Chapter 1

Controllers

Three types of control schemas are compared in this project: random, heuristic, and SCOUt's memory based reinforcement learning. All three controller, are designed to operate within unknown environments using whatever sensors the agent has. Controller's are compared based on their ability to complete a defined goal, the number of actions that the controller had to perform before completing the goal, and the remaining health and energy levels of the agent. The random controller will select valid actions at random until the operation is completed. This behavior provides a primary base line for determining what levels of performance are considered intelligent. Intelligent controllers would need to exceed the performance of a controller that simply selects actions at random. Both the RL and heuristic approaches can be considered intelligent, as they use knowledge of their environment to select helpful actions. It is up to each controller type to effectively use the information provided in the agent's current state to guide them towards success. Heuristic controllers follow a set of logical steps to choose actions. This type of approach offers practical solutions to operations, but are not expected to be optimal. In addition to this, heuristic controllers are not adaptive to new situations as their logical schemas must be defined for each specific goal. Experiments in chapter five **??** simulate operations for two goals: Find Human and Map Water. Separate heuristic controllers are created for each of these, and provide a secondary performance base line. For SCOUt's RL schema to be considered both intelligent *and* adaptive, it would need to perform at the same level or better than the heuristic schemas designed specifically for each goal. The following sections will cover the solutions used by heuristic and SCOUt controllers for analyzing states, and selecting actions.

1.1 Heuristic Controllers

Two heuristic controllers are used in testing: $Heuristic_{FH}$ and $Heuristic_{MW}$. $Heuristic_{FH}$ is designed for the Find Human goal, and $Heuristic_{MW}$ is designed for Map Water. Both use the same action decision schema with slight variations.

The schemas will consider every valid action, and give each a score based on the agent's current state. The action with the highest score is then selected. Different score calculations are used for scanning and movement actions, but scores will always be a value between 1 and 0 (1 being the best possible score). The difference between the two heuristic controllers is found in the way they score movement actions. $Heuristic_{FH}$ influences movement to cells that have higher decibel and temperature differentials, as a human anomaly will likely be indicated by increased values of these element types. $Heuristic_{MW}$ focuses movement into quadrants that have fewer known element values, so that it can gather new data from unexplored area. Both controllers' movement action scores also factor in hazard avoidance. Movement into cells with the presence of water, or large elevation differentials will be discourage as they could potentially cause harm to the agent. After action scores have been calculated using their respective function, a penalty will be given to any repetitive actions. During an operation, the heuristic controller keeps a history of actions performed at each (x, y) location in the environment. If the controller has previously selected one of the considered actions while in the same location, the calculated score will be cut in half. This will encourage the controllers to make new choices resulting in exploration of new areas, and a more efficient use of sensors.

Valid scanning actions are all scored using the same function, 1.1. Higher scores will be given to scan actions for an element type that is considered more important, and has fewer known values within the corresponding sensor's range. Importance of an element type is determined by whether it is flagged as hazardous and/or as an indicator. The amount of known values in the corresponding sensor's range is calculated by referencing the agent's `internalMap`. The resulting score should influence the controller to use sensors efficiently, assist with hazard avoidance, and emphasize goal completion.

style function

```
def scoreScanAction(elementType: String, state: AgentState): Double = state.ge
  case None => 0.0
  case Some(elementState) => {
    // Weights
    val indicatorWeight = 0.5
    val hazardWeight = 0.5
    val pkirWeight = 3.0
    val immediatesKnownWeight = 2.0
    val weightTotals = indicatorWeight + hazardWeight + pkirWeight + immediates
    // Scores
    val iScore = (if (elementState.indicator) 1.0 else 0.0) * indicatorWeight
    val hScore = (if (elementState.hazard) 1.0 else 0.0) * hazardWeight
    val pkirScore = (1.0 - elementState.percentKnownInRange) * pkirWeight
    val immediatesKnownScore = ((4.0 - elementState.immediateValuesKnown.toDou
    return (iScore + hScore + pkirScore + immediatesKnownScore) / weightTotals
  }
}
```

Scoring each valid movement actions is based on the controller's implementation of the `scoreMovmentAction` function, which involves a series of sub-functions tied to each sensor's element type. Each of the sub-functions calculate their own sub-score for their element type. These sub-functions use threshold analyses on the `QuadrantState` corresponding to the considered movement direction. Once each element type's sub-score has been returned to the `scoreMovmentAction` function, an overall score is determined by a weighted average. The overall scoring functions used for $Heuristic_{FH}$ (1.1) and $Heuristic_{MW}$ (1.1) follow the same logic, but contain different sub-functions and related weights. For an example of how threshold analyses are conducted within a sub-function, see $Heuristic_{FH}$'s `scoreElevation` function 1.1.

photo?

```
def scoreMovementAction(quadrant: String, state: AgentState): Double = {
  // Weights
  val elevationWeight = 1.0
  val decibelWeight = 1.0
  val temperatureWeight = 1.0
  val waterDepthWeight = 1.0
  val weightTotals = elevationWeight + decibelWeight + temperatureWeight + wat
  // Scores
  val elevationScore = scoreElevation(quadrant, es) * elevationWeight
  val decibelScore = scoreDecibel(quadrant, es) * decibelWeight
  val temperatureScore = scoreTemperature(quadrant, es) * temperatureWeight
  val waterDepthScore = scoreWaterDepth(quadrant, es) * waterDepthWeight
  return (elevationScore + decibelScore + temperatureScore + waterDepthScore)
}

def scoreMovementAction(quadrant: String, state: AgentState): Double = {
  // Weights
  val elevationWeight = 1.0
  val waterDepthWeight = 1.0
  val weightTotals = elevationWeight + waterDepthWeight
  // Scores
  val elevationScore = scoreElevation(quadrant, es) * elevationWeight
  val waterDepthScore = scoreWaterDepth(quadrant, es) * waterDepthWeight
  return (elevationScore + decibelScore + temperatureScore + waterDepthScore)
}

def scoreElevation(quadrant: String, elementState: ElementState): Double = {
  val qs = elementState.getQuadrantState(quadrant)
  // Weights
  val percentKnownWeight = 0.5
  val averageValueWeight = 0.0
  val immediateValueWeight = 2.0
  val weightTotals = percentKnownWeight + averageValueWeight + immediateValueW
  // Scores
  val pkScore = (1.0 - qs.percentKnown) * percentKnownWeight
```

```
    val avScore = 0.0 * averageValueWeight
    val imScore = qs.immediateValueDifferential match {
      case Some(v) if (Math.abs(v) > 12.0) => 0.0
      case Some(_) => 1.0
      case None => 0.0
    }
    return (pkScore + avScore + imScore) / weightTotals
  }
```

1.2 SCOUt Controller

The SCOUt controller uses reinforcement learning to build a memory of past actions and rewards for planning future actions. After each operation, the SCOUt controller will store state-action pairs into memory. A state-action pair (SAP) contains the action that the agent took, the state that the agent was in when it chose this action, and the short and long term rewards that the action received. In future operations, the SCOUt controller's action decision model (figure 1.1) will search in memory to find SAPs whose states are similar to the agent's current state. Utilizing data from the agent's current state and the controller's collection of past action-state pairs, the controller will predict rewards of each possible action, and chose an action based on its predicted reward.

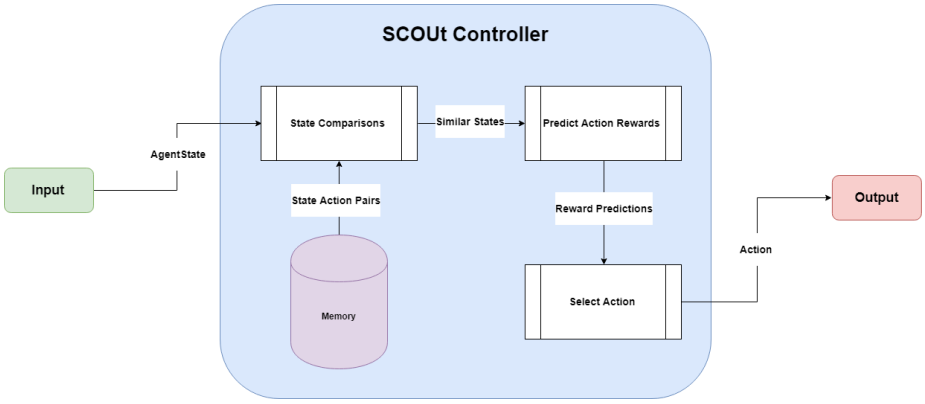

Figure 1.1: Action decision model for the SCOUt controller.

Calculations used for action decisions rely on several weights to assist in state comparisons and decision making processes. Because of the large number of weights required for these calculations, a basic genetic algorithm (GA) was used to generate and evolve different sets of weights. The GA initialized a population of 10 weight sets and evolved them for 50 generations. Each generation creates five mutated copies and five crossover copies of individuals in the current population. The individual(s) that are copied for mutation or crossover are chosen

using roulette selection. Fitness scores are calculated for each of the resulting 20 individuals based on their performance within a series of 50 operations. Ten survivors are then selected for the next generation. Survivor selection keeps the two individuals with the highest fitness scores, and uses roulette selection for choosing the remaining seven. The weight set with the highest fitness in the final generation was selected for use in experimentation, and are listed in table 1.2.

ref roulette selection?

ref table

Table 1.2: Evolved Weight Set Set of values and weights used by the SCOUt controller for action selection. These values/weights were produced using a GA.

Attribute	**Use**	**Value/Weight**
similarityThreshold	state comparison qualification	0.26
health	state comparison	0.41
energy	state comparison	0.78
elementState	state comparison	0.61
quadrantStates	state comparison	0.16
elementState.indicator	state comparison	0.31
elementState.hazard	state comparison	0.07
elementState.percentKnownInRange	state comparison	1.0
elementState.immediateKnown	state comparison	0.41
quadrantState.indicator	state comparison	0.38
quadrantState.hazard	state comparison	0.23
quadrantState.percentKnown	state comparison	0.2
quadrantState.averageValue	state comparison	0.19
quadrantState.immediateValue	state comparison	0.29
minimumSimilarStates	confidence calculation	10
repetitionPenalty	action selection	0.1
predictedShortTermReward	movement action selection	0.87
predictedLongTermReward	movement action selection	0.45
confidence	movement action selection	0.25
predictedShortTermReward	scanning action selection	0.61
predictedLongTermReward	scanning action selection	0.34
confidence	scanning action selection	1.0

1.2.1 Memory

Memory can be gathered from every operation that the SCOUt controller is used in. When an operation has finished, and long term rewards have been assigned to each action, the controller creates new state-action pairs, and selects a sub-set of them to be stored in memory. Saving only a sub-set cuts back on the size of the memory file and the computational time required while searching for similar states. The current memory selection method in this project saves the last 20 SAPs, and a uniformly sampled sub-set of remaining SAPs from the operation. The last 20 are always saved because they typically hold the most important events leading up to the success or failure of the operation. Five

percent of the remaining SAPs are uniformly sampled so that the memory will also contain information related to the agent's initial and intermediate search behavior. Each state-action pair is added to the controller's memory file as a Json object. The collection of SAPs can then be decoded from the file the next time the controller's memory is loaded.

1.2.2 State Normalization

SCOUt begins each operation by loading in all `StateActionPair`'s from memory. The data within each SAP's state is relative to the operation they were recorded in. To handle the variances found between all of these states, SCOUt normalizes them using a Gaussian approach suggested by McCaffrey [1]. Normalization helps make data values more meaningful when studied by the controller. For example, if the controller was seeking out a human, it may look for increases in decibel values. In order for the controller to determine how much of an increase is significant enough to investigate, it needs to first understand what variations are considered normal. Gaussian distribution provides this functionality through the calculation of average and standard deviation (SD) values in a data set. If the agent has gathered decibel readings in its north quadrant that are well outside the SD, it should be encouraged to investigate. All numerical attributes within an `AgentState` are normalized using this Gaussian method. This applies to each of the following attributes:

- `health`
- `energyLevel`
- `elementState`$[0-n]$`.percentKnownInSensorRange`
- `elementState`$[0-n]$`.quadrantState`$[N,S,W,E]$`.percentKnown`
- `elementState`$[0-n]$`.quadrantState`$[N,S,W,E]$`.averageValueDifferential`
- `elementState`$[0-n]$`.quadrantState`$[N,S,W,E]$`.immediateValueDifferential`

First, each attribute type is extracted from all SAP states within the loaded memory. Next the mean and standard deviation values are calculated and stored in an instance of a `GuassianData` class **??**. Once mean and standard deviation values are known, the controller will go back through every SAP state and normalize their attribute using each corresponding `GaussianData` set. The normalization function 1.1 will produce a "normal" value that reflects how many standard deviations the attribute falls above or below the mean. A value of 0 represents no difference between the attribute's value and the mean. Values of 1 and -1 represent a difference of one standard deviation from the mean, and so on. When SCOUt searches for similar states, it will also normalize the current state using the existing sets of Gaussian data. By normalizing the current state to memory states, the numerical attributes compared will all be relative to the existing memory pool.

Normalization of an attribute value, x, based on the gaussian mean, m, and gaussian standard deviation, sd, for the given attribute.

normalization eq

$$x_{normal} = \frac{(x - m)}{sd} \tag{1.1}$$

1.2.3 State Comparisons

Now that all state attributes are normalized, the controller can use a more intuitive approach for calculating differences between two states. Several difference comparisons are used to build a collection of state-action pairs that contain states similar to the current state. These SAPs will later be used to assist in reward prediction. For an SAP to be considered during reward prediction, it must have an overall difference below the *similarityThreshold* specified in table 1.2. Overall state difference is calculated using a series of differences between state attributes, which will all be averaged using a weighted average (equation 1.2). Comparing attributes within `AgentState` separately allows the controller to assign a level of importance to each one through the use of weights (found in table 1.2). Weight values are between 0 and 1. The higher the attribute's weight it, the more influence it will have in the overall difference. An attribute with a weight of 0 will be completely ignored in a weighted average equation. Different weighted average equations are used for overall state difference calculation when the considered SAP's action is movement or scanning type. This allows the controller to compare only attributes that are relevant to the action that was taken.

A general equation that takes a list of n attribute values (V) and a list of n corresponding weights (W) and calculates a weighted average of all attribute values.

$$WeightedAverage = \frac{\sum_{i=0}^{n} A_i * W_i}{\sum_{i=0}^{n} W_i} \tag{1.2}$$

Difference comparisons for each attribute in an `AgentState` are calculated based on their data type (boolean value, normalized numerical value, or sub-class). Sub-class comparisons, such as comparing two `ElementState`s, follow the same procedure as `AgentState`. Difference comparisons will be made for each of the attributes within the sub-class, and a weighted average equation 1.2 function is applied to the results. Boolean differences will return 1 when the compared attributes are both true or both false, and return 1 otherwise. For example, *BooleanDifference* is used to calculate whether an element type in two states were both flagged as an indicator or not. Normalized numerical attributes follow the *GaussianDifference* equation 1.4. This equation will produce values that hold the same principal as the normalization process, where the closer the difference is to 0, the more similar they are. If two values are identical, their Gaussian difference will be 0. Otherwise, the *GaussianDifference* will be relative to how many standard deviations away from each other the two values are. Proofs for these behaviors are found in appendix item **??** and appendix item **??** respectively.

Difference calculation for two boolean values, x and y.

$$BooleanDifference = \begin{cases} x = y & 0 \\ x \neq y & 1 \end{cases} \tag{1.3}$$

Difference calculation for two normalized vales, x and y.

$$GaussianDifference = |x_{normal} - y_{normal}| \tag{1.4}$$

Comparisons with SAP's whose chosen action was a scanning action factor in the health, energy, and element state differences from each of the `AgentState`s (equation 1.7). Each `ElementState` within the current `AgentState` will calculate an *elementStateDifference* using its own equation **??**. All of these *elementStateDifference*s will be averaged (non-weighted) into a single difference value, *averageElementStateDifference*. Difference between two `ElementState`s compares the usage of the element type (hazard and/or indicator detection), and knowledge of the element type (percent known values). The *hazard* and *indicator* differences can help the controller determine the importance and usage of the element type's data being collected. The *percentKnown* and *immediateValuesKnown* differences help the controller decide whether usage of an element type's sensor is efficient, or necessary. For example, if an agent does not have knowledge of the Elevation in adjacent cells, it couldn't confidently determine whether moving into one of those cells would result in a successful move without taking damage.

Calculation for the overall state difference when the compared state-action pair had chosen a scanning action, where V is a list of attributes values and W is the list of weights for the attributes.

$$V = health_{diff}, energy_{diff}, averageElementStateDifference_{diff} W = health_{wight}, energy_{wight}, averageElen \tag{1.5}$$

If the action type is movement, overall state difference is calculated using health, energy, element state differences, and quadrant differences (equation **??**). In addition to calculating an *averageElementStateDifference*, quadrant-to-quadrant differences are calculated between every quadrant in the current state, and every quadrant in the SAP state. Only one "orientation" of quadrant-to-quadrant comparisons will be used in the overall difference calculation. Four orientations are considered by rotating the SAP's quadrants in 90 degree intervals (see figure 1.2). The resulting orientation are denoted as North-to-North, North-to-West, North-to-South and North-to-East. An average of the *quadrantToQuadrantStateDifference* values in each orientation is then calculated, and the orientation with the lowest average difference is used for calculating $OverallDifference_m$.

Calculation for the overall state difference when the compared state-action pair had chosen a scanning action, where V is a list of attributes values and W is the list of weights for the attributes.

$$V = health_{diff}, energy_{diff}, averageElementStateDifference_{diff}, lowestQuadrantOrientationDifference_{di} \tag{1.6}$$

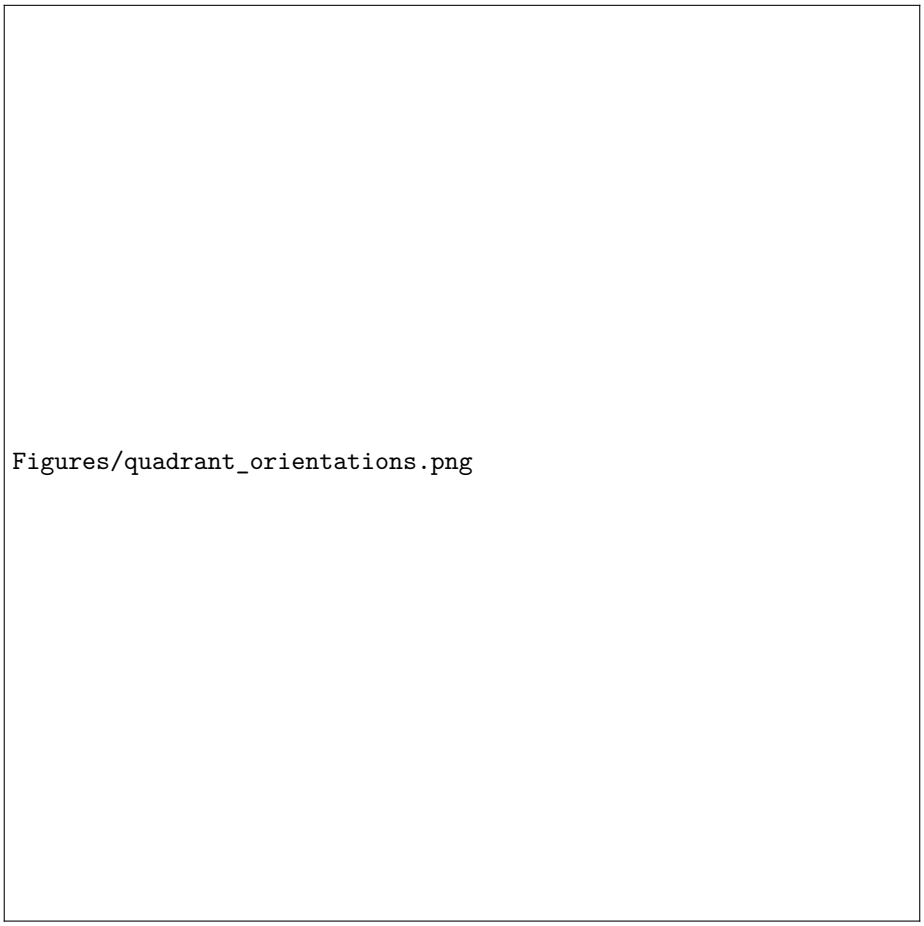

Figure 1.2: Orientation considerations between two compared states.

Each orientation is important to consider because the controller is only concerned with moving towards interesting features in an Environment, regardless of the direction. Considering the orientation with the lowest difference makes the comparison relative to the environments instead of the directionality. Consider an SAP's whose highest matching orientation is found when rotating the quadrants 180 degrees (North-to-South orientation). If the SAP held record that the agent received a high reward for their movement action, the current agent should be encouraged to move towards a quadrant with similar features (not necessarily in the same direction). So if the SAP's agent had chosen to move East in its environment, the current agent should choose to move West in their environment since the two states were most similar in the North-to-South orientation.

Quadrant-to-quadrant comparisons produce a non-weighted average of at-

tribute differences in each of the corresponding *QuadrantState*s between the current state's list of `ElementState`s and the matching `ElementState` in the SAP state. These comparisons are denoted as *quadrantElementStateDifference*s, as they only consider one quadrant of each `ElementState`. For example, making a North-to-South quadrant comparison would consider data in the current states North quadrant against data in the SAP's South quadrant. When making these comparisons, it is not guaranteed that the current state and SAP state will share all of the same types of `ElementState`s. If the current state contains Decibel data and the SAP state does not, it will receive a *quadrantElementStateDifference* of 1. Because we are only comparing against `ElementState`s in the current `AgentState`, if the SAP contains any `ElementState`s not present in the current state, they are simply ignored. These comparisons use equation **??** to examine how much data about the element type is known, and the actual values that are known. Because `averageValueDifferential` and `immediateValueDifferential` are both `Option[Double]`, they have a unique difference equation (equation 1.8).

Calculation for comparing the difference between two `ElementState`s in a given quadrant, where V is a list of attributes values and W is the list of weights for the attributes.

$$V = \{percentKnown_{diff}, \quad averageValueDifferential_{diff}, \quad immediateValueDifferential_{diff}\}$$
$$W = \{percentKnown_{wight}, \quad averageValueDifferential_{wight}, \quad immediateValueDifferential_{wight}\}$$

$$OverallDifference_s = WeightedAverage(V, W) \tag{1.7}$$

A difference calculation used for two values (x and y), where the values are not always known.

$$difference = \begin{cases} x \quad \text{known} \cap y \quad \text{known} & GaussianDifference(x, y) \\ x \quad \text{known} \cup y \quad \text{known} & 1 \\ x \quad \textbf{NOT} \quad \text{known} \cap y \quad \textbf{NOT} \quad \text{known} & 0 \end{cases} \tag{1.8}$$

If the calculated overall difference is below the *similarityThreshold*, an instance of the `StateActionDifference` class (code **??**) is created for use in reward prediction. Each instance stores the overall difference value, the SAP's action taken, and the short and long term rewards. State comparison will be repeated for every SAP in the memory pool, and the resulting collection of `StateActionDifference` instances is passed to the action reward prediction algorithm.

```
class StateActionDifference(
  overallStateDifference: Double
  action: String
  shortTermScore: Double
  longTermScore: Double
)
```

1.2.4 Action Reward Prediction

1.2.5 Action Selection

An action is then selected based on one of two methods. If the controller is training, roulette selection is used. Otherwise, the action with the highest score is always selected. The action is then returned to the agent to perform. If the Operation is still ongoing and the Agent is operational, the process restarts from step using the Agent's new state. If this results in completion or failure of the Operation, the process ends and the Operation event log is added into the controller's memory file for later reference.

step

proof 1

Example:
Gaussian mean = 10
Gaussian standard deviation = 1

x = 12
y = 12
x = y

x(normalized) = (x − Gaussain mean) / Gaussian standard deviation
x(normalized) = (12 − 10) / 1
x(normalized) = 2 / 1
x(normalized) = 2

y(normalized) = (y − Gaussain mean) / Gaussian standard deviation
y(normalized) = (12 − 10) / 1
y(normalized) = 2 / 1
y(normalized) = 2

x(normalized) = y(normalized)

Gaussian difference (x,y) = |x − y|
Gaussian difference (x,y) = |2 − 2|
Gaussian difference (x,y) = 0

proof 2

Example:
Gaussian mean = 10
Gaussian standard deviation = 1

x = 12
y = 7

x(normalized) = (x − Gaussain mean) / Gaussian standard deviation
x(normalized) = (12 − 10) / 1

```
x(normalized) = 2 / 1
x(normalized) = 2

y(normalized) = (y - Gaussain mean) / Gaussian standard deviation
y(normalized) = (7 - 10) / 1
y(normalized) = -3 / 1
y(normalized) = -3

Gaussian difference (x,y) = |x - y|
Gaussian difference (x,y) = |2 - -3|
Gaussian difference (x,y) = 5
```

Bibliography

[1] By James McCaffrey and 01/15/2014. How To Standardize Data for Neural Networks -.