Manual Bachelor Project

M.V. Valk

June 2020

1 Introduction

This document provides details with regard to the program used in the Bachelor Project on comparing Q-learning with Argumentation Based Learning.

2 Running the programs

2.1 Software requirements

Software / Library	**Version**
Python 3	3.7.5
Tensorflow	2.2.0
Tensorflow_gpu	1.14.0
Keras	2.3.1
Numpy	1.16.2
Matplotlib	3.0.2
Tqdm	4.43.0

Table 1: A list of software and libraries which have been used in this study. The source code can be found at `https://github.com/MaxVinValk/Bachelor_Project`

2.2 General information

A program for both performing a genetic algorithm hyperparameter search, and evaluation of specific settings have been provided. These programs can be run using:

```
python3 programName {options}
```

Where options are specified in the following format:

```
--optionName optionValue
```

Each program has a set of valid options, which can be seen in tables 2 and 3. Option values are either integers, floating point numbers, or strings. The latter is represented by using single quotation marks around the option.

2.2.1 Option files

For either programs, options can also be specified in a text file. Option files corresponding to the experiments performed for the project can be found in the folder 'settings'. Options in such a file can be specified using the following format:

```
optionName  optionValue
```

With 1 option on each line of the document. empty lines are allowed in between, if desired. Further, the name and value may be separated by tabs and/or spaces. **Note that in option files, option values that are of the string type are not surrounded with quotation marks.**

A program can be run with an option file using:

```
python3 programName --settingsFile 'settingsFilePath'
```

Additional options may still be specified if required. In addition, for any option specified in both the settings file and the command itself, the latter will be used. This allows the user to override specific options of a settings file while still using the rest.

3 Replication

3.1 Replication - Genetic Algorithm

The program for running the genetic algorithm is MainGen.py. In the settings/geneticAlgorithm folder, all the settings that have been used for each run can be found. Additional commands still need to be specified. A path to an output folder which will store the results of each run needs to be provided, which will be created by the program. If running on peregrine, this flag needs to be set to true. If not running on peregrine, the number of workers that can evaluate individuals needs to be provided. For example:

```
python3 MainGen.py --dir 'boltzman_jong_1' --peregrine true
```

It should be noted that the program has restarting functionality. If for some reason the program was forced to stop unexpectedly, running the command used to launch it originally allows the program to restart where it left of.
For each exploration policy, 4 settings files have been provided, so 4 distinct runs will be generated. Create a new folder, and place all 4 generated directories in it. To read the results, open a python console, and import loadResults from utils/kmeans.py. This method takes as input the folder which holds all 4 runs, as well as the number of individuals you want to load. If not loading all individuals, the program will return the best performing individuals. The result is a min heap, so it will not be sorted.

```
from kmeans import loadResults

resGenes = loadResults(rootFolder, numLoad)
```

To get the best individual, simply retrieve only 1 individual. If one wishes to perform k-means analysis, this can be done as follows:

```
from kmeans import findK, filterOutGenes

filterOutGenes(resGenes, ["explorationPolicy"])
findK(resGenes, numWorkers, numEls)
```

First, as the exploration policies are kept constant, they are removed from the analysis up front.
For findK, resGenes are the genes that have been loaded, numWorkers is an integer specifying how many workers can be used to perform calculations in parallel, and numEls is the amount of clusters that should be considered at most, and should typically be set to be the length of resGenes. This generates a file, "kmeans_wcss", which stores an array in binary encoding using pickle. At each index i it stores the within cluster sum of squares of using i clusters. Note that there is a result of 0 for index 0, but 0 clusters are of course not considered. This is left here simply to allow for easier indexing. To load in the result, use:

```
import pickle

with open("kmeans_wcss", "rb") as f:
    kmeansWCSS = pickle.load(f)
```

Which then can be graphed, for instance, to explore the results further. Calculating the cluster locations for any k can be done using:

```
from kmeans import performClustering
wcss, clusters = performClustering(resGenes, k)
```

Where clusters will hold the resulting k clusters.

3.2 Replication - Evaluation

3.3 Generating data / performing simulations

To test an agent as was done to generate the final results for the project, the program Main.py can be run using the settings found in settings/evaluation. Additional commands still need to be provided. The option for dataRoot needs to be provided with a folder in which the results may be stored. Each run will create its own separate folder within the dataRoot. An example of a valid command is:

```
python3 Main.py --settingsFile 'settingsFileName'\
 --dataRoot 'validExistingFolderPath' --randomSeed 1
```

To recreate each run exactly as was done for the experiment, the program should be run 4 times for each agent using approximation-based Q-learning, with seeds 1 through 4. This will generate 4 separate results that need to be merged into 1. This can be done from a python console using a function from utils/combiner.py

```
from combiner import combineRuns
combineRuns(rootFolder, maxNumRuns)
```

This will take all folders in rootFolder and create a new folder called 'fused' in the root folder. maxNumRuns has a default value of 1000.
If recreating the tabular Q-learning run, it suffices to provide only a rootFolder to store the results in, as it performs all 1000 repetitions in one run.

3.3.1 Analysis

Open up a python console and import the StatCollector class.

```
from Statistic import StatCollector

sc = StatCollector.getInstance()
sc.load(folder)
```

Each run (or combined run) will have a folder called 'rawData' in it. The path to this folder is the argument used in the load method. A run may have session data, data not belonging to any one run, but fused runs do not. A warning may appear when loading a fused run, but this is not an issue.

Exploration The user has a couple of commands for inspecting the data.

```
sc.summarize(detailed)
sc.summarizeRun(runIdx)
```

These commands give some information with regards to what data is stored and for which classes. The variable detailed is by default False. Data is stored per class, and has a name per type of data stored. These names are relevant for other functions.

```
sc.plotAverage(averageOver, shape, plotAll, *toPlot)
sc.plotRun(runIdx, averageOver, shape, plotAll, *toPlot)
```

These functions allow for some exploratory plots. averageOver is a parameter that can be used to determine how many data-points should be considered for each point on the graph, and the default is 1. The shape is a list with [x, y], which specifies per outputted plot how many figures can be at most present at once. Data from separate classes will always appear on separate plots, if present. The default shape is [2, 2]. The variable plotAll is a boolean variable which is true by default. If it is set to false, the specific names of the statistics collected need to be provided in toPlot.

```
sc.getRun(runIdx)
sc.averagedRun()
```

These both return a RunCollector object, which holds the data of a single run. The averagedRun stores the averaged statistic for each statistic in all runs.

```
sc.summarizeGivenRun(run)
sc.getStatistic(run, className, statisticName)
```

These methods allow for the analysis of specific runs. The method getStatistic takes a RunCollector object, the name of the class in which the statistic of interest resides, and the name of the statistic itself.

```
sc.collectStatistic(className, statisticName)
```

This function collects all data for a given statistic and returns it as an array. The index i of this array contains the data for the ith run of the requested statistic.

In-depth analysis & plotting If the program is run as-is, the statistic of interest is how many guesses the algorithm needs to resolve each encountered scenario. This may be retrieved using:

```
rawGuesses = sc.collectStatistic("Agent", "guessesOverTime")
```

As the metric used for plotting is not the total amount of guesses, but accuracy, these results need to be transformed. This can be done with utils/transform.py, in a python console.

```
from transform import convertSetOfRuns, averageRuns

accRaw = convertSetOfRuns(rawGuesses)
accAvg = averageRuns(accRaw)
```

The raw accuracy can be used for statistical analysis, and the averaged accuracy for plotting. The user may wish to only have the accuracy for 1 specific scenario, for instance, the final accuracy at each run. This can be obtained using (assuming 200 scenarios):

```
from transform import getSpecificScenario
finalAccs = getSpecificScenario(accRaw, 199)
```

To perform a permutation test, the following functions can be used from utils/permTest.py:

```
from permTest import permutationTest, getPFromPerm

res = permutationTest(set_1, set_2, numRuns, numWorkers)

getPFromPerm(res, set_1, set_2, numSimulations).
```

The arguments set_1 and set_2 are lists of numbers which are to be compared and should be of the same length. The amount of permutations performed is set with numRuns. Finally, the last argument specifies the number of workers that should be used for the analysis, to run the code in parallel. The function getPFromPerm transforms the obtained information to a p-value (2-tailed). It should be noted that the parameter numSimulations is the same as numRuns in the previous function.

Option name	**Value type**	**Range**	**Description**
settingsFile	String	Path to text file	All other options can also be specified in in a settings file
dataRoot	String (path)	Path to existing folder	A folder that will store the results of the run
randomSeed	Integer	Any positive integer	The seed used for generating random numbers
numSimulations	Integer	Any positive integer	How many states the agent will be presented with each run
numRepetitions	Integer	Any positive integer	How many runs will be done with the agent
lm	String	['nn', 'tab']	Which type of learning will be used. The options are: nn - Approximation-based Q-Learning tab - Tabular Q-Learning
explPolicy	String	['epsilon', 'boltzman']	Which exploration policy should be used (for approx. based). The options are: epsilon - Epsilon Greedy exploration policy boltzman - Boltzman exploration policy
explMin	Floating point	Any positive number	The minimal value the exploration policy variable may have
explDecay	Floating point	[0, 1]	The amount with which the exploration policy variable is multiplied when it is decayed.
explStart	Floating point	Any positive number	The starting value of the exploration policy
discountFactor	Floating point	[0, 1]	The Q-learning discount factor.
learningRate	Floating point	[0, 1]	The learning rate for Q-learning (either for Q-updates or the neural network)
minibatchSize	Integer	Any positive integer	Only used for Approximation-Based Q-learning. The size of the minibatches presented to the neural network.
minReplay	Integer	Any positive integer equal to or greater than the minibatchSize	Only used for Approximation-Based Q-learning. The amount of experiences that needs to be collected prior to beginning training.

Table 2: All valid options for running Main.py, which evaluates a single agent.

Option name	**Value type**	**Range**	**Description**
settingsFile	String	Path to text file	All other options can also be specified in in a settings file
dir	String (path)	Path to existing folder	A folder that will store the resulting runs and genepools. It will be created on run.
poolSize	Integer	Any non-zero positive integer	How many individuals will be in each generation
numGens	Integer	Any non-zero positive integer	How many generations will be simulated sequentially in a single run
numReps	Integer	Any non-zero positive integer	How many runs will be performed
copied	Integer	Smaller than or equal to poolSize minus elitism	How many individuals will be created by copying and mutating a single parent into the next generation. poolSize - copied - elitism must be even.
elitism	Integer	smaller than or equal to poolSize minus copied	How many of the best individuals will be directly copied into the next generation poolSize - copied - elitism must be even.
limitExplPolicy	String	['epsilon', "boltzman]	Allows the search to consider only 1 possible exploration policy instead of encoding the type of exploration policy on a gene.
numSims	Integer	Any positive integer	The amount of training examples given to each agent
numEvals	Integer	Any positive integer	The amount of states used to evaluate final agent performance
peregrine	String	['true', 'false', 't', 'f']	A boolean value. If set to true, the environment variable 'SLURM_JOB_CPUS_PER_NODE' will be used to determine the number of workers used for simulations.
numWorkers	Integer	An integer of 1 up to the amount of available cores.	If not running on peregrine, the user can directly specify the amount of workers that should be used in a simulation.

Table 3: The valid options for the genetic algorithm, which can be ran with MainGen.py.