IN PROGRESS

Table of Contents

Intro

This Bandit Visualization module provides an easy-to-use library for implementing and visualizing various Bandit algorithms and environments in machine learning. If you are looking for the following, this library is not for you:

- You want something fast and efficient
- You want a tool to run very large simulations

It is good to use if you want to

- Visualize your algorithms in various ways
- Quickly make nice looking graphs
- Have the support of a straightforward minilanguage
- Easily add new algorithms to the existing ones (fully in python and numpy)

The library is implemented in python3. This library is dependent also on the following modules, and you must have them installed to run this:

- yaml (text parsing)
- numpy (math)
- scipy (stats)

- matplotlib (creating graphs)

Usage Tutorial

Running Files

Input files are placed in the *Input* directory, and are run using `python3 run.py file_name.txt`, where `file_name.txt` is the input file in minilanguage syntax. When the simulation runs, it will either place an Output file in the *Output* directory, under the name you specified, or it will open an animation window. Because of a `matplotlib` issue, the animation may open behind the terminal window in some situations; running the terminal in full screen should avoid this.

Minilanguage Syntax

The first line of the file always contains an `init:` plus a declaration. Here are three types of supported declarations right now, with a brief description of each:

- `Histogram` : makes a histogram, aggregating the regret from a large number of simulations
- `Variable` : makes a line plot, varying a user-controlled parameter
- `Visualize` : runs an animation on a single cycle

The second most important aspect is the `Simulation` declaration. The number or character following is irrelevant; however, each name must be different if there are multiple `Simulation` declarations. Within the simulation, there are two major sub-classes: the `Algorithm` and the `Bandit`.

The `Algorithm` sub-class describes the algorithm. `algtype` is the type of algorithm, and there are various other arguments depending on the specific `algtype` used. Those are described in detail in the Algorithm Sub-Class secton.

The 'Bandit' sub-class describes the bandit / environment. The `ArmList` sub class takes a list of arms, which must be bracketed in python-style and preceded by a dash (-) and a space, as the example shows. Depending on the arm type, the second parameter will have different numbers of arguments to do different things. There are also other arguments, depending on the type of arm – for example, a Linear Bandit also needs a `MeanVector` declaration. Those specifics are described in detail in the Bandit Sub-Class section.

`Horizon` and `Cycles` denote the horizon that each simulation is to be run to, and the number of cycles that should be run. They can be declared at the top-level or within each simulation independently; top level declarations will be applied to all the simulations.

There are also other various arguments, such as `PlotTitle`, `PlotSave`, and `Animate` which will be described farther down; the names are usually self-explanatory. Those are described in detail in the Additional Arguments section.

Comments are done python-style with a hash (#), and whitespace and blank lines are conveniently ignored. For file examples, see the following three sections.

The `Histogram` init

```
init: Histogram  # comments done python-style

horizon: 500
cycles: 1000

Simulation 1:
    Algorithm:
        algtype: KL_UCB
        incr: B7
    Bandit:
        ArmList:
        - [Bernoulli, [0.3]]
        - [Bernoulli, [0.4]]
        - [Bernoulli, [0.5]]
    label: "KL UCB"

Simulation 2:
    Algorithm:
        algtype: UCB
        incr: B7
        alpha: 0.5
    Bandit:
        ArmList:
        - [Bernoulli, [0.3]]
        - [Bernoulli, [0.4]]
        - [Bernoulli, [0.5]]
    label: "UCB"

PlotTitle: "Horizon 500 -- Means (0.3, 0.4, 0.5) -- Bernoulli"
PlotSave: "example.pdf"
Animate: False
```

The histogram file runs a certain number of simulations (cycles) and produces a histogram plot with regret on the x-axis and frequency on the y-axis. It can support an arbitrary number of histograms, but anything more than three becomes very challenging to read.

The `PlotTitle` argument provides a name for your file. If it is left blank, the program will use the `PlotSave` name split at the period. The `PlotSave` argument provides the name under which to save the file; if left blank, it defaults to "temp.pdf".

If Animate is set to True, an animation window will open and display a live build of the first simulation histogram as it runs, and will continue to run the simulation on a different process. If you close the animation window, the program will still run to completion and save the graph.

The `Variable` init

```
init: Variable

Var:
    domain: [0.01, 0.29]  # you can pass arguments like this, and it does
linear sampling for you
    samples: 10

    # args: [0.01, 0.07, 0.09, 0.16]  # or you can pass arguments explicitly

horizon: 500
cycles: 1000

xlabel: "Delta"

Simulation 1:
    Algorithm:
        algtype: TS_Gauss
    Bandit:
        ArmList:
        - [Normal, [0.3 - &&, 1]]
        - [Normal, [0.3, 1]]
    label: "TS Gauss"

Simulation 2:
    Algorithm:
        algtype: UCB
        incr: B7
        alpha: 2.0
    Bandit:
        ArmList:
        - [Normal, [0.3 - &&, 1  ]]
        - [Normal, [0.3, 1]]
    label: "UCB - B7"

Simulation 3:
    Algorithm:
        algtype: Bayes_Gauss
        incr: B1
    Bandit:
        ArmList:
        - [Normal, [0.3 - &&, 1]]
        - [Normal, [0.3, 1]]
    label: "Bayes_Gauss"

PlotSave: "vari_example.pdf"
PlotTitle: "UCB (B7) vs TS Gauss vs Bayes Gauss -- Normal (0.3, 0.3 - Delta) --
Horizon (500)"
```

The `Variable` init gives you the ability to plot the regret against some controlled variable. This controlled variable can be anything; it is a variable denoted by &&. You can place the && anywhere withing the simulations, or in the horizon top level declarations. The plotter runs each simulation and substitutes in a value for every instance of &&, then runs that simulation a certain number of cycles. These values can be defined in two ways as sub-dictionaries under `Var`:

- specify a domain and a number of samples, and the program does linear sampling
- specify the arguments explicitly as a list, and the program will iterate through the list
    - (TODO: arbitrary list comprehensions for the args)

In the example shown, the mean of each Normal arm varies between 0.01 and 0.29, with 10 sample points. The plot will order the x-axis values for you, so there is no need to worry about argument order. However, you do have to define the `xlabel` variable or it will be left blank. A warning is that a Variable plot can take a long time to run; in the example provided, it needs to run 15 000 000 bandit updates, which may take a while depending on your computer.

Warning: Variable plots use `eval` to evaluate the && substitutions. This results in arbitrarily increased power for good (you can use numpy functions, etc.) but it also means that it can evaluate almost anything!

The `Visualize` init

The `Visualize` init is arguably the most interesting because it runs active animations of Bandit algorithms within a single cycle. The input file must also contain a `visual` argument, in order to determine the type of animation to be run. The two currently supported arguments are

- `ellipse`
- `confidence`
  The `confidence` visual creates an animation of a scalar upper confidence bound used in many algorithms, and the `ellipse` visual creates an animation of the confidence ellipse used by certain linear bandit algorithms.

Every `Visualize` init takes only a single simulation class within the declaration; anything more will be ignored. Furthermore, for a full list of compatibility simulation compatiblility, look in the Argument Summary section.

The `ellipse` visual

```
init: Visualize
visual: "ellipse"

horizon: 5000

Simulation:
    Algorithm:
        algtype: Lin_TS
    Bandit:
        ArmList:
        - [Linear, [1., 1.]]
        - [Linear, [1., 0.]]
        - [Linear, [1., -1.]]
        - [Linear, [-1., 1.]]
        - [Linear, [-1., 0.]]
        - [Linear, [-1., -1.]]
        - [Linear, [0., 1.]]
        - [Linear, [0., -1.]]
        MeanVector: [0.3, 0.4]

    Normalized: True
    NoAxesTick: True
    HelpLines: True
    FPS: 20
```

The ellipse visualization takes a 2D linear bandit; when run, it displays the arm vectors, the actual mean, and the confidence ellipse, as the simulation progresses. There are also some additional optional arguments within the Simulation declaration:

- `Normalized`: This is a more general Linear Bandit argument that takes every arm and mean vector, preserving the direction but dividing by the length. Defaults to False.
- `NoAxesTick`: Option the plot easier to view. If True it removes axes ticks and labels. Defaults to False.
- `HelpLines`: display extension of the mean vector, and perpindicular projections of the arm vectors onto it to see the mean reward that would be recieved from a given arm. Defaults to True.
- FPS: Control the animation update rate. If the animation is running too slowly on your computer, you can decrease this number. Defults to 20.
- `LevelCurves`: For the Lin_TS algorithm, it will display level curves. It is meaningless in any other situation. Defaults to True.

The `confidence` visual

```
init: Visualize

horizon: 5000

visual: "confidence"

Simulation 1:
    Algorithm:
        algtype: KL_UCB
        incr: B3
    Bandit:
        ArmList:
        - [Bernoulli, [0.1]]
        - [Bernoulli, [0.2]]
        - [Bernoulli, [0.4]]
    label: "TS Beta"
```

The `confidence` visual provides an animation of the scalar upper confidence bound used by many algorithms.

the Visualize class is to visualize the behaviour of an algorithm in a single cycle

the bandit and the graph are run concurrently in a single file

Additional Features

Here is a list of currently supported additional features:

- Error Checking: YAML does the syntax error checking if you have mistyped arguments. There is also a small error parser which tries to catch argument-based errors and inconsistent declarations.
- Data Saving: The data generated is saved in the Data folder, in a subfolder named using the first four letters of the `init` and a timestamp created when you start the program.
- Safe Plot Saving: When you specify a plot name, the program attempts to save it without overwriting another file by appending a number to the file name. If you want the existing file under the name to be overwritten, start your file name with `temp`, eg. `temp_plot.pdf` and the program will overwrite any existing file with the same name.
  Here is a list of features that will be implemented in the future:
- Multiprocessing Control: You can specify how many cores you want to use using the `Cores` argument, and it will open the appropriate number of processes to generate the data. This feature is incompatible with the `Animate` argument.

Argument Summary

Using this Summary

When you are preparing an input file, you can use this section to determine compatibliity. For more detail, see the PDF reference file under documentation; this provides a more detailed overview of each algorithm.

The `Simulation` Class

The `Algorithm` Sub-Class

Algorithms describe the behaviour of the bandit arm-choosing behaviour. Here is a list of the currently supported algorithms (called using `algtype`), with description, compatibility, and additional arguments needed as support:

- `random`:
    - Bandit Support: Bernoulli, Normal
    - init Support: Histogram, Variable
    - Additional Arguments: none
- `greedy`:
    - Bandit Support: Bernoulli, Normal
    - init Support: Histogram, Variable
    - Additional Arguments: none
- `greedy_ep`:
    - Bandit Support: Bernoulli, Normal
    - init Support: Histogram, Variable
    - Additional Arguments: `epsilon`
- `UCB`:
    - Bandit Support: Bernoulli, Normal
    - init Support: Histogram, Variable, Visualize {confidence}
    - Additional Arguments: `incr`, `alpha`
- `KL_UCB`:
    - Bandit Support: Bernoulli
    - init Support: Histogram, Variable, Visualize {confidence}
    - Additional Arguments: `incr`
- `Bayes_Gauss`:
    - Bandit Support: Normal
    - init Support: Histogram, Variable, Visualize {confidence}
    - Additional Arguments: `incr`
- `TS_Beta`:
    - Bandit Support: Bernoullli
    - init Support: Histogram, Variable

    - Additional Arguments: none

- `TS_Gauss`:

    - Bandit Support: Normal
    - init Support: Histogram, Variable
    - Additional Arguments: none

- `Lin_UCB`:

    - Bandit Support: Linear
    - init Support: Histogram, Variable, Visualize {ellipse}
    - Additional Arguments: none

- `Lin_TS`:

    - Bandit Support: Linear
    - init Support: Histogram, Variable, Visualize {ellipse}
    - Additional Arguments: none

The `Bandit` Sub-Class

Additional Arguments

The simulation class currently has the following additional arguments:

- `Label`: This is the label used for the Legend, to mark your plot.

Additional Arguments

Example Files

Example files can be found in the Example folder. It contains a semi-comprehensive overview of what this program can do.

Further Details

Note: in the future, this section will likely be moved to the Documentation folder as a PDF / tex file
Here is an overview of what the program does:

- The user inputs a file using `python3 run.py user_file.txt` from the command line. `run.py` is the general process manager that calls the appropriate functions when necessary
- The *user_file* is passed to the *text_parse* module which uses YAML to convert the user input into a rudimentary dictionary. This dictionary is passed to a dictionary checker which checks general consistency and establishes some defaults.
- Now that the dictionary is finished, it will no longer be modified. It is passed as an argument to the various functions in the *DataGen* module, depending on the type of data that is desired, then passed to the *Plot* module to make various plots. *Histogram* and *Variable* plots depend on generated or existing data to build the plots.
- For animations, the bandit is run inline using an update function, without generating

external data.

Overall File Structure

(note: the current file structure is temporary and bound to change)

Rules

Multiprocessor Rules

Notes

The critical importance of core_dict
everything is stored here; during run.py, it is made by calling Parse on some input file
once the core_dict is made, you NEVER change it; if you need a local version make a deepcopy
also, you can't add anything to it; all additions etc. should be done during Parse or dictCheck
since all information is stored here, this is sufficient to do anything desired, in theory
after the core_dict is made, you can pass it around to various functions
DataGen functions use the core dict to make data files
Plot functions use the core dict and data files to make plots
Animate functions use the core dict to make animations
the efficiency of this paradigm is that it allows various functions to operate completely independently of each other!
this makes process management trivial; very few checks are needed (only using join() to make sure data is done before making a plot, etc)
all communication between DataGen and Plot functions is done by reading and writing to text files; this is described in more detail under Data generation / saving procedure

sub_dict notation: // explain notation used in general
sub_dict is some sub dictionary of *core_dict*

Processes, opening subprocesses, mac limitations, etc.
all graphical processes must be in main
all code data generation must be a subprocess
static graphic generation must wait until all data subprocesses are done
no writing can be done in a BuildData file; opened in different process and islated

Data generation / saving procedure
file system / data saving (timestamped)
file creation rules
file writing rules
only open when writing
always use append
writing with newlines

Output file procedure

interface.py and run.py
Outline general code structure