Authors: Nedas Necepurenko, Tikhon McGill

F20BC - Coursework Report

Implementation

The approach we took was to create classes for the following: Artificial Neural Network, Particle Swarm Optimisation(PSO), Particle, and Hyperparameter Profiles.

Hyperparameters Profile

We created hyperparameter profiles to store all hyperparameters, namely:

1. For Neural Networks:
    a. Number of layers and number of neurons per layer
    b. Activation functions for each layer
2. For PSO:
    a. Inertial Weight
    b. Cognitive Weight
    c. Social Weight
    d. Global Weight
    e. Number of iterations
    f. Number of particles.

When a PSO Class is created, it takes in a Hyperparameter Profile into its constructor, setting its hyperparameters based on the profile.

Artificial Neural Network

The ANN class stores the following:

1. The “layer sizes” - An array of numbers, each number representing the number of neurons in that layer – the first element in the array is the input layer, and the last element the output layer
2. The activation functions for each layer (except input layer)
3. The weights for reach layer (except input layer)
4. Biases for each layer (except input layer)

Activation functions and layer sizes come from the Hyperparameter profiles.

As we’re using PSO to optimise weights and biases, we only need the ANN class to have forward propagation method to get the output.

Forward propagation works as follows:

1. For each layer except the input layer, multiply the input matrix (the output of the previous layer, or just the input in the first hidden layer) and the weight matrix together
2. Add the resultant matrix with the bias matrix
3. Apply the activation function of the layer to the output
4. After getting the outputs for each hidden layer we then do the same on the output layer consisting of 1 neuron as this is a binary classification problem, applying our activation function for the output layer we get the prediction.

Authors: Nedas Necepurenko, Tikhon McGill

Particle

The particle class stores the velocity and position of the particle and handles the position update of the particle. It also stores the particle's best position and best fitness value.

The method that calculates the new position has been completed with boundaries of the min and max values (Xiaogang Gao, 2011) found in the dataset, which turned out to be min: -13 & max: 18.

Position update is simply adding the current position and the velocity vectors together. If the new position falls outside the boundary, we randomly reset the particle's position within the boundary.

Particle Conversion

To discuss how to convert a Particle into a Neural Network, we need to, conceptually, understand how a Neural Network would be converted into a Particle (which wasn't implemented, but provided an understanding of how to convert a Particle to an ANN). It would be done in 5 steps:

1. Go through every layer (except the input layer)
2. Get the weight matrix from the previous layer to this layer
3. Flatten the weight matrix into a vector
4. Append the biases of this layer onto the end of this vector
5. The concatenation of all of these layers' vectors in order produces a Particle's Position

Thus, the conversion of a Particle into a Neural Network is as follows:

1. There exists a helper function "**get_layer_vectors**", which, given a Particle, converts its position into an array of vectors. Each vector in this array represents all flattened weights *and* biases for that layer, as described above.
2. The conversion of a Particle to a Neural Network is as follows:
    a. Take in a Particle, and, using "**get_layer_vectors**", get the arrangement of weights *and* biases in each layer, based on its position
    b. Since we know that biases would be after the weights, we extract them using a slicing operation
    c. This leaves us with just a vector of flattened weights
    d. Based on no. in the previous layer **m** and the no. neurons of this layer, **n**, we use NumPy's "**reshape**" method to convert the flattened weight vector into a Weight matrix
    e. This is done for every layer except the input, producing weights and biases
    f. The resulting Neural Network is returned

Authors: Nedas Necepurenko, Tikhon McGill

Particle Swarm Optimisation

Particle swarm optimisation has 4 functions, further explained below:

1. The PSO algorithm itself
2. Getting informants of a particle
3. Accessing the fitness of the particle
4. Updating the velocity of the particle.

PSO algorithm

For N iterations (a hyperparameter), the PSO Algorithm does the following:

1. Update each Particle's position (add velocity to position)
2. Check the fitness of the Particle and update its personal best and the "global best" if the Particle's fitness exceeds any of the two
3. Update the Particle's Velocity (explained in "Updating Velocity")

At the end of the iterations, the Global Best Accuracy is known, and that result used in our experiment.

Getting informants

Before Informants are acquired, the array of all particles is copied, and the particle looking for its informants is removed from this copy, producing just a list of possible informants.

Using NumPy's "**random.choice**" method, **N** particles, where **N** is number of informants (a hyperparameter), are selected randomly, without replacement.

It was chosen to randomly select informants at each time step, instead of pre-selecting informants and retaining them. This was done for two reasons:

1. It would be less memory-intensive to pick random informants for particles rather than storing a Particle's informants in memory.
2. If all of a Particle's preselected informants converge on a Local Optimum, there would be no way for the Particle to do any better. However, if its informants are selected randomly, then there is a chance for the Particle to come across informants whose positions are close to the Global Optimum.

Updating Velocity

The velocity update function was implemented using this equation:

$$v_i \leftarrow \alpha v_i + b(x_i^* - x_i) + c(x_i^+ - x_i) + d(x_i^! - x_i)$$

Figure 1 - Velocity Update Equation

Specifically, *each dimension* of the velocity Vector is updated as follows:

1. Take the velocity of each particle and multiply it by the inertia
2. Take the difference between the current position and its personal best position, and multiply it by a random number from 0 to Cognitive Weight **b**
3. Take the difference between current position and its informants' best position, and multiply it by a random number from 0 to Social Weight **c**
4. Take the difference between current position and all particles' best position, and multiply it by a random number from 0 to Global Weight **d**
5. Add all of the above together, and set the that dimension to be the result

Authors: Nedas Necepurenko, Tikhon McGill

Accessing fitness

The specification mentioned to just use accuracy of the ANN for the entire data and not worry about overfitting. Therefore, the fitness function looks at the entire dataset.

To get the fitness of a Particle, we do the following:
1. Get the dataset
2. Remove the classifications from the dataset, into their own dataset – the “labels”
3. Convert the Particle into a Neural Network
4. For every item in the dataset, do a forward propagation, giving a predicted result. If the result is above 0.5, make the output 1, otherwise make it 0.
5. The fitness function is (**no. correctly-predicted labels / total no. labels**) * 100

Experimental investigation

Experiment Aim
The chosen aim of this Experiment was to investigate how global best performance of Particles would change, by altering the Cognitive, Social or Global weights individually, and keeping all other hyperparameters the same.

Experiment Variables
The following three Hyperparameters were changed – the Independent Variables:
1. Cognitive Weight – how strongly a Particle’s Velocity should be updated to move towards its personal best position in the search space
2. Social Weight – how strongly a Particle’s Velocity should be updated to move towards the best position of its informants
3. Global Weight – how strongly a Particle’s Velocity should be updated to move towards the best position of *all* particles

The following Hyperparameters were unchanged – Control Variables:
1. PSO Hyperparameters:
    a. Inertia was set to 0.7 – when preparing the experiment, it was found early on that inertia did not much affect the performance of PSO
    b. Number of Iterations was set to 200 – we experimented with 100, 200, 250 and 300 iterations. 200 was chosen as a balance between performance and time to execute PSO.
    c. Number of particles was set to 20, for similar reasons as b
    d. Number of informants was set to 4, as, experimentally, they seldom affected performance of PSO
2. Neural Network Hyperparameters:
    a. The Neural Network has 4 Input Neurons (the size of the Dataset’s inputs)
    b. The Neural Network has 1 Output Neuron (the classification in the Dataset)
    c. The Neural Network has 2 Hidden Layers – the first with 3 Neurons and the second with 2 Neurons

d. The Activation function for both Hidden layers is ReLU, and for the output layer it is Sigmoid (since we have a binary classification problem and a value between 0 and 1 is needed)

The Variable we are measuring is the Global Best performance of all particles.

Experiment Methodology

The Experiment operated as follows:

1. Change each of the three independent variables in increments of 0.25, from 0.25 to 1.5
2. These values are looked at individually (first Cognitive Weight is changed in these increments and performance measured, then Social Weight, then Global Weight)
3. The values not being changed have a default value of 1.25 as a control value
4. For each of these increments, the PSO Algorithm is run 10 times, and an average performance acquired, to mitigate stochasticity of the PSO Algorithm
5. The experiment is carried out automatically, using a script "**experimentation.py**" (do note – experiments take a long time to run)

Results

Table 1 – Results

profile_type	weight	average	standard_deviation
Cognitive	0.25	98.227571	1.531555
Cognitive	0.50	97.892050	1.184353
Cognitive	0.75	98.482859	0.944653
Cognitive	1.00	97.589019	1.343138
Cognitive	1.25	97.913931	1.190203
Cognitive	1.50	96.980306	1.283403
Social	0.25	99.985412	0.030754
Social	0.50	99.927061	0.230655
Social	0.75	99.905179	0.194657
Social	1.00	98.913202	0.633496
Social	1.25	96.725018	1.070027
Social	1.50	97.097009	1.680178
Global	0.25	97.782640	1.330443
Global	0.50	98.460977	0.359720
Global	0.75	97.957695	0.950392
Global	1.00	97.935813	1.057580
Global	1.25	97.257476	1.316148
Global	1.50	97.804522	1.440213

Graphs for average accuracies and their standard deviations can be found in the appendices.

Discussion and Conclusions

Table 1 shows the results of the testing, with the profile and weight tested, the average over 10 runs and the standard deviation of those 10 runs.

Authors: Nedas Necepurenko, Tikhon McGill

Based on the data acquired, the following conclusions can be drawn:

1. Cognitive weight appears to be the least impactful of the test cases, with the deviation being rather high across most tested weights except 0.75.
2. Despite this, as the Cognitive Weight increases, there is a downward trend in terms of accuracy. That being said, there are two “peaks” where the accuracy goes up again.
3. When the Social Weight is between 0.25 and 0.75, the PSO Algorithm performs reliably well, at an average accuracy of very close to 100%. After that, there is a sharp drop-off, and then a slight increase.
4. As the Global Weight increases, there doesn’t seem to be any particular trend – it goes up and down again. What’s interesting is that, at Global Weight = 0.25 and Global Weight = 1.5, similar accuracies were achieved. The best accuracy is reached when the Global Weight is at 0.5, but even that is below the Social Weight

What follows are possible explanations for the conclusions:

1. High deviation applied to both very High and Very Low Cognitive Weights. This could be explained by the stochastic nature of PSO – with low Cognitive Weights, Particles depended on informants and the global best, which could by chance be high or low-performing. At high Cognitive weights, particles depended on their personal bests, which could by chance be high or low-performing.
2. The downward trend in accuracy with increasing Cognitive Weight can be explained by Particles reaching local optima and remaining at them, due to the high Cognitive Weight.
3. The small size of the Social Weight leading good performance could be explained by a Particle’s velocity not being too dependent on its informants, which could be stuck in local optima. Despite that, however, there is still a slight amount by which a Particle’s velocity *does* change based on informants, which would alter its course through the search space and allow for more exploration and better results.
4. At small global weights, Particles were not as swayed by the best position in the search space, so, even if their own or their informants’ bests were inferior, did not improve themselves that way. At large weights, Particles were so swayed by the best position of all particles that they ended up in a local optimum. At a value of 0.5, the Global Weight was small enough to not force Particles to conform to one best position, but also large enough to let Particles take into account a good general direction to move to.

In conclusion, given our experiment, to solve the given binary classification problem, the following hyperparameters are optimal (give 99.99% accuracy):

1. An ANN with 4 Input Neurons, 2 hidden layers of 3 and 2 neurons, and 1 output neuron
2. ReLU activation applied to the hidden layers, Sigmoid to the output
3. A PSO Algorithm with 200 iterations, 20 particles, inertial weight of 0.7, cognitive weight of 1.25, social weight 0.25, global weight 1.25

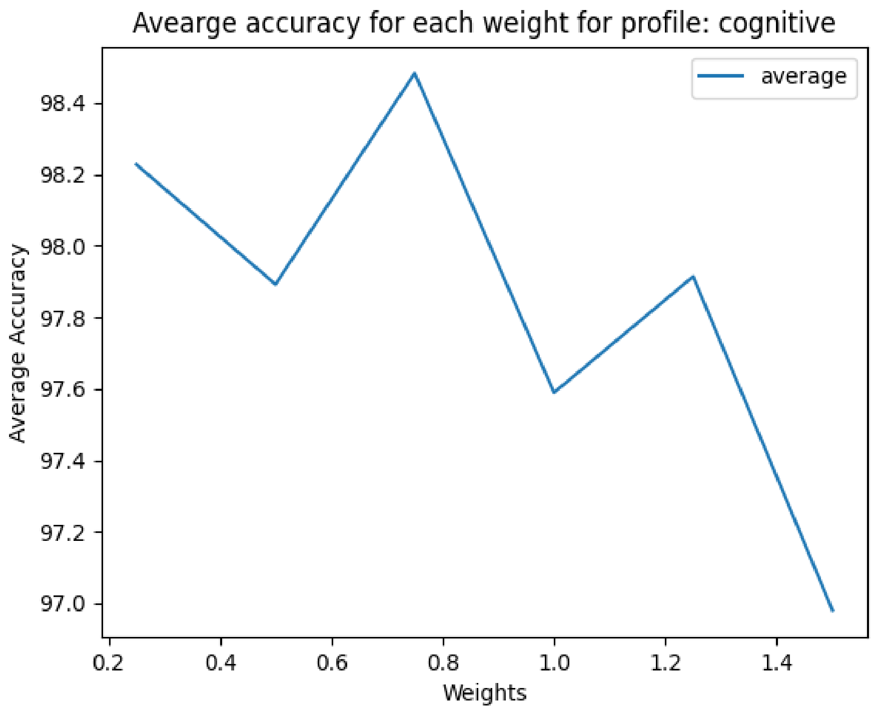
Authors: Nedas Necepurenko, Tikhon McGill

References

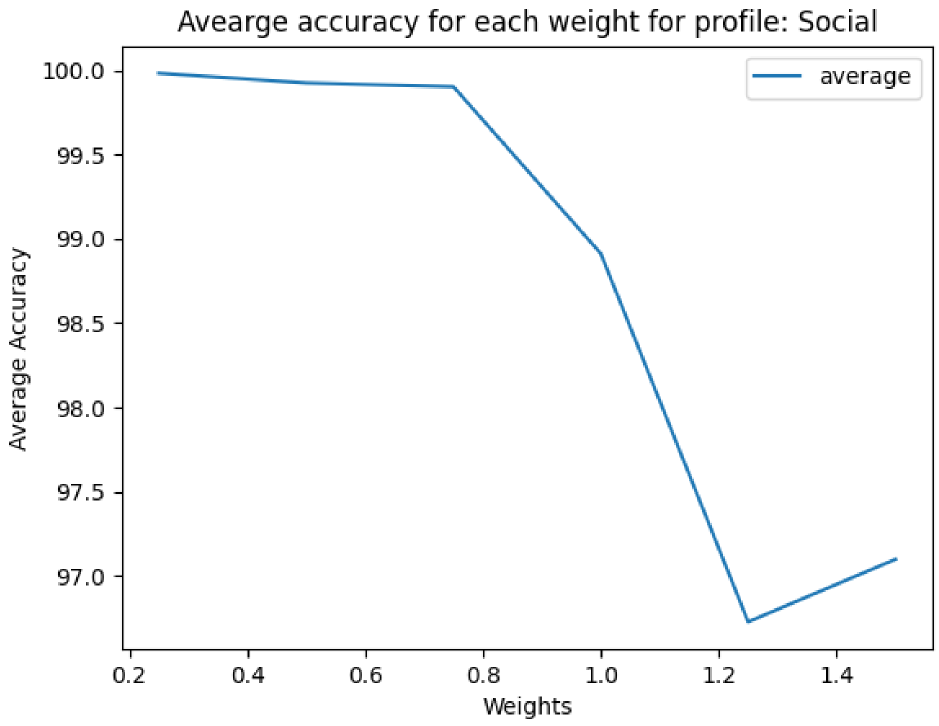
Xiaogang Gao, S. S., 2011. Handling boundary constraints for particle swarm optimization in high-dimensional search space. Information Sciences, 181(20), pp. 4569-4581.

Authors: Nedas Necepurenko, Tikhon McGill

Appendices

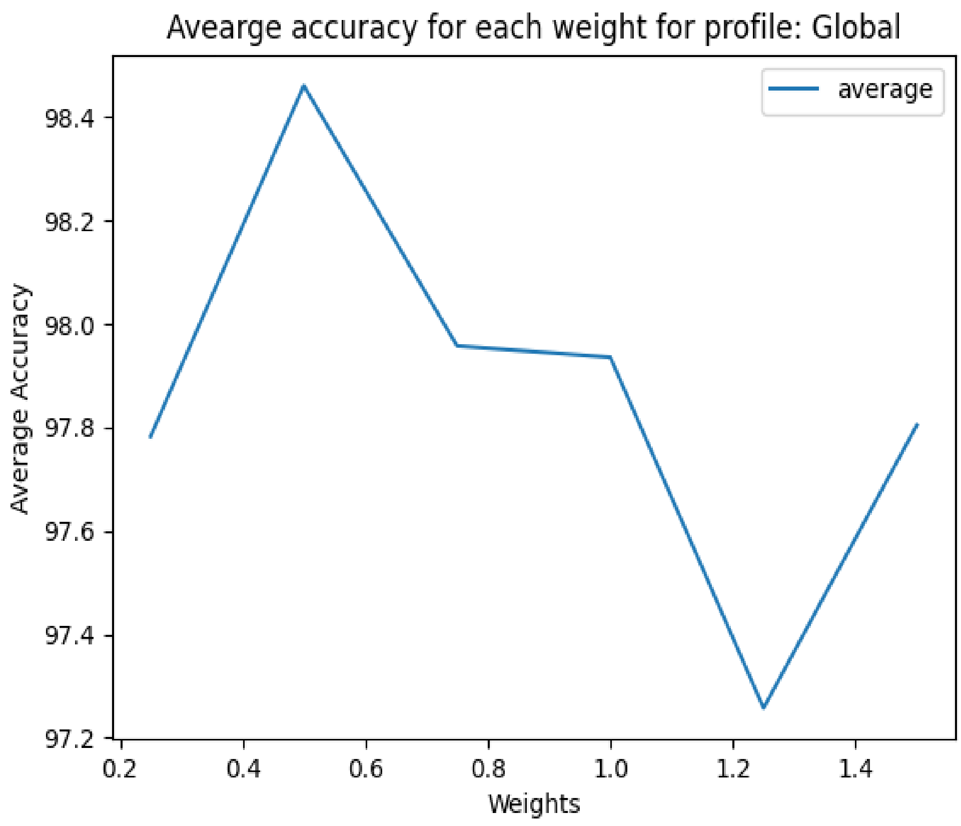


Appendix 1 - Average accuracy for each cognitive weight

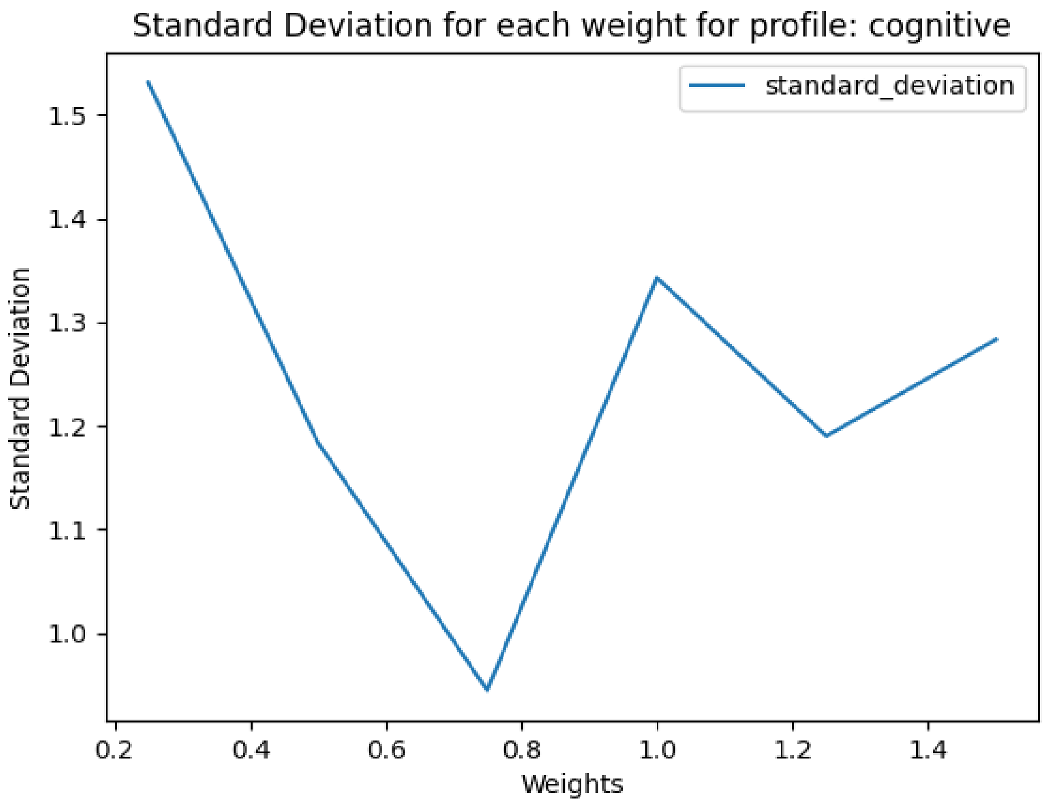


Appendix 2 - Average accuracy for each social weight

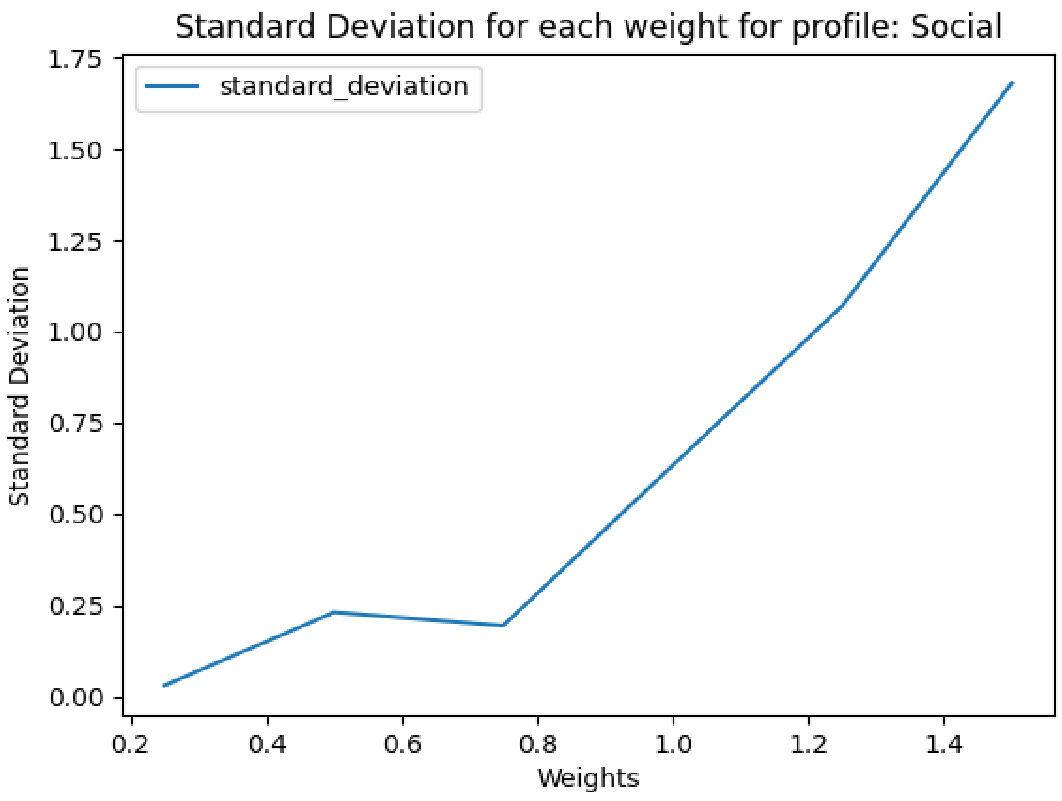
Authors: Nedas Necepurenko, Tikhon McGill



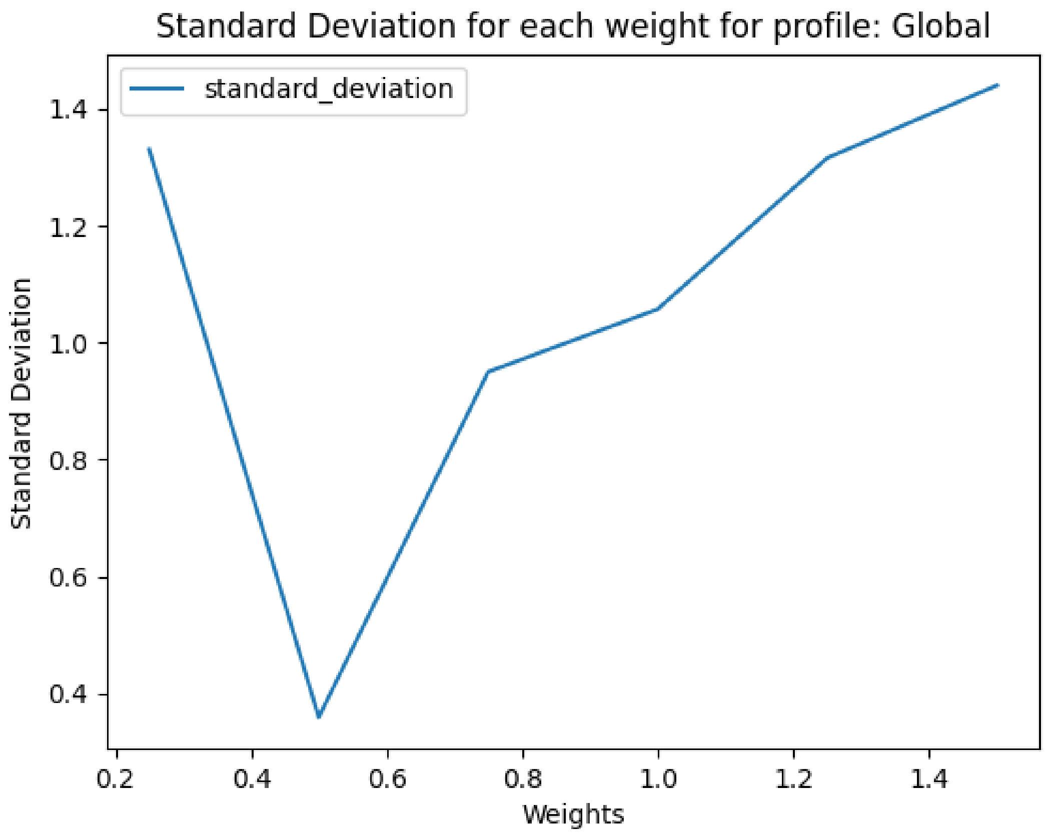
Appendix 3 - Average accuracy for each Global weight



Appendix 4 - Standard deviation for accuracies for cognitive weight



Appendix 5 - Standard deviation for accuracies for social weight



Appendix 6 - Standard deviation for accuracies for global weight