i]
218                                and match_fns[i].endswith(".csv")]
219         else:
220             print("Third arg ('fn') must be the short name of a file (e.g. 'D2' or 'all') within", source_dir)
221             sys.exit()
222
223     #Sets 'choice' equal to the 'choice' argument if it's one of the allowed feature selection/reduction techniques
224     #(e.g. pca, thresh, rf, etc.)
225     if args.choice in choices:
226         choice = args.choice
227     else:
228         print("Fourth arg ('choice') must be the name a feature selection/reduction and must be one of 'pca' (to carry "
229               "out PCA on the file), 'grp' (to reduce dimensionality through a Gaussian random projection), 'agglom' "
230               "(to carry out feature agglomeration), 'thresh' (to do features selection based on a minimal variance "
231               "threshold for each features), and 'rf' (to fit the data to a random forest and use this to select "
232               "the most useful features).")
```

```
233             sys.exit()
234
235
236
237         #Sets the scope of variables set within the 'if args.combine_files:' statement
238         x_combine_files, y_combine_files = None, None
239         empty_arrays = True
240         file_lens = []
241
242         #For each of the full file names of the files that we wish to reduce the dimensions of, get the new 'x' and↘
                'y'
243         #components of their reduced form, concatenate them together, add the overall and single-act NSAA scores if↘
                possible
244         #to do so (otherwise, don't continue with this file), and write this to the same directory it was sourced ↘
                from as a
245         #.csv file with the same name except with a 'FR_' at the front of the file name. If the 'args.↘
                commbine_files' argument
246         #was set, however, just concatenate all the data along axis 0 to prepare for further processing
247
248         for full_file_name in full_file_names:
249             print("\nExtracting data of " + full_file_name + "...")
250             df = pd.read_csv(full_file_name)
251             col_names = df.columns.values[2:]
252             # Splits the loaded file into the 'y' parts (the original .mat source file column and file label) and '↘
                    x' parts (all
253             # the statistical values extracted via 'comp_stat_vals.py')
254             x = df.iloc[:, 2:].values
255             y = df.iloc[:, :2].values
256             if not args.combine_files:
257                 print("Reducing dims of " + full_file_name + "...")
258                 try:
259                     new_x, new_y = ft_red_select(x, y, choice, args.no_normalize, args.dis_kept_features, args.↘
                            num_features) \
260                         if args.num_features else ft_red_select(x, y, choice, args.no_normalize, args.↘
                            dis_kept_features)
261                 except ValueError:
262                     print("'" + full_file_name + "': number of rows too few for number of features, skipping...")
263                     continue
264                 #Recombine the now-reduced 'x' data with the source file name and label columns
265                 new_df = pd.DataFrame(np.concatenate((new_y, new_x), axis=1))
266
267                 #Add a column of NSAA scores to the DataFrame by referencing the external .csvs
268                 if not args.new_subject:
269                     try:
270                         new_df_nsaa = add_nsaa_scores(new_df, batch)
271                     except KeyError:
272                         print(full_file_name + " not found as entry in either 'nsaa_6mw_info', skipping...")
273                         continue
274                 else:
275                     new_df_nsaa = new_df
276                 #Writes the new data to the same directory as before with the same name except with 'FR_' on the ↘
                        front
277                 split_full_file_name = full_file_name.split("\\")
278                 split_full_file_name[-1] = "FR_" + split_full_file_name[-1]
279                 new_full_file_name = "\\".join(split_full_file_name)
280                 if os.path.exists(new_full_file_name):
281                     os.remove(new_full_file_name)
282                 new_df_nsaa.to_csv(new_full_file_name)
283             #Creates a new data storage table if it doesn't exist; otherwise, appends to existing table, along with↘
                    adding the
284             #length of the file to the list of file lengths used to separate the file data after dimensionality ↘
                    reduction
285             else:
286                 file_lens.append(len(x))
287                 if empty_arrays:
288                     x_combine_files = x
289                     y_combine_files = y
290                     empty_arrays = False
291                 else:
292                     x_combine_files = np.append(x_combine_files, x, axis=0)
293                     y_combine_files = np.append(y_combine_files, y, axis=0)
294
295
296         #If we're combining the files for dimensionality reduction, take the concatenated data table of all the ↘
                files, reduce
297         #the dimensions using the chosen technique and, for each of the source file names and their corresponding ↘
                file lengths,
298         #extract the first 'file_len' rows from the table, remove these rows from the table, and write these rows ↘
                to a new
299         #file name with 'FRC_' appended at the front
300         if args.combine_files:
301             print("\nReducing dims of all files...\n")
302             x_total, y_total = ft_red_select(x_combine_files, y_combine_files, choice, args.no_normalize, args.↘
                    dis_kept_features, args.num_features) \
303                 if args.num_features else ft_red_select(x_combine_files, y_combine_files, choice, args.no_normalize↘
                        , args.dis_kept_features)
304             for full_file_name, file_len in zip(full_file_names, file_lens):
305                 new_x, new_y = x_total[:file_len], y_total[:file_len]
306                 x_total, y_total = x_total[file_len:], y_total[file_len:]
307                 #Recombine the now-reduced 'x' data with the source file name and label columns
308                 new_df = pd.DataFrame(np.concatenate((new_y, new_x), axis=1))
309                 #Add a column of NSAA scores to the DataFrame by referencing the external .csvs
310                 if not args.new_subject:
311                     try:
```

```
312                     new_df_nsaa = add_nsaa_scores(new_df, batch)
313                 except KeyError:
314                     print(full_file_name + " not found as entry in either 'nsaa_6mw_info', skipping...")
315                     continue
316             else:
317                 new_df_nsaa = new_df
318 
319             #Writes the new data to the same directory as before with the same name except with 'FRC_' on the front
320             split_full_file_name = full_file_name.split("\\")
321             split_full_file_name[-1] = "FRC_" + split_full_file_name[-1]
322             new_full_file_name = "\\".join(split_full_file_name)
323             if os.path.exists(new_full_file_name):
324                 os.remove(new_full_file_name)
325             new_df_nsaa.to_csv(new_full_file_name)
326 
327 
328 if __name__ == "__main__":
329     main()
```

A.9 'graph_creator.py'

```
1  import argparse
2  import pandas as pd
3  import sys
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from settings import results_path, local_dir, model_pred_path
7  from sklearn.metrics import mean_absolute_error
8  
9  parser = argparse.ArgumentParser()
10 parser.add_argument("choice",
11                     help="Choice of graph creator mode. Must be one of: 'trues_preds' to call the 'plot_trues_preds()' "
12                          "function, 'model_preds_altdirs' to call the 'plot_model_preds_altdirs()' function, "
13                          "'model_preds_trues_preds' for 'plot_model_preds_trues_preds()', 'model_preds_single_acts' "
14                          "for 'plot_model_preds_single_acts()', or 'rnn_results' to call the 'plot_rnn_results() function.")
15 parser.add_argument("arg_one", nargs="?", default=None,
16                     help="Arg based on 'choice' that is specified. If choice='trues_preds', 'arg_one' is the name of "
17                          "the file to load from 'RNN_outputs' (not inc. file extension) to plot the trues and preds "
18                          "from. If choice='model_preds_altdirs', 'arg_one' is the name of the source directory to use "
19                          "for the rows to be selected from 'model_predictions.csv'. If choice='model_preds_trues_preds', "
20                          "'arg_one' is the row number within 'model_predictions.csv' to start from to draw the true and "
21                          "predicted values from (inclusive). If choice='model_preds_single_acts', 'arg_one' is the short "
22                          "file name of the file of which to load the 'single act' rows. If choice='rnn_results', "
23                          "'arg_one' is the start experiment number to load from 'RNN Results.xlsx'.")
24 parser.add_argument("arg_two", nargs="?", default=None,
25                     help="Arg based on 'choice' that is specified. If choice='model_preds_altdirs', 'arg_two' is the "
26                          "name of the model trained dirs to use for the rows to be selected from 'model_predictions.csv' "
27                          "(comma separated). If choice='model_preds_trues_preds', 'arg_two' is the row number within "
28                          "'model_predictions.csv' to end from to draw the true and predicted values from (inclusive). "
29                          "If choice='rnn_results', 'arg_two' is the end experiment number to load from 'RNN Results.xlsx.")
30 parser.add_argument("arg_three", nargs="?", default=None,
31                     help="Only used when choice='model_preds_altdirs' to select only the rows where 'Measurements "
32                          "tested' are the same as 'arg_three'.")
33 parser.add_argument("out_type", nargs="?", default=None,
34                     help="Only used when choice='rnn_results'. Name of metric of which we wish to plot the results."
35                          " Must be one of 'acc', 'mse', 'mae', 'rmse', 'r2', 'ind_act', or 'all_act'.")
36 parser.add_argument("xaxis", nargs="?", default=None,
37                     help="Only used when choice='rnn_results'. Values to be plotting on the x-axis. Must be one of "
38                          "'seq_len' (for plotting sequence length along the x-axis), 'ft' (for plotting the different "
39                          "file types along x-axis), 'seq_over' (for plotting sequence overlaps along the x-axis), "
40                          "or 'features'.")
41 parser.add_argument("--save_img", type=bool, nargs="?", const=True,
42                     help="Save the image to the 'documentation\\Graphs' directory.")
43 parser.add_argument("--no_display", type=bool, nargs="?", const=True,
44                     help="Specify if specifically don't want to open the plot before writing.")
45 parser.add_argument("--split_rows_overlap", type=bool, nargs="?", const=True,
46                     help="Set if wish to divide up the rows to be plotted as 2 separate lines on the plot.")
47 parser.add_argument("--split_rows_disc", type=bool, nargs="?", const=True,
48                     help="Set if wish to divide up the rows to be plotted as 2 separate lines on the plot.")
```

```
parser.add_argument("--x_log", type=bool, nargs="?", const=True,
                    help="Specify if wish to plot x-axis values in log scale (e.g. when plotting very large "
                         "sequence lengths.")
parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
                    help="Option that is only set if the script is run from a batch file to access the external files "
                         "in a correct way.")
args = parser.parse_args()


choices = ["trues_preds", "model_preds_altdirs", "model_preds_trues_preds", "model_preds_single_acts", "rnn_results"]
if args.choice not in choices:
    print("First arg ('arg_one') must be one of 'trues_preds', 'model_preds_altdirs', or 'rnn_results'...")
    sys.exit()



def plot_trues_preds():
    """
        Loads the file, reads in the predicted and true values, plots them, shows it to the user,
        and saves the image to the 'Graphs' directory
    """
    try:
        trues_preds = pd.read_csv(local_dir + "output_files\\RNN_outputs\\" + args.arg_one + ".csv")
        trues, preds = trues_preds["Trues"], trues_preds["Predictions"]
        fig, ax = plt.subplots()
        ax.scatter(trues, preds, alpha=0.03)
        x = np.linspace(*ax.get_xlim())
        ax.plot(x, x)
        plt.title("Plot of true overall NSAA scores against predicted overall NSAA scores")
        plt.xlabel("True overall NSAA scores")
        plt.ylabel("Predicted overall NSAA scores")
        if args.save_img:
            plt.savefig(graphs_path + args.arg_one)
        plt.gcf().set_size_inches(10, 10)
        if not args.no_display:
            plt.show()
    except FileNotFoundError:
        print("First arg ('arg_one') must be the name of a file within 'RNN_outputs' and cannot load '" +
              local_dir + "output_files\\RNN_outputs\\" + args.arg_one + "' ...")
        sys.exit()



def plot_model_preds_altdirs():
    """
        Handles the case where we are graphing with the 'model_predictions.csv' file, i.e. the 'test_altdirs.py' output
    """
    model_preds = pd.read_csv(local_model_pred_path)
    #Select the relevant rows from the file based on the args that we passed into 'graph_creator.py'
    model_preds = model_preds.loc[(model_preds["Source dir"] == args.arg_one) &
                                  (model_preds["Model trained dir(s)"] == str(args.arg_two.split(","))) &
                                  (model_preds["Measurements tested"] == str(args.arg_three.split(",")))]
    model_train_dirs = args.arg_two.split(",")

    #Removes any 'NaN' columns that arise from only zero or one 'alt dirs'
    model_preds = model_preds.dropna(axis=1)


    cols = model_preds.columns.values.tolist()
    #Gets the index positions of the separator columns (i.e. the cols like '------- NSAA Predictions -------')
    model_dir_offsets = [cols.index(c) for c in cols if "-------" in c]
    #Gets the index positions of all the 'true overall NSAA' and 'predict overall NSAA' columns
    tp_col_indices = np.ravel([[off + 8, off + 9] for off in model_dir_offsets]).tolist()
    #Select only the true/pred columns and reassemble them into a dictionary with keys corresponding to model dir names
    overall_tps = model_preds.iloc[:, tp_col_indices].values.T
    overall_tps = [[overall_tps[i], overall_tps[i+1]] for i in range(0, len(overall_tps), 2)]
    overall_tps_dict = {mtd: otp for mtd, otp in zip(model_train_dirs, overall_tps)}

    #Plots true overall NSAA scores against predicted overall NSAA scores
    for i, mtd in enumerate(overall_tps_dict):
        fig, ax = plt.subplots()
        ax.scatter(overall_tps_dict[mtd][0], overall_tps_dict[mtd][1], alpha=0.03)
        x = np.linspace(*ax.get_xlim())
        ax.plot(x, x)
        plt.title("'model_predictions.csv', Source dir = " + args.arg_one + ", Model trained dir(s) = " +
                  args.arg_two.split(",")[i] + "\n" +
                  "Plot of true overall NSAA scores against predicted overall NSAA scores")
        plt.xlabel("True overall NSAA scores")
        plt.ylabel("Predicted overall NSAA scores")
        plt.gcf().set_size_inches(10, 10)
        if args.save_img:
            plt.savefig(graphs_path + "Model_predictions_" + args.arg_one + "_" + args.arg_two + "_" +
                        args.arg_two.split(",")[i] + "_" + args.arg_three + "_true_pred_overall_NSAA")
        if not args.no_display:
            plt.show()
        plt.close()

    #Plots distributions of the percentages of correctly predicted sequence D/HC labels
    for i, md_off in enumerate(model_dir_offsets):
        percents = []
```

```
        for j, row in model_preds.iterrows():
            #Based on the value within the 'True D/HC label' column, pick either the percentage of predicted 'D'
            #sequences or predicted 'HC' sequences, remove the '%', round it to 0 decimal places, and append to
            #percents as an int
            if row[md_off+4] == "D":
                percents.append(int(round(float(row[md_off+6].split("%")[0]))))
            else:
                percents.append(int(round(float(row[md_off+7].split("%")[0]))))
        plt.hist(percents, 100)
        plt.hist(percents, 100, histtype='step', cumulative=True)
        plt.title("Distribution of percentage of correctly predicted sequence D/HC labels over files "
                  "w/ " + args.arg_two.split(",")[i] + " model pred dir")
        plt.xlabel("Percentage of correctly predicted sequence D/HC labels for given file")
        plt.ylabel("Number of files")
        plt.gcf().set_size_inches(10, 10)
        if args.save_img:
            plt.savefig(graphs_path + "Model_predictions_" + args.arg_one + "_" + args.arg_two + "_" +
                        args.arg_two.split(",")[i] + "_" + args.arg_three + "_distrib_seq_labels")
        if not args.no_display:
            plt.show()
        plt.close()

    #Plots distributions of the percentages of correctly predicted single acts for each file
    for i, md_off in enumerate(model_dir_offsets):
        percents = [int(round(float(row[md_off+3].split("%")[0]))) for j, row in model_preds.iterrows()]
        plt.hist(percents, 100)
        plt.hist(percents, 100, histtype='step', cumulative=True)
        plt.title("Distribution of percentage of individual acts correctly predicted over files "
                  "w/ " + args.arg_two.split(",")[i] + " model pred dir")
        plt.xlabel("Percentage of correctly predicted sequence individual acts labels for given file")
        plt.ylabel("Number of files")
        plt.gcf().set_size_inches(10, 10)
        if args.save_img:
            plt.savefig(graphs_path + "Model_predictions_" + args.arg_one + "_" + args.arg_two + "_" +
                        args.arg_two.split(",")[i] + "_" + args.arg_three + "_distrib_perc_indiv_acts")
        if not args.no_display:
            plt.show()
        plt.close()

    #Computes and prints MAE between true and predicted NSAA scores over files
    for i, md_off in enumerate(model_dir_offsets):
        mae = round(mean_absolute_error(model_preds.iloc[:, md_off+8].tolist(),
                                        model_preds.iloc[:, md_off+9].tolist()), 2)
        print("MAE between true and predicted NSAA scores over files (" +
              args.arg_two.split(",")[i] + ") = " + str(mae))

    #Computes and prints percentage of correct predicted file D/HC label
    for i, md_off in enumerate(model_dir_offsets):
        correct_labels = 0
        for j, row in model_preds.iterrows():
            correct_labels += 1 if row[md_off+4] == row[md_off+5] else 0
        print("Percentage of correct predicted file D/HC label (" + args.arg_two.split(",")[i] + ") = " +
              str(round(((correct_labels/len(model_preds))*100), 2)) + "%")

    #Computes and prints MAE of percentage predicted wrong sequence D/HC classification over files
    for i, md_off in enumerate(model_dir_offsets):
        pred_percents = []
        for j, row in model_preds.iterrows():
            if row[md_off+4] == "D":
                pred_percents.append(float(row[md_off+6][:-1]))
            else:
                pred_percents.append(float(row[md_off+7][:-1]))
        mae = round(mean_absolute_error(pred_percents, [100.0 for k in range(len(pred_percents))]), 2)
        print("MAE of percentage predicted wrong sequence D/HC classification over files (" +
              args.arg_two.split(",")[i] + ") = " + str(mae))

    #Computes and prints average percentage of single acts correctly predicted over files
    for i, md_off in enumerate(model_dir_offsets):
        pred_percents = [float(row[md_off+3][:-1]) for j, row in model_preds.iterrows()]
        mean_perc = round(np.mean(pred_percents), 2)
        print("Average percentage of single acts correctly predicted over files (" +
              args.arg_two.split(",")[i] + ") = " + str(mean_perc) + "%")


def plot_model_preds_trues_preds():
    """
        Plots the true and predicted NSAA values for the selected rows (given by 'arg_one' and 'arg_two', inclusive)
        on a graph, with true values plotted on 'x-axis' and predicted values along the 'y-axis'
    """
    model_preds = pd.read_csv(local_model_pred_path)

    try:
        start_row, end_row = int(args.arg_one), int(args.arg_two)
    except ValueError:
        print("First two args ('arg_one' and 'arg_two') must be integers to select the row range "
              "of 'model_predictions.csv...")
        sys.exit()

    model_preds = model_preds.iloc[start_row-2:end_row-1, :]
    trues = model_preds.iloc[:, 6].astype(float).tolist()
    preds = model_preds.iloc[:, 7].astype(float).tolist()
```

```
231         fig, ax = plt.subplots()
232         ax.scatter(trues, preds, alpha=0.3)
233         x = np.linspace(*ax.get_xlim())
234         ax.plot(x, x)
235         plt.title("Plot of true overall NSAA scores against predicted overall NSAA scores")
236         plt.xlabel("True NSAA scores")
237         plt.ylabel("Predicted NSAA scores")
238         if args.save_img:
239             if args.batch:
240                 plt.savefig("..\\..\\documentation\\Graphs\\" + "model_preds_trues_preds_" + args.arg_one + "_" + ↘
                        args.arg_two)
241             else:
242                 plt.savefig(graphs_path + "model_preds_trues_preds_" + args.arg_one + "_" + args.arg_two)
243         plt.gcf().set_size_inches(10, 10)
244         if not args.no_display:
245             plt.show()
246
247
248
249     def plot_model_preds_single_acts():
250         """
251             Plots 3 subplots where the act number (between 1 and 17) are along the x-axis and the results of the 3 ↘
                  output
252             types for each activity is plotted along the y-axis.
253         """
254         model_preds = pd.read_csv(local_model_pred_path)
255         #Select the relevant rows from the file based on the args that we passed into 'graph_creator.py'
256         model_preds = model_preds.loc[(model_preds["Short file name"].str.contains(args.arg_one)) &
257                                       (model_preds["Short file name"].str.contains("\(act")))]
258
259         #Creates a dictionary with the act number as the keys and the value being a single list of 3 values, with ↘
               values of
260         #percent of acts correctly predicted, percent of predicted D/HC (i.e. correct) sequences, and predicted ↘
               overall NSAA
261         act_dict = {line[1].loc["Short file name"].split(" ")[-1][:-1] : [] for line in model_preds.iterrows()}
262         for line in model_preds.iterrows():
263             act_num = line[1].loc["Short file name"].split(" ")[-1][:-1]
264             act_dict[act_num].append(float(line[1].loc["Percent of acts correctly predicted"][:-1]))
265             if line[1].loc["True 'D/HC Label'"] == "D":
266                 act_dict[act_num].append(float(line[1].iloc[11][:-1]))
267             else:
268                 act_dict[act_num].append(float(line[1].iloc[12][:-1]))
269             act_dict[act_num].append(abs(line[1].loc["Predicted 'Overall NSAA Score'"] - line[1].loc["True 'Overall↘
                    NSAA Score'"]))
270
271         #Constructs the subplots for the 3 types of output type metrics for the single act numbers using the
272         #entries in 'act_dict'
273         fig, axes = plt.subplots(nrows=3)
274         sc1 = axes[0].plot([act for act in act_dict], [act_dict[act][0] for act in act_dict])
275         sc2 = axes[1].plot([act for act in act_dict], [act_dict[act][1] for act in act_dict])
276         sc3 = axes[2].plot([act for act in act_dict], [act_dict[act][2] for act in act_dict])
277         axes[0].set(xlabel="Act Number", ylabel="Percent of acts \ncorrectly predicted")
278         axes[0].set_ylim(bottom=0, top=100)
279         axes[1].set(xlabel="Act Number", ylabel="Percent of predicted \ncorrect D/HC sequences")
280         axes[1].set_ylim(bottom=0, top=100)
281         axes[2].set(xlabel="Act Number", ylabel="Absolute error between true\n and predicted overall NSAA score")
282         axes[2].set_ylim(bottom=0, top=34)
283
284         plt.suptitle("Plots of '" + args.arg_one + "' single activities performance for 'acts', 'dhc', and "
285                                                    "'overall' output types")
286         plt.subplots_adjust(hspace=0.5, top=0.95, bottom=0.05)
287         plt.gcf().set_size_inches(18.5, 10.5)
288         if args.save_img:
289             plt.savefig(graphs_path + "Model_predictions_" + args.arg_one + "_single_acts")
290         if not args.no_display:
291             plt.show()
292         plt.close()
293
294
295
296     def plot_rnn_results():
297         out_type_map = {"acc": "Test Accuracy", "mse": "Mean Squared Error", "mae": "Mean Absolute Error",
298                         "rmse": "Root Mean Squared Error", "r2": "R^2 Score",
299                         "ind_act": "Individual Activity Accuracy", "all_act": "All Activities Accuracy"}
300         xaxis_choices = ["seq_len", "ft", "seq_over", "features"]
301
302         #Load the table of RNN results and find the minimum and maximum experiment numbers to use in argument ↘
               validation
303         results_table = pd.read_excel(results_path)
304         results_table_min_max_vals = [int(min(results_table["Experiment Number"].values)),
305                                       int(max(results_table["Experiment Number"].values))]
306
307         #Checks there is a valid starting experiment number within the table
308         if int(args.arg_one) >= results_table_min_max_vals[0]:
309             start_pos = int(args.arg_one)
310         else:
311             print("First arg ('arg_one') must be >= the available experiment numbers...")
312             sys.exit()
313
314         #Checks there is a valid ending experiment number within the table
315         if int(args.arg_two) <= results_table_min_max_vals[1] and int(args.arg_two) > start_pos:
316             end_pos = int(args.arg_two)
317         else:
318             print("Second arg ('arg_two') must be <= the available experiment numbers and > the 'arg_one' arg...")
```

```
319             sys.exit()
320
321         #Ensures that the choice of metric to plot in the graph is a valid choice
322         if args.out_type in out_type_map:
323             choice = args.out_type
324         else:
325             print("Third arg ('out_type') must be one of the available choices for plotting metrics...")
326             sys.exit()
327
328         #Ensures that the choice of x-axis plotting element is a valid choice
329         if args.xaxis in xaxis_choices:
330             xaxis_choice = args.xaxis
331         else:
332             print("Fourth arg ('xaxis') must be one of the available choices for what is plotted along the x-axis↘
                    ...")
333             sys.exit()
334
335         #Extracts the experiment rows that we are concerned with from the table of RNN results
336         start_index = results_table.index[results_table["Experiment Number"] == start_pos].tolist()[0]
337         end_index = results_table.index[results_table["Experiment Number"] == end_pos].tolist()[0]
338         exper_data = results_table.iloc[start_index:end_index+1, :]
339
340         #Appends the measurement names, sequence lengths, and required results (based on the 'out_type' argument) ↘
                from the
341         #relevant cells in each row to lists
342         dirs, measures, seq_lens, seq_overs, results, features = [], [], [], [], [], []
343         for i, row in exper_data.iterrows():
344             dirs.append(row["'rnn' arguments"].split(" ")[2])
345             measures.append(row["'rnn' arguments"].split(" ")[3])
346             seq_lens.append(int(row["'rnn' arguments"].split("--seq_len=")[-1].split(" ")[0]))
347             features.append(int(row[11].split(" ")[-1]))
348             res_names = [j.split(" = ")[0] for j in row["Results"].split(", ")]
349             res_res = [j.split(" = ")[1] for j in row["Results"].split(", ")]
350             added_num = False
351             for j in row["Results"].split(", "):
352                 if out_type_map[choice] == j.split(" = ")[0]:
353                     num = j.split(" = ")[1]
354                     num = float(num[:-1]) if num.endswith("%") else float(num)
355                     results.append(num)
356                     added_num = True
357             if not added_num:
358                 results.append(None)
359             seq_overs.append(None if "seq_overlap" not in row["'rnn' arguments"]
360                              else float(row["'rnn' arguments"].split(" ")[-1].split("=")[-1]))
361
362         if xaxis_choice == "seq_len":
363             #Combine these measurements into a dictionary with each key being a measurement name and each value ↘
                    being a list
364             #of two lists, with each of these sub-lists containing the sequence lengths and required results
365             if args.split_rows_overlap:
366                 measures = np.ravel([["Same overlap" for i in range(len(measures)//2)],
367                                      ["Scaling overlap w/ seq_len" for i in range(len(measures)//2)]])
368             elif args.split_rows_disc:
369                 measures = [[], []]
370                 for i, row in exper_data.iterrows():
371                     if "discard_prop" not in row["'rnn' arguments"]:
372                         measures[0].append("No discard_prop")
373                     else:
374                         measures[1].append("Discard prop")
375                 measures = [elem for inner in measures for elem in inner]
376             df_dict = {measure: [[], []] for measure in measures}
377             for i in range(len(measures)):
378                 df_dict[measures[i]][0].append(seq_lens[i])
379                 df_dict[measures[i]][1].append(results[i])
380             #Creates a line from each entry in the table, with the label being the measurement name, the 'x' values↘
                    being the
381             #sequence lengths, and the 'y' values being the results values
382             for measure in df_dict:
383                 plt.plot(df_dict[measure][0], df_dict[measure][1], label=measure, marker="o")
384             plt.title("Plot of " + str(out_type_map[choice]) + " and sequence length for experiments " +
385                       str(start_pos) + " to " + str(end_pos))
386             plt.xlabel("Sequence length")
387             plt.ylabel(out_type_map[choice])
388             plt.legend()
389
390         elif xaxis_choice == "ft":
391             plt_list = [[], []]
392             for dir, measure, result in zip(dirs, measures, results):
393                 if result:
394                     dir_measure = measure if dirs.count(dirs[0]) == len(dirs) else dir + " : " + measure
395                     plt_list[0].append(dir_measure)
396                     plt_list[1].append(result)
397             plt.plot(plt_list[0], plt_list[1], marker="o")
398             plt.title("Plot of " + str(out_type_map[choice]) + " and dir/file_type for experiments " +
399                       str(start_pos) + " to " + str(end_pos))
400             plt.xlabel("Source directory and file type")
401             plt.xticks(rotation=10, fontsize=5)
402             plt.ylabel(out_type_map[choice])
403
404         elif xaxis_choice == "seq_over":
405             plt.plot(seq_overs, results, marker="o")
406             plt.title("Plot of " + str(out_type_map[choice]) + " and sequence overlap for experiments " +
407                       str(start_pos) + " to " + str(end_pos))
408             plt.xlabel("Sequence overlap")
```

```
409             plt.ylabel(out_type_map[choice])
410
411         elif xaxis_choice == "features":
412             plt.plot(features, results, marker="o")
413             plt.title("Plot of " + str(out_type_map[choice]) + " and number of features for experiments " +
414                       str(start_pos) + " to " + str(end_pos))
415             plt.xlabel("Number of features")
416             plt.ylabel(out_type_map[choice])
417
418         #Sets the scaling of the x-axis to logarithmic if the required argument is set
419         if args.x_log:
420             plt.xscale("log")
421
422         #Only save the plot image to file (with a name based on the rows of data used and metric plotted) if ↘
                argument is specified
423         if args.save_img:
424             file_name = "Exp" + str(start_pos) + "-" + str(end_pos) + "_" + choice
425             plt.savefig(graphs_path + file_name)
426
427         #Display the image to the user if the '--no_display' argument is not given
428         if not args.no_display:
429             plt.gcf().set_size_inches(10, 10)
430             plt.show()
431
432     local_model_pred_path = "..\\" + model_pred_path if args.batch else model_pred_path
433     graphs_path = "..\\..\\documentation\\Graphs\\" if args.batch else "..\\documentation\\Graphs\\"
434
435     if args.choice == "trues_preds":
436         plot_trues_preds()
437     elif args.choice == "model_preds_altdirs":
438         plot_model_preds_altdirs()
439     elif args.choice == "model_preds_trues_preds":
440         plot_model_preds_trues_preds()
441     elif args.choice == "model_preds_single_acts":
442         plot_model_preds_single_acts()
443     elif args.choice == "rnn_results":
444         plot_rnn_results()
```

A.10 'mat_act_div.py'

```
1   #Note: for this to work, we must *first* download the google sheets and place in the 'NSAA' file directory that↘
        is
2   #local to the user. Hence, the user must first download the google sheet at:
3   #'https://docs.google.com/spreadsheets/d/1OvkGU6kwmMxD6zdZqXcNKUvur1uFbAx5IND7_dXibjE', place it in the ↘
        location of
4   #their NSAA directory, and change the global variable 'local_dir' below to reflect the path to their 'NSAA' ↘
        directory
5
6   import argparse
7   import pandas as pd
8   import numpy as np
9   import sys
10  import os
11  import scipy.io as sio
12  from settings import local_dir, nsaa_subtasks_path
13
14  """Section below encompasses all the required arguments for the script to run. Note there are only 2 arguments ↘
        and they
15  are BOTH required."""
16  parser = argparse.ArgumentParser()
17  parser.add_argument("fn", help="Specify the patient ID to split the .mat file of. Specify 'all' for all files ↘
        available.")
18  parser.add_argument("--concat_act_files", type=bool, nargs="?", const=True, default=False,
19                      help="Specify if wish to combine all the files together for a each subject (i.e. stack the ↘
                            tables "
20                           "within the .mat files of each of the activities for a given subject).")
21  args = parser.parse_args()
22
23  local_dir += "NSAA\\"
24
25
26  def extract_act_times():
27      """
28      :return: list of start and end times of activities, as determined by the downloaded Google sheet (these are↘
             of a
29      shape '# of subjects' x '# of total activities (i.e. 17)' x 3 (first being file name containing the ↘
            activity,
30      second being the start time and third being end time); also returns a complete list the subject names of ↘
            which we
31      are extracting the activities
32      """
33      #Reads in the downloaded Google sheet and makes a list of tuples, where each element in the list is a tuple↘
             of two
34      #values: the name of an 'ID' cell and its index location within a different list of IDs
35      data = pd.read_csv(nsaa_subtasks_path, delimiter=";")
36      #Removes the final 12 columns loaded from the table, as these are concerning the '6-min walk' data
37      data = data.iloc[:, :-12]
38      #Drops the columns from the table that we don't need
39      data = data.drop(labels=["AnnotatedBy", "GeneralNotes", "ReferenceVideo", "ReferenceVideoTime",
40                               "RefMVNXFile", "ReferenceMVNXFrame"], axis=1)
```

```
41      #Remove all the 'complete' and 'notes' columns in the table
42      data = data.loc[:, ~(data == "completed").any()]
43      data = data.loc[:, ~(data == "notes").any()]
44
45      patient_ids = data["Patient"].values.tolist()
46      id_pairs = list(zip(patient_ids, np.arange(0, len(patient_ids)+1)))
47      #Removes the first pair in list, as this corresponds to row of table w/ 'filename', 'start', 'finish' etc.
48      id_pairs = id_pairs[1:]
49
50
51      #Gets a list of the first elements of the pairs, i.e. the new list contains just the ID names for the
52      #given version
53      ids = [id[0] for id in id_pairs]
54      if args.fn in ids:
55          #Retrieves the relevant row from the table and only includes the columns containing the activity times
56          act_times = data.iloc[id_pairs[ids.index(args.fn)][1], 1:].values
57          #'Pair up' each source file name, start time, and end time for each activity within 'act_times'; note ↘
                that it
58          #is wrapped in an outer list to make it the same type of list as would be returned if the 'fn' argument↘
                was 'all'
59          act_times_outer = [[[act_times[i], act_times[i+1], act_times[i+2]] for i in range(0, len(act_times), 3)↘
                ]]
60          return act_times_outer, [args.fn]
61      elif args.fn == "all":
62          act_times_outer = []
63          #Repeats the process outlined in the previous 'if' section but for every ID name in the previously
64          #specified list
65          for id in ids:
66              act_times = data.iloc[id_pairs[ids.index(id)][1], 1:].values
67              act_times_outer.append([[act_times[i], act_times[i+1], act_times[i+2]] for i in range(0, len(↘
                    act_times), 3)])
68          return act_times_outer, ids
69      else:
70          print("Second arg ('fn') must be a patient ID within the specified version heading.")
71          sys.exit()
72
73
74
75  def divide_mat_file(act_times_outer, chosen_ids):
76      """
77      :param 'act_times_outer', which contains the list of start and end times of activites for all required ↘
            files,
78      and 'chosen_ids', which isa list of the subject ID names (list of one value if 'fn' is not 'all',
79      else it's all the ID names)
80      :return: no return, but instead divides up each subject's relevant files by the file times specified in the
81      'act_times_outer' argument (i.e. for a given file name and a sequence of start and end times, chops up the
82      .mat file to contain the data from the original of the time period specified by start and end time pairs)
83      """
84
85      #Appends the name of the relevant subdirectory within the 'NSAA' directory to correctly source the .mat ↘
            files
86      global local_dir
87      local_dir += "matfiles\\"
88
89      #Gets the list of files in the 'NSAA\matfiles' directory
90      fns = [fn for fn in os.listdir(local_dir)]
91
92      #Filters down the list of files to split to either a list with one file name if 'fn' is a single name or a ↘
            list
93      #of all .mat files within 'local_dir' to use if 'fn' is 'all'
94      if args.fn != "all":
95          fns = [fn for fn in fns if args.fn in fn]
96      else:
97          fns = [fn for fn in fns if fn.endswith(".mat")]
98
99      if not args.concat_act_files:
100         #Make the relevant subdirectory within 'local_dir' to host the divided-up act files or, if it already ↘
                exists,
101         #then remove all currently-held .mat files within 'act_files'
102         if not os.path.exists(local_dir + "act_files\\"):
103             os.mkdir(local_dir + "act_files\\")
104         else:
105             for fn in os.listdir(local_dir + "act_files\\"):
106                 if fn.endswith(".mat"):
107                     os.remove(local_dir + "act_files\\" + fn)
108     else:
109         if not os.path.exists(local_dir + "act_files_concat\\"):
110             os.mkdir(local_dir + "act_files_concat\\")
111         else:
112             for fn in os.listdir(local_dir + "act_files_concat\\"):
113                 if fn.endswith(".mat"):
114                     os.remove(local_dir + "act_files_concat\\" + fn)
115
116
117
118     #For each of the subjects we are extracting the activities of (either just one if 'fn' is single name or ↘
            numerous if 'fn' is 'all')...
119     for id, act_times in zip(chosen_ids, act_times_outer) :
120         print("\nDividing up '" + id + "' activities...")
121
122         mat_table = None
123         mat_exists = False
124         #For each of the 17 activities that the subject will have performed...
125         for i, act in enumerate(act_times):
```

```
126             #Finds first file beginning with the 'filename' for the activity and subject in question within '↘
                    local_dir'
127             #If it's the second activity ('walking'), search for the file in the two 'walk' source directories
128             walk_dirs = None
129             if i == 1:
130                 #Removes any leading 'All-' from the file name
131                 fn = act[0].split("All-")[1] if act[0].startswith("All-") else act[0]
132                 #Capitalizes the leading 'd' if necessary
133                 fn = fn.split("-")[0].upper() + "-" + "-".join(fn.split("-")[1:])
134                 #Accounts for 'D11b' being 'D11B' instead
135                 fn = fn.split("-")[0][:-1] + "b-" + "-".join(fn.split("-")[1:]) if fn.split("-")[0][-1] == "B" ↘
                        else fn
136                 walk_dirs = ["\\".join(local_dir.split("\\")[:-3]) + "\\6MW-matFiles\\",
137                              "\\".join(local_dir.split("\\")[:-3]) + "\\6minwalk-matfiles\\all_data_mat_files\\↘
                                    "]
138                 #Finds the file name in 'walk_dir' that exactly matches that of the 'filename' column for the
139                 #activity in the table (excluding variations of 'nsaa' in the file name, e.g. 'nsaa1')
140                 file_name = [[walk_dir + f for f in os.listdir(walk_dir) if f.split("-")[0] == fn.split("-")↘
                        [0]]
141                              for walk_dir in walk_dirs]
142             else:
143                 # Capitalizes the leading 'd' if necessary
144                 fn = act[0].split("-")[0].upper() + "-" + "-".join(act[0].split("-")[1:])
145                 #Accounts for the 'HC7l' edge case
146                 fn = fn.split("-")[0][:-1] + "-" + "-".join(fn.split("-")[1:]) if "L" in fn.split("-")[0] else ↘
                        fn
147                 #Accounts for the 'HC03' instead of 'HC3'
148                 fn = fn.split("-")[0][:-2] + fn.split("-")[0][-1] + "-" + "-".join(fn.split("-")[1:]) if fn.↘
                        split("-")[0][-2] == "0" else fn
149                 #Finds the file name in 'local_dir' that closely matches that of the 'filename' column for the
150                 #activity in the table
151                 file_name = [local_dir + f for f in os.listdir(local_dir) if fn.split("-")[0] + "-" in f.split(↘
                        "-")[0] + "-"
152                              and fn.split("-")[1][-2:] in f.split("-")[1][-2:]]
153             try:
154                 if i == 1:
155                     #If the file name doesn't exist in either of the two walking directories, continue to next ↘
                            activity
156                     if all(not walk_dir for walk_dir in walk_dirs):
157                         raise IndexError
158                     else:
159                         #Assuming there's at least 1 walk file for the subject found in one of the directories,
160                         #get the one from '6MW-matFiles', if it exists, otherwise get the one from '6minwalk-↘
                                matfiles'
161                         file_name = [file_name[0][0] if file_name[0] else file_name[1][0]]
162                         file = sio.loadmat(file_name[0])
163                 else:
164                     file = sio.loadmat(file_name[0])
165                 print("\tLoaded activity " + str(i+1) + " for subject '" + id + "' from file '" + file_name[0] ↘
                        + "'...")
166 
167             except IndexError:
168                 print("\tCannot extract activity " + str(i+1) + " for subject '" + id + "', as file '"
169                       + act[0] + "' does not exist in relevant directory, continuing to next activity...")
170                 continue
171 
172             #Gets the start and end point of the activity to extract from within 'file', measured in rows
173             start_time, end_time = int(act[1]), int(act[2])
174 
175             if start_time == -1 or end_time == -1:
176                 print("\tActivity contains '-1' in its 'start' and/or 'finish' times, meaning activity was not ↘
                        "
177                       "performed by the subject...")
178                 continue
179 
180 
181             #Reads in the table of data, cuts out the bit we want for this activity based on 'start_time' and
182             #'end_time', replaces the old table with this one, and writes this new '.mat' file with this ↘
                    smaller table
183             #to a new file based on the source file name and activity number
184             try:
185                 inner_table = file["tree"][0][0][6][0][0][10][0][0][3][0]
186                 new_inner_table = inner_table[start_time:end_time]
187                 if len(new_inner_table) == 0:
188                     print("\tEmpty activity " + str(i+1) + " (i.e. 'end_time' - 'start_time' = 0) in table, ↘
                            skipping...")
189                     continue
190                 np.delete(file["tree"][0][0][6][0][0][10][0][0][3], 0)
191                 file["tree"][0][0][6][0][0][10][0][0][3] = [new_inner_table]
192             except IndexError:
193                 inner_table = file["tree"][0][0][6][0][0][10][0][0][2][0]
194                 new_inner_table = inner_table[start_time:end_time]
195                 if len(new_inner_table) == 0:
196                     print("\tEmpty activity " + str(i+1) + " (i.e. 'end_time' - 'start_time' = 0) in table, ↘
                            skipping...")
197                     continue
198                 np.delete(file["tree"][0][0][6][0][0][10][0][0][2], 0)
199                 file["tree"][0][0][6][0][0][10][0][0][2] = [new_inner_table]
200             if not args.concat_act_files:
201                 new_file_name = str(file_name[0].split("\\")[-1].split(".mat")[0]) + "_act" + str(i + 1) + ".↘
                        mat"
202                 sio.savemat(local_dir + "act_files\\" + new_file_name, file)
203                 print("\tWritten " + new_file_name + " to '" + local_dir + "act_files\\'...")
204             else:
```

```
                if not mat_exists:
                    mat_table = new_inner_table
                    mat_exists = True
                else:
                    try:
                        mat_table = np.append(mat_table, new_inner_table, axis=0)
                    #Accounts for rare problem w/ the combining of tables, specifically for 'D12' act 2
                    except TypeError:
                        print("\tCannot add activity '" + str(i+1) + "' for subject '" + id + "' due to 'TypeError'...")
                        continue
            file["tree"][0][0][6][0][0][10][0][0][2] = [inner_table]
        if args.concat_act_files:
            new_file_name = id + "_concat_acts.mat"
            #Loads a template file that we use to get the formating right, but whose table we replace with 'np.delete()'
            template_file = sio.loadmat(local_dir + [f for f in os.listdir(local_dir) if f.endswith(".mat")][0])
            # Deletes the templates data table and replaces it with the concatenated data
            np.delete(template_file["tree"][0][0][6][0][0][10][0][0][2], 0)
            template_file["tree"][0][0][6][0][0][10][0][0][2] = [mat_table]
            #Saves this concatenated data to a new file
            sio.savemat(local_dir + "act_files_concat\\" + new_file_name, template_file)
            print("\tWritten concat " + new_file_name + " to '" + local_dir + "act_files_concat\\'...")


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

A.11 'model_predictor.py'

```
import argparse
import sys
import os
import pandas as pd
import numpy as np
import tensorflow as tf
from collections import Counter
from matplotlib import pyplot as plt
from settings import local_dir, batch_size, source_dir, output_dir, \
    model_dir, sub_dirs, sub_sub_dirs, file_types, output_types, model_pred_path, model_shapes_path, nsaa_6mw_path
from ft_sel_red import ft_red_select


"""Section below encompasses the three arguments that are required to be passed to the script. This arguments ensures
that the right file is loaded in to be tested with the pre-trained models, along with the arguments themselves informing
the script about which pre-trained model to select to test the file on."""
parser = argparse.ArgumentParser()
parser.add_argument("dir", help="Specifies which source directory the prediction file is contained in so as to process "
                                "the file accordingly. Must be one of '6minwalk-matfiles', '6MW-matFiles', "
                                "'NSAA', 'allmatfiles', or 'left-out'.")
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). Alternatively, supply a name of a "
                               "measurement (e.g. 'position', 'velocity', 'jointAngle', etc.) if the file is being "
                               "predicted is a raw measurement, or multiple raw measurements split by commas if the "
                               "file is an 'AD' file and user wishes to test it on raw measurement models.")
parser.add_argument("fn", help="Specify the short file name of a .csv to be the predictor from 'source_dir'; e.g. for file "
                               "'All_D2_stats_features.csv', enter 'D2'.")
parser.add_argument("--alt_dirs", type=str, nargs="?", const=True, default=False,
                    help="Optional argument to use directory types other than 'dir' to predict from(e.g. if "
                         "'dir'=allmatfiles, can specify '--alt_dirs'=NSAA,6minwalk-matfiles to predict the "
                         "'allmatfile' file on models trained on 'NSAA' and '6minwalk-matfiles' files.")
parser.add_argument("--show_graph", type=bool, nargs="?", const=True, default=False,
                    help="Option to show the 'trues-preds' graph that is created and saved by the script.")
parser.add_argument("--handle_dash", type=bool, nargs="?", const=True, default=False,
                    help="Set this to add a dash ('-') to the beginning of the 'fn' arg; primarily used to get the "
                         "correct short name of files within 'allmatfiles', as 'fn' can't start with a dash.")
parser.add_argument("--file_num", type=str, nargs="?", const=True, default=False,
                    help="Optional arg for 'test_altdirs' to use to write the file number as it appears in its "
                         "source directory.")
parser.add_argument("--use_seen", type=bool, nargs="?", const=True, default=False,
                    help="Specify this if, given a file name, the script should seek out models to use which correspond "
                         "to the given arguments but specifically have seen the corresponding file name before; i.e. the "
                         "models are being assessed on subjects from data it has already seen.")
parser.add_argument("--use_balanced", type=str, nargs="?", const=True, default=False,
```

```
45                      help="Specify this if, given a file name, the script should seek out models to use which ↘
                            correspond "
46                           "to the given arguments but specifically are from a model trained on a 'class balanced↘
                                ' dataset.")
47  parser.add_argument("--single_act", type=int, nargs="?", const=True, default=False,
48                      help="Specify this is intending to use the single-act files that are produced by '↘
                            mat_act_div' to "
49                           "predict with. Only able to be used when 'dir'=NSAA.")
50  parser.add_argument("--single_act_concat", type=str, nargs="?", const=True, default=False,
51                      help="Specify this if intending to use the single-act-concat files that are produced by "
52                           "'mat_act_div' to predict with. Only able to be used when 'dir'=NSAA. If set to '↘
                                src_sac', "
53                           "source files from the 'act_files_concat' directory, whereas if set to 'src_normal', ↘
                                source "
54                           "files from there 'normal' location (i.e. as if the '--single_act_concat' arg wasn't ↘
                                set.")
55  parser.add_argument("--use_frc", type=bool, nargs="?", const=True, default=False,
56                      help="Option to use the 'FRC_' files for 'AD' files instead of 'FR_' files, i.e. use files ↘
                            where "
57                           "dimensionality reduction has been applied the same way over all files rather than on ↘
                                a "
58                           "file-by-file basis.")
59  parser.add_argument("--standardize", type=bool, nargs="?", const=True, default=False,
60                      help="Option to use models that have the '--standardize' argument set. Selects the relevant↘
                            models "
61                           "from the 'rnn_models' directory along with standardizing the file's data.")
62  parser.add_argument("--noise", type=str, nargs="?", const=True, default=False,
63                      help="Option to use models that have the '--noise' argument set, which selects the relevant↘
                            models "
64                           "from the 'rnn_models' directory, but does not add noise to the data now being ↘
                                predicted.")
65  parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
66                      help="Option that is only set if the script is run from a batch file to access the external↘
                            files "
67                           "in a correct way.")
68  parser.add_argument("--use_ft_concat", type=str, nargs="?", const=True, default=False,
69                      help="Specify this is the user wishes to load models that have had file types concatenated ↘
                            "
70                           "together (i.e. where the 'ft' arg to 'rnn.py' was multiple file types separated by ↘
                                commas).")
71  parser.add_argument("--use_indiv", type=bool, nargs="?", const=True, default=False,
72                      help="Specify this if wish to use models that have been trained only on the 'indiv' output ↘
                            type "
73                           "when only using 'single_act' files to test w/ the model.")
74  parser.add_argument("--other_lo", type=str, nargs="?", const=True, default=False,
75                      help="Specify this with a name of other files to have been left out of the model training ↘
                            and "
76                           "testing data set. For example, with 'fn'=D3 and '--other_lo'=D10, search for models ↘
                                that "
77                           "have been built containing '--leave_out=D3,D10' in the title. Separate multiple "
78                           "files by commas.")
79  parser.add_argument("--add_dir", type=str, nargs="?", const=True, default=False,
80                      help="Specify this with the name of another 'dir' if we wish to use models that are trained↘
                            from "
81                           "both 'dir' and '--add_dir' types. For example, if 'dir'=NSAA and '--add_dir_6MW-↘
                                matFiles', "
82                           "ensures that both 'NSAA' and '6MW-matFiles' directory files are used to train the ↘
                                models "
83                           "with which we are concerned.")
84  parser.add_argument("--ft_red", type=str, nargs="?", const=True, default=False,
85                      help="Specify this is using models that have had their raw measurement inputs feature ↘
                            reduced; note "
86                           "that this, like '--use_ft_concat', loads '--pca' models but specifically not models ↘
                                with "
87                           "their input features concatenated.")
88  parser.add_argument("--single_sequence", type=bool, nargs="?", const=True, default=False,
89                      help="Specify this if we only wish to draw a single sequence from the source file(s) for ↘
                            the subject "
90                           "that is drawn evenly across the complete file(s). Note that this is replicated '↘
                                batch_size' "
91                           "times so it can be accepted into the pre-built models.")
92  parser.add_argument("--combine_preds", type=bool, nargs="?", const=True, default=False,
93                      help="Specify this if we wish to combine the predictions made using models built for ↘
                            different "
94                           "output types to get an aggregate prediction for the overall NSAA score.")
95  parser.add_argument("--no_testset", type=bool, nargs="?", const=True, default=False,
96                      help="Specify this to select models that have the train/test ratio set to 0.")
97  parser.add_argument("--new_subject", type=bool, nargs="?", const=True, default=False,
98                      help="Specify this if the we wish to treat the subject, 'fn', as one with no 'true' y-↘
                            labels. "
99                           "Hence, specify this if the subject does not have 'true' values recorded by ↘
                                specialists that "
100                          "is stored in the 'nsaa_6mw_info.xlsx' file (e.g. if the subject is a new subject with↘
                                no "
101                          "true labels or if we otherwise wish to treat them as such).")
102 parser.add_argument("--leave_out_version", type=str, nargs="?", const=True, default=False,
103                     help="Specify this with the name of a version of subjects (e.g. 'V2') to assess the 'fn' ↘
                            file "
104                          "on models with the specified version of files not used to train the models.")
105 parser.add_argument("--final_models", type=bool, nargs="?", const=True, default=False,
106                     help="Specify this if wishing to use the 'final' model directory within the project ↘
                            directory "
107                          "instead of the default directory within the local directory. This is used by several ↘
                                of the "
```

```
108                         "'assess_' batch scripts so one can assess files entirely using the project directory.")
109 parser.add_argument("--no_nsaa_flag", type=bool, nargs="?", const=True, default=False,
110                     help="Flag that is set if the previous 'ft_sel_red.py' script has not added any NSAA information to "
111                         "the 'AD' file. This is used in cases where we are assessing a 'new' subject, e.g. via "
112                         "the 'assess_nsaa_nmb_file.py' script.")
113 args = parser.parse_args()
114
115
116 #Ensures 'dir'=NSAA when '--single_acts' is set
117 if args.single_act and args.dir != "NSAA":
118     print("First arg ('dir') must be 'NSAA' when '--single_acts' is set, as there are only single acts files from "
119           "the files from the 'NSAA' directory.")
120     sys.exit()
121
122 #Ensures 'dir'=NSAA when '--single_acts_concat' is set
123 if args.single_act_concat and args.dir != "NSAA":
124     print("First arg ('dir') must be 'NSAA' when '--single_acts_concat' is set, as there are only single acts files from "
125           "the files from the 'NSAA' directory.")
126     sys.exit()
127
128
129 #Appends the sub_dir name to 'source_dir' if it's one of the allowed names
130 if args.dir + "\\" in sub_dirs:
131     source_dir += args.dir.upper() + "\\"
132 else:
133     print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
134           "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'direct_csv', 'allmatfiles', or 'left-out'.")
135     sys.exit()
136
137 #Ensures the corresponding second argument when dealing with files from 'allmatfiles'
138 if args.dir == "allmatfiles" and args.ft != "jointAngle":
139     print("Second arg ('ft') must be 'jointAngle when 'dir' argument is \'allmatfiles\'")
140     sys.exit()
141
142 #Ensures that, if '--balance' is set, it is either 'up' to upsample the data or 'down' to downsample the data
143 if args.use_balanced:
144     if not args.use_balanced == "up" and not args.use_balanced == "down":
145         print("Optional arg ('--balance') must be set to either 'up' or 'down'.")
146         sys.exit()
147
148 #Ensures that, if '--single_act_concat' is set, it is one of the two allowed strings
149 if args.single_act_concat:
150     if not args.single_act_concat == "src_sac" and not args.single_act_concat == "src_normal":
151         print("'--single_act_concat' must be set to one of 'use_sac' or 'use_normal'...")
152         sys.exit()
153
154 #Ensures that, if the '--final_models' optional argument is set, the directory of models within the project
155 #directory is used
156 if args.final_models:
157     model_dir = "..\\rnn_models_final\\" if args.batch else "rnn_models_final\\"
158
159 fts, sds = [], []
160 for ft in args.ft.split(","):
161     fts.append(ft)
162     if args.dir == "left-out" and ft == "AD":
163         sds.append(local_dir + "output_files\\left-out\\AD\\")
164     elif args.dir == "left-out" and ft in file_types:
165         sds.append(local_dir + "left-out\\" + ft + "\\")
166     elif ft + "\\" in sub_sub_dirs and not args.single_act and not args.single_act_concat:
167         sds.append(source_dir + ft + "\\")
168     elif ft + "\\" in sub_sub_dirs and args.single_act:
169         sds.append(source_dir + ft + "\\act_files\\")
170     elif ft + "\\" in sub_sub_dirs and args.single_act_concat=="src_sac":
171         sds.append(source_dir + ft + "\\act_files_concat\\")
172     elif ft + "\\" in sub_sub_dirs and args.single_act_concat == "src_normal":
173         sds.append(source_dir + ft + "\\")
174     elif ft in file_types and args.dir == "NSAA" and args.single_act_concat=="src_sac":
175         sds.append(local_dir + "NSAA\\matfiles\\act_files_concat\\" + ft + "\\")
176     elif ft in file_types and args.dir == "NSAA" and args.single_act_concat == "src_normal":
177         sds.append(local_dir + "NSAA\\matfiles\\" + ft + "\\")
178     elif ft in file_types and args.dir == "NSAA" and not args.single_act:
179         sds.append(local_dir + "NSAA\\matfiles\\" + ft + "\\")
180     elif ft in file_types and args.dir == "NSAA" and args.single_act:
181         sds.append(local_dir + "NSAA\\matfiles\\act_files\\" + ft + "\\")
182     elif ft in file_types and args.dir == "NSAA" and not args.single_act_concat:
183         sds.append(local_dir + "NSAA\\matfiles\\" + ft + "\\")
184     elif args.dir == "allmatfiles":
185         sds.append(local_dir + "allmatfiles\\" + ft + "\\")
186     elif args.dir == "NMB" and ft != "AD":
187         sds.append(local_dir + "NMB\\" + ft + "\\")
188     elif args.dir == "NMB":
189         sds.append(source_dir + "NMB\\AD\\")
190     else:
191         print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
192               "\'JA', or \'DC\' (unless dir is give as 'NSAA', where 'ft' can be a measurement name), and each part "
193               "must be separated by a comma for each measurement to use in the ensemble.")
194         sys.exit()
```

```
195
196 #Adds a dash to the beginning of the 'fn' argument input if the '--handle_dash' optional argument is set
197 if args.handle_dash:
198     args.fn = "-" + args.fn
199
200 local_nsaa_6mw_path = "..\\" + nsaa_6mw_path if args.batch else nsaa_6mw_path
201
202 #Section below encompasses all the necessary rules to fetch files from various directories, subdirectories, and so on
203 #based on the 'dir', 'fn', 'ft', and other option arguments. Result of this section is a list of short file names, 'fns',
204 #that contain the list of all names of files (not including their full paths)
205 fns = []
206 for ft, sd in zip(fts, sds):
207     if args.dir == "allmatfiles":
208         if any(args.fn.upper() == s.split("jointangle")[-1].split(".mat")[0].upper() for s in os.listdir(sd)):
209             fns.append([s for s in os.listdir(sd) if args.fn.upper() == s.split("jointangle")[-1].split(".mat")[0].upper()][0])
210         else:
211             print("Cannot find '" + str(args.fn) + "' in '" + sd + "'")
212             sys.exit()
213     elif args.dir == "left-out":
214         fns.append([s for s in os.listdir(sd) if args.fn.split("-")[0] in s][0])
215     elif args.dir == "NMB" and ft != "AD":
216         fns.append([s for s in os.listdir(sd) if args.fn in s][0])
217     elif args.dir == "NMB":
218         fns.append([s for s in os.listdir(sd) if "FR_AD_" + args.fn.split("-")[0] + "_" in s][0])
219     elif ("AD\\" in sd and any(args.fn.upper().split("-")[0] == s.upper().split("_")[2] for s in os.listdir(sd)
220                                if s.endswith(".csv"))) \
221             or any(args.fn.upper() == "-".join(s.upper().split("-")[:2]) for s in os.listdir(sd)) \
222             or any(args.fn.upper() == s.upper().split("-")[0] for s in os.listdir(sd)) \
223             or any(args.fn.upper() == s.upper().split("_")[2] for s in os.listdir(sd) if s.endswith(".csv")):
224         if "AD\\" in sd:
225             if not args.single_act:
226                 fns.append([s for s in os.listdir(sd) if args.fn.upper().split("-")[0] == s.upper().split("_")[1]][0])
227             else:
228                 fns.append([s for s in os.listdir(sd) if args.fn.upper() == s.upper().split("_")[1]
229                             and "act" + str(args.single_act) + "_" in s][0])
230         else:
231             if not args.single_act:
232                 fns.append([s for s in os.listdir(sd) if args.fn in s][0])
233             else:
234                 try:
235                     fns.append([s for s in os.listdir(sd) if args.fn in s and "act" + str(args.single_act) + "_" in s][0])
236                 except IndexError:
237                     print("Act " + str(args.single_act) + " not found for " + args.fn + " in '" + sd + "', skipping...")
238                     sys.exit()
239     elif ft == "AD" and args.single_act:
240         fns.append([s for s in os.listdir(sd) if args.fn.upper() == s.upper().split("_")[1]
241                     and "act" + str(args.single_act) + "_" in s][0])
242     elif args.single_act_concat:
243         try:
244             fns.append([s for s in os.listdir(sd) if args.fn.upper() == s.upper().split("_")[0]][0])
245         except IndexError:
246             print("Cannot find subject '" + args.fn.upper() + "' in " + sd + "...")
247             sys.exit()
248     else:
249         print("Cannot find '" + str(args.fn) + "' in '" + sd + "'")
250         sys.exit()
251
252
253 #Creates a list of the full file path names to load the models of, given the type of file and short file names ('fns')
254 full_file_names = []
255 for sd, fn in zip(sds, fns):
256     if fn.startswith("AD"):
257         if not args.use_frc:
258             full_file_names.append(sd + "FR_" + fn)
259         else:
260             full_file_names.append(sd + "FRC_" + fn)
261     elif args.dir == "allmatfiles":
262         full_file_names.append(sd + "jointAngle\\" + fn.split(".")[0] + "_jointAngle.csv")
263     else:
264         full_file_names.append(sd + fn)
265
266
267 #Specifies the directories (i.e. names of file groupings that can be used to train a model, e.g. 'NSAA' or
268 #'6MW-matFiles') on which to test the file. If the 'alt_dirs' optional argument is not provided, then 'dir' is used
269 #(i.e. the file in question is tested on the same directory type as it comes from).
270 if args.alt_dirs:
271     search_dirs = args.alt_dirs.split("_")
272     if not all(sd + "\\" in sub_dirs for sd in search_dirs):
273         print("Optional arg ('--alt_dirs') must be directory names separated by commas and each must be one of "
274               "'NSAA', '6minwalk-matfiles', '6MW-matFiles', or 'direct_csv'.")
275         sys.exit()
276 else:
277     search_dirs = [args.dir]
278
279 #If the '--final_models' optional argument is set, get the mapping of long description names of models to
```

```
#short names as contained within 'rnn_models_final' and use these to select the models we need
model_map_dict = {}
if args.final_models:
    df = pd.read_excel("rnn_models_final\\model_map.xlsx")
    model_map_dict = pd.Series(df.ModNum.values,index=df.ModString).to_dict()
    model_names = list(model_map_dict.keys())
else:
    model_names = os.listdir(model_dir)

#The models that are to be tested by the script are selected based on their names; e.g. if 'dir'=NSAA and 'ft'=position,
#then 'NSAA_position_all_acts', 'NSAA_position_all_dhc', and 'NSAA_position_all_overall' will be loaded as directory
#names containing the trained models
models = []
for sd in search_dirs:
    inner_models = []
    if args.use_indiv:
        output_types = ["indiv"]
    for ot in output_types:
        inner_inner_models = []
        #If the '--use_ft_concat' arg is set, for each of the output types, find the model that contains all of the
        #file types that are given to 'model_predictor' within its title
        if args.use_ft_concat:
            match_dirs = [fn for fn in model_names if fn.split("_")[0] == sd and
                          fn.split("_")[3] == ot]
            for md in match_dirs:
                if "--pca=" + args.use_ft_concat in md and all(ft for ft in args.ft.split(",") if ft in md):
                    if not args.use_seen:
                        if ("--leave_out=" + args.fn) in md:
                            inner_inner_models.append(md)
                            print("Using model: '" + str(md) + " '...")
                            break
                    else:
                        if "--leave_out=" not in md:
                            inner_inner_models.append(md)
                            print("Using model: '" + str(md) + " '...")
                            break
            inner_models.append(inner_inner_models)
        elif args.ft_red:
            match_dirs = [fn for fn in model_names if fn.split("_")[0] == sd and
                          fn.split("_")[3] == ot and "--pca=" + args.ft_red in fn]
            for ft in fts:
                for md in match_dirs:
                    if not args.use_seen:
                        if ("--leave_out=" + args.fn) in md and md.split("_")[1] == ft:
                            inner_inner_models.append(md)
                            print("Using model: '" + str(md) + " '...")
                    else:
                        if "--leave_out=" not in md and md.split("_")[1] == ft:
                            inner_inner_models.append(md)
                            print("Using model: '" + str(md) + " '...")
            inner_models.append(inner_inner_models)
        else:
            for i, ft in enumerate(fts):
                #Handles the special case when we're using NSAA files that are placed in the 'left-out' source dir
                if args.dir == "left-out":
                    inner_inner_models.append([md for md in model_names if md.split("_")[0] == "NSAA"
                                               and md.split("_")[1] == ft and md.split("_")[3] == ot
                                               and "--leave_out" not in md and "--balance" not in md
                                               and "--other_dir" not in md][0])
                elif any(fn for fn in model_names if fn.split("_")[0] == sd and
                                                     fn.split("_")[3] == ot and fn.split("_")[1] == ft):
                    #Gets the first inner model directory that match the file type, directory name, and output type, with
                    #additional preference of model trained on 'left out file' if one exists in 'model_dir. Also
                    #ensures that if '--add_dir' is used that its value is within the directory name, otherwise ensures
                    #that any models containing '--add_dir' values aren't used.
                    if not args.add_dir:
                        match_dirs = [fn for fn in model_names if fn.split("_")[0] == sd and
                                      fn.split("_")[3] == ot and fn.split("_")[1] == ft]
                    else:
                        match_dirs = [fn for fn in model_names if fn.split("_")[3] == ot and
                                      fn.split("_")[1] == ft and sd in fn.split("_")[0] and
                                      args.add_dir in fn.split("_")[0]]
                    #Ensures that if '--other_lo' is used that its value is within the directory name, otherwise ensures
                    #that any models containing 'args.fn + ',' (i.e. models with more than 1 file left out) aren't used
                    match_dirs = [fn for fn in match_dirs if args.other_lo in fn] if args.other_lo \
                        else [fn for fn in match_dirs if args.fn + "," not in fn]
                    match_dirs = [fn for fn in match_dirs if "--use_sac" in fn] if args.single_act_concat \
                        else [fn for fn in match_dirs if "--use_sac" not in fn]
                    match_dirs = [fn for fn in match_dirs if "--no_testset" in fn] if args.no_testset \
                        else [fn for fn in match_dirs if "--no_testset" not in fn]
                    match_dirs = [fn for fn in match_dirs if "--leave_out_version=" + args.leave_out_version in fn] \
                        if args.leave_out_version \
                        else [fn for fn in match_dirs if "--leave_out_version=" not in fn]
```

```
                    if any(args.fn in md for md in match_dirs):
                        #If 'use_balanced' is set, specifically use ones which have '--balance' in directory name
                        if args.use_balanced and args.use_balanced == "up":
                            if not args.use_seen and not args.leave_out_version:
                                #If 'use_seen' is not set, specifically use ones that have '--leave_out='fn'' in directory name
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--balance=up" in md and "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--balance=up" in md and "--leave_out=" not in md][0])
                        elif args.use_balanced and args.use_balanced == "down":
                            if not args.use_seen and not args.leave_out_version:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--balance=down" in md and "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--balance=down" in md and "--leave_out=" not in md][0])
                        elif args.use_frc and ft == "AD":
                            if not args.use_seen and not args.leave_out_version:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--use_frc" in md and "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--use_frc" in md and "--leave_out=" not in md][0])
                        elif args.standardize:
                            if not args.use_seen and not args.leave_out_version:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--standardize" in md and "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--standardize" in md and "--leave_out=" not in md][0])
                        elif args.noise:
                            if not args.use_seen and not args.leave_out_version:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--noise=" + args.noise in md and "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in match_dirs
                                                           if "--noise=" + args.noise in md and "--leave_out=" not in md][0])
                        else:
                            option_args = ["--balance=up", "--balance=down", "--use_frc", "--use_frc",
                                           "--standardize", "--noise"]
                            filtered_match_dirs = [md for md in match_dirs
                                                   if not any(opt_arg in md for opt_arg in option_args)]
                            if not args.use_seen and not args.leave_out_version:
                                inner_inner_models.append([md for md in filtered_match_dirs if "--leave_out=" + args.fn in md][0])
                            else:
                                inner_inner_models.append([md for md in filtered_match_dirs if "--leave_out=" not in md][0])
                    else:
                        inner_inner_models.append(match_dirs[0])
                else:
                    inner_inner_models.append(None)
                print("Using model: '" + str(inner_inner_models[-1]) + " '...")
            inner_models.append(inner_inner_models)
    models.append(inner_models)

#If the '--final_models' optional argument is set, maps the description names of the models to their true names
#within 'rnn_models_final' by mapping using the 'model_map.csv' file
if args.final_models:
    models = [[[model_map_dict[inner_inner_model] for inner_inner_model in inner_inner_models]
               for inner_inner_models in inner_models] for inner_models in models]

#Removes empty 'inner_model' lists (i.e. ones only populated by 'Nones') in the case where all measurements don't have
#a specific output type model
new_models = []
for inner_models in models:
    if not all(not model for model in inner_models):
        new_models.append(inner_models)
models = new_models

#Reads in the .xlsx file with the file shapes in them and extracts the sequence lengths from the most recent entry in
#the tabel that corresponds to the model with a specific source dir, file type, and output type
model_shapes_path = "..\\" + model_shapes_path if args.batch else model_shapes_path
model_shape = pd.read_excel(model_shapes_path)

if args.add_dir:
    sequence_lengths = [[[model_shape.loc[(model_shape["dir"] == model.split("_")[0].split(",")[0]) &
                                          (model_shape["ft"] == model.split("_")[1]) &
                                          (model_shape["measure"] == model.split("_")[3])].iloc[
```

```
                                    -1, -1] if model else None
                                for model in inner_inner_models] for inner_inner_models in inner_models] for inner_models in
                            models]
    elif not args.use_ft_concat:
        sequence_lengths = [[[model_shape.loc[(model_shape["dir"] == model.split("_")[0]) &
                                              (model_shape["ft"] == model.split("_")[1]) &
                                              (model_shape["measure"] == model.split("_")[3])].iloc[-1, -1] if model else None
                                for model in inner_inner_models] for inner_inner_models in inner_models] for inner_models in models]
    else:
        sequence_lengths = [[[model_shape.loc[(model_shape["dir"] == model.split("_")[0]) &
                                              (model_shape["measure"] == model.split("_")[3])].iloc[-1, -1] if model else None
                                for model in inner_inner_models] for inner_inner_models in inner_models] for inner_models in
                            models]


    def preprocessing(full_file_name, sequence_length):
        #Loads the .csv data from the given file name and divides it up into a 3D format based on 'sequence_length'; for
        #example, if 'df_x' is of shape (1000, 50) and 'sequence_length' = 60, then 'x_data' becomes (16, 60, 50). Note that
        #the 'leftover' data at the end of 'df_x' that does not fit into a (60, 50) shape is discarded.
        if not "AD" in full_file_name.split("\\")[-1]:
            df_x = pd.read_csv(full_file_name, index_col=0).values
        elif "AD" in full_file_name.split("\\")[-1] and args.no_nsaa_flag:
            df_x = pd.read_csv(full_file_name, index_col=0).iloc[:, 2:].values
        else:
            df_x = pd.read_csv(full_file_name, index_col=0).iloc[:, 20:].values

        #If we only wish to use a single sequence drawn from the subject file, draw 'sequence_length' frames of data
        #from the file that is evenly distributed across the file
        if args.single_sequence:
            df_x = np.array([df_x[i] for i in range(0, len(df_x), int(len(df_x)/sequence_length)+1)])


        x_data = [df_x[sequence_length*i:(sequence_length*(i+1)), :] for i in range(int(len(df_x)/sequence_length))]

        #Handles the case where the '.mat' file doesn't even contain enough data for one sequence (i.e. the number of rows
        #in the file is < 'sequence_length')
        if len(x_data) == 0:
            print(full_file_name.split("\\")[-1].split(".")[0], "too short in length to use, skipping....")
            sys.exit()

        #If the '--new_subject' optional arg is split, returns without fetching the 'y' label information
        #from the 'nsaa_6mw_info.xlsx' file
        if args.new_subject:
            return x_data, None, None, None

        #Loads the file that contains information on each subject and their corresponding overall and individual NSAA scores,
        #and three 'true' labels for the file are extracted: 'y_label_dhc' gets 1 if the short name of the file (e.g. 'D11')
        #begins with a 'D', else it gets a 0; 'y_label_overall' gets an integer value between 0 and 34 of the overall NSAA score,
        #and 'y_label_acts' gets the 17 individiual NSAA acts as a list as scores between 0 and 2
        df_y = pd.read_excel(local_nsaa_6mw_path)
        if args.dir != "direct_csv" and not source_dir.startswith(local_dir):
            y_label_dhc = 1 if full_file_name.split("\\")[-1].split("_")[2][0].upper() == "D" else 0
        elif source_dir.startswith(local_dir):
            if "FR_" in full_file_name or "FRC_" in full_file_name:
                y_label_dhc = 1 if full_file_name.split("\\")[-1].split("_")[2][0].upper() == "D" else 0
            elif full_file_name.split("\\")[-1].startswith("jointangle"):
                y_label_dhc = 1 if full_file_name.split("\\")[-1].split("angle")[1][0].upper() == "D" else 0
            else:
                y_label_dhc = 1 if full_file_name.split("\\")[-1].split("_")[0][0].upper() == "D" else 0
        else:
            y_label_dhc = 1 if full_file_name.split("\\")[-1].split("_")[1][0].upper() == "D" else 0
        y_index = df_y.index[df_y["ID"] == args.fn.upper().split("-")[0]]
        #Raises an exception
        if len(y_index) == 0:
            #raise FileExistsError
            print(args.fn.upper().split("-")[0] + " not an entry in 'nsaa_6mw_info.xlsx' for " + full_file_name + ", skipping...")
            sys.exit()
        y_label_overall = df_y.loc[y_index, "NSAA"].values[0]
        try:
            y_label_acts = [int(num) for num in df_y.iloc[y_index, 5:].values[0]]
        except ValueError:
            print("\nEntry for '" + args.fn + "' single-act scores not in table, skipping file...\n")
            sys.exit()


        #If the '--standardize' optional argument is given, reshape the 'x' data into a 2D array, standardize
        #each of the features, and reshape it back into its original shape (note: np.concatenate and np.reshape aren't used
        #in order to ensure the data is written back as we would expect with the same number of lines)
        if args.standardize:
```

```
520             x_data = [sample for sequence in x_data for sample in sequence]
521             print("Standardizing the data set...")
522             x_data = StandardScaler().fit_transform(x_data)
523             x_data = [[x_data[i * x_shape[1] + j] for j in range(x_shape[1])] for i in range(x_shape[0])]
524
525         return x_data, y_label_dhc, y_label_overall, y_label_acts
526
527
528
529     def create_batch_generator(x, y=None, batch_size=64):
530         """
531         :param 'x', which is the data to create batches out of and, if 'y' is supplied as well, create batches out ↘
                of
532         this as well based on the 'batch_size' provided
533         :return: the next batch of data to the calling function
534         """
535         n_batches = len(x)//batch_size
536         #Ensures 'x' is a multiple of batch size
537         x = x[:n_batches*batch_size]
538         if y is not None:
539             #Ensures 'y' is a multiple of batch size
540             y = y[:n_batches*batch_size]
541         #For each 'batch_size' chuck of x, yield the next part of the 'x' and 'y' data (i.e. the next 'batch size' ↘
                samples)
542         for ii in range(0, len(x), batch_size):
543             if y is not None:
544                 yield x[ii:ii+batch_size], y[ii:ii+batch_size]
545             else:
546                 yield x[ii:ii+batch_size]
547
548
549
550     def combine_preds(output_strs):
551         """
552             :return: gets the results that are contained within the output strings, computes a method for ↘
                    predicting an
553             overall NSAA score based on the D/HC label and percentage of predicted sequences, and aggregates the
554             predictions made by all three output types to get an average aggregated overall NSAA score prediction
555         """
556
557         #Gets the numerical predictions made for each of the output metrics as contained within 'output_strs'
558         acts_sum_pred = sum(eval([out_str for out_str in output_strs if "Predicted 'Acts Sequence'" in out_str][0].↘
                split(" = ")[1]))
559         dhc_pred = [out_str for out_str in output_strs if "Predicted 'D/HC Label'" in out_str][0].split(" = ")[1]
560         dhc_prop_pred = float([out_str for out_str in output_strs if "Percentage of predicted '" + dhc_pred + "'↘
                sequences"
561                 in out_str][0].split(" = ")[1].split("%")[0])/100
562         overall_pred = int([out_str for out_str in output_strs if "Predicted 'Overall NSAA Score'" in out_str][0].↘
                split(" = ")[1])
563
564         #Gets the median value of all the subjects true overall NSAA scores according to the 'nsaa_6mw_info.xlsx' ↘
                file
565         #and uses this as the basis for the overall NSAA score associated with the 'D' label
566         df_y = pd.read_excel(local_nsaa_6mw_path)
567         df_y = df_y.loc[df_y["ID"].str.startswith("D")]
568         d_median = int(np.median(df_y["NSAA"].tolist()))
569
570         #Averages the overall NSAA scores made by each of the predictions, with the average of the predicted ↘
                overall NSAA
571         #score, the sum of the indivudal acts predictions, and the assigned numerical value for the 'D' or 'HC' ↘
                label
572         dhc_value = 34 if dhc_pred == "HC" else d_median
573         average_pred = round((acts_sum_pred + overall_pred + (dhc_value*dhc_prop_pred)) / (2 + dhc_prop_pred))
574
575         output_strs.append("Aggregated predicted 'Overall NSAA Score' = " + str(average_pred))
576
577         return output_strs
578
579
580
581     pred_overalls, true_overalls = [], []
582     #Stores the strings to write AFTER all the models have run so all results are printed at the end, rather than ↘
            some being
583     #printed and then obscured by the model setup text
584     output_strs = []
585     #For each directory type that we wish to assess on...
586     for i, inner_models in enumerate(models):
587         output_strs.append("\t\t ------- " + str(search_dirs[i]) + " Predictions ------- ")
588         #For each output type that we are training towards (e.g. 'overall', 'acts', etc.)...
589         for j, inner_inner_models in enumerate(inner_models):
590             #'output_type' contains one of 'acts', 'indiv', or 'dhc''
591             output_type = None
592             for model in inner_inner_models:
593                 if model:
594                     output_type = model.split("_")[3]
595             preds = []
596             y_label_dhc, y_label_overall, y_label_acts = (None,)*3
597             #For each measurement type (i.e. for every model name, e.g. 'jointAngle' or 'AD')
598             ft_concat_data = [[] for i in range(5)]
599             data_entered = False
600             for k, full_file_name in enumerate(full_file_names):
601                 #If the argument is set then, for each of the file types given, concatenate the data together
602                 #and if it's the last file type in the sequence of given file types, reduce the dimensions of the ↘
                        sequences
```

```
603             #to the value given by the argument
604             if args.use_ft_concat:
605                 x_data, y_label_dhc, y_label_overall, y_label_acts = preprocessing(full_file_name, ↘
                        sequence_lengths[0][j][0])
606                 ft_concat_data[0] = np.concatenate((ft_concat_data[0], x_data), axis=2) if data_entered else ↘
                        x_data
607                 ft_concat_data[1] = ft_concat_data[1] if data_entered else y_label_dhc
608                 ft_concat_data[2] = ft_concat_data[2] if data_entered else y_label_overall
609                 ft_concat_data[3] = ft_concat_data[3] if data_entered else y_label_acts
610                 data_entered = True
611                 if k == len(full_file_names)-1:
612                     x_data, y_label_dhc, y_label_overall, y_label_acts = \
613                         ft_concat_data[0], ft_concat_data[1], ft_concat_data[2], ft_concat_data[3]
614                     try:
615                         pca = int(args.use_ft_concat)
616                         x_shape = np.shape(x_data)
617                         x_data = [sample for sequence in x_data for sample in sequence]
618                         x_data, y_data = ft_red_select(x_data, None, "pca", False, False, pca)
619                         x_data = [[x_data[i * x_shape[1] + j] for j in range(x_shape[1])] for i in range(↘
                                x_shape[0])]
620                     except ValueError:
621                         if args.use_ft_concat != "all":
622                             print("Optional arg '--pca' must contain int value or 'all' to keep all features...↘
                                ")
623                             sys.exit()
624                 else:
625                     continue
626             #Else, prepare the data as normal for the given output type and file
627             else:
628                 #If there is a 'None' sequence length (i.e. a corresponding model for the output type and file ↘
                        type doesn't
629                 #exist), then skip over it
630                 if not sequence_lengths[i][j][k]:
631                     continue
632                 x_data, y_label_dhc, y_label_overall, y_label_acts = preprocessing(full_file_name, ↘
                        sequence_lengths[i][j][k])
633                 if args.ft_red:
634                     try:
635                         pca = int(args.ft_red)
636                         x_shape = np.shape(x_data)
637                         x_data = [sample for sequence in x_data for sample in sequence]
638                         x_data, y_data = ft_red_select(x_data, None, "pca", False, False, pca)
639                         x_data = [[x_data[i * x_shape[1] + j] for j in range(x_shape[1])] for i in range(↘
                                x_shape[0])]
640                     except ValueError:
641                         if args.use_ft_concat != "all":
642                             print("Optional arg '--pca' must contain int value or 'all' to keep all features...↘
                                ")
643                             sys.exit()
644
645
646
647             #Complete model path to the directory of the model in question
648             model_path = model_dir + inner_inner_models[k] if not args.use_ft_concat else model_dir + ↘
                    inner_inner_models[0]
649             inner_preds = []
650             with tf.Session(graph=tf.Graph()) as sess:
651                 #Loads the model and restores from it's final checkpoint
652                 new_saver = tf.train.import_meta_graph(model_path + "\\model.ckpt.meta", clear_devices=True)
653                 new_saver.restore(sess, tf.train.latest_checkpoint(model_path))
654                 #If there aren't enough sequences within the file being tested to make up a full batch (i.e. ↘
                        len(x_data)
655                 #< batch_size), then replicate it until it's the size of at least batch size
656                 new_x_data = []
657                 while len(new_x_data) < batch_size:
658                     new_x_data += list(x_data)
659                 x_data = new_x_data
660                 #For each batch of the data from 'full_file_name', feed it through the trained model to get ↘
                        predictions based
661                 #on the type of model (e.g. '1's or '0's if output_type == "dhc", ints between 0 and 34 if ↘
                        output_type ==
662                 #"overall", and lists of 17 values between 0 and 1 if output_type == "dhc") and appends these ↘
                        to 'preds'
663                 for ii, batch_x in enumerate(create_batch_generator(x_data, None, batch_size=batch_size), 1):
664                     feed = {'tf_x:0': batch_x, 'tf_keepprob:0': 1.0}
665                     if output_type == "overall" or args.use_indiv:
666                         inner_preds.append(sess.run('logits_squeezed:0', feed_dict=feed))
667                     else:
668                         inner_preds.append(sess.run('labels:0', feed_dict=feed))
669             #Flatten these values to a 1D list
670             inner_preds = np.concatenate(inner_preds)
671             #Gets rid of errant 'nan's in the predictions (cause unknown)
672             inner_preds = [ip for ip in inner_preds if "nan" not in str(ip)]
673             #Add these predictions to aggregate list of predictions for the output type in question
674             preds.append(inner_preds)
675             #Appends all predicted overall NSAA scores for a given model to the list containing the aggregated ↘
                    predicted scores
676             if output_type == "overall":
677                 pred_overalls += inner_preds
678
679
680         #Aggregate results for measurements
681         if output_type == "acts":
682             #Makes sure that each measurement's worth of predictions are the same length (e.g. if there are far
```

```
683             #fewer 'AD' predictions than 'jointAngle', then duplicate 'AD' predictions to be of the same length↘
                    )
684             preds_lens = [len(preds[m]) for m in range(len(preds))]
685             for m in range(len(preds)):
686                 #If the length of one part of 'preds' is shorter than the rest, add 'batch_size' chunks of that↘
                        part
687                 #of preds until it matches that of the largest part. This ensures they all have the same length
688                 #to be able to use 'np.transpose()'.
689                 while len(preds[m]) < max(preds_lens):
690                     preds[m] += preds[m][:batch_size]
691             preds = np.transpose(preds, (1, 2, 0))
692             preds = [[Counter(elems).most_common()[0][0] for elems in row] for row in preds]
693         elif output_type == "dhc":
694             preds = np.transpose(preds)
695             preds = [Counter(elems).most_common()[0][0] for elems in preds]
696         else:
697             preds = np.transpose(preds)
698             preds = [np.mean(elems) for elems in preds]
699
700
701         #Based on what type the model is, add output lines to 'output_strs' that gives info on the true 'y'↘
                label of the
702         #file (e.g. true 'D' or 'HC' label, true overall NSAA score, or true single acts scores) and what the↘
                model predicted
703         if output_type == "overall":
704             true_overalls = [y_label_overall for m in range(len(pred_overalls))]
705             if not args.new_subject:
706                 output_strs.append(str("True 'Overall NSAA Score' = " + str(y_label_overall)))
707             output_strs.append(str("Predicted 'Overall NSAA Score' = " + str(int(round(float(np.mean(preds)),↘
                    0)))))
708         elif output_type == "dhc":
709             true_label = "D" if y_label_dhc == 1 else "HC"
710             pred_label = "D" if max(set(list(preds)), key=list(preds).count) == 1 else "HC"
711             d_percen = np.round(((np.sum(preds)/len(preds))*100), 2)
712             hc_percen = np.round(100 - d_percen, 2)
713             if not args.new_subject:
714                 output_strs.append(str("True 'D/HC Label' = " + true_label))
715             output_strs.append(str("Predicted 'D/HC Label' = " + pred_label))
716             output_strs.append(str("Percentage of predicted 'D' sequences = " + str(d_percen) + "%"))
717             output_strs.append(str("Percentage of predicted 'HC' sequences = " + str(hc_percen) + "%"))
718         elif args.use_indiv:
719             if not args.new_subject:
720                 true_label = y_label_acts[int(args.single_act)-1]
721                 output_strs.append(str("True individual activity score = " + str(true_label)))
722             output_strs.append(str("Predicted individual activity score = " + str(round(float(np.mean(preds)),↘
                    2))))
723         else:
724             preds = np.transpose(preds)
725             pred_acts = [Counter(preds[i]).most_common()[0][0] for i in range(len(preds))]
726             perc_correct = 0
727             if not args.new_subject:
728                 num_correct = 0
729                 for m in range(len(y_label_acts)):
730                     if y_label_acts[m] == pred_acts[m]:
731                         num_correct += 1
732                 perc_correct = round(((num_correct / 17)*100), 2)
733                 output_strs.append(str("True 'Acts Sequence' = " + str(y_label_acts)))
734             output_strs.append(str("Predicted 'Acts Sequence' = " + str(pred_acts)))
735             if not args.new_subject:
736                 output_strs.append(str("Percent of acts correctly predicted = " + str(perc_correct) + "%"))
737         output_strs.append("")
738
739
740 #If the optional '--combine_preds' argument is set, compute the aggregated predicted 'Overall NSAA Score' and↘
        add this
741 #to the list of output strings
742 if args.combine_preds:
743     output_strs = combine_preds(output_strs)
744
745
746 #Prints to the user all the output lines that were generated by testing on all the models
747 print("\n")
748 for output_str in output_strs:
749     print(output_str)
750
751
752
753 #Plot the predicted values against the true values for NSAA overall (only if *not* run from 'test_altdirs.py';↘
        does not
754 #worth if '--new_subject' is set, as won't have the 'true' values
755 if not args.file_num and not args.new_subject:
756     fig, ax = plt.subplots()
757     ax.scatter(true_overalls, pred_overalls, alpha=0.03)
758     plt.xlim(0, 34)
759     plt.ylim(0, 34)
760     x = np.linspace(*ax.get_xlim())
761     ax.plot(x, x)
762     plt.title("Plot of true overall NSAA scores against predicted overall NSAA scores")
763     plt.xlabel("True overall NSAA scores")
764     plt.ylabel("Predicted overall NSAA scores")
765     graphs_path = "..\\..\\documentation\\Graphs\\Model_predictor_" + args.dir + "_" + args.ft + "_" + args.fn↘
            if args.batch \
766         else "..\\documentation\\Graphs\\Model_predictor_" + args.dir + "_" + args.ft + "_" + args.fn
767     plt.savefig(graphs_path)
```

```
768         plt.gcf().set_size_inches(10, 10)
769         if args.show_graph:
770             plt.show()
771 
772     #Copies the 'fn' argument value to 'write_fn', which will be modified and then added to the list of '↘
            output_strs'
773     write_fn = args.fn
774     #Appends a short string to the file name being written to file if the models have already seen the file name in↘
            training
775     if args.use_seen:
776         write_fn += " (already seen)"
777     if args.other_lo:
778         write_fn += " (other lo = " + args.other_lo + ")"
779     if args.single_act_concat == "src_sac":
780         write_fn += " (src sac)"
781     elif args.single_act_concat == "src_normal":
782         write_fn += " (src normal)"
783     if args.use_ft_concat:
784         write_fn += " (feature concat = " + args.use_ft_concat + ")"
785     if args.ft_red:
786         write_fn += " (feature reduced = " + args.ft_red + ")"
787     if args.add_dir:
788         write_fn += " (additional dir = " + args.add_dir + ")"
789     if args.single_act:
790         write_fn += " (act " + str(args.single_act) + ")"
791     if args.single_sequence:
792         write_fn += " (single sequence)"
793     if args.combine_preds:
794         write_fn += " (aggregate overall)"
795     if args.no_testset:
796         write_fn += " (no testset)"
797     if  args.use_indiv:
798         write_fn += " (indiv)"
799     if args.use_frc:
800         write_fn += " (FRC)"
801     if args.noise:
802         write_fn += " (noise = " + args.noise + ")"
803     if args.use_balanced and args.use_balanced == "down":
804         write_fn += " (downsampled)"
805     elif args.use_balanced and args.use_balanced == "up":
806         write_fn += " (upsampled)"
807     if args.leave_out_version:
808         write_fn += " (leave out version = " + args.leave_out_version + ")"
809 
810 
811     #If we're predicting on a subject where we know the true values of their overall score, individual scores, etc↘
            ., then
812     #we write to 'model_predictions.csv' as this is a table that stores the true values for each output type as ↘
            well as predicted
813     if not args.new_subject:
814         #Creates a list of outputs of the model in a way more fitting of a .csv file (e.g. reducing writing "↘
                Percent correct = 10%"
815         #to "10%" when writing it to a cell, while the corresponding header becomes "Percent correct")
816         header, new_output_strs = [], []
817         for out_str in output_strs:
818             if out_str == "":
819                 pass
820             elif " = " in out_str:
821                 header.append(out_str.split(" = ")[0])
822                 new_output_strs.append(out_str.split(" = ")[1])
823             else:
824                 header.append(out_str)
825                 new_output_strs.append(out_str)
826 
827         #Adds in additional strings to the row that are based on the arguments based in to 'model_predictor' and ↘
                also creates
828         #additional corresponding header strings
829         header = ["Short file name", "Source dir", "Model trained dir(s)", "Measurements tested"] + header
830         if args.alt_dirs:
831             new_output_strs = [write_fn, args.dir, args.alt_dirs.split("_"), args.ft.split(",")] + new_output_strs
832         else:
833             #If same directory is used for training models as is used for assessing, add a 'N/A' to the column
834             new_output_strs = [write_fn, args.dir, "N/A", args.ft.split(",")] + new_output_strs
835 
836         #Adds a index name that is set to what number it corresponds to within 'dir' if passed from 'test_altdirs'
837         #(e.g. '8/470') or 'N/A' if the optional '--file_num' arg is not provided
838         ind_lab = args.file_num if args.file_num else "N/A"
839         #Creates a single-line DataFrame from the predictions made by the model(s) with corresponding headers
840         output_strs_df = pd.DataFrame([new_output_strs], columns=header, index=[ind_lab])
841 
842 
843         #Writes the single-line DataFrame to a new .csv file if it doesn't exist or, if it does exist, appends it ↘
                to the end
844         #of the existing one
845         model_pred_path = "..\\" + model_pred_path if args.batch else model_pred_path
846         if not os.path.exists(model_pred_path):
847             with open(model_pred_path, 'w', newline='') as file:
848                 output_strs_df.to_csv(file, header=header)
849         else:
850             with open(model_pred_path, 'a', newline='') as file:
851                 output_strs_df.to_csv(file, header=False)
852         print("Results have been written to '" + model_pred_path + "'...")
853 
854     #If '--new_subject' is set (i.e. we're using a subject with a new version or a completely new subject ↘
```

```
        altogether that
#doesn't exist within the 'nsaa_6mw_info' table), then write to 'model_predictions_newfiles.csv'
else:
    df = pd.read_excel(local_nsaa_6mw_path)

    #Gets the ID of the row we are fetching in the table. If 'fn' is a different version of a subject that exists in
    #the table (e.g. 'fn'=D4V2), then we retrieve the subject's previous version's info (e.g. the 'D4' row).
    id = args.fn.split("-")[0][:-2] if "V2" in args.fn else args.fn.split("-")[0]
    df_row = df.loc[df["ID"] == id]

    #Gets the overall score, individual scores, and D/HC classification of the 'comparison' entry in the table if one
    #exists (e.g. if the 'fn' args is a 'V2' file with a 'V1' file entry in 'nsaa_6mw_info.xlsx'), else if not in the
    #table then add corresponding '(Not in table)' values
    dhc_label = ("D" if id.startswith("D") else "HC") if not df_row.empty else "(Not in table)"
    overall_score = int(df_row.iloc[0, 4]) if not df_row.empty else "(Not in table)"
    acts_scores = df_row.iloc[0, 5:22].astype(float).values.tolist() if not df_row.empty else "(Not in table)"
    acts_scores = [int(act_score) for act_score in acts_scores]


    #Gets the values that we have predicted for the 'fn' file we are predicting upon
    dhc_pred = [out_str for out_str in output_strs if "Predicted 'D/HC Label'" in out_str][0].split(" = ")[1]
    pred_str = "Aggregated predicted 'Overall NSAA Score'" if args.combine_preds else "Predicted 'Overall NSAA Score'"
    overall_pred = int([out_str for out_str in output_strs if pred_str in out_str][0].split(" = ")[1])
    acts_pred = [out_str for out_str in output_strs if "Predicted 'Acts Sequence'" in out_str][0].split(" = ")[1]

    header = ["Short file name", "Source dir", "Model trained dir(s)", "Measurements tested",
              "True D/HC label (other version)", "Predicted D/HC label (short file name)",
              "True overall NSAA score (other version)", "Predicted overall NSAA Score (short file name)",
              "True single act scores (other version)", "Predicted single acts scores (short file name)"]

    new_output_strs = [write_fn, args.dir, args.ft.split(","), dhc_label, dhc_pred, overall_score,
                       overall_pred, acts_scores, acts_pred]

    if args.alt_dirs:
        new_output_strs.insert(2, args.alt_dirs.split("_"))
    else:
        new_output_strs.insert(2, "N/A")

    output_strs_df = pd.DataFrame([new_output_strs], columns=header)

    model_pred_newfiles_path = model_pred_path.split(".csv")[0] + "_newfiles.csv"
    model_pred_newfiles_path = "..\\" + model_pred_newfiles_path if args.batch else model_pred_newfiles_path

    if not os.path.exists(model_pred_newfiles_path):
        with open(model_pred_newfiles_path, 'w', newline='') as file:
            output_strs_df.to_csv(file, header=header, index=False)
    else:
        with open(model_pred_newfiles_path, 'a', newline='') as file:
            output_strs_df.to_csv(file, header=False, index=False)
    print("Results have been written to '" + model_pred_newfiles_path + "'...")
```

A.12 'predictions_selector.py'

```
import argparse
from settings import sub_dirs, model_pred_path
import pandas as pd
import sys


parser = argparse.ArgumentParser()
parser.add_argument("sfn", help="Specify the short file names of the subjects the user wishes to observe. Specify 'all' "
                                "if wish to select all possible rows given the other arguments.")
parser.add_argument("sd", help="Specify the source dir that the selected rows should be from in 'model_predictions'.")
parser.add_argument("--mtd", type=str, nargs="?", const=True, default=False,
                    help="Optional argument to filter rows based on the values within the 'Model trained dir(s)'. "
                         "Separate these 'altdirs' by commas.")
parser.add_argument("--best", type=str, nargs="?", const=True, default=False,
                    help="Optional argument to retrieve the best of the filtered rows. Provide two parts (separated by "
                         "a comma): first part to select the necessary output rows and being one of 'pacp' ('percent of "
                         "acts correctly predicted'), 'ppcs' ('percent of predicted correct sequences'), or 'overall "
                         "('rue..' and 'predicted overall NSAA score'). Second part being an integer of the number of "
                         "best performing rows according to this output metric.")
parser.add_argument("--worst", type=str, nargs="?", const=True, default=False,
                    help="Identical to '--best', except the second part of the arg is used to select the worst "
                         "performing rows according to the selected output metric.")
parser.add_argument("--str_filt", type=str, nargs="?", const=True, default=False,
```

```
                    help="Specify this with a string to filter rows to only those containing this string within "
                         "the 'Short file name' cell. Separate by commas if wishes to have more than one filter (note: "
                         "it acts as an 'OR' of comma-separate values, i.e. row can contain just one of the parts).")
parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
                    help="Option that is only set if the script is run from a batch file to access the external files "
                         "in a correct way.")
args = parser.parse_args()


local_model_pred_path = "..\\" + model_pred_path if args.batch else model_pred_path

#Loads in the 'model_predictions.csv' file
model_preds = pd.read_csv(local_model_pred_path)

metrics = ["pacp", "ppcs", "overall"]


#Filters the rows based on the 'sfn' arg if it isn't 'all'
if not args.sfn == "all":
    try:
        model_preds = model_preds.loc[int(args.sfn)-2:int(args.sd), :]
    except ValueError:
        model_preds = model_preds.loc[model_preds["Short file name"].str.contains(args.sfn)]
        if len(model_preds.index) == 0:
            print("No row names matching the first arg ('" + args.sfn + "') for the 'Short file name' column...")
            sys.exit()

if args.sd + "\\" not in sub_dirs and not args.sd.isdigit():
    print("Second arg ('" + args.sd + "') must be one of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'direct_csv', " +
          "'allmatfiles', or 'left-out'...")
    sys.exit()
else:
    try:
        test_sd = int(args.sd)
    except ValueError:
        model_preds = model_preds.loc[model_preds["Source dir"] == args.sd]

if args.mtd:
    altdirs = args.mtd.split(",")
    for altdir in altdirs:
        if altdir + "\\" not in sub_dirs:
            print("Optional arg '--mtd' must be one or more of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'direct_csv', " +
                  "'allmatfiles', and/or 'left-out', with each separated by a comma...")
            sys.exit()
        else:
            model_preds = model_preds.loc[model_preds["Model trained dir(s)"].str.contains(altdir, na=False)]

#If the '--str_filt' optional argument is given, reduces 'model_preds' to contain just the rows that have at least
#one part (separated by commas) of the '--str_file' argument in the 'Shot file name' column
if args.str_filt:
    new_model_preds = []
    for str_f in args.str_filt.split(","):
        for i, row in model_preds.iterrows():
            if str_f in row["Short file name"]:
                new_model_preds.append(row)
    model_preds = pd.DataFrame(new_model_preds)

selected_rows = [[], []]
best_worst_rows = [0, 0]
best_worst_metrics = [None, None]

#Given that the rows are now filtered correctly, display the remainder based on optionally-provided arguments
if args.best:
    metric, num_rows = args.best.split(",")
    if metric not in metrics:
        print("Optional arg '--best' must have first part (of two, split by comma) of one of 'pacp', 'pcs', "
              "or 'overall'...")
        sys.exit()
    else:
        best_worst_metrics[0] = metric
    try:
        best_worst_rows[0] = int(num_rows)
    except ValueError:
        print("Optional arg '--best' must have second part (of two, split by comma) to be an integer of the number of "
              "'n' best rows of filtered rows.")
        sys.exit()
if args.worst:
    metric, num_rows = args.worst.split(",")
    if metric not in metrics:
        print("Optional arg '--worst' must have first part (of two, split by comma) of one of 'pacp', 'pcs', "
              "or 'overall'...")
        sys.exit()
    else:
        best_worst_metrics[1] = metric
    try:
```

```
109             best_worst_rows[1] = int(num_rows)
110         except ValueError:
111             print("Optional arg '--worst' must have second part (of two, split by comma) to be an integer of the number of "
112                   "'n' worst rows of filtered rows.")
113             sys.exit()
114
115     #Enables us to select the correct column values for each of the directories that are writing for the row
116     offsets = [0, 10] if args.mtd and len(args.mtd.split(",")) == 2 else [0]
117
118
119
120     #Selects the columns that we are interested in, including combinations of some of the output columns (e.g. difference
121     #between 2 of them for 'overall', chosing one column or the other for 'ppcs', removing percentage signs, etc.
122
123     #For each directory that is producing results for the given file's row (defaults to just 1 directory)...
124     for i, row in model_preds.iterrows():
125         #For each '--best' or '--worst' optional arguments set
126         for j, bwm in enumerate(best_worst_metrics):
127             #If it's set by the appropriate arg...
128             if bwm:
129                 sel_row = [row["Short file name"], row["Source dir"], row["Model trained dir(s)"], row["Measurements tested"]]
130                 #For each of the model directories contained within the row...
131                 for offset in offsets:
132                     if bwm == "pacp":
133                         #Adds the percentage value w/o the percentage symbol
134                         sel_row.append(float(row[offset + 8][:-1]))
135                     elif bwm == "ppcs":
136                         #Selects the correct column value to append to the list based on the value in 'True 'D/HC Label''
137                         ppcs = float(row[offset + 11][:-1]) if row[offset + 9] == "D" else float(row[offset + 12][:-1])
138                         sel_row.append(ppcs)
139                     else:
140                         #Finds the absolute difference between the 'True 'Overall NSAA Score'' and 'Predicted 'Overall NSAA Score''
141                         overall_diff = abs(int(row[offset + 13]) - int(row[offset + 14]))
142                         sel_row.append(overall_diff)
143                 #Add the row w/ columns removed + redesigned to list of rows that we're interested in
144                 selected_rows[j].append(sel_row)
145
146
147     #Dictionary to map metric selected for given '--best' or '--worst' to a column name
148     metric_dict = {"pacp": "Percentage of acts correctly predicted", "ppcs": "Percentage of predicted correct sequences",
149                    "overall": "Diff true/pred overall NSAA score"}
150     #Sets up the column names for the 5 or 6 columns needed, based on the metrics stored in them and the names of the
151     #alt dirs or (if not set), the source dir, for each of the '--best' and '--worst' optional args set
152     col_names = [[], []]
153     for i in range(len(col_names)):
154         if selected_rows[i]:
155             col_names[i] += ["Short file name", "Source dir", "Model trained dir(s)", "Measurements tested"]
156             for j in range(len(offsets)):
157                 met_col = metric_dict[best_worst_metrics[i]]
158                 col_names[i].append(met_col)
159
160
161
162     #Finally, selects the top or bottom (or both) 'n' number of lines, depending on which of '--best' or '--worst' has been
163     #selected and the number of lines to extract from each of them, having reversed them if needed for percentage metrics
164     #('pacp' and 'ppcs'), before printing out the selected rows to the console as a DataFrame.
165
166
167     for i in range(len(selected_rows)):
168         if selected_rows[i]:
169             #Create a DataFrame of each group of rows with the previously-obtained column names
170             df = pd.DataFrame(selected_rows[i], columns=col_names[i])
171             #The rows are now sorted in either ascending or descending order (based on the metrics recorded in the rows),
172             #with priority given to first ordering by the 5th column and secondary ordering importance to the 6th column
173             if args.mtd and len(args.mtd.split(",")) == 2:
174                 if best_worst_metrics[i] == "pacp" or best_worst_metrics[i] == "ppcs":
175                     df = df.sort_values(by=[col_names[i][4], col_names[i][5]], ascending=False)
176                 else:
177                     df = df.sort_values(by=[col_names[i][4], col_names[i][5]], ascending=True)
178             else:
179                 if best_worst_metrics[i] == "pacp" or best_worst_metrics[i] == "ppcs":
180                     df = df.sort_values(by=[col_names[i][4]], ascending=False)
181                 else:
182                     df = df.sort_values(by=[col_names[i][4]], ascending=True)
183             #Prints either the top or bottom 'best_worst_rows[i]' rows of the DataFrame, depending on the if it's printing
184             #for '--best' or '--worst'
185             if args.mtd and len(args.mtd.split(",")) == 2:
186                 if i == 0:
187                     print("\nBest performing " + str(best_worst_rows[i]) + " rows by " + col_names[i][4] +
188                           " and " + col_names[i][5] + "...\n")
189                     df = df.iloc[:best_worst_rows[i], :]
```

```
                print(df)

            else:
                print("\nWorst performing " + str(best_worst_rows[i]) + " rows by " + col_names[i][4] +
                      " and " + col_names[i][5] + "...\n")
                df = df.iloc[(best_worst_rows[i]*-1):, :]
                #Reverses the order so the worst rows appear first
                df = df.iloc[::-1]
                print(df)
        else:
            if i == 0:
                print("\nBest performing " + str(best_worst_rows[i]) + " rows by " + col_names[i][4] + "...\n")
                df = df.iloc[:best_worst_rows[i], :]
                print(df)

            else:
                print("\nWorst performing " + str(best_worst_rows[i]) + " rows by " + col_names[i][4] + "...\n"
                      )
                df = df.iloc[(best_worst_rows[i] * -1):, :]
                #Reverses the order so the worst rows appear first
                df = df.iloc[::-1]
                print(df)
```

A.13 '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, mean_absolute_error, r2_score
from sklearn.metrics import confusion_matrix as cm
from sklearn.preprocessing import StandardScaler
import pyexcel as pe
from matplotlib import pyplot as plt
from math import floor
import data_balancer
from settings import local_dir, source_dir, output_dir, sub_dirs, sub_sub_dirs, raw_measurements, batch_size, \
    nsaa_6mw_path, model_shapes_path
from ft_sel_red import ft_red_select



#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)

#Necessary for imported variable to work as global
source_dir = source_dir

"""Section below encompasses all the arguments that are required to setup and train the model. This includes the
name of the directory to source the files from to train the model, the type of files that this directory contains,
the name(s) of the file to train the model, and the choice of training model (i.e. what the model should be setup
to predict as output). Option arguments include those to specify the sequence length the RNN should deal with, the
percentage of sequence overlap between train/test samples, the number of epochs to use, and whether to write the
settings to the output results file (these go to default values if not specified)."""
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'. "
                                "Alternatively, specify 'cnn_data' if building models from the data used for 'cnn' project.")
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). Alternatively, supply a name of a "
                               "measurement (e.g. 'position', 'velocity', 'jointAngle', etc.) if the file is to be "
                               "trained on raw measurements. Additionally, if user wishes to concatenate file types "
                               "along the 'features' dimension, specify multiple file types separated by commas.")
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 "
                                   "
```

```
48                                  "regression to predict the overall north star scores for sequences, 'acts' ↘
                                        for "
49                                  "multi-target classification of possible NSAA activities that have taken ↘
                                        place "
50                                  "in the sequence and what their corresponding individual NSAA would be, or '↘
                                        indiv' "
51                                  "for predicting a score of 0, 1, or 2 from 'single-act' source files.")
52  parser.add_argument("--seq_len", type=float, nargs="?", const=1,
53                      help="Option to split a source file (be it a raw joint-angle file or a JA/DC/AD stats ↘
                           features file) "
54                           "into multiple parts for training purposes, where '--seq_len' is the number of 'rows' ↘
                                of data "
55                           "to correspond to each file label/score(s).")
56  parser.add_argument("--seq_overlap", type=float, nargs="?", const=1,
57                      help="Option to specify the proportion of overlap in divided up sequences. E.g. set to ↘
                           '0.75' to "
58                           "have each of e.g. 742 sequences to overlap 75% of each other, resulting in 927 new ↘
                                sequences.")
59  parser.add_argument("--discard_prop", type=float, nargs="?", const=1,
60                      help="Option to discard every nth element to reduce the size of the sequence length while ↘
                           keeping "
61                           "context high (e.g. if seq_len=180 and discard_prop=0.2, then it discards every 5th ↘
                                element of "
62                           "the sequences and is left with a sequence length of 144 but with the context window ↘
                                of the "
63                           "original sequence length of 180).")
64  parser.add_argument("--write_settings", type=bool, nargs="?", const=True, default=False,
65                      help="Option to write the hyperparameters of the RNN directly to the RNN results .csv file ↘
                           in "
66                           "the appropriate row.")
67  parser.add_argument("--create_graph", type=bool, nargs="?", const=True, default=False,
68                      help="Option to create a graph of the true values against the predicted values and save ↘
                           them "
69                           "to the 'Graphs' subdirectory within the 'documentation' directory.")
70  parser.add_argument("--epochs", type=int, nargs="?", const=1,
71                      help="Option to set number of epochs to use in this run of the model.")
72  parser.add_argument("--other_dir", type=str, nargs="?", const=True, default=False,
73                      help="Option to include an additional source directory as part of the data to train the ↘
                           model on. "
74                           "Must not be the same as 'dir' and must be one '6minwalk-matfiles', 6MW-matFiles', or ↘
                                'NSAA'.")
75  parser.add_argument("--leave_out", type=str, nargs="?", const=True, default=False,
76                      help="Option to specify the short file names of the file to leave out of the train and test↘
                           set "
77                           "altogether, and thus use it reliably in 'model_predictor.py'. Note that it removes ↘
                                ALL files "
78                           "with a matching short name, so '--leave_out=D4' would exclude 'D4', 'd4' and 'D4v2' ↘
                                files. "
79                           "Alternatively, specify short files names so particular files are left out (e.g. "
80                           "'--leave_out=HC8-008' would leave 'HC8-009' in the data set. Split multiple files to ↘
                                be left "
81                           "out by commas.")
82  parser.add_argument("--balance", type=str, nargs="?", const=True, default=False,
83                      help="Option to balance the data based on overall NSAA scores. Set as 'up' to upsample ↘
                           samples with "
84                           "overall NSAA y-labels to that of the most frequent value or 'down' to downsample to ↘
                                that of "
85                           "the least frequent value.")
86  parser.add_argument("--use_frc", type=bool, nargs="?", const=True, default=False,
87                      help="Option to use the 'FRC_' files for 'AD' files instead of 'FR_' files, i.e. use files ↘
                           where "
88                           "dimensionality reduction has been applied the same way over all files rather than on ↘
                                a "
89                           "file-by-file basis.")
90  parser.add_argument("--use_sac", type=bool, nargs="?", const=True, default=False,
91                      help="Option to use the concatenated files for 'single-act' files over all activities per ↘
                           subject "
92                           "instead of separate files for each activity.")
93  parser.add_argument("--standardize", type=bool, nargs="?", const=True, default=False,
94                      help="Option to standardize all the data (i.e. so each feature column of the total data set↘
                           "
95                           "(following splitting into sequences) is N(0, 1).")
96  parser.add_argument("--noise", type=float, nargs="?", const=True, default=False,
97                      help="Option to add Gaussian-distributed noise to each of the features. Specify a ↘
                           proportion of "
98                           "N(0, 1) noise to the add to the data set.")
99  parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
100                     help="Option that is only set if the script is run from a batch file to access the external↘
                           files "
101                          "in a correct way.")
102 parser.add_argument("--pca", type=str, nargs="?", const=True, default=False,
103                     help="Specify if wishing reduce the feature size of the data set training the model to the ↘
                           number "
104                          "of features specified by the value passed to this argument.")
105 parser.add_argument("--balance_allmatfiles", type=int, nargs="?", const=True, default=False,
106                     help="Specify this to only use a smaller sample of the overall 'allmatfiles' data set, as ↘
                           the "
107                          "base contains ~8x as many files as NSAA. Value given is the maximum number of files ↘
                                per "
108                          "subject that we can draw from the 'allmatfiles' directory.")
109 parser.add_argument("--balance_nmb", type=int, nargs="?", const=True, default=False,
110                     help="Specify this to only use a smaller sample of the overall 'nmb' data set, as the "
111                          "base contains ~8x as many files as NSAA. Value given is the maximum number of files ↘
                                per "
```

```
                        "subject that we can draw from the 'allmatfiles' directory.")
parser.add_argument("--no_testset", type=bool, nargs="?", const=True, default=False,
                    help="Specify this to set the train/test ratio to 0 so to use all the available data as training. "
                        "Note that this results in no results being written to console or the 'RNN Results' directory.")
parser.add_argument("--leave_out_version", type=str, nargs="?", const=True, default=False,
                    help="Specify this with the name of a version of subjects (e.g. 'V2') to leave out of the files "
                        "used to train the models. Use this if wish to train models on one version of subjecst to "
                        "then evaluate model generalisation to other versions of already-seen subjects.")
args = parser.parse_args()

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

#If no optional argument given for '--seq_overlap', defaults to seq_overlap = 0, i.e. defaults to no overlap of sequences
if not args.seq_overlap:
    args.seq_overlap = 0

#Ensures the sequence overlap isn't '1', which would result in an infinite number of sequences as the sequence window
#would never 'move along' the data
if args.seq_overlap == 1:
    print("Can't set overlap to be 100% of each sequence...")
    sys.exit()

#If no optional argument given for '--discard_prop', defaults to discard_prop = 0, i.e. defaults to not discarding
#any parts of the sequences
if not args.discard_prop:
    args.discard_prop = 0

#Ensures that, if '--balance' is set, it is either 'up' to upsample the data or 'down' to downsample the data
if args.balance:
    if not args.balance == "up" and not args.balance == "down":
        print("Optional arg ('--balance') must be set to either 'up' or 'down'.")
        sys.exit()


#Location at which to store the created model that will be used by 'model_predictor.py'
model_path = local_dir + "output_files\\rnn_models\\" + '-'.join(sys.argv[1:]) + "\\model.ckpt"


#Available choices of model outputs for 'choice' argument to take and available measurement names for 'ft' to take
#if not one of 'sub_sub_dirs'
choices = ["dhc", "overall", "acts", "indiv"]
choice = None

#Global variable to hold strings that should be printed at the end of 'rnn' running if '--balance' is set
balance_strs = []

"""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)
num_lstm_cells = 128
num_rnn_hidden_layers = 2
learn_rate = 0.001
num_epochs = 20
test_ratio = 0.2
num_acts = 17

#Only sets 'num_epochs' to different value if '--epochs' optional argument is provided
if args.epochs:
    num_epochs = int(args.epochs)

#Only sets the 'test_ratio' to 0 (i.e. use all available data for model training) if '--no_testset' optional
#argument is provided
if args.no_testset:
    test_ratio = 0

#If '--other_dir' is set, make sure it's a permitted name before continuing
if args.other_dir:
    if not args.other_dir + "\\" in sub_dirs or args.other_dir == args.dir:
        print("Optional arg '--other_dir' must be one of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', "
              "'NMB', or 'direct_csv' and must not be the same name as 'dir'....")
        sys.exit()

#Sets the paths for external files based on the whether the script was called from a batch file or not
nsaa_6mw_path = "..\\" + nsaa_6mw_path if args.batch else nsaa_6mw_path
model_shapes_path = "..\\" + model_shapes_path if args.batch else model_shapes_path



def preprocessing(dir, ft):
    """
```

```
198         :return given a short file name for 'fn' command-line argument, finds the relevant file in 'source_dir' and
199         adds it to 'file_names' (or set 'file_names' to all file names in 'source_dir'), reads in each file name in
200         'file_names' as .csv's into DataFrames, create corresponding 'y-labels' (with one 'y-label' corresponding to
201         each row in the .csv, with it being a 1 if it's from a file beginning with 'D' or 0 otherwise), and shuffling/
202         splitting them into training and testing x- and y-components
203     """
204     file_names = []
205     x_data, y_data = [], []
206     #Declare 'sequence_length' as global to be able to modify later on if '--discard_prop' is set
207     global sequence_length
208     #Appends the sub_dir name to 'source_dir' if it's one of the allowed names
209     global source_dir
210     #Sets 'y_value_balance' and 'y_data_balance' to be used in function scope if '--balance' is set
211     y_label_balance, y_data_balance = None, []
212     #If 'dir'=cnn, call the 'cnn_preprocessing' function instead and return the x/y train/test splits that the
213     #function returns
214     if dir == "cnn_data":
215         return cnn_preprocessing()
216 
217     if dir + "\\" in sub_dirs:
218         source_dir += dir + "\\"
219     else:
220         print("First arg ('dir') must be a name of a subdirectory within source dir and must be one of "
221               "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'NMB', or 'direct_csv'.")
222         sys.exit()
223     #Change 'source_dir' to point to the directory of raw measurement files, single-act raw measurements files,
224     #or another type of file (e.g. 'AD' or 'JA')
225     if dir == "NSAA" and args.choice == "indiv" and ft in raw_measurements:
226         source_dir = local_dir + "NSAA\\matfiles\\act_files\\" + ft + "\\"
227     elif dir == "NSAA" and ft in raw_measurements and args.use_sac:
228         source_dir = local_dir + "NSAA\\matfiles\\act_files_concat\\" + ft + "\\"
229     elif dir == "NSAA" and args.choice == "indiv" and ft + "\\" in sub_sub_dirs:
230         source_dir += ft + "\\act_files\\"
231     elif dir == "NSAA" and ft + "\\" in sub_sub_dirs and args.use_sac:
232         source_dir += ft + "\\act_files_concat\\"
233     elif ft + "\\" in sub_sub_dirs and dir != "6MW-matFiles":
234         source_dir += ft + "\\"
235     elif dir == "allmatfiles" and ft in raw_measurements:
236         source_dir = local_dir + "allmatfiles\\" + ft + "\\"
237     elif dir == "NSAA" and ft in raw_measurements:
238         source_dir = local_dir + "NSAA\\matfiles\\" + ft + "\\"
239     elif dir == "6minwalk-matfiles":
240         if ft == "AD" or ft == "JA":
241             source_dir += + ft + "\\"
242         elif ft in raw_measurements:
243             source_dir = local_dir + "6minwalk-matfiles\\all_data_mat_files\\" + ft + "\\"
244         else:
245             print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
246                   "\'JA', or \'DC\' (unless dir is give as 'NSAA', where 'ft' can be a measurement name).")
247             sys.exit()
248     elif dir == "6MW-matFiles" or dir == "NMB":
249         if ft == "AD":
250             source_dir = local_dir + "\\output_files\\" + dir + "\\AD\\"
251         elif ft in raw_measurements:
252             source_dir = local_dir + dir + "\\" + ft + "\\"
253         else:
254             print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
255                   "\'JA', or \'DC\' (unless dir is give as 'NSAA', where 'ft' can be a measurement name).")
256             sys.exit()
257     else:
258         print("Second arg ('ft') must be a name of a sub-subdirectory within source dir and must be one of \'AD\',"
259               "\'JA', or \'DC\' (unless dir is give as 'NSAA', where 'ft' can be a measurement name).")
260         sys.exit()
261 
262     #Appends to the list of files the single file name corresponding to the 'fn' argument or all available files within
263     #'source_dir' if 'fn' is 'all'
264     if args.fn.lower() != "all":
265         if any(args.fn == s.upper().split("-")[0] for s in os.listdir(source_dir)):
266             file_names.append([s for s in os.listdir(source_dir) if args.fn in s][0])
267         else:
268             print("Cannot find '" + str(args.fn) + "' in '" + source_dir + "'")
269             sys.exit()
270     else:
271         try:
272             file_names = [fn for fn in os.listdir(source_dir)]
273         except FileNotFoundError:
274             print(dir, "not a valid 'dir' name for when choice='" + args.choice + "'...")
275             sys.exit()
276 
277     #Sets 'choice' equal to the 'choice' argument if it's one of the allowed output RNN choices (i.e. 'dhc', 'acts',
278     #'indiv', or 'overall')
279     global choice
280     if args.choice in choices:
281         choice = args.choice
282     else:
```

```
283             print("Must provide a choice of 'dhc' for simple binary classification of sequences, 'overall' for ↘
                    single-target "
284                   "regression to predict the overall north star scores for sequences, 'acts' for multi-target "
285                   "classification of possible NSAA activities that have taken place in the sequence and what their ↘
                          "
286                   "corresponding individual NSAA would be, or 'indiv' for predicting a score of 0, 1, or 2 from "
287                   "'single-act' source files.")
288             sys.exit()
289
290         #Ensures that only the written files from feature select/reduct script are used if present (i.e. if the ↘
                directory
291         #we are concerned with is not within 'direct_csv')
292         if dir != "direct_csv" and source_dir.startswith(local_dir + "output_files\\"):
293             if not args.use_frc:
294                 file_names = [fn for fn in file_names if fn.startswith("FR_")]
295             else:
296                 file_names = [fn for fn in file_names if fn.startswith("FRC_")]
297         elif ft == "AD":
298             if not args.use_frc:
299                 file_names = [fn for fn in file_names if fn.startswith("FR_")]
300             else:
301                 file_names = [fn for fn in file_names if fn.startswith("FRC_")]
302
303         if not file_names:
304             print("No files found in given directory for given args...")
305             sys.exit()
306
307         #Removes any file that contains in the name the optional argument '--leave_out', split by commas
308         if args.leave_out:
309             lo_names = args.leave_out.split(",")
310             for lo_n in lo_names:
311                 file_names = [fn for fn in file_names if lo_n not in fn]
312
313         #Removes any file that contains in the name the optional argument '--leave_out_version'
314         if args.leave_out_version:
315             file_names = [fn for fn in file_names if args.leave_out_version not in fn]
316
317         #If the '--balance_allmatfiles' optional argument is set, rebalance the 'allmatfiles' data set to use a ↘
                reduced
318         #number of files
319         if dir == "allmatfiles" and args.balance_allmatfiles:
320             print("Initial number of files within 'allmatfiles': " + str(len(file_names)))
321             #Ensures that the number of files selected for each subject name in 'allmatfiles' are randomly chosen ↘
                    for a
322             #given subject, as opposed to the first occuring files
323             np.random.seed(42)
324             np.random.shuffle(file_names)
325
326             #Setups a dictionary to count the number of files we have selected per subject and an empty list to ↘
                    hold the
327             #selected file names
328             fn_counts = {fn.split("jointangle")[1].split("-")[0]:0 for fn in file_names}
329             new_file_names = []
330
331             #For each file name, adds it to the new list of file names to include if we haven't reached the maximum↘
                    number
332             #allowed ('--balance_allmatfiles') for that subject
333             for fn in file_names:
334                 short_fn = fn.split("jointangle")[1].split("-")[0]
335                 if fn_counts[short_fn] < args.balance_allmatfiles:
336                     new_file_names.append(fn)
337                     fn_counts[short_fn] += 1
338
339             #Sets the newly chosen file names to the list of files from which to build the model
340             file_names = new_file_names
341             print("Reduced number of files within 'allmatfiles' to use to train models (max = " +
342                   str(args.balance_allmatfiles) + " per subject) : " + str(len(file_names)))
343         elif dir == "NMB" and args.balance_nmb:
344             print("Initial number of files within 'nmb': " + str(len(file_names)))
345             #Ensures that the number of files selected for each subject name in 'nmb' are randomly chosen for a
346             #given subject, as opposed to the first occuring files
347             np.random.seed(42)
348             np.random.shuffle(file_names)
349
350             #Setups a dictionary to count the number of files we have selected per subject and an empty list to ↘
                    hold the
351             #selected file names
352             fn_counts = {fn.split("-")[0]: 0 for fn in file_names}
353             new_file_names = []
354
355             #For each file name, adds it to the new list of file names to include if we haven't reached the maximum↘
                    number
356             #allowed ('--balance_nmb') for that subject
357             for fn in file_names:
358                 short_fn = fn.split("-")[0]
359                 if fn_counts[short_fn] < args.balance_nmb:
360                     new_file_names.append(fn)
361                     fn_counts[short_fn] += 1
362
363             # Sets the newly chosen file names to the list of files from which to build the model
364             file_names = new_file_names
365             print("Reduced number of files within 'NMB' to use to train models (max = " +
366                   str(args.balance_nmb) + " per subject) : " + str(len(file_names)))
367
```

```
368     #For each file name that we are dealing with (all files names in 'source_dir' if 'fn' is 'all', else a single
369     #file name), adds 'y' labels based on what type of model output we are training for and divide up both 'x' and 'y'
370     #data into sequences
371     for file_name in file_names:
372         print("Extracting '" + file_name + "' to x_data and y_data....")
373         #Read in the data from the corresponding .csv
374         data = pd.read_csv(source_dir + file_name)
375 
376         #If the model output type is 'overall', add the NSAA scores if they aren't included already,
377         #and get the overall NSAA score from the first row's first cell in the file as the 'y_label'
378         if choice == "overall":
379             if data.columns.values[1] != "NSS":
380                 try:
381                     data = add_nsaa_scores(data.values)
382                 except KeyError:
383                     print(file_name + " not found as entry in either 'nsaa_6mw_info', skipping...")
384                     continue
385             data = data.values
386             if dir != "direct_csv" and source_dir.startswith(local_dir + "output_files\\"):
387                 y_label = data[0, 1]
388             else:
389                 y_label = data[0, 0]
390         #If the model output type is 'dhc', set 'y_label' to 1 if the first letter of the short file name within the
391         #file name is 'D', else set it to 0 (i.e. if it's a 'HC' file)
392         elif choice == "dhc":
393             data = data.values
394             if dir != "direct_csv" and not source_dir.startswith(local_dir):
395                 y_label = 1 if file_name.split("_")[2][0].upper() == "D" else 0
396             elif ft == "AD":
397                 y_label = 1 if file_name.split("_")[2][0].upper() == "D" else 0
398             elif file_name.startswith("All"):
399                 y_label = 1 if file_name.split("_")[0].split("-")[1][0].upper() == "D" else 0
400             elif source_dir.startswith(local_dir) and ft != "AD":
401                 y_label = 1 if file_name.split("_")[0][0].upper() == "D" else 0
402             else:
403                 y_label = 1 if file_name.split("_")[1][0].upper() == "D" else 0
404         #If the model output type is 'acts', add the NSAA scores if they aren't included already, and get the
405         #individual NSAA activity scores from 17 cells in the first row of the data
406         elif choice == "acts":
407             if data.columns.values[1] != "NSS":
408                 try:
409                     data = add_nsaa_scores(data.values)
410                 except KeyError:
411                     print(file_name + " not found as entry in either '6mw_matfiles.xlsx', 'nsaa_matfiles.xlsx',"
412                                       "or 'KineDMD data updates Feb 2019.xlsx', 'skipping...")
413                     continue
414             data = data.values
415             if dir != "direct_csv" and not source_dir.startswith(local_dir):
416                 y_label = data[0][2:19]
417             elif ft == "AD":
418                 y_label = data[0][2:19]
419             else:
420                 y_label = data[0][1:18]
421         #If the model output type is 'indiv', add the NSAA scores if they aren't included already, get the name of the
422         #activity the file is sourced from (e.g. 'act5') and, based on this, retrieve the value of the correct column
423         #from the first row of data (e.g. retrieve the 5th column of the first row if file contains 'act5')
424         else:
425             if data.columns.values[1] != "NSS":
426                 try:
427                     data = add_nsaa_scores(data.values)
428                 except KeyError:
429                     print(file_name + " not found as entry in either '6mw_matfiles.xlsx', 'nsaa_matfiles.xlsx',"
430                                       "or 'KineDMD data updates Feb 2019.xlsx', 'skipping...")
431                     continue
432             data = data.values
433             y_label = data[0][int(file_name.split(".")[0].split("_")[1].split("act")[1])]
434 
435         #If the '--balance' optional argument is set, then set the 'y_label_balance' to overall NSAA score for the file,
436         #regardless of the 'choice' arg, to use as a means to balance the data
437         if args.balance:
438             y_label_balance = data_balancer.ext_label_dist(file_name=file_name, batch=args.batch)
439 
440         #Determine the number of data splits needed based on the size of the data file and the desired sequence length,
441         #including rounding it down and accounting for the sequence overlap proportion)
442         num_data_splits = int(len(data) / sequence_length)
443         num_data_splits = int(num_data_splits * (1/(1-args.seq_overlap)))
444         start, end = 0, 0
445         #For each desired sequence, determine the start and end positions of the sequence in 'data', extract this from
446         #the body of 'data', append this to 'x_data', and append the previously-determined 'y_label' to 'y_data'
447         for i in range(num_data_splits):
448             end = start + sequence_length
449             #Prevents moving window from 'clipping' the end of the data rows and getting a number of rows of
450             #less than 'sequence_length'
```

```
451             if end > len(data):
452                 continue
453             #Discards every 'nth' row of a sequence if the user sets the optional '--discard_prop' argument to reduce
454             #the sequence size while keeping the original context window the same
455             split_data = data[start:end]
456             if args.discard_prop:
457                 if args.discard_prop == 0:
458                     print("Cannot set '--discard_prop to '0' as would give a zero division error when trying to"
459                           "take the 'nth' element...")
460                     sys.exit()
461                 if args.discard_prop <= 0.5:
462                     split_data = np.asarray([row for i, row in enumerate(split_data)
463                                              if i % floor(1/args.discard_prop) != 0])
464                 else:
465                     split_data = np.asarray([row for i, row in enumerate(split_data)
466                                              if i % floor(1/round((1-args.discard_prop), 5)) == 0])
467
468             if dir == "direct_csv" and choice == "dhc":
469                 x_data.append(split_data[:, 1:])
470             elif dir == "direct_csv":
471                 x_data.append(split_data[:, 19:])
472             elif source_dir.startswith(local_dir) and choice == "dhc" and not ft == "AD":
473                 x_data.append(split_data[:, 1:])
474             elif source_dir.startswith(local_dir) and "output_files" not in source_dir:
475                 x_data.append(split_data[:, 19:])
476             else:
477                 x_data.append(split_data[:, 21:])
478             y_data.append(y_label)
479             #If we wish to balance the data, append the overall NSAA score to a separate list used to balance the data
480             if args.balance:
481                 y_data_balance.append(y_label_balance)
482             #Note: overlap lengths are rounded DOWN via 'int'
483             start += int(sequence_length*(1-args.seq_overlap))
484
485     global x_shape
486     global y_shape
487     x_shape = np.shape(x_data)
488     y_shape = np.shape(y_data)
489     print("X shape =", x_shape)
490     print("Y shape =", y_shape)
491     #Sets the global 'sequence_length' if '--discard_prop' is called to ensure RNN is setup with the correct
492     #placeholder variable shape
493     if args.discard_prop:
494         sequence_length = len(x_data[0])
495
496     #Rebalances the data based on the value of 'args.balance'
497     if args.balance:
498         if args.balance == "up":
499             x_data, y_data, balance_s = data_balancer.upsample(x_data, y_data, y_data_balance)
500         else:
501             x_data, y_data, balance_s = data_balancer.downsample(x_data, y_data, y_data_balance)
502         #Sets the global variable to print out balance strings at a later point
503         global balance_strs
504         balance_strs = balance_s
505         x_shape = np.shape(x_data)
506         y_shape = np.shape(y_data)
507         print("Balanced X shape =", x_shape)
508         print("Balanced Y shape =", y_shape)
509
510     #If the '--standardize' optional argument is given, reshape the 'x' data into a 2D array, standardize
511     #each of the features, and reshape it back into its original shape (note: np.concatenate and np.reshape aren't used
512     #in order to ensure the data is written back as we would expect with the same number of lines)
513     if args.standardize:
514         x_data = [sample for sequence in x_data for sample in sequence]
515         print("Standardizing the data set...")
516         x_data = StandardScaler().fit_transform(x_data)
517         x_data = [[x_data[i*x_shape[1] + j] for j in range(x_shape[1])] for i in range(x_shape[0])]
518     #If the '--noise' argument is given, add Gaussian noise to every feature of every sample of every sequence
519     #in the data set
520     if args.noise:
521         x_data = [sample for sequence in x_data for sample in sequence]
522         print("Adding Gaussian noise to the data set...")
523         x_data += np.random.normal(0, np.std(np.array(x_data, dtype=np.float64), axis=0) * args.noise, np.shape(x_data))
524         x_data = [[x_data[i * x_shape[1] + j] for j in range(x_shape[1])] for i in range(x_shape[0])]
525
526     #Appends the arguments that were used to invoke the model and its sequence length to a file that stores
527     #the sequence lengths (to be used by the 'model_predictor.py' script
528     model_shape = pd.read_excel(model_shapes_path)
529     new_model_shape = model_shape.append(
530         {"dir": dir, "ft": ft, "measure": args.choice, "seq_len": x_shape[1]}, ignore_index=True)
531     new_model_shape.to_excel(model_shapes_path, index=False)
532
533     return x_data, y_data
534
535
536
537 def cnn_preprocessing():
538     """
539         :return:given the name for a 'left-out' file (i.e. the subject to be used as a testing set) via the '--
```

```
            left_out'
        optional argument (required here), the relevant training and testing .csv files (one for each) are loaded in
        and sequences are extracted from this and given corresponding 'y' labels based on the 'choice' arg, followed by
        shuffling these sets are passing them along to the calling 'preprocessing' function, which terminates with the
        returned data from this function
    """

    global sequence_length
    cnn_data_dir = local_dir + "cnn_data\\"

    train_test_data = [[[], []], [[], []]]
    if any(dir for dir in os.listdir(cnn_data_dir) if dir.startswith(args.leave_out)):
        leave_out_dir = [dir for dir in os.listdir(local_dir + "cnn_data\\") if dir.startswith(args.leave_out)][0]
    else:
        print("Must provide '--leave_out' arg as name of subject to leave out of training...")
        sys.exit()

    global choice
    if args.choice == "dhc":
        choice = "dhc"
    elif args.choice == "overall":
        choice = "overall"
    else:
        print("For 'dir'=cnn, the 'choice' arg must be one of 'overall' or 'dhc'...")
        sys.exit()



    for i, t_t in enumerate(("train", "test")):
        file_name = cnn_data_dir + leave_out_dir + "\\" + t_t + ".csv"
        file_data = pd.read_csv(file_name)
        print("\n" + t_t.title() + " file '" + file_name + "' of shape: " + str(np.shape(file_data)))
        print("Extracting '" + file_name + "' data...")
        x_data = file_data.iloc[:, 1:29].values
        if choice == "dhc":
            y_data = file_data.iloc[:, 29].values
        else:
            y_data = file_data.iloc[:, 30].values

        #Determine the number of data splits needed based on the size of the data file and the desired sequence length,
        #including rounding it down and accounting for the sequence overlap proportion)
        num_data_splits = int(len(x_data) / sequence_length)
        num_data_splits = int(num_data_splits * (1 / (1 - args.seq_overlap)))

        print("Splitting '" + file_name + "' into sequences....")
        start, end = 0, 0
        # For each desired sequence, determine the start and end positions of the sequence in 'data', extract this from
        # the body of 'data', append this to 'x_data', and append the previously-determined 'y_label' to 'y_data'
        for j in range(num_data_splits):
            end = start + sequence_length
            # Prevents moving window from 'clipping' the end of the data rows and getting a number of rows of
            # less than 'sequence_length'
            if end > len(x_data):
                continue
            # Discards every 'nth' row of a sequence if the user sets the optional '--discard_prop' argument to reduce
            # the sequence size while keeping the original context window the same
            split_data = x_data[start:end]
            if args.discard_prop:
                if args.discard_prop == 0:
                    print("Cannot set '--discard_prop to '0' as would give a zero division error when trying to "
                          "take the 'nth' element...")
                    sys.exit()
                if args.discard_prop <= 0.5:
                    split_data = np.asarray([row for j, row in enumerate(split_data)
                                             if j % floor(1 / args.discard_prop) != 0])
                else:
                    split_data = np.asarray([row for j, row in enumerate(split_data)
                                             if j % floor(1 / round((1 - args.discard_prop), 5)) == 0])

            train_test_data[i][0].append(split_data)
            #Appends the label that corresponds to the label of the middle row of the sequence
            train_test_data[i][1].append(y_data[int((end + start)/2)])
            # Note: overlap lengths are rounded DOWN via 'int'
            start += int(sequence_length * (1 - args.seq_overlap))


    global x_shape
    global y_shape
    x_shape = np.shape(np.concatenate((train_test_data[0][0], train_test_data[1][0]), axis=0))
    y_shape = np.shape(np.concatenate((train_test_data[0][1], train_test_data[1][1]), axis=0))

    print("\n")
    print("X train shape =", np.shape(train_test_data[0][0]))
    print("Y train shape =", np.shape(train_test_data[0][1]))
    print("X test shape =", np.shape(train_test_data[1][0]))
```

```
624         print("Y test shape =", np.shape(train_test_data[1][1]))
625
626         # Rebalances the data based on the value of 'args.balance'
627         if args.balance:
628             if args.balance == "up":
629                 train_test_data[0][0], train_test_data[0][1], balance_s = \
630                     data_balancer.upsample(train_test_data[0][0], train_test_data[0][1], train_test_data[0][1])
631             else:
632                 train_test_data[0][0], train_test_data[0][1], balance_s = \
633                     data_balancer.downsample(train_test_data[0][0], train_test_data[0][1], train_test_data[0][1])
634             # Sets the global variable to print out balance strings at a later point
635             global balance_strs
636             balance_strs = balance_s
637             x_shape = np.shape(train_test_data[0][0])
638             y_shape = np.shape(train_test_data[0][1])
639             print("Balanced X train shape =", x_shape)
640             print("Balanced Y train shape =", y_shape)
641
642         # Sets the global 'sequence_length' if '--discard_prop' is called to ensure RNN is setup with the correct
643         # placeholder variable shape
644         if args.discard_prop:
645             sequence_length = len(train_test_data[0][0][0])
646
647         #Shuffles the training and testing data sets, as it isn't automatically done in this 'preprocessing' ↘
                function
648         #due to not using the 'train_test_split' function
649         z = list(zip(train_test_data[0][0], train_test_data[0][1]))
650         np.random.shuffle(z)
651         x_train, y_train = zip(*z)
652         z = list(zip(train_test_data[1][0], train_test_data[1][1]))
653         np.random.shuffle(z)
654         x_test, y_test = zip(*z)
655
656         return np.concatenate((x_train, x_test), axis=0), np.concatenate((y_train, y_test), axis=0)
657
658
659
660 def add_nsaa_scores(file_df):
661     """
662     :param 'file_df', which contains the values in a 2D numpy array, to have the values NSAA scores appended on↘
            each
663     of its rows
664     :return: the same data as before, but with the overall and individual NSAA scores appended at the beginning↘
            of
665     each row of the data
666     """
667     #To make sure that accepted parameter is as a DataFrame
668     file_df = pd.DataFrame(file_df)
669
670     #For the table of data that we have on the subjects, load in the table, find the columns with ID and
671     #overall NSAA scores, and create a dictionary of matching values, e.g. {'D4': 15, 'D11: 28,...}, with all ↘
            values
672     #from each table
673     nsaa_6mw_tab = pd.read_excel(nsaa_6mw_path)
674     nsaa_6mw_cols = nsaa_6mw_tab[["ID", "NSAA"]]
675     nsaa_overall_dict = dict(pd.Series(nsaa_6mw_cols.NSAA.values, index=nsaa_6mw_cols.ID).to_dict())
676
677     #If the first column's names begins with "jointangle", remove this part
678     if file_df.iloc[0, 0].startswith("jointangle"):
679         for i, row in file_df.iterrows():
680             row[0] = row[0].split("jointangle")[1]
681
682     #Adds column of overall NSAA scores at position 0 of every row of the data values, with the NSAA score ↘
            being
683     #appended determined by the short file name of the data as found at the beginning of each row of the data
684     nss = [nsaa_overall_dict[i.upper()[:-2] if i.upper().endswith("V2") else i.upper()]
685            for i in [j.split("_")[0] for j in file_df.iloc[:, 0].values]]
686     file_df.insert(loc=0, column="NSS", value=nss)
687
688     #Loads the data that contains information about single act NSAA scores from the .xlsx file, extracts the
689     #file names and single-acts columns, and creates a list of label names (i.e. the names of the activities) ↘
            and a
690     #dictionary that maps the label names to a list of single-act scores
691     nsaa_single_dict = {}
692     for name, acts in zip(nsaa_6mw_tab.loc[:, "ID"].values, nsaa_6mw_tab.iloc[:, 5:].values):
693         if not any(np.isnan(acts)):
694             nsaa_single_dict[name] = acts
695     nsaa_act_labels = nsaa_6mw_tab.columns.values[5:]
696
697     #For each label name and for every row, adds the score that is found in the single-acts dictionary for the ↘
            relevant
698     #activity for a given short file name (if it isn't found in the dictionary, add a '2' as we're assuming it'↘
            s a
699     #healthy control patient), add these together, and insert each new row of values at the beginning of the ↘
            old rows
700     #so each now have the additional single-act scores and overall NSAA scores at the beginning of each row and↘
             return it
701     label_sample_map = []
702     for i in range(len(nsaa_act_labels)):
703         inner = []
704         for j in range(len(file_df.index)):
705             fn = file_df.iloc[j, 1].split("_")[0].upper()
706             fn = fn[:-2] if fn.endswith("V2") else fn
707             if fn in nsaa_single_dict:
```

```
708                 inner.append(nsaa_single_dict[fn][i])
709             elif fn.startswith("HC"):
710                 inner.append(2)
711             else:
712                 #If patient isn't found in the table (and thus we don't have info on the individual NSAA scores),
713                 #don't continue with the file and move onto the next one
714                 raise KeyError
715         label_sample_map.append(inner)
716     for i in range(len(nsaa_act_labels)):
717         file_df.insert(loc=(i+1), column=nsaa_act_labels[i], value=label_sample_map[i])
718     return file_df
719
720
721
722 def create_batch_generator(x, y=None, batch_size=64):
723     """
724     :param 'x', which is the data to create batches out of and, if 'y' is supplied as well, create batches out of
725     this as well based on the 'batch_size' provided
726     :return: the next batch of data to the calling function
727     """
728     n_batches = len(x)//batch_size
729     #Ensures 'x' is a multiple of batch size
730     x = x[:n_batches*batch_size]
731     if y is not None:
732         #Ensures 'y' is a multiple of batch size
733         y = y[:n_batches*batch_size]
734     #For each 'batch_size' chuck of x, yield the next part of the 'x' and 'y' data (i.e. the next 'batch size' samples)
735     for ii in range(0, len(x), batch_size):
736         if y is not None:
737             yield x[ii:ii+batch_size], y[ii:ii+batch_size]
738         else:
739             yield x[ii:ii+batch_size]
740
741
742
743 def write_to_csv(trues, preds, output_strs, open_file=False):
744     """
745     :param 'trues', which are a list of 'true' values for each of the test sequences that have been tested on the model,
746     'preds', which are the predicted values corresponding to each 'true' value, and 'output_strs' which are the strings
747     that have been printed to the console and will also be stored in the output file (with 'open_file' being whether to
748     immediately open it upon writing to it
749     :return: no return, but instead writes the sequence numbers, true values, predicted values, console output, and
750     model settings to a new file within 'output_dir'
751     """
752     print("\nWriting true and predicted values to .csv....")
753     df = pd.DataFrame()
754     df["Sequence Number"] = np.arange(1, len(trues)+1)
755     if choice != "acts":
756         df["Predictions"] = preds
757         df["Trues"] = trues
758     else:
759         df["Predictions"] = ["[" + " ".join(str(num) for num in preds[i]) + "]" for i in range(len(preds))]
760         df["Trues"] = ["[" + " ".join(str(num) for num in trues[i]) + "]" for i in range(len(trues))]
761     df["Results"] = ""
762     df.iloc[0, -1] = ' '.join(sys.argv[1:])
763     df.iloc[1, -1] = ", ".join(output_strs)
764     settings = ["X shape = " + str(x_shape), "Y shape = " + str(y_shape), "Test ratio = " + str(test_ratio),
765                 "Sequence length = " + str(sequence_length), "Features length = " + str(len(x_train[0][0])),
766                 "Num epochs = " + str(num_epochs), "Num LSTM units per layer = " + str(num_lstm_cells),
767                 "Num hidden layers = " + str(num_rnn_hidden_layers), "Learning rate = " + str(learn_rate)]
768     df["Settings"] = pd.Series(settings)
769
770     #Create 'output_dir' if it doesn't exist and, if it does exist, remove the existing file and write the data frame
771     #to this file, opening it afterwards if required; if the '--write_settings' argument is given, append the settings
772     #written to this file to the 'RNN Results.ods' file directly rather than manually copying them over
773     if not os.path.exists(output_dir):
774         os.mkdir(output_dir)
775     file_index = 1
776     full_output_name = output_dir + '_'.join(sys.argv[1:]) + "_RNN_trues_preds" + str(file_index) + ".csv"
777     while os.path.exists(full_output_name):
778         file_index += 1
779         full_output_name = "_".join(full_output_name.split("_")[:-1]) + "_" + str(file_index) + ".csv"
780     df.to_csv(full_output_name, index=False, float_format="%.2f")
781     if open_file:
782         os.startfile(full_output_name)
783     if args.write_settings:
784         settings = ["" for i in range(7)] + settings
785         sheet = pe.get_sheet(file_name="..\\RNN Results.ods")
786         sheet.row += settings
787         sheet.save_as("..\\RNN Results.ods")
788     if args.create_graph:
789         fig, ax = plt.subplots()
790         ax.scatter(trues, preds, alpha=0.10)
791         x = np.linspace(*ax.get_xlim())
792         ax.plot(x, x)
```

```
793         plt.title("Plot of true overall NSAA scores against predicted overall NSAA scores")
794         plt.xlabel("True overall NSAA scores")
795         plt.ylabel("Predicted overall NSAA scores")
796         file_name = args.dir + "_" + args.ft + "_" + args.choice + "_trues_preds"
797         plt.savefig("..\\documentation\\Graphs\\" + file_name)
798         plt.gcf().set_size_inches(10, 10)
799         plt.show()
800
801
802
803 class RNN(object):
804     def __init__(self, features_length, seq_len, lstm_size, num_layers, batch_size, learning_rate, num_acts):
805         """
806         :param sets the hyperparameters as 'RNN' object attributes, builds the RNN graph, and initializes
807         the global variables
808         """
809         self.features_length = features_length
810         self.seq_len = seq_len
811         self.lstm_size = lstm_size
812         self.num_layers = num_layers
813         self.batch_size = batch_size
814         self.learning_rate = learning_rate
815         self.num_acts = num_acts
816
817         self.g = tf.Graph()
818         with self.g.as_default():
819             tf.set_random_seed(123)
820             self.build()
821             self.saver = tf.train.Saver()
822             self.init_op = tf.global_variables_initializer()
823
824     def build(self):
825         """
826         :return: no return, but sets up the complete RNN architecture to be called upon initialization
827         """
828         #Placeholders to hold the data that is fed into the RNN (where each batch has shape 'seq_len' x
829         # 'features_length' for 'x' data and a single '1' or '0' for 'y' data)
830         tf_x = tf.placeholder(tf.float32, shape=(self.batch_size, self.seq_len, self.features_length), name='↘
                tf_x')
831         if choice != "acts":
832             tf_y = tf.placeholder(tf.float32, shape=(self.batch_size), name='tf_y')
833         else:
834             tf_y = tf.placeholder(tf.float32, shape=(self.batch_size, self.num_acts), name='tf_y')
835         tf_keepprob = tf.placeholder(tf.float32, name='tf_keepprob')
836
837         #Defines several hidden RNN layers, with 'self.num_layers' layers, 'self.lstm_size' number of cells per
838         #layer, and each implementing dropout functionality
839         cells = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.DropoutWrapper(tf.contrib.rnn.BasicLSTMCell(
840             self.lstm_size), output_keep_prob=tf_keepprob) for i in range(self.num_layers)])
841         #Sets the initial state for the RNN layers
842         self.initial_state = cells.zero_state(self.batch_size, tf.float32)
843         #With the RNN layer architecture defined in 'cells', sets up the layers to feed from input 'x' ↘
                placeholder
844         #into 'cells' and with the above defined 'initial_state'
845         lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, tf_x, initial_state = self.initial_state)
846
847
848         #Defines the cost based on the sigmoid cross entropy with the RNN output and the 'y' labels, along with
849         #the Adam optimizer for the optimizer of choice with a learning rate set by 'self.learning_rate'
850         if choice != "acts":
851             logits = tf.layers.dense(inputs=lstm_outputs[:, -1], units=1, activation=None, name='logits')
852             logits = tf.squeeze(logits, name='logits_squeezed')
853         else:
854             logits = tf.layers.dense(inputs=lstm_outputs[:, -1], units=self.num_acts, activation=None, name='↘
                    logits')
855
856         if choice == "overall" or choice == "indiv":
857             cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=tf_y, predictions=logits), name='cost')
858             predictions = {'cost': cost}
859         elif choice == "dhc":
860             # Adds an output layer that feed from the final values emitted from the 'cells' layers with a ↘
                    single neuron
861             # to classify for a binary value
862             y_proba = tf.nn.sigmoid(logits, name='probabilities')
863             cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf_y, logits=logits), name='↘
                    cost')
864             predictions = {'probabilities': y_proba, 'labels': tf.cast(tf.round(y_proba), tf.int32, name='↘
                    labels')}
865         else:
866             cost = tf.reduce_mean(tf.losses.mean_squared_error(labels=tf_y, predictions=logits), name='cost')
867             predictions = {'labels': tf.cast(tf.round(logits), tf.int32, name='labels'), 'cost': cost}
868         optimizer = tf.train.AdamOptimizer(self.learning_rate)
869         train_op = optimizer.minimize(cost, name='train_op')
870
871
872     def train(self, x_train, y_train, num_epochs):
873         """
874         :param the training data for both 'x' and 'y' data is provided, along with the number of training ↘
                epochs
875         that the training will run for
876         :return no return, but for each epoch, fetches the next batch of data from the 'x' and 'y' training ↘
                sets with
877         a dropout probability and initial state, and feeds it into the RNN for training following the ↘
                architecture
```

```
878             and hyperparameters of 'build()', while printing the loss at every 20 iterations for a given epoch
879             """
880             with tf.Session(graph=self.g) as sess:
881                 sess.run(self.init_op)
882                 iteration = 1
883                 for epoch in range(num_epochs):
884                     state = sess.run(self.initial_state)
885                     for batch_x, batch_y in create_batch_generator(x_train, y_train, self.batch_size):
886                         feed = {'tf_x:0': batch_x, 'tf_y:0': batch_y, 'tf_keepprob:0': 0.5, self.initial_state: ↘
                                state}
887                         loss, _, state = sess.run(['cost:0', 'train_op', self.final_state], feed_dict=feed)
888                         if iteration % 20 == 0:
889                             print("Epoch: %d/%d Iteration: %d | Train loss: %.5f" % (epoch+1, num_epochs, iteration ↘
                                    , loss))
890                         iteration += 1
891                 self.saver.save(sess, model_path)
892             print("\n\n")
893
894
895
896     def predict(self, x_test, return_proba=False):
897             """
898                 :param feeds in the 'x' testing data with which we wish to compute the predicted values (1 or 0) ↘
                        for
899                 each of the 2-dimensional samples (where each sample is 'self.seq_length' x 'self.features_length')↘
                        , with
900                 an option to return the labels rather than the probabilities for assessment purposes
901                 :return: list of predicted values that are the result of feeding through the 'x' testing data ↘
                        through the
902                 now-trained RNN model
903             """
904             preds = []
905             with tf.Session(graph=self.g) as sess:
906                 self.saver.restore(sess, model_path)
907                 test_state = sess.run(self.initial_state)
908                 for ii, batch_x in enumerate(create_batch_generator(x_test, None, batch_size=self.batch_size), 1):
909                     feed = {'tf_x:0': batch_x, 'tf_keepprob:0': 1.0, self.initial_state: test_state}
910                     if return_proba:
911                         pred, test_state = sess.run(['probabilities:0', self.final_state], feed_dict=feed)
912                     elif choice == "overall" or choice == "indiv":
913                         pred, test_state = sess.run(['logits_squeezed:0', self.final_state], feed_dict=feed)
914                     else:
915                         pred, test_state = sess.run(['labels:0', self.final_state], feed_dict=feed)
916                     preds.append(pred)
917             return np.concatenate(preds)
918
919
920
921 #Sets the default sequence length source dir so the originals can be retrieved later after having been modified↘
        by
922 #different 'ft's or by 'altdirs'
923 original_seq_len = sequence_length
924 original_source_dir = source_dir
925
926 #Separates the 'ft' arg into list of file types, separated by a comma
927 fts = [ft for ft in args.ft.split(",")]
928 #Sets the scope of the variables to store all data that is concatenated together to global scope and a variable↘
        to
929 #check whether any data has been entered into this variable yet
930 x_data, y_data = None, None
931 data_entered = False
932
933
934 #For each of the directories that we wish to use as a source of the file data...
935 for dir in args.dir.split(","):
936     #For each of the file types of the argument (separated by commas), preprocesses the data from all files for↘
            this
937     #file type, concatenates it with any previous data from other file types along the features dimension (i.e.↘
            horizontal
938     #concatenation (while checking that it is possible to do so, assuming the first 2 dimensions are the same)
939     fts_x_data, fts_y_data = None, None
940     fts_data_entered = False
941     for ft in fts:
942         sequence_length = original_seq_len
943         source_dir = original_source_dir
944         #Extracts the training and testing data
945         ft_x_data, ft_y_data = preprocessing(dir, ft)
946         #if ft != "AD" or x_data is None:
947         fts_x_data = np.concatenate((fts_x_data, ft_x_data), axis=2) if fts_data_entered else ft_x_data
948         fts_y_data = fts_y_data if fts_y_data else ft_y_data
949         fts_data_entered = True
950         """
951         else:
952             print("'AD' is not a valid 'dir' choice when concatenating file types due to different sized "
953                           "sequence lengths and numbers of samples; not including 'AD'...")
954             continue
955         """
956     x_data = np.concatenate((x_data, fts_x_data), axis=0) if data_entered else fts_x_data
957     y_data = np.concatenate((y_data, fts_y_data), axis=0) if data_entered else fts_y_data
958     data_entered = True
959
960 print("Final concatenated 'X' data shape =", np.shape(x_data))
961 print("Final concatenated 'Y' data shape =", np.shape(y_data))
962
```

```
963 #Repeats the preprocessing for 'other_dir' if specified by setting 'dir' to the value of 'other_dir' and then ↘
      setting
964 #it back to the original value after extracting the data from 'other_dir' and adding it to the 'dir' data
965 if args.other_dir:
966     old_dir = args.dir
967     args.dir = args.other_dir
968     sequence_length = original_seq_len
969     alt_x_data, alt_y_data = preprocessing(args.other_dir, args.ft)
970     x_data = np.concatenate((x_data, alt_x_data), axis=0)
971     y_data = np.concatenate((y_data, alt_y_data), axis=0)
972     args.dir = old_dir
973
974
975 #If the optional arg '--pca' is given, reduces the dimensionality of the data set to the dimensions specified
976 #by the argument via calling the relevant function from 'ft_sel_red.py'
977 if args.pca:
978     try:
979         pca = int(args.pca)
980         print("Reducing the dimensions of data via 'PCA'....")
981         x_data = [sample for sequence in x_data for sample in sequence]
982         x_data, y_data = ft_red_select(x_data, y_data, "pca", False, False, pca)
983         x_data = [[x_data[i * x_shape[1] + j] for j in range(x_shape[1])] for i in range(x_shape[0])]
984         print("Reduced-dim X Shape = " + str(np.shape(x_data)))
985         print("Reduced-dim Y Shape = " + str(np.shape(y_data)))
986     except ValueError:
987         if args.pca != "all":
988             print("Optional arg '--pca' must contain int value or 'all' to keep all features...")
989             sys.exit()
990
991
992
993 print("Splitting the data into training and testing components...")
994 x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, shuffle=True, test_size=test_ratio)
995
996
997
998 #Builds the RNN based on the hyperparameters initially set, and trains the model
999 rnn = RNN(features_length=len(x_train[0][0]), seq_len=sequence_length, lstm_size=num_lstm_cells,
1000                   num_layers=num_rnn_hidden_layers, batch_size=batch_size, learning_rate=learn_rate, num_acts=↘
                        num_acts)
1001
1002 rnn.train(x_train, y_train, num_epochs=num_epochs)
1003
1004 #If the optional arg '--no_testset' is set, finish the the program after completing the training of the model
1005 if args.no_testset:
1006     print("Model built: finishing without testing as '--no_testset' is set...")
1007     sys.exit()
1008
1009 preds = rnn.predict(x_test)
1010 #Ensures the true 'y' values are the same length and the predicted values (so 'preds' and 'y_true' have the ↘
      same shape)
1011 y_true = y_test[:len(preds)]
1012
1013
1014 #Based on what type of model was created (i.e. with which output type), create several strings based on various
1015 #evaluation metrics (i.e. the performance results of the model in question on test data) and append them to a ↘
      list,
1016 #which is then printed to the console and also passed to the 'write_to_csv' so they can be written to file
1017 output_strs = []
1018 if choice == "overall" or choice == "indiv":
1019     output_strs.append("Mean Squared Error = " + str(round(mean_squared_error(y_true=y_true, y_pred=preds), 4))↘
          )
1020     output_strs.append("Mean Absolute Error = " + str(round(mean_absolute_error(y_true=y_true, y_pred=preds), ↘
          4)))
1021     output_strs.append("Root Mean Squared Error = " + str(round(np.sqrt(mean_squared_error(y_true=y_true, ↘
          y_pred=preds)), 4)))
1022     output_strs.append("R^2 Score = " + str(round(r2_score(y_true=y_true, y_pred=preds), 4)))
1023 elif choice == "dhc":
1024     output_strs.append(str(cm(y_true=y_true, y_pred=preds)) + "\n(Note: row names = true vals, col names = pred↘
           vals;"
1025                        + "cm[0,0] = True HC, cm[0,1] = False D, cm[1,0] = False HC, c[1,1] = True D)")
1026     output_strs.append("Test Accuracy = " + str(round(((np.sum(preds == y_true) / len(y_true)) * 100), 2)) + "%↘
          ")
1027 else:
1028     ind_sum, all_sum = 0, 0
1029     for i in range(len(preds)):
1030         ind_sum += np.sum(preds[i] == y_true[i])
1031         all_sum = all_sum + 1 if list(preds[i]) == list(y_true[i]) else all_sum
1032     output_strs.append("Individual Activity Accuracy = " + str(round(((ind_sum / (len(y_true)*num_acts)) * 100)↘
          , 2)) + "%")
1033     output_strs.append("All Activities Accuracy = " + str(round(((all_sum / len(preds)) * 100), 2)) + "%")
1034
1035
1036 print("\n\n")
1037 for output_str in output_strs:
1038     print(output_str)
1039
1040 #Prints to the console the before and after shapes of the data if the '--balance' optional arg is set
1041 if args.balance:
1042     for balance_str in balance_strs:
1043         print(balance_str)
1044
1045 write_to_csv(trues=y_true, preds=preds, output_strs=output_strs)
```

A.14 'settings.py'

```
#This file is only for the storing of variable names that are used throughout 'source' and variables are simply
#imported from here, rather than repeatedly re-defining things like 'local_dir' throughout the script. It also means
#there is only one point needed to change the variables in the directory rather than every script that uses it.


#Note: CHANGE THIS to location of the 3 sub-directories' encompassing directory local to the user.
local_dir = "C:\\msc_project_files\\"

#Other locations and sub-dir nameswithin 'local_dir' that will contain the files we need, as dictated by assuming the
#user has previously run the required scripts (e.g. 'comp_stat_vals', 'ext_raw_measures', etc.)
source_dir = local_dir + "output_files\\"
output_dir = local_dir + "output_files\\RNN_outputs\\"
model_dir = local_dir + "output_files\\rnn_models\\"
sub_dirs = ["6minwalk-matfiles\\", "6MW-matFiles\\", "NSAA\\", "direct_csv\\", "allmatfiles\\", "left-out\\", "NMB\\"]
sub_sub_dirs = ["AD\\", "JA\\", "DC\\"]

#Other paths to documentation files that are referenced in several locations within 'source'
doc_path = "..\\documentation\\"
model_pred_path = doc_path + "model_predictions.csv"
results_path = doc_path + "RNN Results.xlsx"
nsaa_6mw_path = doc_path + "nsaa_6mw_info.xlsx"
model_shapes_path = doc_path + "model_shapes.xlsx"
nsaa_subtasks_path = doc_path + "nsaa_17subtasks_matfiles.csv"

#Types of files that are used to train models on
file_types = ["AD", "position", "velocity", "acceleration", "angularVelocity", "angularAcceleration",
              "sensorFreeAcceleration", "sensorMagneticField", "jointAngle", "jointAngleXZY"]
#The types of 'output' that models are trained towards
output_types = ["acts", "dhc", "overall"]



"""Section below includes names of parts of data that are defined outside the scope of this project (i.e. defined
as part of the bodysuit equipment and that are given to us as part of the bodysuit user manual)"""

#List of possible measurements to extract from the source .mat files. Note that 'orientation' and 'sensorOrientation'
#are NOT included due to having '92' and '68' dimensions, respectively and not being in form of values for x,y,z dims
raw_measurements = ["position", "velocity", "acceleration", "angularVelocity", "angularAcceleration",
                    "sensorFreeAcceleration", "sensorMagneticField", "jointAngle", "jointAngleXZY"]

axis_labels = ["X", "Y", "Z"]
short_file_types = ["JA", "AD", "DC"]


#Below 3 lists are labels for the 23 segments, 22 joints, and 17 sensors, respectively, 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}


#Other random constants that are referenced in several points in 'source'
batch_size = 64
sampling_rate = 60      #In Hz
```

A.15 'test_altdirs.py'

```
import argparse
import sys
import os
```

```
4  from settings import local_dir, sub_dirs, file_types
5
6
7  """Section below specifies the arguments that are required to run the script, which needs a directory from which to
8  source the files to test, the directories that are used to train the models that we shall be testing on, and
9  the names of the file types (i.e. measurements, be they raw measurements or 'AD' files) from the source directory that
10 we are testing upon."""
11 parser = argparse.ArgumentParser()
12 parser.add_argument("dir", help="Specifies the directory that we wish to use to source the testing files from. Must be "
13                                 "one of '6minwalk-matfiles', '6MW-matFiles', 'NSAA', or 'allmatfiles'.")
14 parser.add_argument("testdirs", help="Specifies the directories of files that are used to train the models used "
15                                      "for file assessment. Must be one of '6minwalk-matfiles', '6MW-matFiles', "
16                                      "'NSAA', or 'allmatfiles', and should be separated with '_'.")
17 parser.add_argument("ft", help="Specifies the measurements that will be used from the files in 'dir' to test on "
18                                "the models sourced from 'testdirs'.")
19 parser.add_argument("--batch", type=bool, nargs="?", const=True, default=False,
20                     help="Option that is only set if the script is run from a batch file to access the external files "
21                          "in a correct way.")
22 args = parser.parse_args()
23
24
25
26 #Section below ensures that the arguments provided to the script are valid in the context of how we will be
27 #using them and exits out if they aren't allowed.
28
29 if args.dir + "\\" not in sub_dirs:
30     print("First arg ('dir') must be one of '6minwalk-matfiles', 6MW-matFiles', 'NSAA', 'allmatfiles', or 'direct_csv'.")
31     sys.exit()
32
33 test_dirs = args.testdirs.split("_")
34 for td in test_dirs:
35     if td + "\\" not in sub_dirs:
36         print("Second arg ('testdirs') must be one or more dirs separated by '_' and each must be one of "
37               "'6minwalk-matfiles', '6MW-matFiles', 'NSAA', 'allmatfiles', or 'direct_csv'.")
38
39 #Ensures the corresponding second argument when dealing with files from 'allmatfiles'
40 if args.dir == "allmatfiles" and args.ft != "jointAngle":
41     print("Third arg ('ft') must be 'jointAngle when 'dir' argument is \'allmatfiles\'")
42     sys.exit()
43
44 for ft in args.ft.split(","):
45     if ft not in file_types:
46         print("Third arg ('ft') must be comma separated file types and must be within 'file_types'.")
47         sys.exit()
48
49
50 #Based on which 'dir' we are using, load the 'short names' of each of the '.mat' files that correspond to the full
51 #file names within the 'dir' directory
52
53 if args.dir == "allmatfiles":
54     source_dir = local_dir + args.dir + "\\"
55     #Note: the check for files not containing 'AllTasks' or 'ttest' is to prevent 'model_predictor' trying to predict
56     #on those files since 'ext_raw_measures' doesn't extract measures from these
57     short_file_names = [f.split("jointangle")[-1].split(".mat")[0].upper() for f in os.listdir(source_dir)
58                         if f.endswith(".mat") and "AllTasks" not in f and "ttest" not in f and "Session" not in f]
59 elif args.dir == "6MW-matFiles":
60     source_dir = local_dir + args.dir + "\\"
61     short_file_names = [f.split("-")[1] for f in os.listdir(source_dir) if f.startswith("All") and f.endswith(".mat")]
62     short_file_names += [f.split("-")[0] for f in os.listdir(source_dir) if not f.startswith("All") and f.endswith(".mat")]
63 elif args.dir == "NSAA":
64     source_dir = local_dir + args.dir + "\\matfiles\\"
65     short_file_names = [f.upper().split("-")[0] for f in os.listdir(source_dir) if f.endswith(".mat")]
66 elif args.dir == "NMB":
67     source_dir = local_dir + args.dir + "\\"
68     short_file_names = [f.split(".")[0] for f in os.listdir(source_dir) if f.endswith(".mat")]
69 else:
70     source_dir = local_dir + args.dir + "\\all_data_mat_files\\"
71     short_file_names = [f.split("-")[1] for f in os.listdir(source_dir) if f.endswith(".mat")]
72
73
74
75 len_sfn = len(short_file_names)
76
77 #For each of the short file names that are contained within 'dir', run 'model_predictor.py' for every short file name
78 #and with arguments specified by the arguments given to 'test_altdirs'
79 for i, sfn in enumerate(short_file_names):
80     print("\nFile: " + str(i+1) + " / " + str(len(short_file_names)) + ": " + sfn + "\n")
81     str_part = " ..\model_predictor.py " if args.batch else "model_predictor.py"
82     if sfn.startswith("-"):
83         mod_pred_str = "python" + str_part + args.dir + " " + args.ft + " " + sfn[1:] \
```

```
84                          + " --alt_dirs=" + args.testdirs + " --handle_dash --file_num=" \
85                          + str(i+1) + " / " + str(len_sfn) + " --no_testset"
86          else:
87              mod_pred_str = "python" + str_part + args.dir + " " + args.ft + " " + sfn + \
88                          " --alt_dirs=" + args.testdirs + " --file_num=" \
89                          + str(i+1) + "/" + str(len_sfn) + " --no_testset"
90              mod_pred_str = mod_pred_str + " --batch" if args.batch else mod_pred_str
91          os.system(mod_pred_str)
```