Machine Learning Optimiser

Jack Bracewell, Milan Misak, and Craig Ellis
Group 25

January 7, 2013

Executive Summary

Our project was to create an extensible Machine Learning Optimiser (MLO), using Python. The final product includes an intuitive GUI, a very straightforward framework for creating new regressors, classifiers and meta-heuristics, multi-threading and multi-processing to increase performance, crash-recovery, and more.

The objective of the MLO algorithm is to minimise the number of fitness function evaluations when solving a parameter optimisation problem, which is clearly more beneficial than traditional approaches, when the fitness function takes a long time to return. The motivation for this was for optimising parameters when designing heterogeneous computer systems, but it is useful in any parameter optimisation problem with a time consuming fitness function.

This project is important because a program like this does not yet exist, and (if made open-source) this has the potential to become a very popular tool for those looking into Machine Learning research. The goal was to provide a system to make it easier to continue research into the development of the MLO algorithm - when actually put to use, the algorithm would be implemented in a non-scripting language where performance would be drastically increased.

There initially existed a prototype version of MLO, which had been created with no clear specification, and had resulted in quite chaotic code which was hampering further development of the algorithm and related research. We were to make an improved version, starting from scratch, and also adding new features that would make using the program far less arduous for new users.

The result is organised and modular; not only are the algorithm-integral components more easy to define and interchange, but so is much of the program structure itself, making it easy to change many of the features such as graph visualisation and user interface.

Future directions for the project might include using Cloud Computing, to remove the workload from the user's machine, and creating a web interface for it. Also, some simple added functionality could be useful: adding support to the design space for non-numeric dimensions means that a wider variety of algorithms are possible, including ones which take, for example, strings as design space parameters.

Contents

1 Introduction

This project concerns the creation of an easy-to-use parameter optimisation tool, with the possibility of making the finished product open-source. The tool itself is a Machine Learning Optimiser (MLO), which automates the task of iterating over an optimisation algorithm (eg. Particle Swarm algorithm, Genetic algorithms) for a given problem, and producing the discovered 'optimal result'. The MLO differs from pure implementations of such optimisation algorithms in its use of Surrogate Models. The pseudocode for the MLO is given below (Algorithm 1).

The program we create must be modular, which will allow new classifiers and regressors to be created based on a simple format. The classifiers and regressors to use in a specific run can then be chosen from the library of those previously created, in a configuration file passed to the program at runtime. This flexibility will allow for much greater ease of use than was ever possible before - since no program like this yet exists, every trial currently has to be hand-coded, which is far less efficient.

1.1 The problem

Reconfigurable applications whose behaviour can be modified by changing a few input parameters before execution are usually optimised manually. This is a very tedious task, as the designer has to analyse the application, create benchmarks ('fitness functions') and run the application with different combinations of input parameters. Since this process involves execution of the application code it is very likely going to take a very long time. Parameter optimisation algorithms such as Particle Swarm Optimisation (PSO) are often used instead of calculating every combination of parameters (this may take an impractical amount of time if the design space is large).

The aim of the MLO algorithm is to further reduce the number of fitness function evaluations of pure optimisation algorithms, by using machine learning techniques. For long running fitness functions, the added time taken to train a Surrogate Model and apply extra filtering of the algorithm population is vastly outweighed by the time saved on fitness evaluations.

In this context, a Surrogate Model is a method used to try to approximate the fitness function, and any valid/invalid regions of the design space. Using a classifier, constraints on a given design can be placed to render certain sections valid or invalid. New designs can then be made inside a region that will most likely produce a valid result. Using a regressor, which will attempt to model the fitness function, allows us to forgo the evaluation of the fitness function in places where the regressor models a particle with a low standard deviation.

At the beginning of this project, we were given a simple version of such a Machine Learning Optimiser. When this program was developed there was no clear specification or design, and this resulted in some big problems with the program:

- The code became very hard to maintain and extend over time.
- It was not particularly stable, resulting in program crashes which often meant losing a few hours of computation results.

This state of the program was hampering further development of the algorithm; it was very difficult to add or remove any features from the program, and very hard for a new user to get to grips with the code base.

Algorithm 1: MLO algorithm [1] [2]

```
1  for $x_* \in X_*$ do
2      $x_*.fit \leftarrow f(x_*)$
3  end
4  repeat
5      for $x_* \in X_*$ do
6          $\mu, \sigma \leftarrow \text{Regressor}(D_r, x_*)$
7          $c \leftarrow \text{Classifier}(D_c, x_*)$
8          if $\sigma > max_{stdv}$ and $c = 0$ then
9              $x_*.fit \leftarrow f(x_*)$
10         else
11             if $c = 0$ then
12                 $x_*.fit \leftarrow \mu$
13             else
14                 $x_*.fit \leftarrow \text{max}_{val}$ or $\text{min}_{val}$
15             end
16         end
17     end
18     $X_* \leftarrow \text{Meta}(X_*)$
19 until Termination condition satisfied
```

(X_* represents the current population, 0 is assumed to be the valid class)

1.2 Outcome

Our task was to create a brand new and greatly improved Machine Learning Optimiser, completely from scratch. The following are some of the requirements we were given for the project:

- A modular design must be created, allowing simple addition of new regressors and classifiers, and optimization meta-heuristics
- There must be a clear separation of different parts of the application as defined by the Model-View-Controller (MVC) design pattern.
- Benchmark and configuration scripts should follow same conventions – their interfaces should remain relatively unchanged.
- The program should continuously save its state, and be crash-recoverable. This state must be human readable in .csv format as much as possible
- We should create visualisations of the computation, such as the state of the surrogate model.

On top of these core requirements, we set ourselves some extra goals, to improve the basic functionality of the program:

- The program should use threading to allow concurrent execution of different computations, and possibly parallelism in some areas to increase performance.
- We should try to create a graphical user interface (GUI), to make it easier for others to pick up and use the program.
- There should be a better logging system, separating error, debugging and information messages.
- The program should be cross compatible to different operating systems as much as possible.

The following report details the design, including use of the MVC design pattern and what each part of it contains; various implementation details, such as the technologies we used, problems we encountered, and how we overcame them; we also describe further directions for the project in the evaluation section.

2 Design and Implementation

2.1 Design

Following one of the core requirements, we used the Model-View-Controller (MVC) design pattern for decoupling the business and view logic of our code.

There is a single main controller which, depending on how the program is executed, gets assigned a view which can be either terminal or GUI based. This view then displays the state of models, which in our case are *runs* containing *trials* (trials of the same configuration are beneficial to check failure rate, and the configuration's sensitivity to randomisation). Then there is also a secondary controller running as a separate thread which is used for producing visualisations.

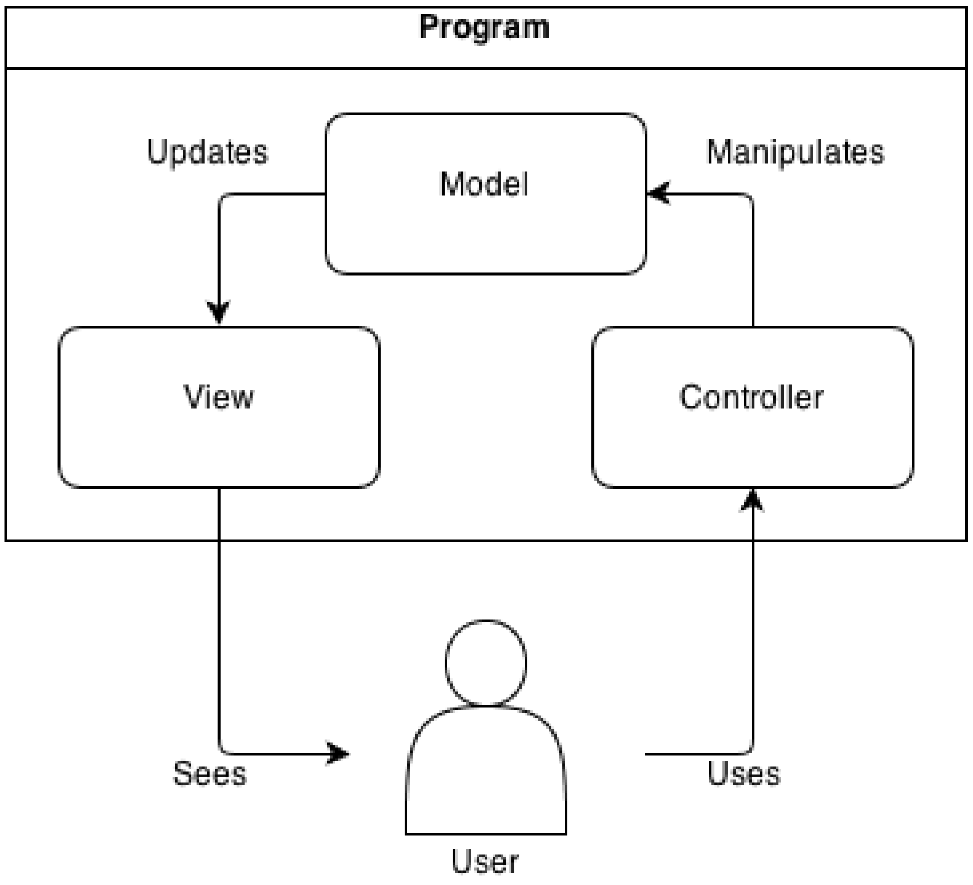


Figure 1: Relationships of MVC components

The architecture of our program follows the core principles of Object-Oriented Programming (OOP). Often, we make use of polymorphism by having an abstract base class providing an interface (*Classifier*, for example. See Section 2.1.1) and one or more implementations (such as the *SupportVectorMachineClassifier*). Other concepts, like encapsulation, are very important as well. Not hiding internal workings and attributes of classes would make maintenance harder, as different entities would be highly coupled - this must be avoided.

2.1.1 Model

The most important models we have are *runs* and *trials*. A run is a single unit of an optimisation problem. Each run then contains one or more trials, which all run the same problem-solving procedure but do not necessarily yield the same result (as optimisation is non-deterministic). Runs are very generic while trials are algorithm-specific. We have implemented a trial class for the Particle Swarm Optimisation (PSO) algorithm.

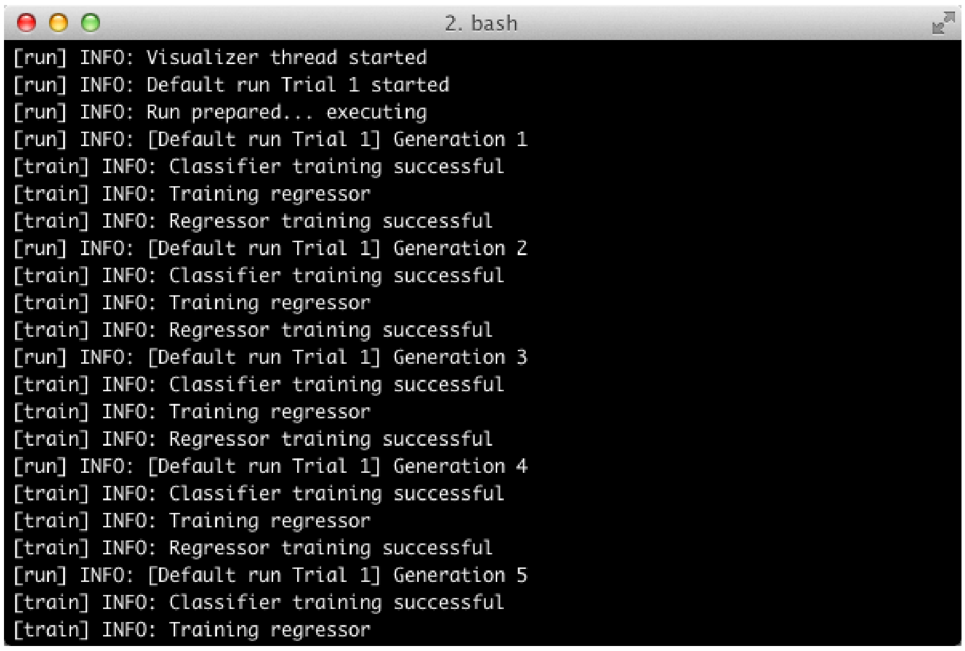
Next, we have *classifiers* and *regressors*. These are both designed so that a user can provide their own implementation by extending a base class we provide. We have implemented a Support Vector Machine classifier and Gaussian Process regressor.

There are also Surrogate Model models which act as a container for the classifier and regressor, making it simpler to interact with from the main algorithm code. We have also implemented a dummy surrogate model which will make the algorithm perform the relevant meta-heuristic in a pure sense.

2.1.2 View

The 'View' of our project's MVC is contained in the *views* package; *gui/windows.py* and *modes.py* deal with the GUI, while *plot.py* is used for graph generation. The design of the view was split into two parts: a terminal view, and a GUI view (which of the two to use is determined by the the arguments passed to the program when it starts).

The terminal view was relatively simple to create, since all we needed to do was inform the user of key events in the algorithm's execution. For example, the message *'[TRIALNAME] Generation G'* is displayed for each generation *G* that is completed by the trial *TRIALNAME*, which lets the user know how far the algorithm has progressed. An example is shown in Figure 2.



```
[run] INFO: Visualizer thread started
[run] INFO: Default run Trial 1 started
[run] INFO: Run prepared... executing
[run] INFO: [Default run Trial 1] Generation 1
[train] INFO: Classifier training successful
[train] INFO: Training regressor
[train] INFO: Regressor training successful
[run] INFO: [Default run Trial 1] Generation 2
[train] INFO: Classifier training successful
[train] INFO: Training regressor
[train] INFO: Regressor training successful
[run] INFO: [Default run Trial 1] Generation 3
[train] INFO: Classifier training successful
[train] INFO: Training regressor
[train] INFO: Regressor training successful
[run] INFO: [Default run Trial 1] Generation 4
[train] INFO: Classifier training successful
[train] INFO: Training regressor
[train] INFO: Regressor training successful
[run] INFO: [Default run Trial 1] Generation 5
[train] INFO: Classifier training successful
[train] INFO: Training regressor
```

Figure 2: Terminal view

The GUI view was obviously more difficult, and involved creating one or two quick mock-ups before we began implementing it (Figure 3). As we added new features to the program, such as the ability to rename and regenerate certain graphs, the GUI had to evolve to accommodate these features, without becoming too untidy and confusing.

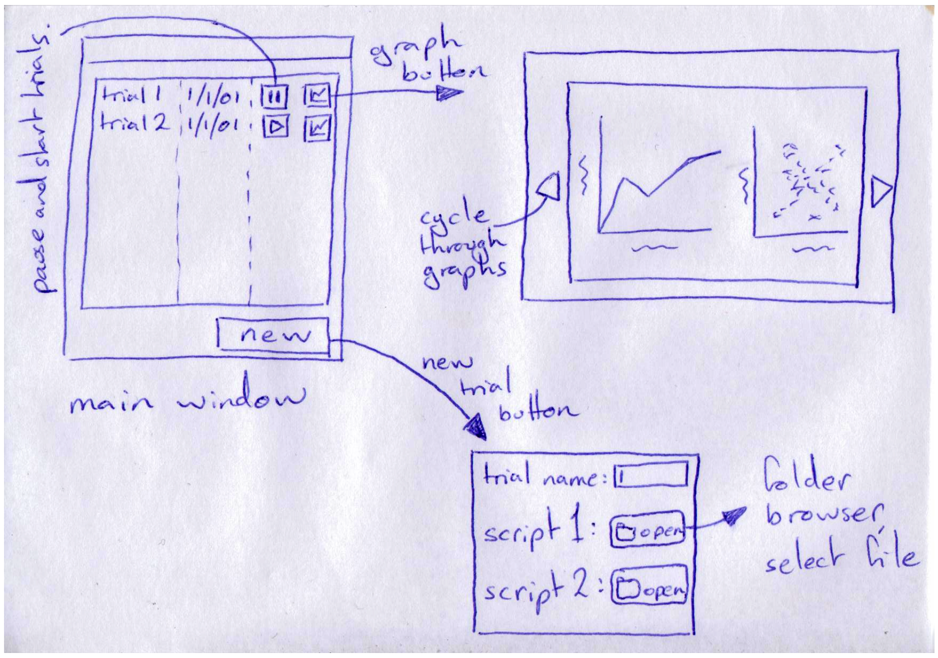


Figure 3: Original design mock-up for the GUI

For many of these new features, we found that simply having a button to a separate window could improve navigation in the program. For example, having the 'New trial' button open a new window allowed us to add some explanatory text to describe the different files required to create a new trial. On the graph window, too, we found that having all of the options to rename and regenerate a graph on the same screen as the graph itself made the window more confusing. Therefore we created a 'Graph options' window to accommodate this functionality, leaving the graph window looking very simple, and the GUI very easy to use.

More examples of the GUI windows are shown in section 2.2.6: Graphical User Interface.

2.1.3 Controller

As previously described, there would main controller class *controller.py* which deals with the spawning of the view, and individual runs. The controller also provides a base for synchronisation, it would hold any relevant synchronisation primitives that might be needed to keep the other threads in check.

It also became apparent that visualisation of graphs was something rather time consuming and was not on the critical path. This would require a visualisation daemon *visualizer.py* to be spawned by main controller to perform such tasks as they were added to the queue by runs.

2.2 Implementation

2.2.1 Technologies Used

As per the requirements, we used Python as our programming language throughout the project. Development with Python is very fast, thanks to its concise syntax and powerful features built into the language. Python is also very suitable for any sort of programming which requires numerical computation and machine learning, due largely to popular open-source libraries such as *numpy*, *scipy* and *deap*. These were used by the original program together with *matplotlib* for plotting graph figures. We decided to stick with these trustworthy libraries for our implementation (see Section 7.3).

- NumPy is described on numpy.org as 'the fundamental package for scientific computing in Python'. We have made extensive use of the array object it provides, as well as various functions necessary for manipulating graphing data.
- SciPy is a mathematics tool similar to (and dependent on) NumPy, which we used to manipulate graphing data.
- DEAP stands for Distributed Evolutionary Algorithms in Python, and was necessary for the more complex function management done in *trial.py*, as well as some algorithm-specific methods elsewhere.
- scikit-learn is a machine learning tool for Python, and as such was used largely to assist in writing our classifiers and regressors.
- matplotlib is a graph-plotting library, used only in *views/plot.py.*
- wxPython is the library we chose to use for making our GUI, and will allow the finished program to run on any platform.

Apart from these third-party libraries, we also used many parts of the extensive standard library of Python. For example when dealing with threads, processes, or I/O (input/output).

Another benefit of using Python is that it is an interpreted language, so all compilation to executable machine code can be done on the fly. This makes it suitable for creating cross-platform applications. Obviously, some dependencies, such as those listed above, must be installed on each platform for the program to run, but then the program will run unmodified on all platforms that have a Python interpreter. We have tested our program on Linux, and OS X with no problems. The program will theoretically run on Windows: it managed to boot with the GUI, but the relevant numerical libraries are not free on Windows, so have not been obtained.

2.2.2 Technical Challenges

Unfortunately, some of the libraries we used have certain crucial functions which are *not* thread-safe (*matplotlib*, *sklearn*), due to their use of global variables. When running multiple trials/runs concurrently it therefore becomes a problem. Of course the obvious reaction would be to put a lock on these functions, but the sklearn functions in particular are used liberally, in every generation of the algorithm, and would create an unacceptable bottleneck.

There are 2 contrasting solutions for dealing with these non-thread-safe libraries. The *matplotlib* solution will be discussed below in the Daemons section (Section 2.2.4). The *sklearn* functions are on the critical path of the MLO algorithm, whereas visualisation (*matplotlib*) was not, so we could not apply the daemon approach.

The solution for fitting Gaussian Processes using *sklearn* is to spawn a new process to execute the function. A pipe is created, and the child end given to the newly spawned process. The trial thread then receives data from the parent end of the pipe, which will block the trial until input is received [3]. The spawned process simply fits the Gaussian Process using the *sklearn* function, and sends the result down the pipe before it terminates, thus keeping the logical ordering of the algorithm.

2.2.3 Process and Thread structure

For each trial to run concurrently, we opted to create a new thread per trial; this was for a couple of reasons:

- Firstly, the entire state of each trial needs to be saved every generation - and in the worst case, most of it needs to be made into a snapshot every generation. This would lead to a high overhead as this data is sent through pipes. Avoiding this by using shared memory would be a messy implementation, because it would require using the only 2 shared memory types in Python: Value and Array [3].

- The second reason is that spawning many trials would put the CPU under great strain, and may even crash the computer before anything could be done about it. It would also be more wasteful of memory. The most expensive part of the algorithm is training regressors, which is done by spawning new processes anyway, though this can be limited arbitrarily by using a semaphore to control the number of processes spawned when using threads for trials

If the GUI is being used, there is another thread for WxApp, which is concerned with maintaining the GUI and performing calls to the controller. There is also the visualiser thread, which will be described more in Section 2.2.4 below. This is on top of any processes spawned to perform regressor fitting tasks, described in Section 2.2.2.

2.2.4 Visualizer Daemon

For the visualisation of the algorithm, some graphs need to be produced to show the state of the surrogate model, and of the algorithm in general. In the original version of the MLO, the visualisation was on the critical path of each trial - logically this did not make sense as it is not something with a strict deadline. The visualisation code does not take a negligible amount of time either, so the sensible option was to create a daemon to perform the task of visualisation.

Whenever a trial encounters a generation which requires visualisation, it takes a snapshot of its current state (all the information needed for producing the graphs), and pushes this onto the visualiser queue. Note that the queue is one of the thread-safe structures in Python [4]. The visualiser loops round, spawning new processes to visualise each snapshot, notifying the view to update when the new image has been generated. A configurable limit is also placed on the number of processes that can be running visualisation tasks at one time – otherwise it can be quite easy to swamp the CPU if there are many trials running, or if dummy surrogate models are being used.

2.2.5 Program recovery after crashes

The original version of the MLO program suffered from occasional crashes that often wasted a few hours of computation, since there was no way of starting the optimisation from the point when the program crashed.

Our program continuously saves its state to the disk with every finished generation of any *trial* in every *run*. Should the program crash for some reason, it can then be rerun with a *--restart* option and the computation will simply keep going from where it was terminated.

It was a requirement that all the program state data that is stored to the disk is human-readable, and preferably in the CSV format. In a certain directory (taken from the configuration script, with a time-stamped subdirectory) for a run you can always find a *run_data.txt* file containing paths to the fitness and configuration scripts and various *trial-X* subdirectories (where X is a trial number). In these subdirectories there is always a pair of files, *Y.csv* and *Y-population.csv* (where Y is a generation number), that contain the program state during that generation.

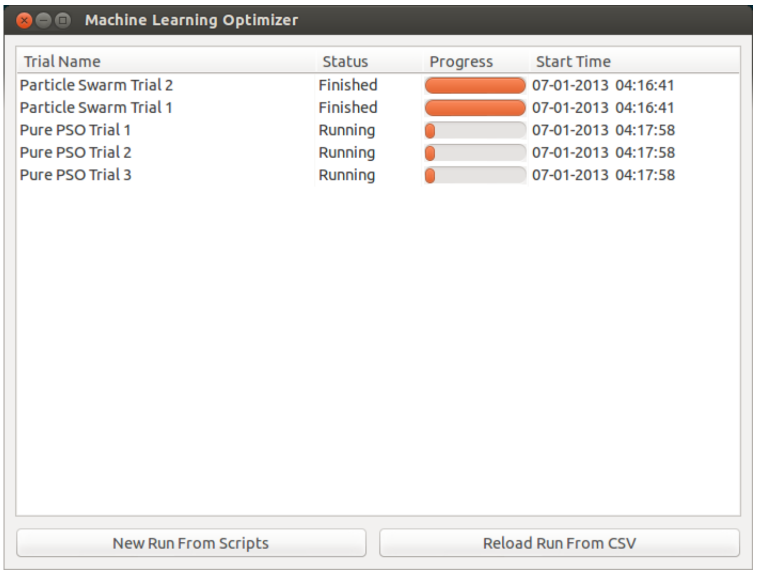
2.2.6 Graphical User Interface

The library we decided to use to create our GUI was *wxPython*. We debated the merits of several different libraries that might be suitable, using the Python wiki as a starting point [7]. In the end, we settled on wxPython because of its excellent multi-platform versatility, its methodical format and object class structure, and its high-level functionality. This made it an excellent choice for us, as novices in Python GUI coding; the high-level style meant that we did not have to spend time building up our own collection of tools - it was all provided for us.

A library that we also considered using (and may have ended up using, had we not discovered wxPython) was *Tkinter*. Tkinter is the standard, and indeed is usually installed as part of Python, so is used in many applications. However, in the end we chose wxPython for 3 simple reasons:

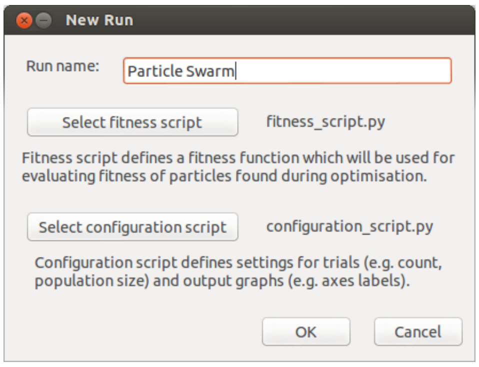
- Tkinter is a lower-level library, and would take longer to get to grips with in the short time we had for the project.
- wxPython automatically creates a native look for the application, on each operating system. In Tkinter, this look must be made manually. For our program, multi-platform use was relatively important.
- From our research, it seems to be a common opinion that Tkinter is reaching the end of its useful life, and newer toolkits like wxPython are becoming the future of GUI coding in Python. Developing a new

application, it seems sensible to use a library that is likely to last for as long as possible.



Trial Name	Status	Progress	Start Time
Particle Swarm Trial 2	Finished		07-01-2013 04:16:41
Particle Swarm Trial 1	Finished		07-01-2013 04:16:41
Pure PSO Trial 1	Running		07-01-2013 04:17:58
Pure PSO Trial 2	Running		07-01-2013 04:17:58
Pure PSO Trial 3	Running		07-01-2013 04:17:58

Figure 4: The main window, displaying the state of current computations



Run name: Particle Swarm

fitness_script.py

Fitness script defines a fitness function which will be used for evaluating fitness of particles found during optimisation.

configuration_script.py

Configuration script defines settings for trials (e.g. count, population size) and output graphs (e.g. axes labels).

Figure 5: The window used for creating a new run

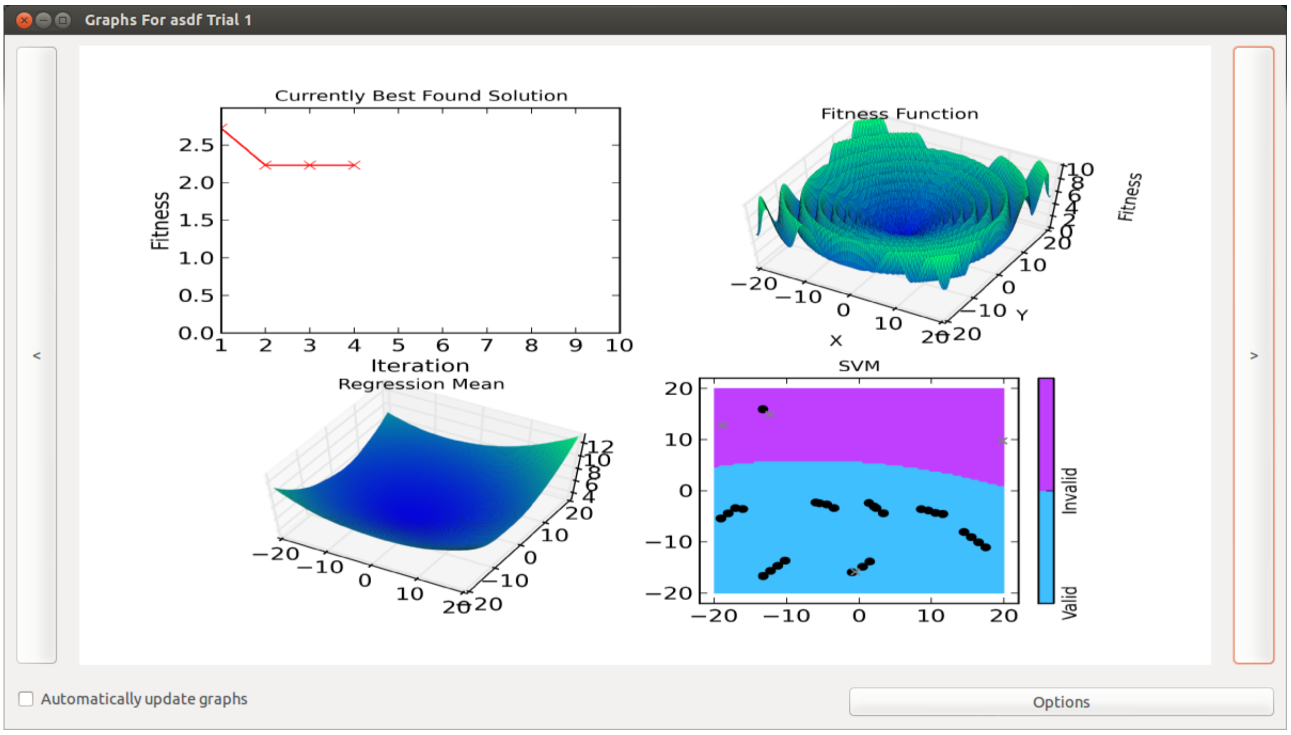
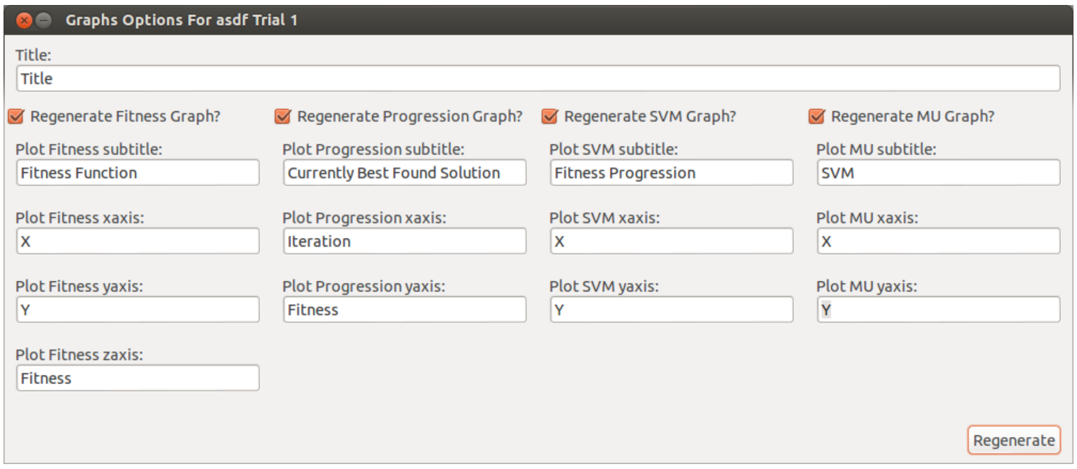


Figure 6: The graph window, showing graphs from a PSO run



Graphs Options For asdf Trial 1

Title:
Title

✓ Regenerate Fitness Graph?	✓ Regenerate Progression Graph?	✓ Regenerate SVM Graph?	✓ Regenerate MU Graph?
Plot Fitness subtitle: Fitness Function	Plot Progression subtitle: Currently Best Found Solution	Plot SVM subtitle: Fitness Progression	Plot MU subtitle: SVM
Plot Fitness xaxis: X	Plot Progression xaxis: Iteration	Plot SVM xaxis: X	Plot MU xaxis: X
Plot Fitness yaxis: Y	Plot Progression yaxis: Fitness	Plot SVM yaxis: Y	Plot MU yaxis: Y
Plot Fitness zaxis: Fitness			

Regenerate

Figure 7: Some options used when regenerating very simple graphs

2.2.7 PEP8

PEP stands for Python Enhancement Proposal, which is a design for proposing features or providing certain information. Python has a set of code style guidelines, *PEP8*, written by its author Guido van Rossum. These guidelines define what Python code comprising the standard library should look like. It is, however, a de facto standard for most Python code, as it greatly improves readability and unifies code styles by defining guidelines for code layout, whitespace use, comments, version bookkeeping and naming conventions.

We think that following the PEP8 guidelines is of a great importance for this project as not only will it make the development easier for us but also because the project will become open-source. Then there will potentially be thousands of people reading through the code and making their own improvements.

To check compliance with the guidelines we used *pep8* from PyPI [5]. Our code almost fully passes, though there are a few lines which do not, as many of these we felt were more readable in their current state. Overall, that is a great result and the code is easily readable.

3 Evaluation

Our entire project was based on a brand new piece of software being developed for a niche set of users. Normally a project like this would have been able to set many more goals per iteration than we had time for, and thus could spend more time on extra features, and ironing out every last tiny bug.

However, the product we produced is an enormous improvement on the initial code, enhancing flexibility and execution time, as well as providing a simple-to-use interface for the user, which wasn't present at all in the code we were given.

3.1 Strengths and Weaknesses

3.1.1 Strengths

- The system is very resistant to failure
- The GUI is simple to use
- The application has cross-platform compatibility
- The design is modular, many of the components are treated as black boxes and can be interchanged with new ones
- Installation and setup is relatively simple (especially on Linux)

3.1.2 Weaknesses

- Only population-based meta-heuristics can be used currently
- There are many dependencies on third-party libraries (see Section 7.3)
- Components such as Gaussian Processes are numerically unstable, this means runtime prediction can vary. This complicates computation of elasticity (See Section 4.2)

3.2 Testing Procedures

Our final program is one which cannot be fully tested, since it is designed for user extension, and it would be impossible to test everything that might be added. However, by running the MLO with the Particle-Swarm algorithm we created, we could make sure that the framework we built was stable and any bugs introduced later would be algorithm-specific.

One of the intended uses for the MLO will be for optimising heterogeneous computer systems. The fitness function for this will take many parameters, and will take a long time to compute (not to mention the difficulty of visualising a surrogate model with 9 dimensions).

What we decided to do was use some 2-dimensional benchmark functions that are provided by *deap* (see Figure 8 for examples), which can be minimisation or maximisation functions, clean or noisy, and single- or multi-objective. We then chose an arbitrary valid function to simulate exit codes of 1 and 0, which makes it possible to visualise the fitness function, and be able to compare it with the surrogate model at points during the algorithm's execution. This would not be possible outside of testing, since the design space needs to be almost exhaustively searched in order to produce a graph of the fitness function, and the idea behind the MLO is to avoid this because the fitness function has a typically long run time.

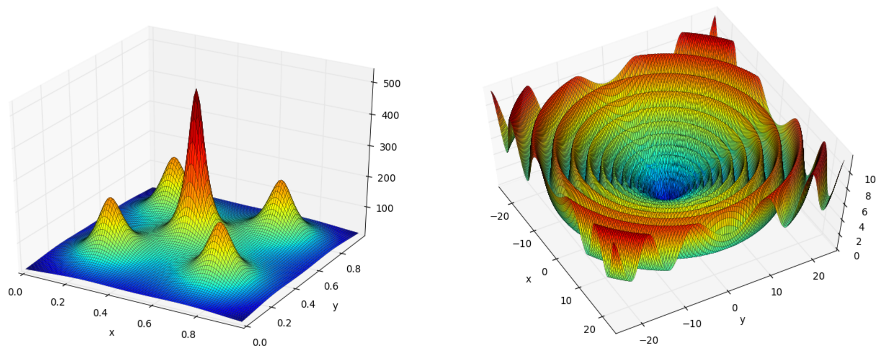


Figure 8: The deap benchmarks range from the simple shekel, to the more complex schaffer

MVC as such is well-suited for testing, as there are separate components (units) which can be tested on their own. It is also possible to automatically test user interfaces. However, this didn't make too much sense in our case, as the user interface of the application is quite simple with only a few different windows, buttons, and other controls. Another flaw is that for most of the tests we would actually need to run the algorithm, which would make testing very slow.

Another challenge posed by testing this program is dealing with nondeterminism. Optimisation problems do not have single valid solutions, but rather a set of solutions - some of which are better than others. Creating unit tests is therefore not quite as straightforward as it could be for simpler programs.

4 Conclusion and Future Extensions

4.1 What we learnt

- Many open source libraries are not thread-safe, due to liberal use of global variables. This is likely to be poorly documented and it greatly complicates the library use.
- Many Python libraries are just wrappers around C libraries, and on occasion will be subject to undefined behaviour. Whenever an error occurs in the corresponding C library, it will bring down the whole Python Interpreter
- Despite claiming to follow the same specification, some functions that require operating system calls (such as BoundedSemaphore in Python, and calls to spawn processes) may behave differently on different operating systems.
- There are liable to be small, unexpected quirks when making a multi-platform program, as certain operating systems may expect more from the code than others. For example, we found that OS X required an extra method call (exec*()) to properly spawn a process.

4.2 Elastic Algorithms

An interesting property of the MLO is that it fits the definition of an elastic algorithm, the attributes of which we list below [6]:

1. Interpretability and Preemption: The algorithm is designed to be stoppable at any time to provide a valid answer, and to be easily suspended and then resumed with minimal overhead.
2. Quality function: You need to be able to model and measure quality and reason about how it progresses with computation.
3. Step-wise incremental: Algorithm proceeds in iterations, starting from initial results and at each step improves it (time may expire at any time). Implicit concept of scale-up but not scale-down.
4. Diminishing returns: For most problems improvement in solution quality is larger at the early stages of the computation, and it diminishes over time. This is not a problem when we have idle resources and deadline has not expired. For pay-as-you-go this may be very costly.

The MLO satisfies the first property with the restarting functionality we implemented: if the computation should crash or be stopped, it can be restarted again from the most recently completed generation. The second property is also satisfied, because a fitness function must be provided that the MLO is trying to maximise or minimise. The third property is intrinsic of the MLO, as the execution proceeds in generations. Generally there will be diminishing returns for the computation, though of course this is dependent on the shape of the fitness function. The benefits that come from this are discussed more in the Future extensions section (Section 4.3).

4.3 Future extensions

4.3.1 Cloud Computing

A benefit of the MLO being an elastic algorithm is that it is a prime candidate for being run in the cloud, which would stop the program from slowing down the user's machine (and thereby slowing down the MLO) to any high degree. To calculate the elasticity between quality and price, it would require an estimation of the quality improvement in the next iteration. This could perhaps be calculated by taking the gradient of the fitness to generation graph, or by estimating the improvement based on the regression model of the fitness function.

Another variable that we would need to estimate is the time taken for each iteration, which has been seen to increase constantly. A regressor could attempt to predict the time for the next iteration by extrapolating the duration of previous generations. This calculation should probably be done locally, as training the regressor is not on the critical path of the MLO algorithm, and so does not need to waste processing power on the cloud. It is possibly the most expensive task in the MLO, but unlike visualisation the amount of data contained by this regressor is very small, so the act of sending the data is unlikely to prove more expensive than performing the regression on the cloud.

It would take a minimal amount of work to get the MLO to run on Platform-as-a-Service provider: Heroku, for example, allows users 1 free dyno. Slightly more work would also allow execution on Amazon's EC2.

4.3.2 Web interface

For a user-friendly implementation of the program in the cloud there would also be a need to provide a web interface. Users could then use any device they like from desktop computers to mobile phones to run optimisations and see the progress and results later.

4.3.3 Other meta-heuristics and Surrogate Models

One of the objectives of this project was to create a better structure, leading to the easy integration of different classifiers, regressors, or meta-heuristics. One of the obvious extensions would be to create some more trial types based around different meta-heuristics other than PSO, or to try using different classifiers or regressors such as neural networks. In terms of speed, the MLO would probably benefit from using a surrogate model that can more easily add new cases, as the retraining of the gaussian process takes the most time out of any task on the critical path.

4.3.4 Support for non-numeric dimensions in the design space, and dynamic dimensions

During this year's Machine Learning coursework, there was a time when we would have liked to use the MLO for finding an optimal topology for a neural network, but some of the parameters were string values, so were incompatible with the design space. Also the actual number of dimensions in the design space may be different, depending on some of the parameters (number of layers and number of neurons per layer). There are probably many other problems like this that have long running fitness functions, and would benefit from this functionality.

There was a simple workaround that we discovered in the meantime, which was to use a discrete numerical dimension, and using this to select values in an array in the fitness function. This is far from perfect because the shape of the fitness function is going to be very dependant on the ordering of this array, which will affect the performance of the MLO.

5 Project Management

Our group was exceptional in that it contained only three members, while all others contained five. The three of us live in the same flat, so meetings were common and easy to organise, which meant that the entire group could meet to discuss any tough design or implementation decisions. We did, however, decide to designate a scrum manager (Jack) to keep the group on track, and to make sure deadlines were met.

We decided to work in weekly iterations. Meetings with our supervisor also occurred every week, so that coincided well with our plan. At these meetings, we showed our supervisor what we had done in the past week, and were given feedback and suggestions for the next week's goals. There was also time for us to ask questions if something about the project wasn't clear.

We each tried to work on all areas of the project, so we could have at least a basic understanding of the overall design, and could hold discussions about the implementation, and any problems that anyone was having. Obviously, some areas were worked on more by certain individuals, for example, Milan was most involved in the saving and reloading of the program state. Craig had the most expertise in the GUI, and Jack spent the most time working on the visualisation and surrogate models.

5.1 Pivotal Tracker

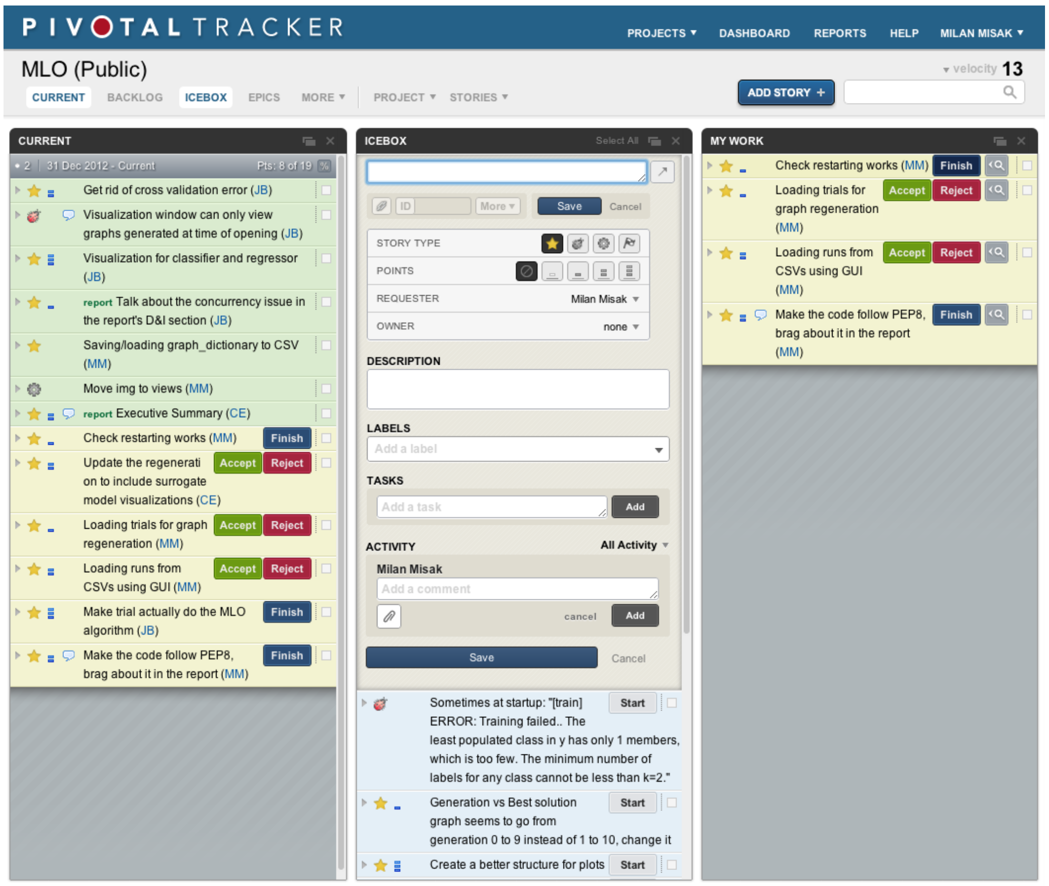
For dividing work between us and keeping track of what everyone was doing we used *Pivotal Tracker*. Pivotal Tracker is a great online project management tool, which we were all familiar with from previous projects. It uses *stories* (tasks) as the basic unit of work; these can be of different types, such as features, bugs or chores. In this way, we also used Pivotal Tracker as a bug tracker during the development of the project.

When stories are created, the creator can assign a difficulty and a person to carry out that task, or leave it open for anybody to look at when they have some spare time (other options are shown in Figure 9). Also when the story is created, it is organised into one of three different categories:

- The *Current* section indicates tasks which are a priority, and are either currently being worked on, or should be looked at by anyone with no current task.

- The *Backlog* contains tasks which have ‘overflowed’ from *Current*. Pivotal Tracker uses a ‘velocity’, or rate of work, to determine how many tasks are likely to be completed in the current iteration. If there are more tasks in *Current* than are expected to be finished in the iteration, the rest are moved to *Backlog*. It is possible to adjust the current velocity so that Pivotal Tracker assigns more or less tasks per iteration.

- The final category is the *Icebox*. These are tasks that should be looked at some time in the future, but are not a top priority at the moment.



CURRENT

2 | 31 Dec 2012 - Current | Pts: 8 of 19

- Get rid of cross validation error (JB)
- Visualization window can only view graphs generated at time of opening (JB)
- Visualization for classifier and regressor (JB)
- report Talk about the concurrency issue in the report's D&I section (JB)
- Saving/loading graph_dictionary to CSV (MM)
- Move img to views (MM)
- report Executive Summary (CE)
- Check restarting works (MM)
- Update the regenerati on to include surrogate model visualizations (CE)
- Loading trials for graph regeneration (MM)
- Loading runs from CSVs using GUI (MM)
- Make trial actually do the MLO algorithm (JB)
- Make the code follow PEP8, brag about it in the report (MM)

ICEBOX

REQUESTER: Milan Misak

OWNER: none

Milan Misak

- Sometimes at startup: "[train] ERROR: Training failed.. The least populated class in y has only 1 members, which is too few. The minimum number of labels for any class cannot be less than k=2."
- Generation vs Best solution graph seems to go from generation 0 to 9 instead of 1 to 10, change it
- Create a better structure for plots

MY WORK

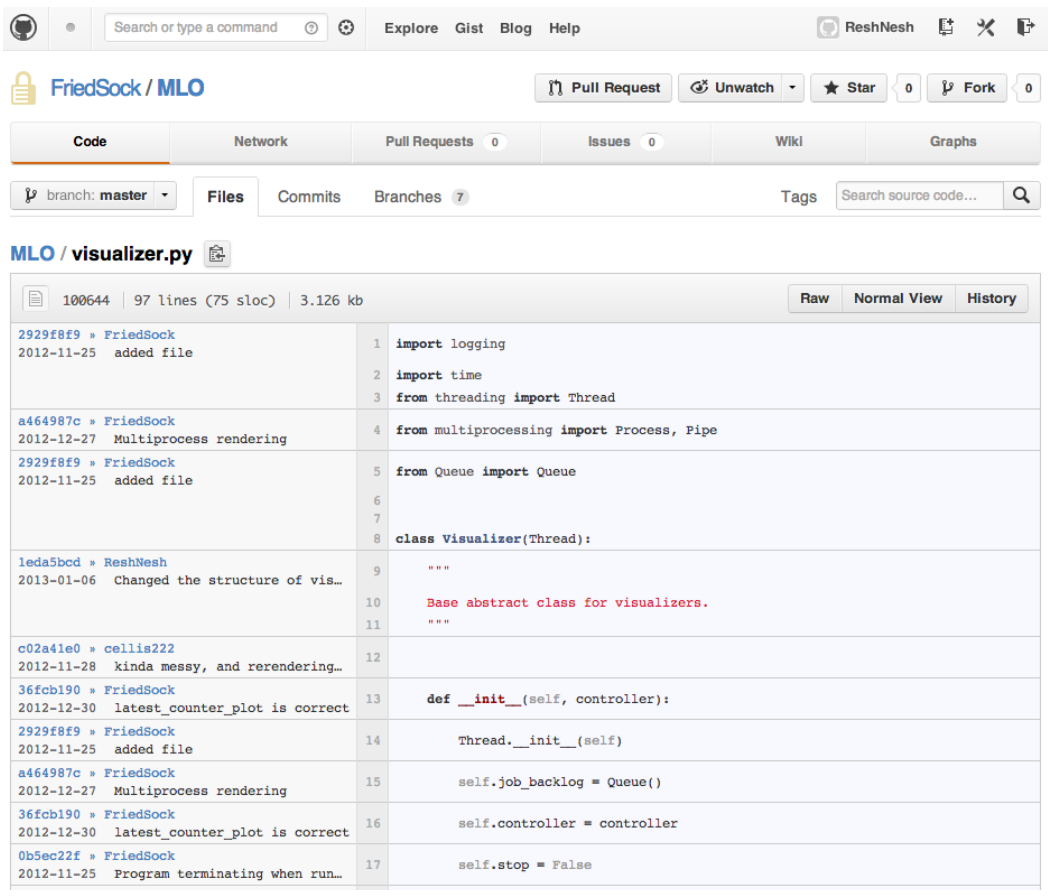
- Check restarting works (MM)
- Loading trials for graph regeneration (MM)
- Loading runs from CSVs using GUI (MM)
- Make the code follow PEP8, brag about it in the report (MM)

Figure 9: Pivotal Tracker and its features such as the *Current* section or a *New story* dialog box

5.2 GitHub

The version control system we used was *git*, which allows the entire group to concurrently work on files, merging each others’ changes and storing backup versions of the code as we go. Specifically, we used *GitHub* to host the repository, as it is free, easy to use, and a site that our entire group was familiar with.

As previously mentioned, this project may become open-source in future, and GitHub is the most well-known git repository host, with over 2.9 million people currently using it. This makes the project widely accessible, and since it is possible to create a GitHub account for free, there are no limitations for those that want to use the program code. At the moment we are using a private repository accessible just by us and our supervisor but it can very easily be made public.



MLO / visualizer.py

```
100644 | 97 lines (75 sloc) | 3.126 kb

2929f8f9 » FriedSock                          import logging
2012-11-25  added file                        import time
                                              from threading import Thread
a464987c » FriedSock                          from multiprocessing import Process, Pipe
2012-12-27  Multiprocess rendering
2929f8f9 » FriedSock                          from Queue import Queue
2012-11-25  added file

                                              class Visualizer(Thread):
1eda5bcd » ReshNesh                               """
2013-01-06  Changed the structure of vis…         Base abstract class for visualizers.
                                                  """
c02a41e0 » cellis222
2012-11-28  kinda messy, and rerendering…
36fcb190 » FriedSock                              def __init__(self, controller):
2012-12-30  latest_counter_plot is correct
2929f8f9 » FriedSock                                  Thread.__init__(self)
2012-11-25  added file
a464987c » FriedSock                                  self.job_backlog = Queue()
2012-12-27  Multiprocess rendering
36fcb190 » FriedSock                                  self.controller = controller
2012-12-30  latest_counter_plot is correct
0b5ec22f » FriedSock                                  self.stop = False
2012-11-25  Program terminating when run…
```

Figure 10: An equivalent of running *git blame* but with a familiar GitHub user interface. This is perfect for accusing people of introducing bugs.

5.3 Auxiliary Methods

We use some of the principles from the Extreme Programming methodology [8]. However, not all of the rules characterised our best options. Pair-programming for all the production code seemed wasteful of our limited human resources, and unit-testing all the code is impossible given the non-deterministic nature of the MLO algorithm.

Pivotal Tracker was helpful for giving us a long-term view of how the project was going, but it was often important to discuss immediate goals and plans for the day. We therefore conducted stand-up meetings in the morning on those days when we were working on the project. There we discussed what was on the critical path for successful and quick completion of the project, and whose task it should be to complete it.

Usually when somebody encountered a particularly difficult problem which they could not solve, we arranged to pair program. Obviously we could have had all three of us trying to find the solution at the same time, but this is often not necessary and it is more efficient if there is some progress being made on a different part of the program concurrently.

We also refactored a large amount of our own previous code. For example, if we discovered redundant code, or some functionality which is no longer used, then it deserved some time to sort this out, to make sure that the codebase stays easy to maintain and lightweight.

6 Bibliography

References

[1] Maciej Kurek, Tobias Becker and Wayne Luk *"Parametric Optimization of Reconfigurable Designs using Machine Learning" - ARC 2012*

[2] Maciej Kurek and Wayne Luk *"Parametric Reconfigurable Designs with Machine Learning Optimizer" - FPL 2012*

[3] Python Multiprocessing docs
http://docs.python.org/2/library/multiprocessing.html

[4] Python Queue docs
http://docs.python.org/2/library/queue.html#module-Queue

[5] PEP8 - Python style guide checker
http://pypi.python.org/pypi/pep8

[6] Prof. Yike Guo and Dr. Moustafa Ghanem *"Elastic Algorithms" Lecture Slides*

[7] The GUI Programming Guide on the Python wiki
http://wiki.python.org/moin/GuiProgramming

[8] The Rules of Extreme Programming
http://www.extremeprogramming.org/rules.html

7 Appendix

7.1 Sample Configuration Script

```
from trial import PSOTrial

results_folder_path = 'data/log'
enable_traceback = True
eval_correct = False


### Basic setup

trials_count = 1
population_size = 10

max_fitness = 250.0
max_iter = 10
max_speed = 0.025
max_stdv = 0.05

surrogate_type = 'proper'  # Can be proper or dummy
F = 10  # The size of the initial training set
M = 2  # How often to perturb the population, used in discrete problems


### Trial-specific variables

trials_type = PSOTrial

phi1 = 2.0
phi2 = 2.0

weight_mode = 'norm'
max_weight = 1.0
min_weight = 0.4
weight = 1.0

mode = 'exp'
exp = 2.0
admode = 'iter'  # Advancement mode, can be fitness
```

```
applyK = False
KK = 0.73


### Visualisation

vis_every_X_steps = 2  # How often to visualize
counter = 'g'  # The counter that visualization uses as a 'step'
max_counter = max_iter  # Maximum value of counter

# Default values for describing graph visualization
graph_title = 'Title'
graph_names = ['Progression', 'Fitness', 'Mean', 'SVM']

graph_dict1 = {'subtitle': 'Currently Best Found Solution',
               'xaxis': 'Iteration',
               'yaxis': 'Fitness'}
graph_dict2 = {'subtitle': 'Fitness Function',
               'xaxis': 'X',
               'yaxis': 'Y',
               'zaxis': 'Fitness'}
graph_dict3 = {'subtitle': 'Regression Mean',
               'xaxis': 'X',
               'yaxis': 'Y'}
graph_dict4 = {'subtitle': 'SVM',
               'xaxis': 'X',
               'yaxis': 'Y'}

all_graph_dicts = {'Progression': graph_dict1,
                   'Fitness': graph_dict2,
                   'Mean': graph_dict3,
                   'SVM': graph_dict4}

plot_view = 'default'


### Regressor and classifier type
regressor = 'GaussianProcess'
classifier = 'SupportVectorMachine'
```

```
### Regression
regr = 'quadratic'
corr = 'squared_exponential'
theta0 = 0.01
thetaL = 0.001
thetaU = 3.0
nugget = 3
```

7.2 Sample Fitness Script

```
from math import e, pow
from deap import benchmarks
from numpy import array

dimensions = 2  # Dimensionality of solution space


# Min and max fitness values
minVal = 0.0
maxVal = 10


# Defines names for classification codes
error_labels = {0: 'Valid', 1: 'Invalid'}
rotate = False


# Defines the problem to be maximization or minimization
def is_better(a, b):
    return a < b


worst_value = maxVal


# Example fitness function for surrogate model testing
def fitnessFunc(part):
    code = 0 if is_valid(part) else 1
    return benchmarks.schaffer(part), code, array([0.0])
```

```
# Example function to define if a design is valid or invalid
def is_valid(part):
    return part[0] ** int(part[1]) / part[1] < e


# Example Termination condition
def termCond(best):
    print '[termCond]: ' + str(best < 0.00001) + ' ' + str(best)
    return best < 0.001


# Name of the benchmark
def name():
    return 'schaffer'


# Example definition of the design space
designSpace = []
for d in xrange(dimensions):
    designSpace.append({'min': -20, 'max': 20, 'step': 0.5,
                        'type': 'continuous'})
```

7.3 Third-Party Library Requirements

These can be installed from PyPI using the *pip* command.

```
numpy
scipy
deap
scikit-learn
infpy
matplotlib
wxpython
```