Evolving Fuzzy Forests: creating Fuzzy Inference Systems with Genetic Programming

A dissertation submitted in partial fulfilment of the requirements for the M.Sc. in Advanced Computing Technologies (Data Analytics specialisation)

David Kirby
Department of Computer Science and Information Systems
Birkbeck, University of London

Summer Term 2021

Academic Declaration

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I have read and understood the sections on plagiarism in the Programme Handbook and the College web site. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

Abstract

This report describes a python framework for evolving sets of fuzzy inference rules using genetic programming. The framework was applied in two ways: Firstly it was used for classification of simple datasets. Although it was found to be slower and sometimes less accurate than other methods, it could be useful in situations where explainability is important since the rule set can be output in a text form understandable by non-experts.

Secondly it was used for reinforcement learning in several openAI gym environments where it was found to be effective at solving the RL problems it was tried on.

Contents

1 Introduction

There have been many machine learning algorithms developed over the last few decades and combined with the increase in computing power and availability of large volumes of data they have transformed many areas of our lives. However their use in some areas such as law, medicine or finance is held back because most of the models are black boxes unable to explain how they came to a decision.

One solution to this problem is to have a machine learning algorithm that creates a set of fuzzy IF-THEN rules that can then be represented in a form understandable to non-specialists. Fuzzy rules were originally hand written for expert systems but writing and debugging rules by hand is time consuming and error prone. An alternative it to learn the fuzzy rules from data. Several ways of doing this have been developed, one of the most successful being through the application of genetic programming (GP).

1.1 Background

1.1.1 Fuzzy Sets and Fuzzy Inference Systems

People have no problem dealing on a daily basis with fuzzy and imprecise terms and reasoning about them. For example "light rain", "heavy rain" and "downpour" have no precise definition and there is no exact point where a change in precipitation moves from one category to another, but people understand and generally agree on what the terms mean. In 1965 Lofti Zadeh (Zadeh, 1965) introduced fuzzy logic and fuzzy set theory, which allowed for representing and reasoning about the sets using the kind of imprecise terms used in human language. Zadeh showed that you could represent these kinds of fuzzy sets by a membership function that maps how strongly an item belongs to the set into a real value between 0 and 1. Ten years later (Zadeh, 1975) he included mapping fuzzy membership functions to linguistic variables. For example in English we refer to people as being "short", "average", "tall", "very tall" and so on. These can be modelled by mapping a linear value (a person's height) to one or more fuzzy sets. A person can have degrees of membership to several "height" sets, ranging from 0 (not a member of this set) to 1 (a full member of the set). So a person who is a little above average height may have a membership of 0.8 to the "average height" class and 0.1 to the "tall" class. Unlike probabilities the membership values do not need to sum to 1 and in most cases will not. The usual Boolean set operators such as AND, OR, UNION, INTERSECTION and NOT can be expressed as mathematical functions on the membership functions.

Figure 1.1 shows how a person's height could map onto linguistic variables using triangular mapping functions. Other shapes than triangles can also be used, such as Gaussian or trapezoid.

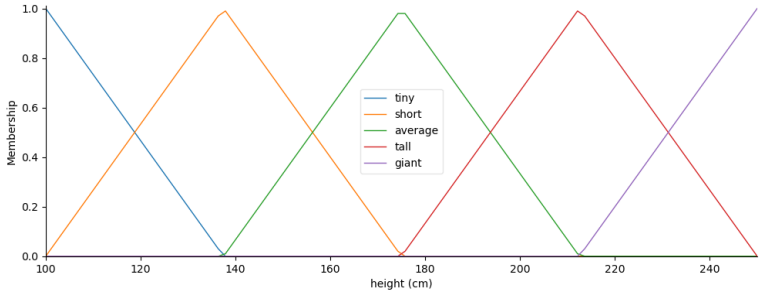

Figure 1.1: fuzzy membership functions for adult human height

A fuzzy inference system (FIS) uses a set of rules composed of fuzzy sets to map one or more inputs into output values. For example, a potential set of fuzzy rules for the classification of irises could be:

```
IF petal-length IS long AND  petal-width IS wide
    THEN virginica is likely
IF NOT petal-width IS wide
  AND (sepal-width is narrow OR petal-length IS long)
  THEN versicolor is likely
IF petal-width IS narrow AND petal-length IS short
    THEN setosa is likely
etc.
```

Fuzzy inference systems are commonly used for feedback control systems but can also be used for other purposes such as classification. Because the rules are expressed in terms of linguistic variables they can be easy for humans to understand.

1.1.2 Genetic Programming

Genetic programming (GP) is part of a family of algorithms inspired by the process of natural evolution, known collectively as Evolutionary Computing (EC). These algorithms include genetic algorithms, evolutionary strategies, particle swarm optimisation, differential evolution and many others. All EC algorithms work by creating a population that randomly samples possible solutions in the problem space, then combines attributes of the best individuals to create a new population. They generally hold the following features in common and differ in implementation and emphasis:

- codification of a potential solution to a problem into some data structure. This is often referred to as their chromosome or genotype
- a population of individuals whose chromosomes are initially randomly generated
- a fitness function that evaluates an individual's chromosome to see how well it fits as a solution to the problem

- a method of selecting individuals from the population according to their fitness
- selected individuals may be mutated by randomly modified their chromosome
- two selected individuals may be mated by swapping over parts of the chromosomes

The evolutionary computing algorithms generally work as follows:

```
an initial population is randomly generated
for N generations:
    evaluate each individual's fitness

    randomly select individuals with a bias towards fitter individuals

    randomly leave them unchanged, mutated or mated
    modified individuals are replaced by the offspring
```

Evolutionary computing algorithms have been applied to many problem domains, and are particularly useful for solving problems that are mathematically intractable, for example because the derivative cannot be calculated for back propagation or the problem is NP-complete. They are also useful for multi-objective problems where they can find a set of solutions that are Pareto optimal.

Most EC algorithms represent the chromosome as a fixed-length list of values. For example Genetic Algorithms (GA) (Holland, 1975) represents the chromosome as a bit string, with mutation done by flipping a randomly selected bit and crossover done by swapping the bits of two individuals that are between random start and end points. Evolutionary Strategy uses a list of floating point values for the chromosome and mutates by adding a value from a Gaussian distribution, with crossover not used. Using a fixed length linear representation works for some problems, but there are limitations on what it can encode.

Genetic Programming (Koza, 1992) avoids this limitation by encoding the chromosome as a tree that can vary in size as it evolves. The tree is usually used to represent a computer program or mathematical expression, but it can also be used to represent other complex structures such as circuit diagrams (Koza *et al.*, 1999).

To mutate a tree a random node is selected and the sub-tree from that node is replaced by a new randomly generated tree. Crossover is done by selecting a random node in each parent and swapping them over to create two new individuals. Figure 1.2 shows an example of two trees representing the expressions $sin(x + 3)$ and $sqrt(log(y * 10))$. After crossover the two offspring represent the expressions $sin(x + log(y * 10))$ and $sqrt(3)$.

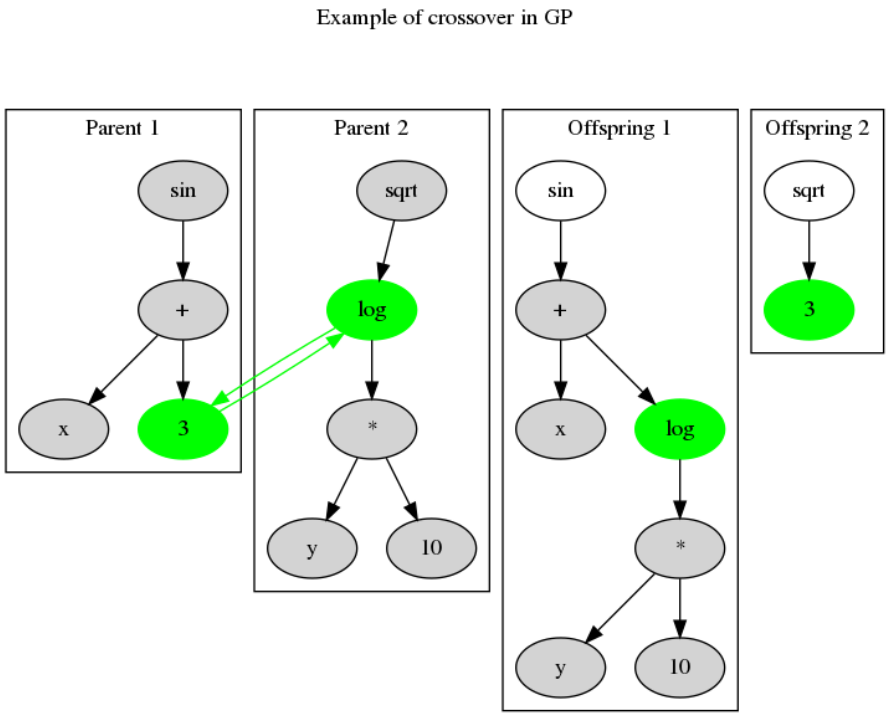

Figure 1.2: Example of crossover in Genetic Programming

2 Project Objectives and Results Summary

2.1 Objectives

The objective of the project was to implement a library for evolving fuzzy logic rules through Genetic Programming with application in two domains:

1. Classification of data sets
2. Reinforcement Learning with the OpenAI Gym RL platform.

A second objective was to explore how well the generated fuzzy Logic system performed for different tasks and investigate ways of improving its performance.

2.2 Results Summary

All the objectives of the project were achieved. The main entry points to the code are two classes, FuzzyClassifier for classification of datasets and GymRunner to learn to play openAI Gym environments.

2.2.1 FuzzyClassifier

A FuzzyClassifier class was created that could learn a set of fuzzy rules from a dataset that could then be used to predict the classification of unseen data. It was implemented in the style of scikit-learn classifiers, with hyperparameters specified in the `__init__` method, a `fit` method to train the classifier and a `predict` method to make predictions.

It was found that the classifier worked well on small data sets with two or three classes, but for large datasets the training time could be prohibitive and the accuracy was found to be poor when tried on a data set with seven classes. However this could be an artefact of the particular dataset used since the No Free Lunch theorem (Wolpert & Macready, 1997) says that no algorithm is suitable for all data sets, so the poor performance could be an artefact of that particular dataset rather than the number of classes. Further work would be needed to determine if it performs as badly on other multi-class problems.

A unique feature of the classifier is the ability to show the top performing rule set as human readable text. For example on the Wisconsin breast cancer dataset, the following set of rules generated during 5-fold cross validation scored 94.8% accuracy in predicting benign and malignant outcomes on the test data:

```
IF NOT-Single_Epi_Cell_Size[low] THEN [malignant[likely], benign[unlikely]]
IF Cell_Shape_Uniformity[low] THEN [malignant[likely], benign[likely]]
IF Cell_Shape_Uniformity[very_low] THEN [benign[likely], malignant[unlikely]]
IF Mitoses[high] THEN benign[unlikely]
```

```
IF Bare_Nuclei[very_high] THEN benign[unlikely]
IF Cell_Size_Uniformity[high] AND Single_Epi_Cell_Size[high] THEN malignant[unlikely]
```

2.2.2 GymRunner

A class called GymRunner was created that could learn to play some reinforcement learning environments in openAI Gym. It was found to be able to master some simple environments such as CartPole (balancing a 2D pole on a movable cart) in as few as ten generations. For LunarLander, a more complex environment, it took several hundred generations to be able to repeatably get a good score. A video demonstration of it playing LunarLander is available at https://youtu.be/Oo6hulwqr9M.

More information about the exploration of the FuzzyClassifier and GymRunner performance against different data sets and gym environments can be found in the Evaluation section of the report.

3 Architecture and Design

3.1 Third Party Libraries

The following third party libraries were chosen for use in the project:

3.1.1 DEAP for genetic programming

DEAP (Fortin *et al.*, 2012) is the de facto standard library for evolutionary computation in Python, supporting a wide range of algorithms such as Genetic Algorithms, Genetic Programming, Particle Swarm Optimisation, and Evolution Strategy. Other evolutionary libraries that were investigated were found to be toy projects not intended for production use, did not support Genetic Programming, or were tailored to a specific application of GP such as symbolic regression.

DEAP is a toolbox of components for implementing evolutionary systems, rather than out-of-the-box ready made algorithms such as those provided by frameworks such as scikit-learn. This makes the learning curve steeper, but gives great flexibility in the problems it can be applied to.

The main components of DEAP are:

1. `deap.creator.create` function for creating new types. This is used to define the custom classes used by the application, for example the type of an individual.
2. `deap.base.Toolbox` is used to register parameterised functions. For example

   ```
   toolbox = deap.base.Toolbox()
   toolbox.register("select", deap.tools.selTournament, tournsize=3)
   ```

   creates the attribute "select" on the Toolbox instance that is the `deap.tools.selTournament` function with its `tournsize` parameter bound to the value 3. When the Toolbox instance is passed to the main evolution algorithm it will expect certain functions to be defined on the toolbox for it to call. The functions that need to be registered depend on the algorithm being used and typically include "mate", "mutate" and "evaluate" as well as "select".
3. A library of functions for different ways of mating, mutating and selecting individuals in the population and for running different kinds of evolution algorithms. Using the toolbox these can be combined like Lego bricks to produce a huge variety of evolutionary computing solutions.
4. The Genetic Programming module. This is the most complex part of DEAP and is explained in more detail in the next section.

3.1.1.1 The deap.gp module The `deap.gp` module contains classes and functions for supporting Genetic Programming. The core components are:

1. The `PrimitiveTree` class encapsulates the tree structure that the genetic programming operations act on. The tree of primitives is stored in a python list in depth-first order. Because the the arity of every primitive is known, the tree can be reconstructed from the list when it is compiled. The `PrimitiveTree` class has methods for manipulating the tree and a `__str__` method converts the tree into the equivalent python code, to be used by the `compile` function.
2. The `PrimitiveSet` and `TypedPrimitiveSet` classes are used to register the nodes that go into a tree structure. There are three node types:
   - `Primitive`s are functions that take a fixed non-zero number of arguments and return a single result. These form the non-leaf nodes of the tree.
   - `Terminal`s are either constants or functions with no arguments that form the leaves of the tree. Terminals that are functions are executed every time the compiled tree is run.
   - `EphemeralConstant`s are Terminal functions that are executed once when they are first created and after that always return the same value. These are used for example to generate a random value that is then used as a constant.

   A `PrimitiveSet` assumes that all the parameter and return types of the primitives and terminals are compatible. A `TypedPrimitiveSet` requires all the types to be defined when they are registered, and will only build trees where the parameter and return types match.
3. The `compile` function takes an individual tree of primitives and compiles it to a python function. It does this by first converting it to a string containing a python lambda function and then calling `eval` on the string. Because of this it is necessary that all the primitives and terminals have `__repr__` methods that will result in that object being created when executed.
4. support functions for creating, mutating and mating trees of primitives.

Full documentation for DEAP can be found at https://deap.readthedocs.io/en/master/index.html.

3.1.2 Scikit-fuzzy for the fuzzy inference system

Several python fuzzy logic libraries were evaluated for the low-level fuzzy inference engine implementation, including:

- fuzzylite python version - https://fuzzylite.com/; https://github.com/fuzzylite/pyfuzzylite
- fuzzylogic - https://github.com/amogorkon/fuzzylogic

- FuzzyLogicToolbox - https://github.com/Luferov/FuzzyLogicToolBox
- FuzzyPy - https://github.com/alsprogrammer/PythonFuzzyLogic
- Simpful - https://github.com/aresio/simpful
- fuzzylab - https://github.com/ITTcs/fuzzylab
- scikit-fuzzy - https://scikit-fuzzy.github.io/scikit-fuzzy/

Several of these were rejected because they were lacking in documentation or unit tests. Others were rejected because they appeared to be abandoned with no commits for at least two years and in some cases requiring python 2.7 or earlier.

The library chosen was scikit-fuzzy (Warner *et al.*, 2019), for the following reasons:

- it has reasonable documentation (including docstrings) and unit tests
- it has been updated in the last six months, so is still in active development
- it supports rules having multiple output variables, for example the rule

  ```
  IF NOT-petal_width[average] AND petal_length[long] THEN
  [setosa[unlikely], versicolor[unlikely]]
  ```

  can set a fuzzy output value for both setosa and versicolor. Most of the libraries would require this to be written as two rules.
- a rule in scikit-fuzzy is created using python operators "&", "|" and "~" on objects. Although this may be a drawback when creating rules by hand since it is less readable than the text-based approach used by many other libraries, it fits in well with the way that the DEAP gp module defines chromosomes as a tree of python functions and objects.

Scikit-fuzzy has a low level API that provides functions for creating and manipulating fuzzy variables, and a high-level API that provides fuzzy rules and an inference engine that evaluate the rules against the user's data. The high-level components of scikit-fuzzy are:

1. The `Antecedent` class represents an input fuzzy variable. It is created with a name string and a "universe of discourse" - a numpy linspace array that defines the range of values it can take.
2. The `Consequent` class represents an output fuzzy variable and is defined in the same way as an `Antecedent` with the addition of a defuzzification method. Several defuzzification methods are available such as centroid, bisector and mean of maximum.
3. Once a fuzzy variable has been created it needs to have terms defined on it. A term is a membership function over part of the universe of discourse. The `Antecedent` and `Consequent` classes have an `automf` function that will automatically create overlapping triangular membership functions over the universe of discourse. The user may provide a list of names for the terms, or default names may be used. In the above example "petal_width" would be an `Antecedent` and the terms could be "very_narrow", "narrow"

"average", "wide", "very_wide". Similarly "setosa" would be a `Consequent` with terms "likely" and "unlikely".

4. `Antecedent` terms can combined to create a new compound term using python operators "&", "|" and "~" for the AND, OR and NOT fuzzy operators.

5. A `Rule` class is created with an `Antecedent` term expression and a list of `Consequent` instances. For example the above rule would be created with:

```
Rule(
    ~petal_width['average'] & petal_length['long'],
    [setosa['unlikely'], versicolor['unlikely']]
)
```

6. To run the Fuzzy Inference System a `ControlSystem` instance is created a with a list of rules, and a `ControlSystemSimulator` is created with the control system as a parameter. The input data is passed into the `ControlSystemSimulator.input` method and `ControlSystemSimulator.calculate` is called. The result for each consequent is then available in `ControlSystemSimulator.output["consequent-name"]`.

3.1.3 OpenAI Gym Reinforcement Learning Platform

Gym (Brockman *et al.*, 2016) is a framework for reinforcement learning and a collection of environments for training RL agents. It provides a simple and flexible API that agents can use to explore and interact with an environment. The environments supported include classical control problems, 2D and 3D physics engines and classic Atari console video games. It has become a standard platform for doing RL in python and a range of compatible third party environments are also available.

The core parts of the API are:

1. The environments are registered with the gym and can be created by passing the environment name to the `gym.make` method, e.g. `env = gym.make("CartPole-v1")`
2. the `env.observation_space` defines what an agent can "see" at each step and the `env.action_space` defines what an agent can do at each step. An agent can interrogate these spaces to determine the range of possible inputs and outputs.
3. The `env.reset()` method set the environment to its initial state and returns an observation of that state.
4. The `env.step(action)` takes an action provided by the agent and returns a tuple of `(observation, reward, done, info)`.

- `observation` is information about the updated state of the environment
- `reward` is the reward (positive or negative) for taking that step

- `done` is a Boolean flag indicating if the run is completed
- `info` may contain diagnostic information specific to an environment, and should not be used by the agent for learning

5. `env.render()` will display the current environment (e.g. one frame of an Atari game) and can be used to create a video display of the agent in action. This is usually omitted during training to speed up the process.

The `observation_space` and `action_space` are subclasses of `gym.spaces.Space` and will be one of several types. The most common are:

1. `Discrete(n)` - the observation or action is a single integer in a range 0-n.
2. `Box(low, high, shape, dtype)` a numpy array of the given shape and dtype where the values are bounded by the high and low values. This may be a simple linear array or multidimensional. For example the Atari games often have an observation space of `Box(low=0, high=255, shape=(210, 160, 3), dtype=np.uint8)`, where the Box is a 3-dimensional representation of the screen pixel RGB values.

Other types of observational space exist, such as dicts or tuples of spaces, but they are rarely used in practice.

3.1.4 Other third party libraries

Other third-party libraries used are:

1. tensorboardX (https://tensorboardx.readthedocs.io/en/latest/tensorboard.html) is used to optionally write performance data in a format that can be displayed by TensorBoard (https://www.tensorflow.org/tensorboard/)
2. Numpy for general numerical and array manipulation
3. Pandas for handing classification on pandas dataframes.
4. scikit-learn in order to make the FuzzyClassifier class a compatible scikit-learn classifier.

3.2 Design Overview

The code for the project is in a python package `evofuzzy` which contains four modules.

- **fuzzygp.py** has the low level functions for handling the genetic programming operations and the conversion between DEAP PrimitiveTree instances and scikit-fuzzy rules. The two main entry points are:
  - `register_primitiveset_and_creators` function creates a `TypedPrimitiveSet` initialised with all the primitives and reg-

isters it with a deap toolbox, along with functions for creating individuals and complete populations.
  - `ea_with_elitism_and_replacement` is the main evaluate / evolve loop, loosely based on the `eaSimple` function provided by DEAP. It takes a population and a toolbox instance and will iterate for a fixed number of generations, evaluating each individual with the `toolbox.evaluate` function then mutating and mating them according to their fitness. It also performs several other services including:
    * handling small batches of data for the classifier
    * parallelising the evaluation of the population members over multiple cores
    * recording statistics about the population, such as mean and best fitness using the DEAP Statistics class.
    * optionally saving the statistics and other information in a format that can be read by TensorBoard.
    * elitism - preserving the best individuals from one generation to the next
    * replacing the worst performing individuals with newly generated ones, to prevent loss of diversity

- **fuzzybase.py** contains one class, `FuzzyBase`. This is the base class for the classifier and reinforcement learning implementations and has methods for initialisation of the hyperparameters and logging of statistics, and for executing the `fuzzygp.ea_with_elitism_and_replacement` function to run the evolve/evaluate loop. It also has methods for saving and loading the state of the class and properties for returning the best performer from the current population, both as a PrimitiveTree and as a human-readable string.

- **fuzzyclassifier.py** has a `FuzzyClassifier` class that inherits from `FuzzyBase` and does classification in a manner consistent with scikit-learn classifiers. it has two public methods in addition to those in the base class:

  - `fit` takes X and y parameters for the training data and ground truth classes and trains the classifier.
  - `predict` takes a X in the same format as it was trained on and predicts the classes.

- **gymrunner.py** has a `GymRunner` class that inherits from `FuzzyBase` and has two public methods in addition to those in the base class:

  - `train` takes an openAI Gym environment and trains a population of fuzzy rules to play it
  - `play` takes an openAI Gym environment and plays it with the current top performer, rendering a video of it in action.

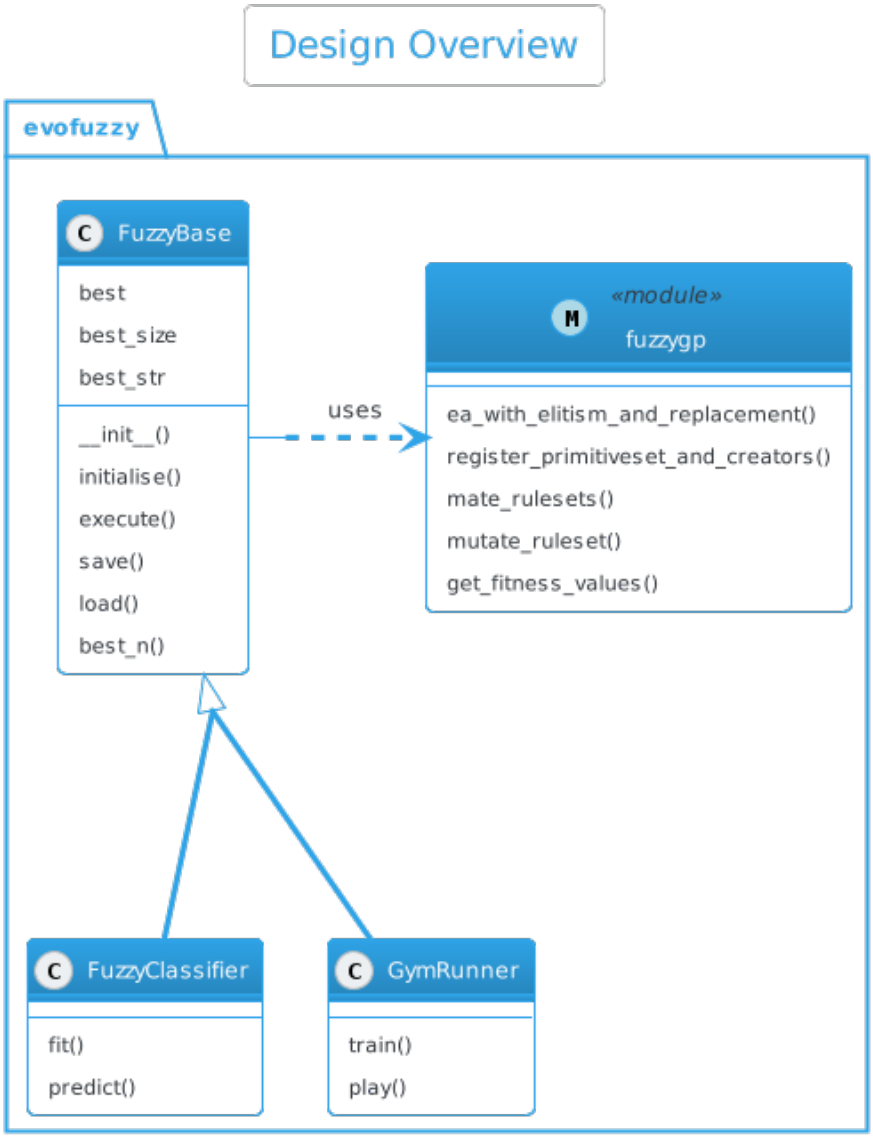

Figure 3.1: Public API of the evofuzzy package

4 Implementation

This section goes through the steps that were taken to implement the evofuzzy package and the decisions made along the way. For instructions on how to use the final version see the User Guide in Appendix I.

The implementation was done iteratively over several stages. The classifier code was developed first then support for reinforcement learning was added. The development process alternated between exploratory programming in Jupiter notebooks and test driven development using a combination of unit tests written with pytest and test scripts running against small data sets such as Fisher's iris dataset or simple gym environments such as CartPole.

4.1 Stage 1: Classification with hand-coded fuzzy rules

The first stage focused purely on using fuzzy rules for classification. Rules for classifying the iris dataset were written by hand after inspecting the distribution of the data. Only two of the features were used, "petal_width" and "petal_length", each with three terms. The rules that were written were

```
rule1 = ctrl.Rule(petal_width['narrow'] & petal_length['short'], setosa['likely'])
rule2 = ctrl.Rule(petal_width['medium'] & petal_length['medium'], versicolor['likely'])
rule3 = ctrl.Rule(petal_width['wide'] & petal_length['long'], virginica['likely'])
```

A function was written that read each row of a pandas DataFrame containing the iris data and passed it to the fuzzy controller. The rules were applied to the features and calculated the activation strength of each of the consequents. The argmax of the activation strengths were used to pick the predicted class.

This method achieved an accuracy of 86.66%, showing that classification with simple fuzzy rules was viable. A `FuzzyClassifier` class was created to encapsulate this functionality in a `predict` method.

4.2 Stage 2: Encoding fuzzy rules in DEAP GP trees

The second stage was to create a DEAP primitive set that could be used to generate and manipulate fuzzy rules. Normally when using the `deap.gp` module, calling `compile` on an expression tree would either produce a function that would then be called with the user data as parameters, or if the expression tree was defined without parameters then to call it directly. For this project what was needed was for the `deap.gp.compile` function to return a scikit-fuzzy `Rule` instance. Fortunately in python functions and classes are somewhat interchangeable in that both are callable objects, and calling a class object creates an instance of that object.

The `evofuzzy.fuzzygp._make_primitive_set` function does the work of defining the primitive set that is at the heart of the evofuzzy code. This takes a list of scikit-fuzzy `Antecedent` instances and a list of `Consequent` instances and returns a primitive set with the types and functions registered on it for creating fuzzy rules.

A `PrimitiveSetTyped` instance was created that takes no parameters and has the type of the return value set to the `skfuzzy.control.Rule` class and the following primitives are added to it:

- The terms of each of the `Antecedent` instances were added as terminal nodes of type `Term`.
- the `Rule` class is added as a primitive that has a `Term` and a `list` as parameters, and returns a `Rule` instance.
- functions for the "and" "or" and "not" operators were added, using `operator.and_`, `operator.or_` and `operator.invert` from the standard library. Each of these are defined as taking two `Term` instances as parameters and returning a new `Term`.
- for the consequents an ephemeral constant was used that creates a list with a single consequent term selected at random from those available. Since DEAP can only handle a fixed number of parameters for each primitive this was found to be the simplest way to handle having a list as parameter. A class is used for the ephemeral constant that has a `__str__` method to return the list as a string that can be compiled as valid python code.
- an identity function was added as a primitive that takes a list and returns the same list unchanged. This is so that if mating or mutating tries to modify the consequents list, the only option is to pass it through the identity function, since this is the only function that takes that type.

Once the `_make_primitive_set` function was completed a function was added to create a rule from the primitive set. The `deap.gp.genGrow` function was used, which generates a random tree from the primitives where the branches may be of different lengths. Another function was added to create a `RuleSet` that consist of a random number of rules. The `RuleSet` class is a subclass of `list` that has the `__len__` method overridden to return the sum of the length of all the contained rules. This is necessary because the DEAP library uses `len(individual)` when controlling bloat. A `length` property was added to return the number of rules. The `register_primitive_set_and_creators` function was written to create the primitive set and register it on the DEAP toolbox, along with the functions for creating individuals and populations.

4.3 Stage 3: Adding rule generation to the classifier

The `FuzzyClassifier` class with hand-coded rules created in stage 1 was updated to generate random rules using the functions created in stage 2. The `__init__` method was added that took the hyperparameters for controlling the tree height and number of rules. Since the FuzzyClassifier is intended to be compatible with scikit-learn, the `__init__` method is only used for assigning hyperparameters to local variables of the same name, as required in the scikit-learn developer's guidelines (https://scikit-learn.org/stable/developers/develop.html#instantiation).

The `fit(X, y, ...)` method was added that created the DEAP toolbox and registered the primitive set and creation functions. A helper function was added to generate the scikit-fuzzy `Antecedent` objects. The upper and lower limits for the fuzzy variable are taken from the min and max values in the X data, while the variable names and terms are either taken from a dictionary passed in by the user or derived from the column names & default terms. Another helper function was added to create the `Consequent` objects that represent the target classes. Originally these had a single fuzzy term "likely" that ranged from 0 to 1, but later an "unlikely" term was added that was the inverse function. This enabled more expressive rules to be created.

4.4 Stage 4: Adding learning from data to the classifier

At this point the classifier would generate a random rule set and use it to predict the classes, but this clearly performs no better than random guessing. The next step was to add learning from the training data by creating a population of individuals and implementing the evaluate-evolve cycle. To do this the core genetic programming operators were added as methods to the class and also registered on the toolbox:

- `evaluate` evaluates an individual by compiling its primitive trees into scikit-learn fuzzy rules and executing them against each row of the input data in turn. The resulting predictions over the entire data set are compared with the actual values in `y` and the accuracy score returned as the fitness value for that individual.

- `mate` takes two individuals and mates them by randomly selecting a rule from each and swapping over randomly selected subtrees.
- `mutate` takes one individual and mutates it by randomly selecting a rule from it and replacing a subtree with a newly created subtree

A function was implemented in the `fuzzygp` module to run the evaluate-mutate loop - originally called `eaSimpleWithElitism` and based on the version in (Wirsansky, 2020), it was later rewritten as features were added and renamed to `ea_with_elitism_and_replacement` to conform to the PEP8 naming conven-

tion. The function loops round a fixed number of times given by the `n_iter` hyperparameter. Each time round the loop it:

- evaluates each member of the population against the training data to calculate their fitness
- creates a new generation by selecting members based on their fitness then randomly mutating or mating them.

The function also optionally prints statistics about the population each generation, including the best and average fitness and sizes.

The original version of the function implemented elitism - preserving the best members of the population between generations - using the DEAP `HallOfFame` class. This was later changed in favour of sorting the population by fitness each iteration and working with slices of the list, which significantly simplified the code.

4.5 Stage 5: Improving the classifier

At this point the system was a successful classifier - the first attempt at classifying the iris dataset got an accuracy of 88% with a population of 20 over 20 generations- slightly better than achieved with the hand-written rules. However there was plenty of room for improvement - the classifier was very slow, taking around 50 seconds to train on the 150 iris data points, and was slow to converge.

Improvements to the initial classifier were added over several iterations:

4.5.1 Parallelising the evaluation with multiprocessing

Evolutionary algorithms are "embarrassingly parallel" so a significant speedup was obtained by using a `multiprocessing.Pool` to evaluate the population in parallel with a pool of workers. The run time for training on the iris dataset went from about 50 seconds down to 12 seconds on an 8-core Linux desktop, for a 4-fold speedup.

4.5.2 Supporting rules with multiple consequents

The initial implementation only allowed for a single `Consequent` term in a rule, which limited their expressiveness and meant more rules were required to cover all the possible output classes. The code for creating an ephemeral constant was modified to create a consequent list containing a random selection of the available consequents. The number of consequents to include was randomly selected to be from 1 to half the number of available consequents, rounded up.

4.5.3 Using Double Tournaments to reduce bloat

A recurring problem in genetic programming is that the trees tend to grow over successive generations as subtrees are replaced through mutation and mating, a problem known as bloat in the GP literature. Having lots of large trees are slower evaluate and slower to converge to good solution, since the search space is correspondingly larger.

To reduce bloat the standard tournament selection was replaced with a double tournament as described in (Luke & Panait, 2002). This selects individuals for the next generation through two rounds of tournaments:

1. A series of fitness tournaments are held where in each round `tournament_size` individuals are selected at random from the population and the fittest is chosen as the winner to go into the next round.
2. a second series of tournaments is held where pairs of candidates from the previous round are selected and the smallest is selected with a probability controlled by the `parsimony_size` hyperparameter. This is a value between 1 and 2, where 1 means no size selection is done and 2 means the smallest candidate is always selected. In the paper cited above, values in the range 1.2 to 1.6 were found to work well for their experiments.

4.5.4 Adding whole-rule mating and mutating

At this point mating and mutating only happen on a single branch of one of the rules that make up an individual, which may potentially only make a small change to its fitness. To enable larger changes to take place the ability to mate or mutate at the level of complete rules was added. When an individual is selected for mutating there is a probability controlled by the `whole_rule_prob` hyperparameter that an entire rule will be replaced by a newly generated rule. Similarly when two individuals are selected for mating there is the same probability that they will swap entire rules.

In practice this was not found to make much difference to the performance on the data sets that have been studied.

4.5.5 Adding new individuals to reduce diversity loss

Another problem common in evolutionary algorithms is loss of diversity, where a moderately good genotype outperforms the others and spreads through the population, resulting in convergence on a sub-optimal local maxima. To avoid this each generation a number of new individuals are created and added to the population. The number to add is controlled by the `replacements` hyperparameter.

4.5.6 Adding support for TensorBoard

To assist with evaluation and tuning of hyperparameters support was added for writing information to disk in a format that can be displayed by TensorBoard (https://www.tensorflow.org/tensorboard/). It used the tensorboardX library (https://tensorboardx.readthedocs.io/) to write the data. The `fit` function was extended to take an optional `tensorboardX.SummaryWriter` instance and this was used to save:

- at the start of a training run:
  - the hyperparameters used for training
- after each iteration:
  - the highest and average fitness of the population
  - the smallest and average size of the individuals
  - the fitness of the entire population as histogram data
  - the size of the entire population as histogram data
  - the number of rules each individual has as histogram data
- at the end of training:
  - the rules of the best individual as human-readable text
  - the size of the best individual

tensorboardX has no dependency on TensorBoard, so it is not necessary to install TensorBoard to save the data, only to view it afterwards.

Figures 4.1 shows an examples of TensorBoard comparing several runs of the classifier on the iris dataset.

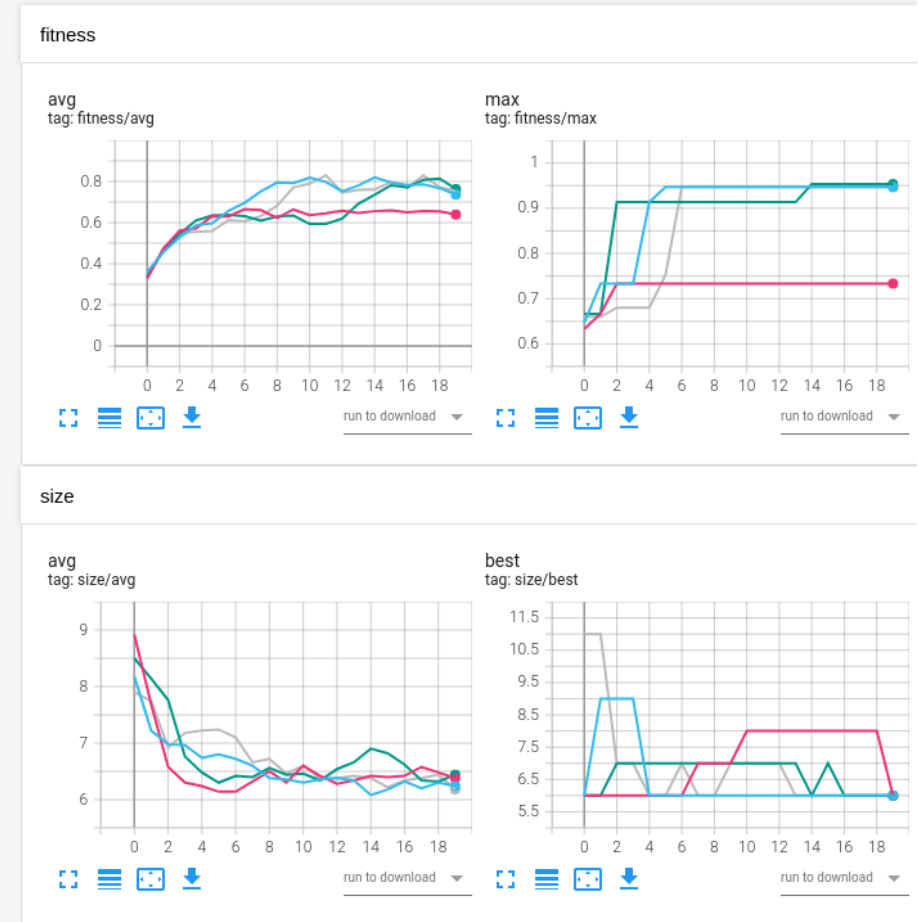

Figure 4.1: example TensorBoard display of scalar values

Figures 4.2 shows an examples of TensorBoard displaying histograms of how the fitness and sizes of the entire population changes over the 20 iterations.

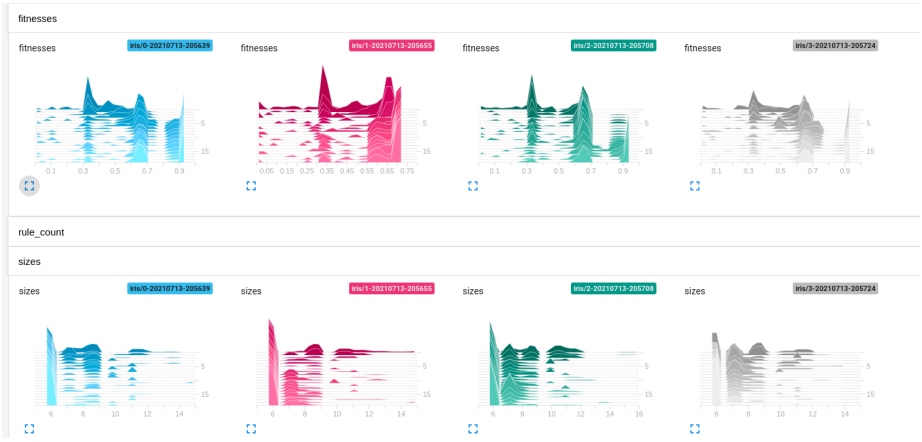

Figure 4.2: example TensorBoard display of histograms of population fitness and sizes

Figure 4.3 shows the TensorBoard display of the rules for the best individual after a training run.

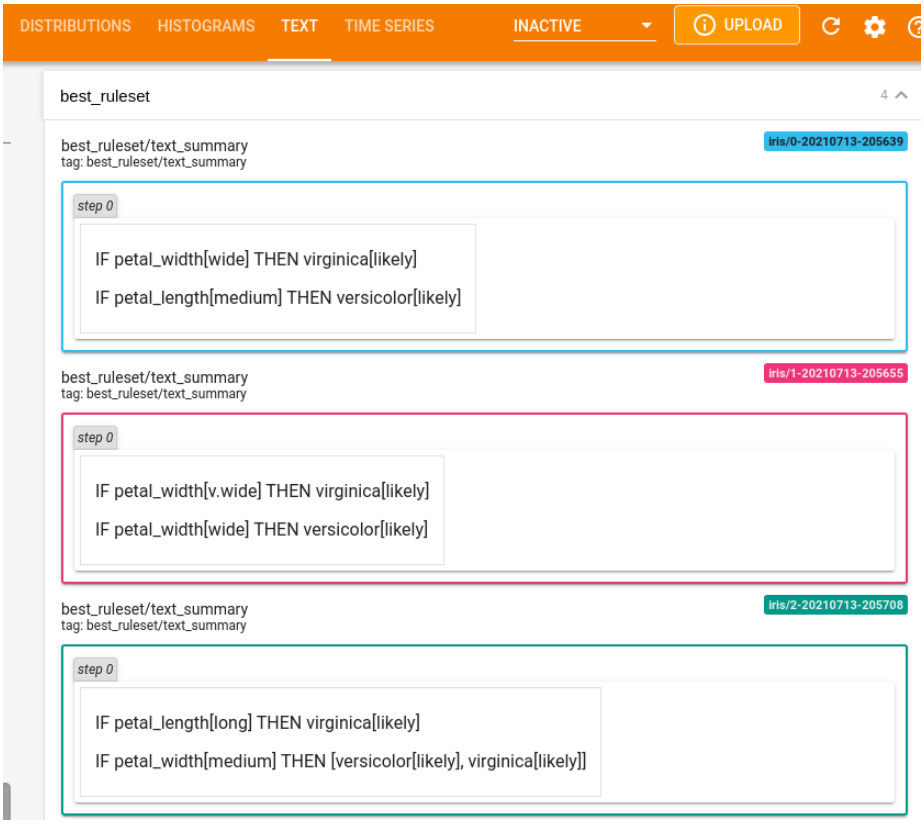

Figure 4.3: example TensorBoard display of best individual

4.5.7 Adding rule pruning

It was noticed that rules were often created with redundant terms, for example (in pseudo-code) "IF NOT(NOT(X)) THEN ...", "IF X AND X THEN ..." and "IF X OR X THEN..." could all be replaced with "IF X THEN ..." without changing the meaning of the expression. This redundancy was unnecessary bloat that slowed down execution of the rules and contributed nothing to the fitness. It also made the final rules less readable. To alleviate this `_prune_rule` and `_prune_population` functions were added that searched for this kind of redundancy and remove it. The population is then pruned just before it is evaluated.

4.5.8 Adding "unlikely" term to consequents

The rules at this stage can only assert that a target class is "likely" which hampers their expressiveness. Adding a second "unlikely" term to the consequents that was the negation of the "likely" term enabled. The rules could now express statements such as

```
IF petal_width[wide] THEN [versicolor[likely], setosa[unlikely]]
```

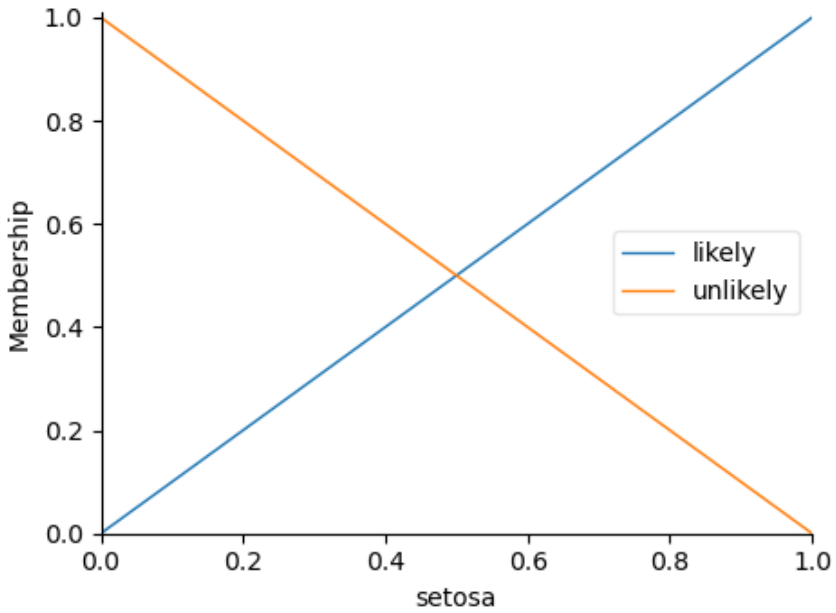

Figure 4.4: Setosa consequent fuzzy mapping with "unlikely" term

4.5.9 Adding mini-batch learning

Although at this point the classifier learns from the data, it only updates the population after each complete pass through the training data. This means that for large datasets the convergence on a good solution is extremely slow. To resolve this mini-batch learning was added, controlled by an optional `batch_size` hyperparameter. The implementation was done by converting the `batch_size` into a list of python `slice` objects and passing the list to the `ea_with_elitism_and_replacement` function. This list is iterated over in the main loop and each individual is evaluated against the current slice of the data then the next generation is evolved. The data and ground truth arrays are shuffled before `ea_with_elitism_and_replacement` is called in case the data is organised in order of the class values - that would have resulted the population being trained on batches where the output classes are all the same leading to poor generalisation.

This code change results in much faster convergence since there are far more opportunities for learning. Previously if the data set had 1000 data points then over 10 iterations the population would have evolved 10 times. With the batch size set to 100 then it would have evolved 100 times.

An iteration is still considered a complete pass through the data, so may now consist of many generations. The output of the statistics, both to TensorBoard and through print statements, still only happened at the end of each complete iteration to keep the output to a manageable level.

Figure 4.5 shows a comparison of the best and mean fitness and sizes when classifying the iris data without batching (grey line) and with a batch size of 20 (red line). It can be seen that with batching the population has reached a better solution after five iterations than the run without batching took after 20 iterations. Not only is the fitness higher but the size of the individuals is also smaller. The run was done over 100 data points and the remaining 50 were used for measuring the performance on unseen data. In this case the version without batching had a test accuracy of 78% while the version with batching had an accuracy of 96%.

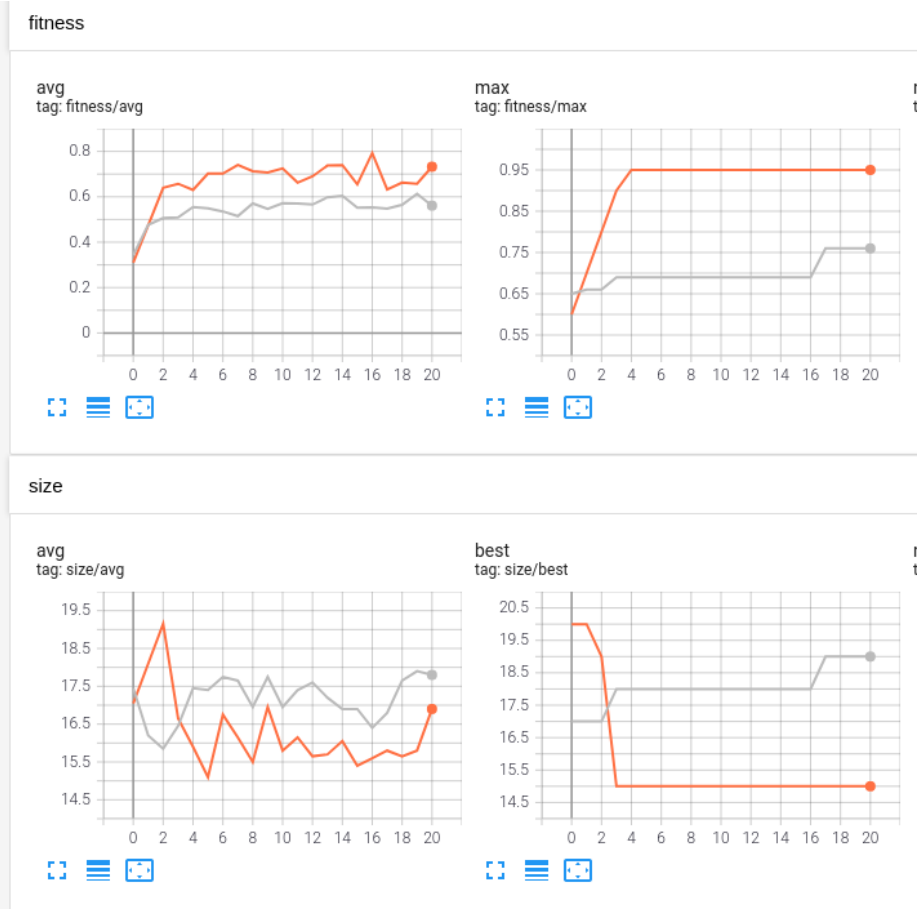

Figure 4.5: iris classification with and without batching

4.6 Stage 6: Refactoring in preparation for adding the GymRunner class

In preparation for adding support for reinforcement learning the code was extensively refactored to create a superclass of `FuzzyClassifier` called `FuzzyBase` and common code was moved to it. Top level functions that would be common to both were moved into the `fuzzygp` module.

At the end of this process the `FuzzyBase` class had the following methods:

- `__init__` that initialised all the hyperparameters
- `_initialise` that set up the toolbox and registered functions on it plus other initialisation tasks
- `execute` that handled calling the `ea_with_elitism_and_replacement` function with the appropriate parameters

- `_mate` and `_mutate` to handle mating and mutating of rule sets.
- methods for getting the top performing individual either as a RuleSet or a string.

The `FuzzyClassifier` class was left with:

- `fit` which was greatly reduced since much of its contents were moved to the `_initialise` and `execute` methods. What was left was mainly for handling creation of the `Antecedent` and `Consequent` objects and calling the methods on the base class.
- `predict` largely unchanged
- helper functions for creating slices of the data for mini-batch processing.

4.7 Stage 7: Adding the `GymRunner` class

The `GymRunner` class was added as a subclass of `FuzzyBase`, initially with these methods:

- `train` for training the population on a Gym environment.
- `play` for displaying a video sequence of the best individual interacting with the environment.
- `_evaluate` to run a single individual in the environment and returns the total reward it has achieved. The environment is reset to a random starting state every time an individual is evaluated.
- `_evaluate_action` to run a single timestep of an individual in the environment. This maps the current observations from the environment into the antecedent terms, runs the FIS for the individual and converts the consequent into the environments action space and returns the action to perform.

At this stage the antecedents and consequents had to be created by the user and passed into the `train` method. The `antecedents` parameter took a list of `Antecedent` instances that map the observation space to fuzzy variables. A `make_antecedent` function was provided to create an `Antecedent` instance from a name and min and max range, plus an optional list of terms to use. For the `consequents` parameter it took a dictionary mapping a name to an action value.

Only environments with `Discrete` action spaces were supported at this stage (see section 4.1.3). The discrete actions were treated the same way as for the classifier - a `Consequent` was created with "likely" and "unlikely" terms and the action with the highest score was chosen as the return value.

At this point it was possible to train a GymRunner instance to play some simple environments with discrete actions such as CartPole and MountainCar.

4.8 Stage 8: Handle Box actions and auto-generate Antecedents and Consequents

The next step was to create the antecedents and consequents automatically by inspecting the observation space and action space respectively and to handle `Box` (continuous) actions.

For the antecedents, a helper function `_make_antecedents_from_env` was written that inspects the environment's `observation_space` attribute to find the shape of the space and the maximum and minimum that each value can take, then creates an `Antecedent` for each one. The names of the antecedents are "obs_0", "obs_1" etc and the terms are the default scikit-fuzzy terms of "lower", "low", "average", "high" and "higher". Currently only one-dimensional `Box` spaces are supported.

One issue that was encountered was that many environments give the upper and lower bounds as "inf" and "-inf", which caused the antecedents to fail. To get round this an `inf_limit` parameter was added to the `train` method that clipped any "inf" antecedents to that value. The default was arbitrarily chosen to be 100.

The user may still provide their own list of antecedents definition and these will be used instead of the auto-generated ones.

For the consequents the `action_space` was inspected to see what type it was and different kinds of consequents created depending on whether it was a `Box` or `Continuous` space. For the `Discrete` space a binary `Consequent` was created for each possible action, as described previously. For `Box` spaces a consequent was created for each output with the min and max values taken from the action space and the terms the same as for the antecedents. Both box and continuous consequents were named "action_0", "action_1" etc.

4.9 Stage 9: Further improvements

4.9.1 Adding EWMA of fitness values

It was observed that because the gym environment is reset to a random state every time an individual is evaluated, the individual may perform well on one run of the gym environment but badly on another. Similarly an individual in the classifier may perform well on one batch but badly on another. This could result in an individual who is a good performer overall being removed from the population prematurely, or a bad performer that got lucky on one batch spreading his genes through population. To counter this an optional exponential weighted moving average (EWMA) was included in the fitness calculation, so that some memory of the fitness from previous evaluations may be preserved. This is controlled by the `memory_decay` hyperparameter which can take a value

between 0 and 1, where 1 is no memory and only the latest fitness value is used, and 0 means only the first fitness value is used. It is calculated as

$fit_t = \alpha \times fit_{calc} + (1 - \alpha) \times fit_{t-1}$

where

- α is the `memory_decay` hyperparameter
- fit_{calc} is the true fitness from evaluating the individual against the environment or data
- fit_{t-1} the adjusted fitness from the previous evaluation
- fit_t is the adjusted fitness for the current evaluation

The `memory_decay` parameter defaults to 1, so no memory of the previous evaluations is preserved.

4.9.2 Adding save and load methods

Convenience methods to save and load the state of a `GymRunner` of `FuzzyClassifier` was added to the base class. This pickles/unpickles the object's `__dict__` to a file. The code was also updated for "warm start" learning - if the `fit` or `train` methods are called on the same object multiple times it will carry on learning from where it left off, instead of starting from scratch with a new population.

4.9.3 Experimental feature added - `predict` or `play` with the top N performers

A parameter `n` was added to the `predict` and `play` methods that take the top `n` performers and combines them into a single individual when predicting or playing. The hypothesis is that this should act as a kind of ensemble and improve performance, but this does not appear to be the case so the feature should be considered experimental.

4.9.4 Final Hyperparameters

The final set of hyperparameters available are given below. For an explanation of each see the User Guide in Appendix I.

- min_tree_height
- max_tree_height
- min_rules
- max_rules
- population_size
- n_iter
- mutation_prob

- crossover_prob
- whole_rule_prob
- elite_size
- replacements
- tournament_size
- parsimony_size
- batch_size (ignored by GymRunner)
- memory_decay
- verbose

4.10 Project Schedule and Progress

The project used the Kanban method for planning where the work is divided into stories and given relative sizes in story points, then as each story finishes the next is taken off the backlog. There is no fixed schedule, instead progress is measured in the rate of completion of story points. Figure 4.6 shows the project progress, with the original story point estimates in brackets. It can be seen that some of the estimates were significantly out, with some of the early stories taking less time than anticipated and later stories longer.

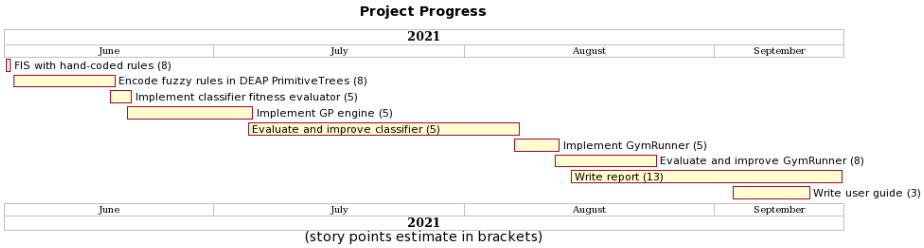

Figure 4.6: Project progress Gantt chart

5 Evaluation

The package was evaluated against several different datasets for classification and gym environments for reinforcement learning. The results were saved to TensorBoard for analysis.

5.1 FuzzyClassifier evaluation

The initial development of the classifier was done by running the rules against the entire iris dataset with no data held out for evaluation since the purpose at that point was to test that the code worked. For evaluating the accuracy on test data a script was written for 5-fold cross validation of the iris dataset. The cross validation functionality was then refactored into a helper function `classifier_cv.cross_validate` that could be used with other datasets. The standard scikit-learn `sklearn.model-selection.cross_validate` function was not used because that would not allow tensorboardX to be used for recording results or the antecedents and consequents for the classifier to be specified.

The `cross_validate` function uses `sklearn.model_selection.StratifiedKFold` to create a 5-fold split of the data with the distribution of the target classes balanced between each split.
For each split:

- If the TensorBoard directory is specified a subdirectory is added with its name containing the split number, current date and time, then a `tensorboardX.SummaryWriter` is instantiated to log into that directory.
- a `FuzzyClassifier` is created with any hyperparameters passed into the function
- the classifier is trained against the training data and evaluated against the test fold
- the accuracy and confusion matrix are printed out and also saved to TensorBoard.

At the end of the five folds, the average accuracy and standard deviation are printed.

Running five-fold CV on a large dataset could be very time consuming, so a flag was added to the parameters to optionally swap the train and test data, so the model was trained on one fifth of the data and evaluated on four-fifths. This gave less accurate results but a five-fold speedup.

5.1.1 Iris dataset results

Figure 5.1 shows a typical result from cross validation of the iris dataset with a batch size of 10 and a population of 20 trained over 10 iterations. Most of the folds had reached 100% training accuracy by the fifth iteration.

The average accuracy on the test data was 93.33% with a standard deviation of 7.81%. The training run time for each fold was 18-20 seconds.

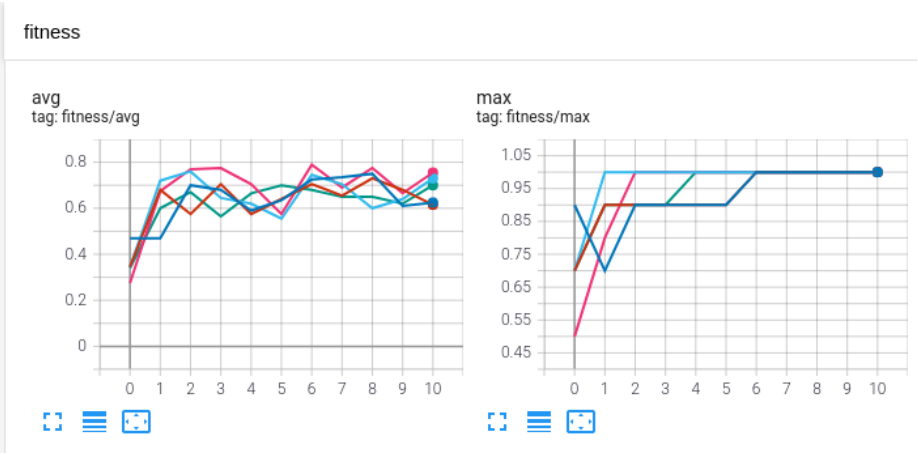

Figure 5.1 typical iris CV result

5.1.2 Wisconsin Breast Cancer dataset results

The Wisconsin Cancer dataset (Wolberg & Mangasarian, 1990) was chosen as a more challenging task. The dataset was originally from the UCI Machine learning repository (Dua & Graff, 2017) and accessed through the OpenML catalogue via the scikit-learn `sklearn.datasets.fetch_openml` function. There are two version of this dataset available, so the smaller version with 10 features and 699 instances was chosen (https://www.openml.org/d/15).

Figure 5.2 shows a typical result with a population of 50, a batch size of 50 and 5 iterations. The run time for each fold ranged from 34 to 42 seconds. The average accuracy on the test data was 93.7% with a standard deviation of 1.9%.

Here is an example of the best rule set generated by one of the runs, with an accuracy on the test set of 94.89%.

```
IF Single_Epi_Cell_Size[very_low] THEN [malignant[unlikely], benign[likely]]
IF Bare_Nuclei[medium] THEN benign[unlikely]
IF Cell_Shape_Uniformity[very_low] THEN [benign[likely], malignant[unlikely]]
IF Bare_Nuclei[very_high] THEN malignant[likely]
IF NOT-Cell_Size_Uniformity[very_low] THEN [benign[unlikely], malignant[likely]]
IF Normal_Nucleoli[low] THEN malignant[likely]
```

This shows how easy it would be for a domain expert to interpret the reason the system gave for a prediction.

The confusion matrix for this run is shown in Table 5.1.

actual \ predicted	benign	malignant
benign	82	7
malignant	0	48

Table 5.1 confusion matrix for Wisconsin cancer classification

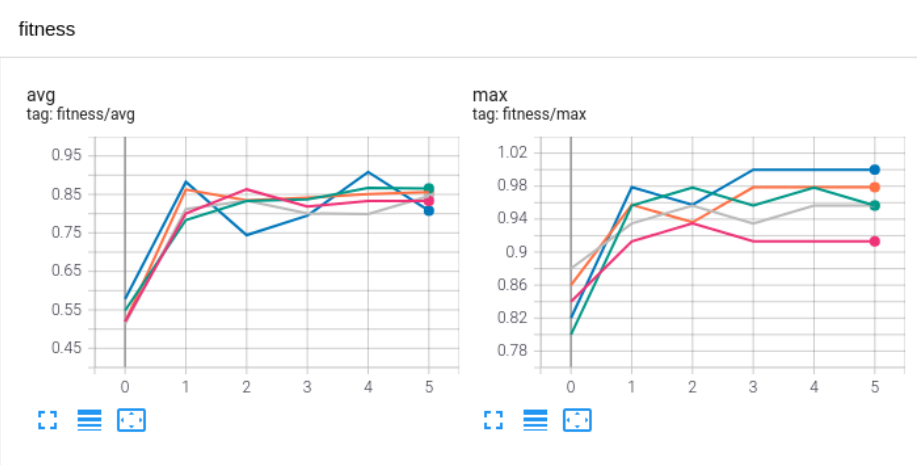

Figure 5.2 typical cancer CV result

5.1.3 Segmentation dataset results

The segmentation dataset (https://www.openml.org/d/40984), also from the UCI ML repository via OpenML, is a much more challenging task than the previous two. It is a dataset of information about 3x3 pixel squares taken from outdoor images, such as mean RGB values, intensity, measure of horizontal and vertical edges etc. The task is to classify the pixels into one of seven classes - “brickface”, “sky”, “foliage”, “cement”, “window”, “path” and “grass”. There are 17 features in total and 2310 rows.

This dataset performed very poorly. It was much slower than the previous datasets, partly because there were far more rows of data, but also because the larger number of feature and classes meant more rules and larger rules were used to model it. Each fold took between 8 and 14 minutes to run. It was also far less accurate, both on the training and test sets. After 5 iterations with a population of 50 and a batch size of 30, it’s best accuracy on the training set was 72.2% but that fold only scored 55.4% accuracy on the test set. The mean accuracy on the test set was 54% with a standard deviation of 6.68%. For comparison, the scikit-learn RandomForestClassifer managed 5-fold CV on the dataset in under 2.5 seconds with an accuracy of 94%.

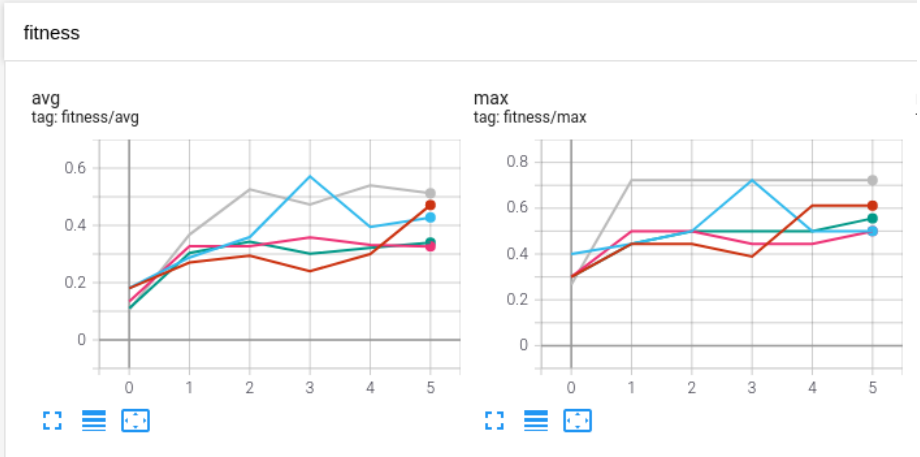

Figure 5.3: typical segmentation CV results

Further research is needed, but my hypothesis is that the reason it performs so poorly is due to the large number of target classes, rather than the number of features. It may also be that in this particular dataset the relationship between the features and the target class simply does not map well into how the fuzzy rules work.

5.2 GymRunner evaluation

5.2.1 CartPole-v1

Evaluation of the GymRunner class started with the CartPole environment (https://gym.openai.com/envs/CartPole-v1/). This is a classic reinforcement learning exercise where a hinged pole on a cart has to be kept upright by moving the cart left and right.

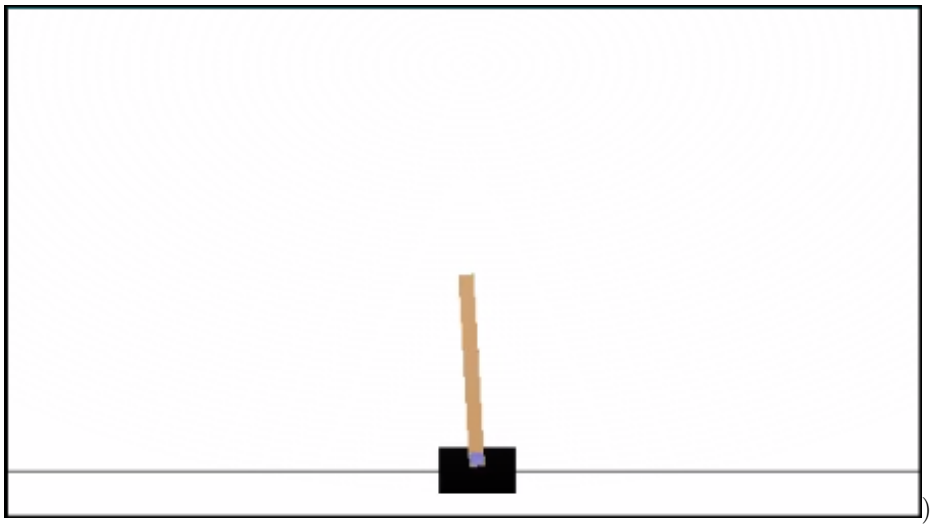

Figure 5.4: CartPole environment in action

The observations that can be made by the agent are:

- 0: cart position relative to the centre. From the centre to the edge of the display is +/- 2.4 units
- 1: cart velocity. The environment's `observation_space` specified the value as -inf to +inf, but in practice was found to be between -1 and 1
- 2: pole angle from the vertical in radians. The `observation_space` gave the limits as -0.418 to 0.418, but the stopping condition effectively limited it to -0.209 to 0.209.
- 3: pole angular velocity. This was also specified as between -inf and inf, but in practice was between -2 and 2.

The action generated by the agent is a discrete value - 0 to push the cart left, 1 to push the cart right. There is no option to not move the cart.

The game ends when the cart reaches the edge of the display, the angle of the pole goes outside the range of +/- 0.209 radians (12 degrees), or the game reaches 500 timesteps. A reward of 1 point is given for every timestep, so the maximum reward is 500.

To run CartPole, these antecedents were used:

```
antecedents = [
    make_antecedent("position", -2.4, 2.4),
    make_antecedent("velocity", -1, 1),
    make_antecedent("angle", -0.25, 0.25),
    make_antecedent("angular_velocity", -2, 2),
]
```

With these antecedents, a population of 50 could learn to get a maximum score in under ten generations, and often in 2-4 generations as shown in Figure 5.5.

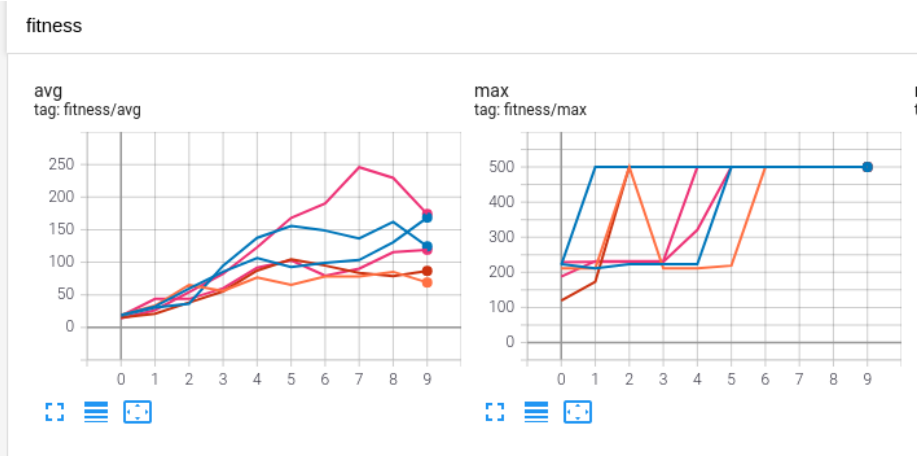

Figure 5.5: CartPole v1 results

5.2.2 MountainCarContinuous-v0

The second task for GymRunner was the MountainCarContinuous-v0 environment (https://gym.openai.com/envs/MountainCarContinuous-v0/). Unlike the CartPole environment, this requires a continuous output value from the agent so it a test of GymRunner's ability to handle `Box` action spaces.

The task is to get a car to the top of a steep climb. The car does not have sufficient energy to do it in one go, so has to go back and forth between the left and right slopes to build up momentum.

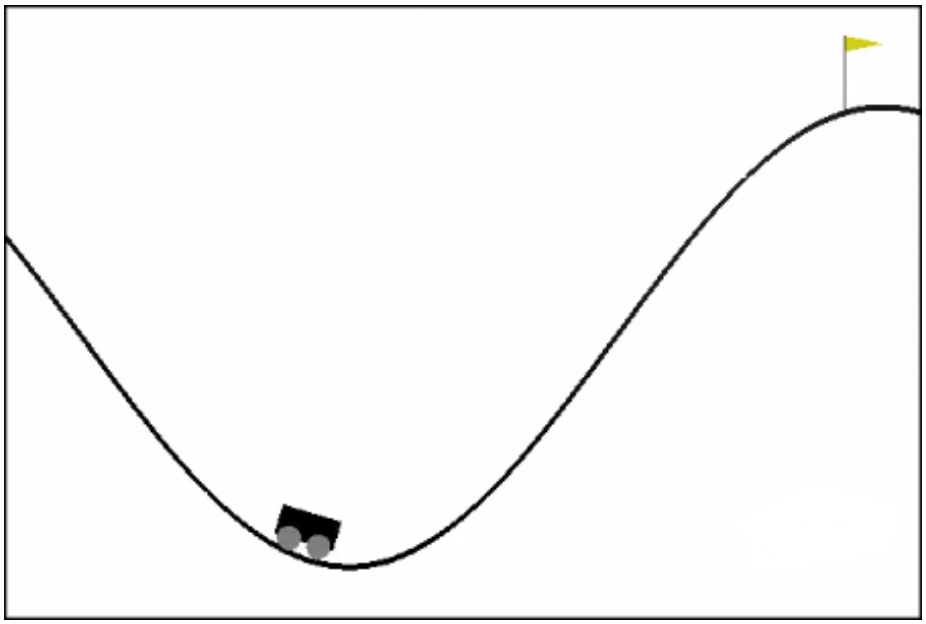

Figure 5.6: MountainCar in action

The observations the agent can make are:

- 0: Car position. The limits are -1.2 to 0.6
- 1: Car velocity. The limits are -0.07 to 0.07

The action the agent can take is a floating point value of the force left or right, ranging from -1.0 to 1.0.

A reward of 100 is given when the car reaches the goal, and a small penalty is given each timestep for the amount of energy expended. A reward total of 90 or more is generally judged to be a success.

Figure 5.7 shows five training episodes with a population of 50 over ten generations. All training episodes manage to achieve a score in excess of 90 after one or two generations.

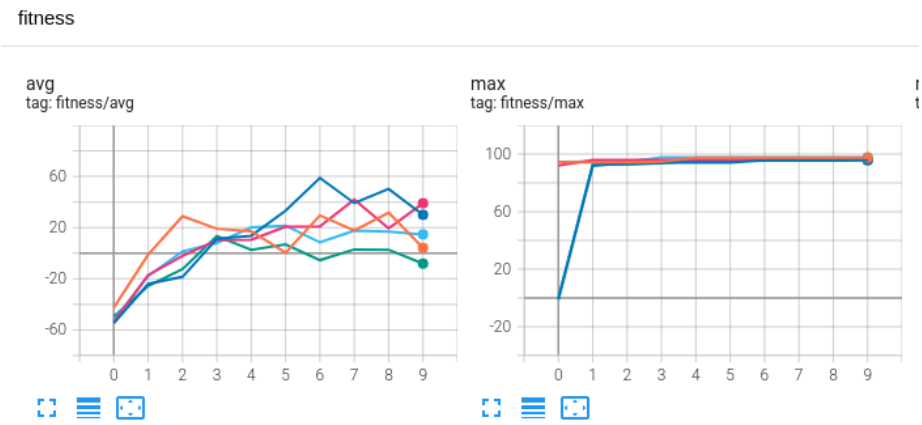

Figure 5.7: MountainCarContinuous results

5.2.3 Pendulum-v0

The Pendulum-v0 environment (https://gym.openai.com/envs/Pendulum-v0/) is a rigid pendulum that rotates about a fixed point. The aim is to move the pendulum into the upright position and keep it there by applying clockwise or anticlockwise torque to it. The pendulum starts at a random angle.

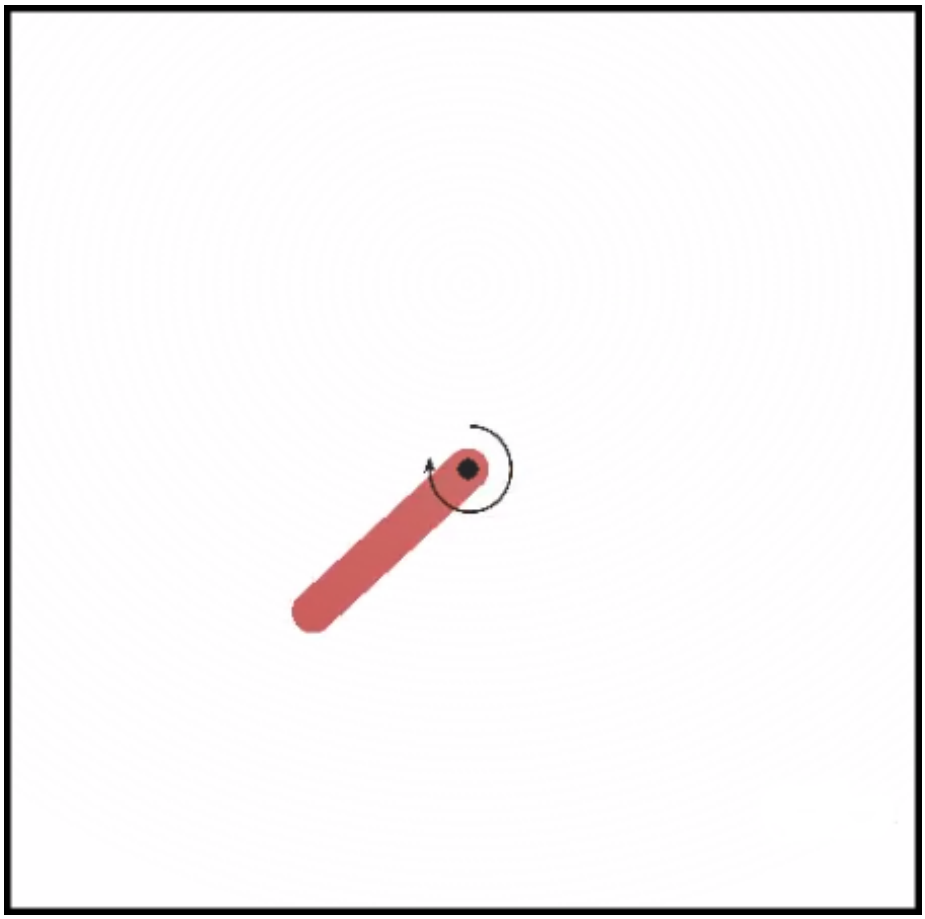

figure 16: Pendulum in action

The observations and action spaces are undocumented, but from the source code it appears that the observations are:

- 0: the x coordinate of the tip of the pendulum, between -1.0 and 1.0
- 1: the y coordinate of the tip of the pendulum, between -1.0 and 1.0
- 2: the angular velocity of the pendulum, between -8.0 and 8.0.

The action is the torque to apply, between -2.0 and 2.0.

The reward is a penalty calculated by how far the pendulum is from the upright position each timestep, so is always negative. The run ends after 200 timesteps.

Figure 5.8 shows the results from for training session with a population of 50 over 50 iterations. Although superficially a similar task to CartPole, the performance was much worse. It took far longer to achieve a reasonable score, and in one of the runs barely learnt anything after 50 generations. For the runs that did achieve a reasonable score in training, replaying them against reset environments

showed that they were highly sensitive to the start position. For some start angles they moved it to the vertical position straight away and held it there for a good score, for other start positions they never managed to gain control and swung wildly.

I hypothesise that part of the reason for this is that the observations are in terms of x and y coordinates of the tip, rather than the angle. To achieve good control the coordinates would need to be translated back into a single angle, which is beyond the realm of the fuzzy rules. Possibly given sufficiently complex rules and enough time to evolve them, they could achieve a reasonable approximation.

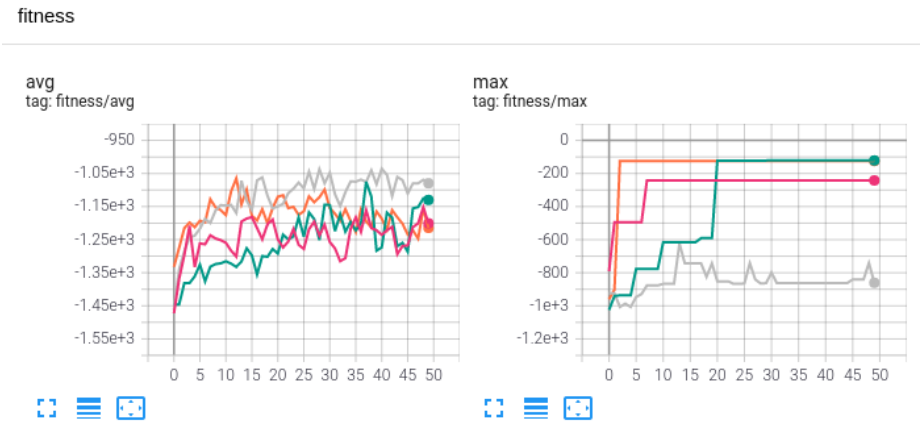

Figure 5.8: Pendulum results

5.2.4 LunarLanderContinuous-v2

The LunarLanderContinuous-v2 environment (https://gym.openai.com/envs/LunarLanderContinuous-v2/) requires the agent to land a 2D lunar module between two flags on a randomly generated terrain and from a random start position.

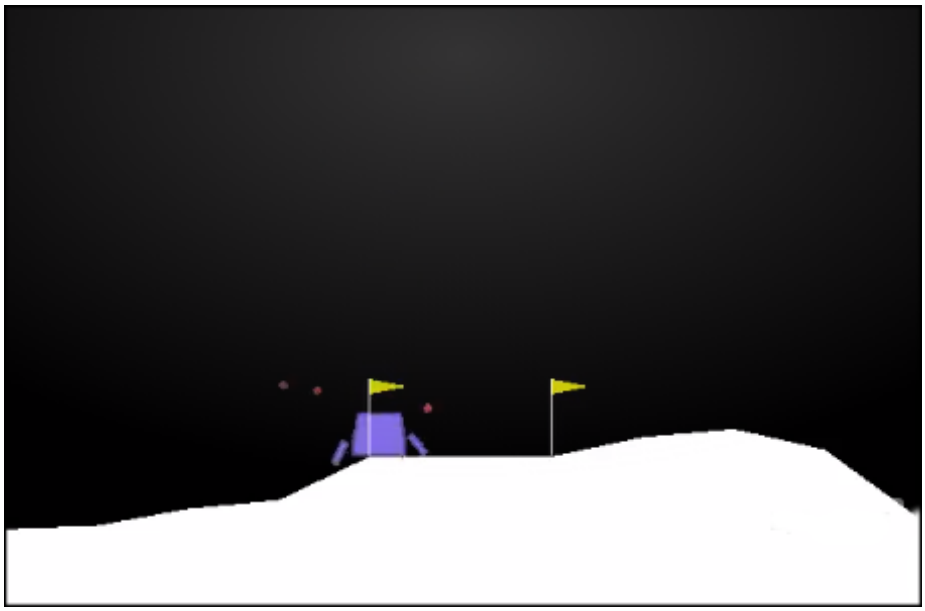

Figure 5.9: Lunar Lander in action

The observations are eight floating point values in the range of -inf to inf. The first two values are the coordinates of the lander, where (0, 0) is the coordinate of the centre of the landing pad. The other values are undocumented.

The actions are two floats, the first is the thrust of the main engine (-1.0 to 1.0) and the second is the thrust for the side engines (-1.0 to 1.0), where negative values are left thrust and positive values are right thrust.

The reward is complex, with up to 300 points for a successful landing, -100 for crashing and a penalty each timestep for firing the engine.

Initial short training runs generally fell into a local optima of not firing the engines at all and so crashing as quickly as possible without getting the engine penalty. However by increasing the population to 100, increasing the maximum number of rules to 10 and training for 1000 iterations it did learn to successfully land most of the time. There is a video demonstrating GymRunner controlling the lander at https://youtu.be/Oo6hulwqr9M.

The best rule set after 1000 generations was:

```
IF obs_4[higher] THEN [action_1[high], action_0[average]]
IF obs_2[high] THEN action_1[lower]
IF obs_4[high] THEN action_1[higher]
IF obs_1[average] THEN action_0[high]
IF obs_5[high] THEN action_1[higher]
IF obs_7[high] THEN [action_1[high], action_0[higher]]
IF obs_4[higher] THEN [action_1[lower], action_0[low]]
IF obs_0[higher] THEN action_0[low]
```

Figure 5.10 shows the score of the best individual over 1000 training iterations. It shows a gradual improvement at first, with occasional high spikes where an individual got lucky. At around 350 iterations there was a marked increase in performance which lasted for around 75 iterations then was forgotten. Then at iteration 650 there was a significant improvement, followed by further gradual improvements. This sort of learning behaviour is common in evolutionary algorithms as "good genes" spread through the population or spread a little way then get weeded out by chance.

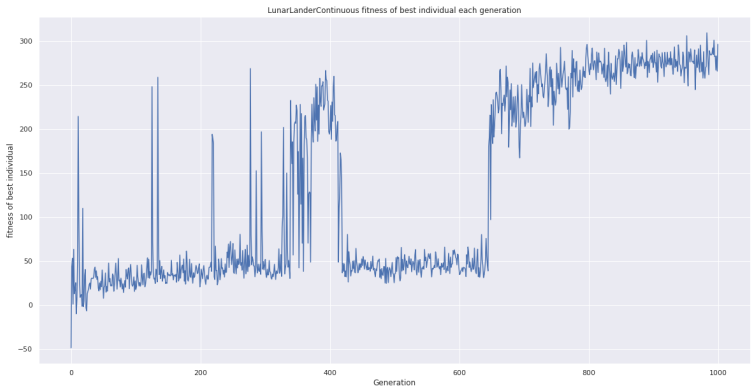

Figure 5.10: Lunar Lander performance

6 Conclusion

This project has shown that genetic programming of fuzzy rules is a viable technique for both classification and reinforcement learning, but does have some weaknesses.

For classification it is best suited to small datasets in situations where explainability is of paramount importance, such as medical diagnosis. With large datasets it is very slow compared to other methods such as Random Forest, and if there are complex dependencies between features or a large number of target classes it may perform poorly.

For reinforcement learning it has been shown to work well in several simple tasks, but has not been tried with more complex environments such as Atari games. Again it has the advantage that the generated rules are highly interpretable, which may be important in some applications such as a decision-making component in an autonomous vehicle.

Some of the weaknesses of the system is that the slow speed and large number of hyperparameters makes tuning the system to a particular dataset or RL environment very time consuming. It may be advisable when tuning the classifier to use a representative subset of the training data and to start with a small number of simple rules then gradually increase the number and complexity to find the best combination for the task.

6.1 Possible improvements and future work

There are a number of improvements that were considered but were omitted due to time constraints.

6.1.1 Re-implement or replace the fuzzy logic library

Profiling of the classifier showed that even with a small batch size, around 90% of the time was spent in the scikit-fuzzy library. That library is implemented in pure python, with the low level maths implemented with numpy. Replacing it with a pure C or Cython version could result in a significant speed improvement. It could also be possible to offload some of the fuzzy calculations to the GPU for further speed improvements, as demonstrated by (Chauhan *et al.*, n.d.) and (Defour & Marin, 2013).

6.1.2 Using weighted rules when merging the top n performers

An experimental feature in the system is to merge the rules for several of the top performers to make one large predictor. At the moment the rules of all

the performers are given equal weight, but it may be beneficial to weight the importance of the rules by the fitness values of the individuals.

6.1.3 Adding predict_proba to the classifier and making the fitness function configurable

Many scikit-learn predictors have a `predict_proba` method that output the class probabilities. It would be possible to add the same method to `FuzzyClassifier` by taking the weights generated for the consequences and normalising them so they sum to one.

The classifier currently uses the prediction accuracy as the metric that it is trying to maximise. This could be made configurable so that other metrics can be used instead, such as F1 score or ROC AUC. Some of the metrics will require the `predict_proba` method to be implemented first.

6.1.4 Add diagnostics to the classifier when making a decision

It would be possible to add a method to the classifier where you pass it a single data point and it outputs the strength with which each rule is triggered when making the prediction. This could be useful for improving explainability such as when using the classifier for medical diagnosis.

6.1.5 Add early stopping to GymRunner

Currently GymRunner will continue running an agent in an environment until the environment signals that it has ended. Some environments may run for a long time before this happens, which slows down the learning process. It could speed things up to run the environment just long enough to make a decision on the fitness of an individual by adding an optional `time_limit` hyperparameter that is the number of time steps to run it for. A refinement of that could be to have a way of increasing the value over time, so it starts off with a small `time_limit` to weed out the bad performers early on, then gradually increases it to allow later generations longer to prove their worth.

6.1.6 Multiple populations - Demes

In biology, demes are populations of the same species that are physically separated so they form separate gene pools. In evolutionary computing the term is used to mean splitting a population into sub-populations with little or no crossover between them. This helps prevent loss of diversity that could result in the entire population converging on a sub-optimal solution. This could be used in the evofuzzy project by splitting the population into two or more demes, training

them in parallel, then combining the predictions of the top performers of each deme.

7 References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., et al. (2016) Openai gym. *arXiv preprint arXiv:1606.01540*.

Chauhan, D., Singh, S., Singh, S. & Banga, V.K. (n.d.) *Speedup of Type-1 Fuzzy Logic Systems on Graphics Processing Units Using CUDA*. 5.

Defour, D. & Marin, M. (2013) *FuzzyGPU: A fuzzy arithmetic library for GPU*. [Online]. Available from: https://hal.archives-ouvertes.fr/hal-00856617 [Accessed: 11 September 2021].

Dua, D. & Graff, C. (2017) *UCI machine learning repository*. [Online]. Available from: http://archive.ics.uci.edu/ml.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., et al. (2012) DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*. 13, 2171–2175.

Holland, J.H. (1975) second edition, 1992. *Adaptation in natural and artificial systems*. Ann Arbor, MI, University of Michigan Press.

Koza, J.R. (1992) *Genetic programming*.

Koza, J.R., Bennett III, F.H., Andre, D. & Keane, M.A. (1999) *The design of analog circuits by means of genetic programming*. 1999. Evolutionary design by computers.

Luke, S. & Panait, L. (2002) Fighting Bloat with Nonparametric Parsimony Pressure. In: Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, Hans-Paul Schwefel, et al. (eds.). *Parallel Problem Solving from Nature — PPSN VII*. Lecture Notes in Computer Science. [Online]. Berlin, Heidelberg, Springer Berlin Heidelberg. pp. 411–421. Available from: doi:10.1007/3-540-45712-7_40 [Accessed: 4 September 2021].

Warner, J., Sexauer, J., scikit-fuzzy, twmeggs, et al. (2019) *JDWarner/scikit-fuzzy: Scikit-Fuzzy version 0.4.2*. [Online]. Zenodo. Available from: doi:10.5281/zenodo.3541386 [Accessed: 3 September 2021].

Wirsansky, E. (2020) *Hands-on genetic algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems*.

Wolberg, W.H. & Mangasarian, O.L. (1990) Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the National Academy of Sciences of the United States of America*. [Online] 87 (23), 9193–9196. Available from: doi:10.1073/pnas.87.23.9193.

Wolpert, D.H. & Macready, W.G. (1997) No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*. [Online] 1 (1), 67–82. Available from: doi:10.1109/4235.585893 [Accessed: 29 August 2021].

Zadeh, L.A. (1965) Fuzzy sets. *Information and Control*. [Online] 8, 338–353. Available from: http://www-bisc.cs.berkeley.edu/Zadeh-1965.pdf.

Zadeh, L.A. (1975) The concept of a linguistic variable and its application to approximate reasoning—I. *Information Sciences*. [Online] 8 (3), 199–249. Available from: doi:10.1016/0020-0255(75)90036-5 [Accessed: 22 August 2021].

Appendices

Appendix I: User Guide

User Manual for the evofuzzy python package

`evofuzzy` is a python package for using Genetic Programming (GP) to evolve a Fuzzy Inference System (FIS) that can be used for classification and for reinforcement learning in the OpenAI Gym environment. It is built on the DEAP evolutionary computation framework and the scikit-fuzzy library. No knowledge of these libraries is needed, but some prior knowledge of FIS and GP will be useful when tuning the system.

Core concepts and terminology

Fuzzy Variables and Terms

A fuzzy variable maps a linear value into a series of linguistic terms, for example if classifying irises then petal length could be represented by a fuzzy variable with the terms "short", "medium", and "long".

Fuzzy Rules, Antecedents and Consequents

A fuzzy rule is an IF - THEN expression that combines fuzzy terms to generate output fuzzy terms. The IF part can combine input variables called Antecedents through AND, OR and NOT operators. The THEN part of the rule specifies one or more output terms, or Consequents. For classification the consequent terms are "likely" and "unlikely" to represent the likelihood of that class.

For example

```
IF petal_length[short] AND sepal_length[short] THEN [setosa[likely], virginica[unlikely]]
```

Fuzzy Inference System (FIS)

A Fuzzy Inference System takes a set of fuzzy rules and some input data, evaluates the rules against that data and de-fuzzifies the output to return a crisp (non-fuzzy) result. This may be a class prediction for classification or actions that a reinforcement learning agent is to perform.

Genetic Programming

In the `evofuzzy` package the fuzzy rules are encoded as expression trees and individual consists of a list of rules.

Creating the population

When the genetic programming system is run an initial population of individuals is randomly generated. Hyperparameters that control the generation of the population are:

- `population_size` sets the size of the population to create
- `min_rules` sets the minimum number of rules that an individual may have
- `max_rules` sets the maximum number of rules that an individual may have
- `min_tree_height` is the minimum depth of a generated tree
- `max_tree_height` is the maximum depth of a generated tree

The latter four hyperparameters control the size and complexity of the individuals and hence the bias-variance trade-off and the execution speed. If an individual has a few small rules it may not have enough complexity to model the relationships in the data and so under-fit. If it has too many rules or they are too large it may over-fit the data. It will also be harder to interpret and run slower.

The evolution cycle

Once the population has been created, the rule set of each individual is evaluated against the data or RL environment and a fitness score calculated. A new population is then created by selecting individuals according to their fitness and randomly mutating or mating them. These are then evaluated again and the process repeated for a number of generations controlled by the `n_iter` hyperparameter.

Also each cycle the best performing individuals are carried over unmodified to ensure that they are not selected out. The number to carry over is controlled by the `elite_size` parameter. Also a number of new individuals given by the `replacements` hyperparameter are created to prevent too much loss of diversity.

By default the FuzzyClassifier will run each individual against the entire training data before creating the next generation. For a large dataset this will be very slow and wasteful, so there is a hyperparameter `batch_size` that will split the input data into small batches and train/evolve the population against each batch in turn. This can lead to much faster convergence. However there is a possibility that an individual may do well against one batch and poorly against another so a good performing individual overall may be weeded out by one poor batch. To counter this there is a `memory_decay` hyperparameter that controls an exponential weighted moving average of the fitness values over

successive evaluations. This is a value between 0 and 1, where 1 (the default) only remembers the most recent fitness value, and 0 only remembers the first fitness value.

The `batch_size` hyperparameter is ignored by the GymRunner class, but the `memory_decay` hyperparameter is used because the individuals may be evaluated against successive randomly initialised environments so may perform well one time and badly another.

Selection algorithm

evofuzzy uses a double tournament algorithm when selecting the next generation, to help prevent trees from growing too large (bloat). The selection is done in two steps:

1. A series of fitness tournaments are held where in each round `tournament_size` individuals are selected at random from the population and the fittest is chosen as the winner to go into the next round.
2. a second series of tournaments is held where pairs of candidates from the previous round are selected and the smallest is selected with a probability controlled by the `parsimony_size` hyperparameter. This is a value between 1 and 2, where 1 means no size selection is done and 2 means the smallest candidate is always selected. Values in the range 1.2 to 1.6 were found to work well for their experiments.

Mutating algorithm

Individuals in the population are selected for mutation with a probability given by the `mutation_prob` hyperparameter. An individual that is mutated has a single rule from its set of rules selected and either the entire rule is replaced with a newly generated one, or a sub-tree is selected and replaced with a new sub-tree. The probability of the entire rule being replaced is controlled by the `whole_rule_prob` hyperparameter.

Mating algorithm

Pairs of individuals in the population may also be selected for mating with a probability given by the `crossover_prob` hyperparameter. A random rule is selected from each parent and either the entire rules are swapped over with a probability of `whole_rule_prob` or a sub-tree of each rule is selected and swapped over.

Using evofuzzy

evofuzzy provided two classes - `FuzzyClassifier` for classification and `GymRunner` for reinforcement learning on OpenAI Gym. They are both subclasses of `FuzzyBase` so have the following in common.

Common interface

Hyperparameters

Both classes are instantiated with the hyperparameters to use during training. All the hyperparameters are explained in the previous section, but here is a summary:

min_tree_height int

minimum height of tree at creation

max_tree_height int

maximum height of a tree at creation

min_rules int

minimum number of rules of an individual

max_rules int

maximum number of rules of an individual

population_size int

the size of the population

n_iter int

number of times to iterate over the dataset/environment

mutation_prob float 0.0 - 1.0

probability of an individual being mutated

crossover_prob float 0.0 - 1.0

probability of a pair of individuals being mated

whole_rule_prob float 0.0 - 1.0

probability of entire rules being mutated / mated

elite_size int

number of top performers to preserve across generations

replacements int

number of new individuals to inject each generation

tournament_size int

number of individuals to include in a tournament

parsimony_size float 1.0 - 2.0

selection pressure for small size

batch_size int or None

number of data points to include each generation. FuzzyClassifier only, this is ignored by the GymRunner class.

memory_decay float 0.0 - 1.0

EWMA weighting for new fitness over previous fitness

verbose bool

if True print summary stats while running

Common methods and properties

Both classes have these methods and properties in common:

save(path_to_file)

Save the state of the FuzzyClassifier or GymRunner instance to a file.

load(path_to_file)

Restore the state of a FuzzyClassifier or GymRunner from a file previously created with the `save` method.

best (property)

Get the current best performing individual. This is a list of list of DEAP GP primitives.

best_str (property)

Return the fuzzy rules of the best performing individual as a human readable string.

individual_to_str(individual)

Convert any individual to a human readable string.

best_n(n) Merge the rules of the top `n` individuals into a single rule set. This is an experimental feature to combine the predictive power of several top performers into a single entity.

Other common features

Both classes support writing information while training into a format that can be viewed in TensorBoard, by using the TensorBoardX library (https://tensorboardx.readthedocs.io/en/latest/index.html). If an instance of the `tensorboardX.SummaryWriter` is passed to the training method (`fit` or `train`) then at the end of each iteration statistics about the current best/average fitness and size is saved, plus a histogram of the fitness and size of the entire population. The hyperparameters for the run are also saved as a text object. The user may also use the SummaryWriter to save additional information before or after a run if they wish.

The FuzzyClassifier class for classification

The FuzzyClassifier class tries to follow the scikit-learn API as far as possible. The class has the following methods in addition to those in the previous section:

fit(X, y, classes, antecedent_terms=None, columns=None, tensorboard_writer=None)

Train the classifier on the training data X and y. The parameters are:

- `X` a pandas DataFrame or numpy-like array of features
- `y` the target data for the classifier
- `classes`: a dictionary mapping the names of the target class to their values in `y`. For example, if `y` contains 0, 1, 2 for "setosa, versicolor and virginica respectively then the `classes` parameter should contain `{"setosa": 0, "versicolor": 1, "virginica": 2}` **NOTE:** the class names must be valid python identifiers and not python keywords.
- `antecedent_terms`: an optional dictionary converting feature names to the list of fuzzy terms that will be used for that feature. For example:

  ```
  {
      'sepal_length': ['short', 'medium', 'long'],
      'sepal_width': ['narrow', 'medium', 'wide'],
      'petal_length': ['short', 'medium', 'long'],
      'petal_width': ['narrow', 'medium', 'wide']
  }
  ```

  This can be used to control the number of terms used for each feature. if provided then the keys must match the column names. If not provided then the terms will default to "lower", "low", "average", "high", "higher".
- `columns`: optional feature names to apply to the columns of X. These must match the keys given in the `antecedent_terms` if provided.

if not provided and X is a pandas DataFrame then the pandas column names will be used. If not provided and X is a numpy array or similar structure then the column names will default to "column_0", "column_1" etc.

NOTE: the feature names, whether they come from the pandas dataframe or from this parameter, must be valid python identifiers and not python keywords.

- `tensorboard_writer`: an optional `tensorboardX.SummaryWriter` instance to log information for display in TensorBoard.

predict(X, n=1) Predict the target class for the data in X.

- `X` a DataFrame or numpy array in the same format that the classifier was trained on.
- `n` optional experimental parameter to use the combined top `n` individuals in the population to make the prediction. By default only the best performer is used.

Example FuzzyClassifier code:

```
from datetime import datetime
from pathlib import Path
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

import pandas as pd
from evofuzzy import FuzzyClassifier
import tensorboardX

"""Script for testing the classifier by running it on the iris dataset.
"""

TO_TENSORBOARD = True  # write results and stats to tensorboard?

data = load_iris()
cols = [c.replace(" ", "_").replace("_(cm)", "") for c in data.feature_names]
iris = pd.DataFrame(data.data, columns=cols)
y = pd.Series(data.target)

train_X, test_X, train_y, test_y = train_test_split(iris, y, test_size=50)


classes = {name: val for (name, val) in zip(data.target_names, range(3))}
antecedent_terms = {
```

```
    col: ["very_narrow", "narrow", "medium", "wide", "very_wide"]
    if "width" in col
    else ["very_short", "short", "medium", "long", "very_long"]
    for col in cols
}

if TO_TENSORBOARD:
    logdir = Path(f"tb_logs/iris/{datetime.now().strftime('%Y%m%d-%H%M%S')}")
    logdir.mkdir(parents=True, exist_ok=True)
    tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
else:
    tensorboard_writer = None

classifier = FuzzyClassifier(
    population_size=20,
    elite_size=5,
    n_iter=5,
    mutation_prob=0.5,
    crossover_prob=0.5,
    min_tree_height=1,
    max_tree_height=3,
    min_rules=2,
    max_rules=4,
    whole_rule_prob=0.2,
    batch_size=20,
)
classifier.fit(
    train_X,
    train_y,
    classes,
    antecedent_terms=antecedent_terms,
    tensorboard_writer=tensorboard_writer,
)

print(f"Best Rule:  size = {len(classifier.best)}")
print(classifier.best_str)

predictions = classifier.predict(test_X)
confusion = pd.DataFrame(
    data=confusion_matrix(test_y, predictions),
    columns=data.target_names,
    index=data.target_names,
)
print(confusion)
if tensorboard_writer:
    tensorboard_writer.add_text("confusion", confusion.to_markdown())
```

```
tensorboard_writer.close()
```

The GymRunner class for reinforcement learning

The GymRunner class has two methods in addition to the common ones give above.

train(env, tensorboard_writer=None, antecedents=None, inf_limit=100.0) Train the GymRunner instance to play the OpenAI Gym environment. The parameters are:

- `env`: the Gym environment created with `gym.make(env_name)`
- `tensorboard_writer`: an optional `tensorboardX.SummaryWriter` instance to log progress to TensorBoard.

- `antecedents`: an optional list of scikit-fuzzy `Antecedent`s, one for each input variable. If this is not provided then the antecedents are created automatically from the environment's `observation_space`. This can be used to give finer control over how the inputs are converted to fuzzy variables, and to give the fuzzy variables meaningful names instead of "obs_0", "obs_1" etc that will be created by default. See below for the `make_antecedent` helper function.
- `inf_limit`: Some Gym environments have observation_spaces with lower and upper limits of (-inf, inf) which would cause problems for the fuzzy inference system when the antecedents are created automatically from the observation_space. This parameter replace +/-inf with +/-`inf_limit`. It defaults to 100 but that is a quite arbitrary choice so should be set to something appropriate for the environment. If the `antecedents` parameter is given or the observation space limits are not +/-inf then this parameter has no effect.

play(env, n=1) Show the GymRunner playing the environment.

- `env`: the Gym environment created with `gym.make(env_name)`
- `n`: experimental parameter to combine the top `n` individuals into a single agent. By default only the top scoring individual in the population is used.

This method returns the total reward the agent accrued from playing the environment.

Helper function

make_antecedent(name, min, max, terms=None) This function can be used to create the values for the `antecedents` parameter to the `train` method. The parameters are:

- `name`: str the name to give the antecedent. This must be a usable as a valid python identifier.
- `min`: the minimum value for the antecedent.
- `max`: the maximum value for the antecedent.
- `terms`: an optional list of names for the fuzzy terms. If not provided then they will default to "lower", "low", "average", "high", "higher".

Example GymRunner code:

```
from datetime import datetime
from pathlib import Path
import tensorboardX
import gym
from evofuzzy import GymRunner
from evofuzzy.fuzzygp import make_antecedent

tensorboard_dir = "tb_logs/cartpole-v0"
if tensorboard_dir:
    logdir = Path(f"{tensorboard_dir}/{datetime.now().strftime('%Y%m%d-%H%M%S')}")
    logdir.mkdir(parents=True, exist_ok=True)
    tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
else:
    tensorboard_writer = None

env = gym.make("CartPole-v1")
runner = GymRunner(
    population_size=50,
    elite_size=1,
    n_iter=10,
    mutation_prob=0.9,
    crossover_prob=0.2,
    min_tree_height=1,
    max_tree_height=3,
    max_rules=4,
    whole_rule_prob=0.2,
)

antecedents = [
    make_antecedent("position", -2.4, 2.4),
    make_antecedent("velocity", -1, 1),
    make_antecedent("angle", -0.25, 0.25),
    make_antecedent("angular_velocity", -2, 2),
]

runner.train(env, tensorboard_writer, antecedents)
```

```
print(runner.best_str)
reward = runner.play(env)
print("Reward:", reward)
```

Appendix II: Source Code

requirements.txt

```
deap>=1.3
scikit-fuzzy>=0.4.2
scikit-learn>=0.24
pandas>=1.3
numpy>=1.21
gym==0.18.3
```

requirements-dev.txt

```
pytest>=6.2
tensorboardx>=2.4
```

evofuzzy/fuzzygp.py

```
import random
from itertools import repeat, count
from typing import Any, List, NamedTuple, Dict, Iterable, Tuple, Optional
import operator
from multiprocessing import Pool
from functools import partial

import deap.tools
import numpy as np
import pandas as pd
import skfuzzy as fuzz
import tensorboardX
from deap import creator, base, algorithms, gp, tools
from skfuzzy import control as ctrl
from skfuzzy.control import Rule, Antecedent, Consequent
from skfuzzy.control.term import Term

BIG_INT = 1 << 64

consequents_counter = count()


class RuleSet(list):
    """Subclass of list that contains gp.PrimitiveTree instances, used to hold a set of fuz
    len(ruleset) will return the total length of all the contained primitive trees.
    The ruleset.length property will return the length of the top level list.
    """

    def __len__(self):
        return sum(len(item) for item in self)

    @property
    def length(self):
        return super().__len__()


creator.create("RuleSetFitness", base.Fitness, weights=(1.0,))
creator.create("Individual", RuleSet, fitness=creator.RuleSetFitness)


def identity(x: Any) -> Any:
    """Identity function - returns the parameter unchanged.
    Used to enable terminal/ephemeral values to be created below a tree's minimum depth.
```

```
    """
    return x


class MakeConsequents:
    """Ephemeral constant that randomly generates consequents for the rules.
    This needs to be a class so that the repr can be defined to return something
    that can be used by toolbox.compile.
    :param cons_terms: dict mapping a skfuzzy Consequent name to a list of "name['term']"
                       strings.
    """

    def __init__(self, cons_terms: Dict[str, str]):
        max_consequents = len(cons_terms) // 2 + 1
        sample_size = random.randint(1, max_consequents)
        candidates = random.sample(list(cons_terms.values()), sample_size)
        self.values = [random.choice(term) for term in candidates]

    def __repr__(self):
        return f"[{', '.join(value for value in self.values)}]"


def _make_primitive_set(
    antecedents: List[Antecedent], consequents: List[Consequent]
) -> gp.PrimitiveSetTyped:
    """Create a typed primitive set that can be used to generate fuzzy rules.

    :param antecedents:  The Antecedent instances for the rules
    :param consequents:  The Consequent instances for the rules.

    :return:  The PrimitiveSetTyped with all the rule components registered
    """
    cons_terms = {
        cons.label: [f"{cons.label}['{name}']" for name in cons.terms.keys()]
        for cons in consequents
    }
    make_consequents = partial(MakeConsequents, cons_terms)

    pset = gp.PrimitiveSetTyped("Rule", [], Rule)

    for ant in antecedents:
        pset.context[ant.label] = ant
        for name, term in ant.terms.items():
            pset.addTerminal(term, Term, f"{ant.label}['{name}']")

    for cons in consequents:
```

```
            pset.context[cons.label] = cons

    # The ephemeral constants must have a globally unique name
    consequents_name = f"consequents_{next(consequents_counter)}"
    pset.addEphemeralConstant(consequents_name, make_consequents, list)
    pset.addPrimitive(Rule, [Term, list], Rule)
    pset.addPrimitive(operator.and_, [Term, Term], Term)
    pset.addPrimitive(operator.or_, [Term, Term], Term)
    pset.addPrimitive(operator.invert, [Term], Term)
    pset.addPrimitive(identity, [list], list)

    return pset


class CreatorConfig(NamedTuple):
    """Hyperparameter configuration for creating instances"""

    min_tree_height: int = 2
    max_tree_height: int = 4
    min_rules: int = 2  # minimum number of rules to have in a chromosome
    max_rules: int = 5  # maximum number of rules to have in a chromosome


def _generate_rule(pset: gp.PrimitiveSetTyped, min_: int, max_: int, type_=None):
    """
    Return a randomly generated PrimitiveTree encoding a fuzzy rule.
    :param pset: The PrimitiveSetTyped to draw the node types from
    :param min_: minimum tree height
    :param max_: maximum tree height
    :param type_:
    :return: the generated primitiveTree
    """
    return gp.PrimitiveTree(gp.genGrow(pset, min_, max_, type_))


def _generate_rule_set(
    pset: gp.PrimitiveSetTyped, type_=None, config: CreatorConfig = None
) -> RuleSet:
    """Generate a RuleSet - a list of rules created by the _generate_rule function

    :param pset: PrimitiveSetTyped registered with the primitives needed
    :param type_:
    :param config: configuration for the _generate_rule function
    :return: RuleSet with the generated rules
    """
    rules_len = random.randint(config.min_rules, config.max_rules)
```

```
    return RuleSet(
        _generate_rule(pset, config.min_tree_height, config.max_tree_height, type_)
        for _ in range(rules_len)
    )


def register_primitiveset_and_creators(
    toolbox: base.Toolbox,
    config: CreatorConfig,
    antecedents: List[Antecedent],
    consequents: List[Consequent],
):
    """Create a primitive set for fuzzy rules and register functions for creating
    individuals and populations with the toolbox.
    Prerequisites:
    - RuleSetFitness class registered in creator module
    - antecedents and consequents have had their terms defined

    :param toolbox: deap.base.Toolbox instance
    :param config:  Config instance holding hyperparameters
    :param antecedents: list of fuzzy antecedents used by the rules
    :param consequents: list of fuzzy consequents used by the rules
    :return: The PrimitiveSet that has been created
    """

    pset = _make_primitive_set(antecedents, consequents)

    toolbox.register("compile", gp.compile, pset=pset)
    toolbox.register(
        "expr",
        _generate_rule,
        pset=pset,
        min_=config.min_tree_height,
        max_=config.max_tree_height,
    )
    toolbox.register(
        "rules_expr",
        _generate_rule_set,
        pset=pset,
        config=config,
    )
    toolbox.register(
        "individualCreator", tools.initIterate, creator.Individual, toolbox.rules_expr
    )
    toolbox.register(
        "populationCreator", tools.initRepeat, list, toolbox.individualCreator
```

```
    )
    toolbox.register("get_pset", identity, pset)
    return pset


def ea_with_elitism_and_replacement(
    population: List[RuleSet],
    toolbox: base.Toolbox,
    cxpb: float,
    mutpb: float,
    ngen: int,
    replacement_size: int = 0,
    stats: tools.Statistics = None,
    tensorboard_writer: "tensorboardX.SummaryWriter" = None,
    elite_size: int = 1,
    verbose: bool = True,
    slices: Optional[Iterable[slice]] = None,
    always_evaluate: bool = False,
    memory_decay: float = 1,
) -> Tuple[List[RuleSet], tools.Logbook]:
    """Modified version of the DEAP eaSimple function to run the evolution process
    while keeping the top performing members from one generation to the next and replacing
    poor performers with new individuals.

    :param population: The initial population
    :param toolbox: the deap toolbox with functions registered on it
    :param cxpb: crossover probability 0 <= cxpb <= 1
    :param mutpb: mutation probability 0 <= mupb <= 1
    :param ngen: number of generations to run the evolution for
    :param replacement_size: number of poor performers to replace with new individuals
    :param stats: DEAP Stats instance for recording statistics
    :param tensorboard_writer: Optional tensorboardX SummaryWriter instance to log
           results to tensorboard.
    :param elite_size: the number of top performers to carry over to the next generation
    :param verbose: boolean flag - if True then print stats while running
    :param slices: optional list of slice objects to run EA on small batches
    :param always_evaluate: flag to force evaluation of fitness
    :param memory_decay: value between 0 and 1 to control how much weight to put on previous
           values
    :return: final population and logbook

    """

    if slices is None:
        batched = False
        slices = [slice(0, BIG_INT)]
```

```
    else:
        batched = True

    def evaluate_population(
        population: List[RuleSet], batch_slice: slice
    ) -> Tuple[List[RuleSet], deap.tools.Logbook]:
        if not always_evaluate and not batched:
            # Only evaluate the individuals in the population that have not been evaluated
            population = [ind for ind in population if not ind.fitness.valid]
        # prune the rules that are going to be evaluated
        _prune_population(population)
        with Pool() as pool:
            fitnesses = pool.starmap(
                toolbox.evaluate, zip(population, repeat(batch_slice))
            )

        for ind, fit in zip(population, fitnesses):
            if ind.fitness.valid:
                old_fitness = ind.fitness.values
                new_fit = fit[0] * memory_decay + (old_fitness[0] * (1 - memory_decay))
                ind.fitness.values = (new_fit,)
            else:
                ind.fitness.values = fit

    logbook = tools.Logbook()
    logbook.header = "gen", "fitness", "size"
    logbook.chapters["fitness"].header = "max", "avg"
    logbook.chapters["size"].header = "min", "avg", "best"

    evaluate_population(population, slices[0])
    population.sort(key=lambda ind: ind.fitness.values)

    write_stats(population, 0, verbose, logbook, stats, tensorboard_writer)

    for gen in range(1, ngen + 1):
        if batched and verbose:
            print("Batch: ", end="")
        for idx, slice_ in enumerate(slices):
            if batched and verbose:
                print(idx, end=", ")

            replacements = toolbox.populationCreator(replacement_size)

            offspring = toolbox.select(
                population[replacement_size:],
                len(population) - (elite_size + replacement_size),
```

```
            )
            offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

            if elite_size:
                offspring.extend(population[-elite_size:])
            if replacement_size:
                offspring.extend(replacements)

            population[:] = offspring
            evaluate_population(population, slice_)
            population.sort(key=lambda ind: ind.fitness.values)

        write_stats(population, gen, verbose, logbook, stats, tensorboard_writer)

    return population, logbook


def write_stats(
    population: List[RuleSet],
    generation: int,
    verbose: bool,
    logbook: deap.tools.Logbook,
    stats: deap.tools.Statistics,
    tensorboard_writer: "tensorboardX.SummaryWriter",
):
    """Gather statistics on the population and
    1. print them to the screen if the verbose flag is set
    2. write them to TensorBoard directory if tensorboard_writer is given

    :param population:
    :param generation:
    :param verbose:
    :param logbook:
    :param stats:
    :param tensorboard_writer:
    :return:
    """
    record = stats.compile(population) if stats else {}
    logbook.record(gen=generation, **record)
    if verbose:
        print()
        print(logbook.stream)
    if tensorboard_writer:
        for (name, val) in record.items():
            if isinstance(val, dict):
                for (subname, subval) in val.items():
```

```
                    tensorboard_writer.add_scalar(
                        f"{name}/{subname}", subval, generation
                    )
            else:
                tensorboard_writer.add_scalar(name, val, generation)
        tensorboard_writer.add_histogram(
            "fitnesses", np.array([i.fitness.values[0] for i in population]), generation
        )
        tensorboard_writer.add_histogram(
            "sizes", np.array([len(i) for i in population]), generation
        )
        tensorboard_writer.add_histogram(
            "rule_count", np.array([i.length for i in population]), generation
        )


def _prune_rule(rule: gp.PrimitiveTree):
    """
    Remove redundancy in a fuzzy rule.  ie:
    - `NOT NOT X` is converted to X
    - `X AND X` is converted to X
    - `X OR X` is converted to X
    rules are modified in-place
    :param rule: the rule to prune
    """
    pos = 0
    while pos < len(rule):
        name = rule[pos].name
        # if there are two consecutive inverts then delete them both
        if name == "invert" and rule[pos + 1].name == "invert":
            del rule[pos : pos + 2]
            continue
        if name in ("and_", "or_"):
            # merge duplicate branches
            lhs = rule.searchSubtree(pos + 1)
            rhs = rule.searchSubtree(lhs.stop)
            if rule[lhs] == rule[rhs]:
                rule[pos : rhs.stop] = rule[lhs]
                continue
        pos += 1


def _prune_population(population: List[RuleSet]):
    """
    Prune all the rules in a population
    :param population: list of RuleSets to prune
```

```
    """
    for ind in population:
        for rule in ind:
            _prune_rule(rule)


def mate_rulesets(
    individual_1: RuleSet, individual_2: RuleSet, whole_rule_prob: float
) -> Tuple[RuleSet, RuleSet]:
    """Mate two individuals by randomly selecting two rules, one from each individual then c
      - swapping the rules over completely
      or
      - swapping randomly selected subtrees of the two rules
    N.B. The individuals are modified in-place as well as returned from the function.

    :param individual_1:  First individual
    :param individual_2:  Second individual
    :param whole_rule_prob: probability of swapping entire rules instead of sub-trees
    :return: the modified individuals
    """
    rule1_idx = random.randint(0, individual_1.length - 1)
    rule2_idx = random.randint(0, individual_2.length - 1)
    if random.random() < whole_rule_prob:
        # swap entire rules over
        rule2 = individual_1[rule1_idx]
        rule1 = individual_2[rule2_idx]
    else:
        rule1, rule2 = gp.cxOnePoint(individual_1[rule1_idx], individual_2[rule2_idx])
    individual_1[rule1_idx] = rule1
    individual_2[rule2_idx] = rule2
    return individual_1, individual_2


def mutate_ruleset(
    toolbox: base.Toolbox, individual: RuleSet, whole_rule_prob: float
) -> Tuple[RuleSet]:
    """Mutate an individual by selecting a rule then either:
    - replacing the entire rule with a newly generated one
    or
    - selecting a subtree of the rule and replacing it with a newly generated subtree
    N.B. The individual is modified in-place as well as returned from the function

    :param toolbox: deap.base.Toolbox instance that has been initialised with the
    register_primitiveset_and_creators function.
    :param individual:  the individual to mutate
    :param whole_rule_prob: probability of replacing an entire rule instead of a sub-tree
```

```
    :return: the modified individual in a 1-tuple
    """
    rule_idx = random.randint(0, individual.length - 1)
    if random.random() < whole_rule_prob:
        rule = toolbox.expr()
    else:
        (rule,) = gp.mutUniform(
            individual[rule_idx], expr=toolbox.expr, pset=toolbox.get_pset()
        )
    individual[rule_idx] = rule
    return (individual,)


def get_fitness_values(individual: RuleSet) -> Tuple[float]:
    """Return the fitness of an individual.  N.B. in DEAP the fitness is always a tuple

    :param individual:
    :return: Tuple with the fitness value(s) as floats
    """
    return individual.fitness.values


def make_antecedent(
    name: str,
    min: float,
    max: float,
    terms: Optional[List[str]] = None,
    inf_limit: Optional[float] = None,
) -> ctrl.Antecedent:
    """Create a skfuzzy Antecedent object and initialise the terms on it.

    :param name: The name of the antecedent
    :param min: minimum value that the antecedent can take
    :param max:  minimum value that the antecedent can take
    :param terms: optional names of the terms that are defined on the antecedent.
                  If omitted then the default names are used:
                    "
    :param inf_limit: if the min and/or max are +/-inf then replace them with +/- this valu
                      This can happen when taking the min and max from a Gym environment ob

    :return: the created Antecedent instance
    """
    if inf_limit is not None and min == -np.inf:
        min = -inf_limit
    if inf_limit is not None and max == np.inf:
        max = inf_limit
```

```
    antecedent = ctrl.Antecedent(np.linspace(min, max, 11), name)
    if terms:
        antecedent.automf(names=terms)
    else:
        antecedent.automf(variable_type="quant")
    return antecedent


def make_antecedents(
    X: pd.DataFrame, antecedent_terms: Dict[str, List[str]]
) -> List[ctrl.Antecedent]:
    """Create a list of Antecedent terms from a dict that maps names of antecendents to
    a list of terms for them.

    :param X: pandas dataframe with the training data
    :param antecedent_terms: mapping from antecedent names to list of terms
    :return: list of Antecendent instances
    """
    if antecedent_terms is None:
        antecedent_terms = {}
    mins = X.min()
    maxes = X.max()
    antecedents = []
    for column in X.columns:
        terms = antecedent_terms.get(column, None)
        antecedent = make_antecedent(column, mins[column], maxes[column], terms)
        antecedents.append(antecedent)
    return antecedents


def make_binary_consequents(classes: Iterable[str]) -> List[ctrl.Consequent]:
    """Create a list of Consequent instances with "likely" and "unlikely" terms.

    :param classes: List of target class names
    :return: list of Consequent instances
    """
    consequents = []
    for cls in classes:
        cons = ctrl.Consequent(np.linspace(0, 1, 10), cls, "som")
        cons["likely"] = fuzz.trimf(cons.universe, (0.0, 1.0, 1.0))
        cons["unlikely"] = fuzz.trimf(cons.universe, (0.0, 0.0, 1.0))
        consequents.append(cons)
    return consequents
```

evofuzzy/fuzzybase.py

```
import pickle
from typing import Optional, Iterable

import numpy as np
import tensorboardX
from deap import base, tools

from evofuzzy.fuzzygp import (
    ea_with_elitism_and_replacement,
    CreatorConfig,
    register_primitiveset_and_creators,
    RuleSet,
    mate_rulesets,
    mutate_ruleset,
    get_fitness_values,
)


class FuzzyBase:
    """Common base class for FuzzyClassifier and GymRunner"""

    always_evaluate_ = False
    population_ = None

    def __init__(
        self,
        min_tree_height: int = 2,
        max_tree_height: int = 4,
        min_rules: int = 2,
        max_rules: int = 5,
        population_size: int = 100,
        n_iter: int = 20,
        mutation_prob: float = 0.1,
        crossover_prob: float = 0.9,
        whole_rule_prob: float = 0.1,
        elite_size: int = 5,
        replacements: int = 5,
        tournament_size: int = 5,
        parsimony_size: float = 1.7,
        batch_size: Optional[int] = None,
        memory_decay: float = 1,
        verbose: bool = True,
    ):
```

```
        """Initialise hyperparameters"""

        self.min_tree_height = min_tree_height
        self.max_tree_height = max_tree_height
        self.min_rules = min_rules
        self.max_rules = max_rules
        self.population_size = population_size
        self.n_iter = n_iter
        self.mutation_prob = mutation_prob
        self.crossover_prob = crossover_prob
        self.whole_rule_prob = whole_rule_prob
        self.elite_size = elite_size
        self.replacements = replacements
        self.tournament_size = tournament_size
        self.parsimony_size = parsimony_size
        self.batch_size = batch_size
        self.memory_decay = memory_decay
        self.verbose = verbose

    def _initialise(self, tensorboard_writer: Optional["tensorboardX.SummaryWriter"]):
        """Initialise the toolbox and register functions needed by DEAP on it.
        Also set up the DEAP statistics object and optionally write hyperparameters to Tens
        :param tensorboard_writer: Object used to write to TensorBoard.
        :return:
        """

        if tensorboard_writer:
            hparams = "\n\n".join(
                f"* {k}: {v}" for (k, v) in self.__dict__.items() if not k.endswith("_")
            )
            tensorboard_writer.add_text("hparams", hparams)

        if hasattr(self, "toolbox_"):
            # already initialised
            return

        self.toolbox_ = base.Toolbox()
        self.config_ = CreatorConfig(
            self.min_tree_height, self.max_tree_height, self.min_rules, self.max_rules
        )
        self.pset_ = register_primitiveset_and_creators(
            self.toolbox_, self.config_, self.antecedents_, self.consequents_
        )
        self.toolbox_.register(
            "select",
            tools.selDoubleTournament,
```

```
            fitness_size=self.tournament_size,
            parsimony_size=self.parsimony_size,
            fitness_first=True,
        )
        self.toolbox_.register("mate", self._mate)
        self.toolbox_.register("mutate", self._mutate)

        self.fitness_stats_ = tools.Statistics(get_fitness_values)
        self.fitness_stats_.register("max", np.max)
        self.fitness_stats_.register("avg", np.mean)
        self.size_stats_ = tools.Statistics(len)
        self.size_stats_.register("min", np.min)
        self.size_stats_.register("avg", np.mean)
        self.size_stats_.register("best", self.best_size)
        self.stats_ = tools.MultiStatistics(
            fitness=self.fitness_stats_, size=self.size_stats_
        )

    def execute(
        self,
        slices: Optional[Iterable[slice]],
        tensorboard_writer: Optional["tensorboardX.SummaryWriter"],
    ):
        """Run the fuzzygp main loop.  Optionally write the best rule to TensorBoard.

        :param slices: sequence of slices to use for batching classifier training data.
        :param tensorboard_writer: Object used to write to TensorBoard.
        :return: None
        """
        if self.population_ is None:
            self.population_ = self.toolbox_.populationCreator(n=self.population_size)

        self.population_, self.logbook_ = ea_with_elitism_and_replacement(
            self.population_,
            self.toolbox_,
            cxpb=self.crossover_prob,
            mutpb=self.mutation_prob,
            ngen=self.n_iter,
            replacement_size=self.replacements,
            stats=self.stats_,
            tensorboard_writer=tensorboard_writer,
            elite_size=self.elite_size,
            verbose=self.verbose,
            slices=slices,
            always_evaluate=self.always_evaluate_,
            memory_decay=self.memory_decay,
```

```
        )
        if tensorboard_writer:
            tensorboard_writer.add_text("best_ruleset", self.best_str)
            tensorboard_writer.add_text("size_of_best_ruleset", str(self.best_size()))

    def _mate(self, ind1: RuleSet, ind2: RuleSet):
        """Do crossover on two individuals"""
        return mate_rulesets(ind1, ind2, self.whole_rule_prob)

    def _mutate(self, individual: RuleSet):
        """Mutate an individual"""
        return mutate_ruleset(self.toolbox_, individual, self.whole_rule_prob)

    @property
    def best(self):
        """Return the best individual"""
        return self.population_[-1] if self.population_ else None

    def best_size(self, *args):
        """Return the size of the best individual.
        N.B. takes additional *args because it is called from the DEAP stats code which
        passes additional arguments.
        """
        return len(self.best) if self.best else 0

    @property
    def best_str(self):
        """Return the string representation of the best individual"""
        return self.individual_to_str(self.best) if self.best else "Unevaluated"

    def individual_to_str(self, individual: RuleSet):
        """Convert any individual to its string representation"""
        return "\n".join(
            str(self.toolbox_.compile(r)).splitlines()[0] for r in individual
        )

    def save(self, filename: str):
        """Save the state of the object as a pickle"""
        with open(filename, "wb") as f:
            pickle.dump(self.__dict__, f, -1)
        return self

    def load(self, filename: str):
        """Load the state of the object from a pickle created by the save method"""
        with open(filename, "rb") as f:
            self.__dict__ = pickle.load(f)
```

```
        return self

    def best_n(self, n: int = 1):
        """Create a new rule set that combines the top n individuals"""
        if n == 1:
            return self.best
        rules = RuleSet()
        for individual in self.population_[-n:]:
            rules.extend(individual)
        return rules
```

evofuzzy/fuzzyclassifier.py

```
from typing import Dict, List, Any, Optional, Tuple
import numpy as np
import pandas as pd
import tensorboardX
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle
from skfuzzy import control as ctrl

from .fuzzybase import FuzzyBase
from .fuzzygp import make_antecedents, make_binary_consequents


class FuzzyClassifier(FuzzyBase, BaseEstimator, ClassifierMixin):
    """Class to create a fuzzy rule classifier"""

    def fit(
        self,
        X,
        y,
        classes: Dict[str, Any],
        antecedent_terms: Optional[Dict[str, List[str]]] = None,
        columns: Optional[List[str]] = None,
        tensorboard_writer: "tensorboardX.SummaryWriter" = None,
    ) -> "FuzzyClassifier":
        """

        :param X: Training data.  May be a numpy array or pandas dataframe (or something co
        :param y: sequence of target classes
        :param classes: Dict mapping the name of a target class to the value used in y to
        represent it.
        :param antecedent_terms: mapping from antecedent names to list of terms
        :param columns: List of column names if X is not a pandas DataFrame or different na
        are needed.
        :param tensorboard_writer: Optional object to write information to TensorBoard
        :return: self
        """

        X, y = shuffle(X, y)
        self.classes_ = classes

        if not columns and not hasattr(X, "columns"):
            columns = [f"column_{i}" for i in range(X.shape[1])]
```

```
        self.columns_ = columns

        X = self._convert_dataframe(X)

        self.antecedents_ = make_antecedents(X, antecedent_terms)
        self.consequents_ = make_binary_consequents(classes.keys())

        self._initialise(tensorboard_writer)

        if hasattr(self.toolbox_, "evaluate"):
            del self.toolbox_.evaluate
        self.toolbox_.register("evaluate", self._evaluate, X=X, y=y)

        slices = list(_batch_slices(len(X), self.batch_size))

        self.execute(slices, tensorboard_writer)
        return self

    def _convert_dataframe(self, X):
        """Convert the training data to a Pandas DataFrame if it is not one already.
        If it is and column names are provided, then rename the columns.

        :returns: The new or modified dataframe
        """
        if not self.columns_:
            return X

        # if columns is provided then assume either X is a numpy array or the user
        # want to rename the dataframe columns
        if isinstance(X, pd.DataFrame):
            X.columns = self.columns_
        else:
            X = pd.DataFrame(data=X, columns=self.columns_)
        return X

    def predict(self, X: pd.DataFrame, n: int = 1) -> List[Any]:
        """Make predictions on the test data.

        :param X: The data to make predictions from
        :param n: The number of best individuals to use
        :return: list of predicted classes
        """
        individual = self.best_n(n)
        X = self._convert_dataframe(X)
```

```
        rules = [self.toolbox_.compile(rule) for rule in individual]
        return _make_predictions(X, rules, self.classes_)

    def _evaluate(
        self,
        individual: "RuleSet",
        batch_slice: Optional[slice],
        X: pd.DataFrame,
        y: pd.Series,
    ) -> Tuple[float]:
        """Calculate the fitness score of an individual by making a prediction on the X and
        data and returning the accuracy in a tuple (as required by DEAP).

        :param individual:  RuleSet to evaluate
        :param batch_slice: the slice of the X and y data to use
        :param X: test data
        :param y: ground truth classes
        :return: tuple containing the accuracy score
        """

        X = X.iloc[batch_slice]
        y = y.iloc[batch_slice]
        rules = [self.toolbox_.compile(rule) for rule in individual]
        predictions = _make_predictions(X, rules, self.classes_)
        return (accuracy_score(y, predictions),)


def _make_predictions(
    X: pd.DataFrame, rules: List[ctrl.Rule], classes: Dict[str, Any]
) -> List[Any]:
    """Apply fuzzy rules to data in a pandas dataframe and
    predict the target class.
    :param X: Pandas dataframe with the data.  Column names must match the antecedent
              names in the rules.
    :param rules:  list of fuzzy rules
    :param classes: dict mapping rule consequent names to target class values

    :returns: list of class predictions
    """
    antecedents = {
        term.parent.label for rule in rules for term in rule.antecedent_terms
    }
    columns = [col for col in X.columns if col in antecedents]
    X = X[columns]
    controller = ctrl.ControlSystem(rules)
    classifier = ctrl.ControlSystemSimulation(controller)
```

```
    prediction = []
    class_names = list(classes.keys())
    class_vals = list(classes.values())
    for row in X.itertuples(index=False):
        classifier.inputs(row._asdict())
        classifier.compute()
        class_idx = np.argmax([classifier.output.get(name, 0) for name in class_names])
        class_val = class_vals[class_idx]
        prediction.append(class_val)
    return prediction


def _batch_slices(max_size: int, batch_size=None):
    """generate slices to split an array-like object into smaller batches.
    If batch size is not given then yield a slice that covers the whole thing.
    """
    if batch_size is None:
        yield slice(0, max_size)
    else:
        range_iter = iter(range(0, max_size, batch_size))
        start = next(range_iter)
        for end in range_iter:
            yield slice(start, end)
            start = end
        if start != max_size:
            yield slice(start, max_size)
```

tests/test_classifier.py

iris_test.py

```
from datetime import datetime
from pathlib import Path
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

import pandas as pd
from evofuzzy import FuzzyClassifier
import tensorboardX

"""Script for testing the classifier by running it on the iris dataset.
"""

TO_TENSORBOARD = True  # write results and stats to tensorboard?

data = load_iris()
cols = [c.replace(" ", "_").replace("_(cm)", "") for c in data.feature_names]
iris = pd.DataFrame(data.data, columns=cols)
y = pd.Series(data.target)

train_X, test_X, train_y, test_y = train_test_split(iris, y, test_size=50)


classes = {name: val for (name, val) in zip(data.target_names, range(3))}
antecendent_terms = {
    col: ["v.narrow", "narrow", "medium", "wide", "v.wide"]
    if "width" in col
    else ["v.short", "short", "medium", "long", "v.long"]
    for col in cols
}

if TO_TENSORBOARD:
    logdir = Path(f"tb_logs/iris/{datetime.now().strftime('%Y%m%d-%H%M%S')}")
    logdir.mkdir(parents=True, exist_ok=True)
    tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
else:
    tensorboard_writer = None

classifier = FuzzyClassifier(
    population_size=20,
    elite_size=3,
    n_iter=5,
    mutation_prob=0.5,
```

```
    crossover_prob=0.5,
    min_tree_height=1,
    max_tree_height=3,
    min_rules=4,
    max_rules=6,
    whole_rule_prob=0.1,
    batch_size=20,
)
classifier.fit(
    train_X,
    train_y,
    classes,
    antecedent_terms=antecendent_terms,
    tensorboard_writer=tensorboard_writer,
)

print(f"Best Rule:  size = {len(classifier.best)}")
print(classifier.best_str)

predictions = classifier.predict(test_X)
confusion = pd.DataFrame(
    data=confusion_matrix(test_y, predictions),
    columns=data.target_names,
    index=data.target_names,
)
print(confusion)
if tensorboard_writer:
    tensorboard_writer.add_text("confusion", confusion.to_markdown())
    tensorboard_writer.close()
```

classifier_cv.py

```
"""
Cross-validation function for the FuzzyClassifier class.
"""

from collections import Counter
from datetime import datetime
from pathlib import Path
from statistics import fmean, stdev
from typing import NamedTuple, Optional, Iterable, List, Dict

import pandas as pd
import tensorboardX
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import StratifiedKFold

from evofuzzy import FuzzyClassifier


class HyperParams(NamedTuple):
    """NamedTuple holding the hyperparameters for the classifier."""

    min_tree_height: int = 2
    max_tree_height: int = 4
    min_rules: int = 2
    max_rules: int = 5
    population_size: int = 100
    n_iter: int = 50
    mutation_prob: float = 0.1
    crossover_prob: float = 0.9
    whole_rule_prob: float = 0.1
    elite_size: int = 5
    replacements: int = 5
    tournament_size: int = 5
    parsimony_size: float = 1.9
    batch_size: Optional[int] = None
    memory_decay: float = 1


def cross_validate(
    train_x: pd.DataFrame,
    train_y: Iterable,
    hyperparams: HyperParams,
    antecendent_terms: Dict[str : List[str]],
```

```
    classes: List[str],
    tensorboard_dir: Optional[str],
    train_test_swap: bool = False,
    number_of_predictors: int = 1,
):
    """Do 5-fold cross validation on training data and training target classes.  Print
    out the results and optionally write them to tensorboard directory.

    :param train_x: DataFrame of training data
    :param train_y: target classes
    :param hyperparams: HyperParams object
    :param antecendent_terms: mapping from antecendent names to list of terms to use
    :param classes: list of target class names
    :param tensorboard_dir: directory to write tensorboard info to (may be None)
    :param train_test_swap: If True then for each fold train on 1/5 the data and test on 4/5
    :param number_of_predictors: number of predictors to use for testing
    :return: None
    """
    kfold = StratifiedKFold(n_splits=5, shuffle=True)
    if tensorboard_dir and tensorboard_dir[-1] == "/":
        tensorboard_dir = tensorboard_dir[:-1]

    results = []
    for (i, (train_idx, test_idx)) in enumerate(kfold.split(train_x, train_y)):
        if train_test_swap:
            train_idx, test_idx = test_idx, train_idx
        if tensorboard_dir:
            logdir = Path(
                f"{tensorboard_dir}/{i}-{datetime.now().strftime('%Y%m%d-%H%M%S')}"
            )
            logdir.mkdir(parents=True, exist_ok=True)
            tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
        else:
            tensorboard_writer = None

        classifier = FuzzyClassifier(
            min_tree_height=hyperparams.min_tree_height,
            max_tree_height=hyperparams.max_tree_height,
            min_rules=hyperparams.min_rules,
            max_rules=hyperparams.max_rules,
            population_size=hyperparams.population_size,
            n_iter=hyperparams.n_iter,
            mutation_prob=hyperparams.mutation_prob,
            crossover_prob=hyperparams.crossover_prob,
            whole_rule_prob=hyperparams.whole_rule_prob,
            elite_size=hyperparams.elite_size,
```

```
            replacements=hyperparams.replacements,
            tournament_size=hyperparams.tournament_size,
            parsimony_size=hyperparams.parsimony_size,
            batch_size=hyperparams.batch_size,
            memory_decay=hyperparams.memory_decay,
        )
        classifier.fit(
            train_x.iloc[train_idx],
            train_y.iloc[train_idx],
            classes,
            antecedent_terms=antecendent_terms,
            tensorboard_writer=tensorboard_writer,
        )

        print(f"Best Rule:  size = {len(classifier.best)}")
        print(classifier.best_str)
        print(
            "Final length of rules sets",
            dict(Counter(x.length for x in classifier.population_)),
        )
        predictions = classifier.predict(train_x.iloc[test_idx], n=number_of_predictors)
        actual = train_y.iloc[test_idx]
        accuracy = sum(actual == predictions) / len(actual)
        results.append(accuracy)
        target_names = classes.keys()
        confusion = pd.DataFrame(
            data=confusion_matrix(actual, predictions),
            columns=target_names,
            index=target_names,
        )
        print("Test accuracy:", accuracy)
        print(confusion)
        if tensorboard_writer:
            tensorboard_writer.add_text("cv_accuracy", str(accuracy))
            tensorboard_writer.add_text("confusion", confusion.to_markdown())
            tensorboard_writer.close()

    print("Accuracy: ", results)
    print(f" -  average {fmean(results)}, std {stdev(results)}")
```

classify_iris.py

```
from sklearn.datasets import load_iris
import pandas as pd

from classifier_cv import cross_validate, HyperParams

"""Script for doing 5-fold cross-validation on the iris dataset.
"""

tensorboard_dir = "tb_logs/iris_cv/"
# tensorboard_dir = None

hyperparams = HyperParams(
    population_size=20,
    elite_size=3,
    n_iter=10,
    mutation_prob=0.5,
    crossover_prob=0.5,
    min_tree_height=1,
    max_tree_height=3,
    min_rules=3,
    max_rules=5,
    whole_rule_prob=0.2,
    batch_size=10,
)

data = load_iris()
cols = [c.replace(" ", "_").replace("_(cm)", "") for c in data.feature_names]
iris = pd.DataFrame(data.data, columns=cols)
y = pd.Series(data.target)

classes = {name: val for (name, val) in zip(data.target_names, range(3))}
antecendent_terms = {
    col: ["v.narrow", "narrow", "medium", "wide", "v.wide"]
    if "width" in col
    else ["v.short", "short", "medium", "long", "v.long"]
    for col in cols
}

cross_validate(
    iris,
    y,
    hyperparams,
    antecendent_terms,
```

```
    classes,
    tensorboard_dir,
    number_of_predictors=1,
)
```

classify_segmentation.py

```
import pandas as pd
from sklearn.datasets import fetch_openml

from classifier_cv import cross_validate, HyperParams

"""Script for doing 5-fold cross-validation on the image segmentation dataset.
"""

tensorboard_dir = "tb_logs/segment_cv/"
# tensorboard_dir = None

hyperparams = HyperParams(
    population_size=50,
    elite_size=3,
    n_iter=5,
    mutation_prob=0.5,
    crossover_prob=0.5,
    min_tree_height=1,
    max_tree_height=2,
    min_rules=20,
    max_rules=25,
    whole_rule_prob=0.1,
    batch_size=20,
    memory_decay=0.7,
)

data, y = fetch_openml(data_id=40984, as_frame=True, return_X_y=True)

# this column should not be used
del data["region.centroid.row"]

data.columns = [c.replace(".", "_") for c in data.columns]

classes = {name: name for name in y.dtype.categories}

antecendent_terms = {
    col: ["very_low", "low", "medium", "high", "very_high"] for col in data.columns
}

cross_validate(
    data,
    y,
    hyperparams,
```

```
    antecendent_terms,
    classes,
    tensorboard_dir,
    train_test_swap=False,
)
```

classify_cancer.py

```
import pandas as pd
from sklearn.datasets import fetch_openml

from classifier_cv import cross_validate, HyperParams

"""Script for doing 5-fold cross-validation on the Wisconsin Cancer dataset.
"""

tensorboard_dir = "tb_logs/cancer_cv/"
# tensorboard_dir = None

hyperparams = HyperParams(
    population_size=50,
    elite_size=3,
    n_iter=5,
    mutation_prob=0.5,
    crossover_prob=0.5,
    min_tree_height=1,
    max_tree_height=3,
    min_rules=4,
    max_rules=7,
    whole_rule_prob=0.1,
    batch_size=50,
)

data, y = fetch_openml(data_id=15, as_frame=True, return_X_y=True)

# the Bare_Nuclei column has some null values, so drop them
na_mask = data["Bare_Nuclei"].notna()
data = data[na_mask]
y = y[na_mask]

classes = {name: name for name in y.dtype.categories}

antecendent_terms = {
    col: ["very_low", "low", "medium", "high", "very_high"] for col in data.columns
}

cross_validate(
    data,
    y,
    hyperparams,
    antecendent_terms,
```

```
    classes,
    tensorboard_dir,
    train_test_swap=False,
    number_of_predictors=1,
)
```

evofuzzy/gymrunner.py

```
from itertools import count
from typing import Optional, List, Tuple, Union, Iterable

import gym
import numpy as np
import tensorboardX
from skfuzzy import control as ctrl

from .fuzzybase import FuzzyBase
from .fuzzygp import make_antecedent, make_binary_consequents, RuleSet


def _make_box_consequent(name: str, low: float, high: float) -> ctrl.Consequent:
    """Create a Consequent for use with the Box action space.
    :param name: name of the action
    :param low: Minimum value
    :param high: Maximum value
    :return: Consequent
    """
    cons = ctrl.Consequent(np.linspace(low, high, 11), name, "som")
    cons.automf(5, "quant")
    return cons


def _antecedents_from_env(env: gym.Env, inf_limit: float) -> List[ctrl.Antecedent]:
    """Create antecedents from a Box observation space.

    :param env: gym Environment
    :param inf_limit: limits to use if the observation_space says they are "inf"
    :return: list of Antecedents
    """
    observations = env.observation_space
    assert isinstance(
        observations, gym.spaces.Box
    ), "Only Box observation spaces supported"
    assert (
        len(observations.shape) == 1
    ), "Only one dimensional observation spaces supported"
    return [
        make_antecedent(f"obs_{i}", low, high, inf_limit=inf_limit)
        for (i, low, high) in zip(count(), observations.low, observations.high)
    ]
```

```
def _consequents_from_env(env: gym.Env) -> Tuple[List[ctrl.Consequent], bool]:
    """Create the consequents from the gym environment's action_space.

    :param env:  Gym environment
    :return: tuple with list of consequents and a flag to indicate if it is a Box or Discre
    """
    actions = env.action_space
    if isinstance(actions, gym.spaces.Box):
        assert len(actions.shape) == 1, "Only one dimensional action spaces supported"
        return (
            [
                _make_box_consequent(f"action_{i}", low, high)
                for (i, low, high) in zip(count(), actions.low, actions.high)
            ],
            True,
        )
    assert isinstance(
        actions, gym.spaces.Discrete
    ), "Only Box and Discrete actions supported"
    return make_binary_consequents(f"action_{i}" for i in range(actions.n)), False


class GymRunner(FuzzyBase):
    always_evaluate_ = True

    def train(
        self,
        env: gym.Env,
        tensorboard_writer: Optional["tensorboardX.SummaryWriter"] = None,
        antecedents: Optional[List[ctrl.Antecedent]] = None,
        inf_limit: float = 100.0,
    ):
        """Train the GymRunner against the gym environment.

        :param env: The environment
        :param tensorboard_writer: Optional object to write information to TensorBoard
        :param antecedents: Optional list of Antecendent objects to use
        :param inf_limit: limit to use if the environment limits are "inf"
        :return: None
        """
        if antecedents:
            self.antecedents_ = antecedents
        else:
            self.antecedents_ = _antecedents_from_env(env, inf_limit)
```

```
        self.consequents_, self.box_actions_ = _consequents_from_env(env)

        self._initialise(tensorboard_writer)

        if hasattr(self.toolbox_, "evaluate"):
            del self.toolbox_.evaluate
        self.toolbox_.register("evaluate", self._evaluate, env=env)
        self.execute(None, tensorboard_writer)

    def play(self, env: gym.Env, n: int = 1) -> float:
        """Display the best individual playing in the environment

        :param env: the Gym environment to play
        :param n: the number of top performing individuals to use - defaults to 1
        :return: the final reward
        """
        reward = self._evaluate(self.best_n(n), None, env, True)
        env.close()
        return reward[0]

    def _evaluate(
        self, individual: RuleSet, batch: None, env: gym.Env, render: bool = False
    ) -> Tuple[float]:
        """Evaluate one individual by running the gym environment with the
        fuzzy rules represented by the individual as the agent.

        :param individual: The individual to evaluate
        :param batch: Not used - needed for compatibility with FuzzyClassifier
        :param env: gym environment to use
        :param render: if true then render the scene every timestep
        :return: tuple containing the total reward
        """
        total_reward = 0
        rules = [self.toolbox_.compile(rule) for rule in individual]
        antecedents = {
            term.parent.label for rule in rules for term in rule.antecedent_terms
        }
        controller = ctrl.ControlSystem(rules)
        simulator = ctrl.ControlSystemSimulation(controller)

        observation = env.reset()
        while True:
            if render:
                env.render()
            action = self._evaluate_action(observation, simulator, antecedents)
            observation, reward, done, _ = env.step(action)
```

```
            total_reward += reward
            if done:
                break
        return (total_reward,)

    def _evaluate_action(
        self,
        observation: gym.spaces.Box,
        simulator: ctrl.ControlSystemSimulation,
        antecedents: Iterable[str],
    ) -> Union[int, List[float]]:
        """Run the FIS on the current observation and return the action to take.

        :param observation: a Box observation space
        :param simulator: scikit-fuzzy controller
        :param antecedents: set of antecedent names
        :return: the action to take.  Will be float or int, depending on if the action
        space is Box or Discrete
        """
        obs_vals = {
            ant.label: ob
            for (ant, ob) in zip(self.antecedents_, observation)
            if ant.label in antecedents
        }
        simulator.inputs(obs_vals)
        simulator.compute()
        if self.box_actions_:
            return self._evaluate_continuous_actions(simulator)
        else:
            return self._evaluate_discrete_actions(simulator)

    def _evaluate_discrete_actions(
        self, simulator: ctrl.ControlSystemSimulation
    ) -> int:
        """Get the Consequent values from the simulator and convert them to a Discrete valu

        :param simulator: scikit-fuzzy controller
        :return: Discrete value for the action
        """
        action_names = [cons.label for cons in self.consequents_]
        return np.argmax([simulator.output.get(name, 0) for name in action_names])

    def _evaluate_continuous_actions(
        self, simulator: ctrl.ControlSystemSimulation
    ) -> List[float]:
        """Get the Consequent values from the simulator and return them as a list of actions
```

```
        :param simulator: scikit-fuzzy controller
        :return: list of floats for the actions
        """
        return [
            simulator.output.get(f"action_{i}", 0)
            for i in range(len(self.consequents_))
        ]
```

run_cartpole.py

```
from datetime import datetime
from pathlib import Path
import tensorboardX
import gym
from evofuzzy import GymRunner
from evofuzzy.fuzzygp import make_antecedent

"""Script for running the CartPole-v0 environment"""

# tensorboard_dir = None
tensorboard_dir = "tb_logs/cartpole-v0"

if tensorboard_dir:
    logdir = Path(f"{tensorboard_dir}/{datetime.now().strftime('%Y%m%d-%H%M%S')}")
    logdir.mkdir(parents=True, exist_ok=True)
    tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
else:
    tensorboard_writer = None

env = gym.make("CartPole-v1")
runner = GymRunner(
    population_size=50,
    elite_size=1,
    n_iter=10,
    mutation_prob=0.9,
    crossover_prob=0.2,
    min_tree_height=1,
    max_tree_height=3,
    max_rules=4,
    whole_rule_prob=0.2,
)

antecedents = [
    make_antecedent("position", -2.4, 2.4),
    make_antecedent("velocity", -1, 1),
    make_antecedent("angle", -0.25, 0.25),
    make_antecedent("angular_velocity", -2, 2),
]

runner.train(env, tensorboard_writer, antecedents)
print(runner.best_str)
reward = runner.play(env)
print("Reward:", reward)
```

run_mountain_car.py

```
from datetime import datetime
from pathlib import Path
import tensorboardX
import gym
from evofuzzy import GymRunner

"""Script for running the MountainCar-v0 environment."""

tensorboard_dir = "tb_logs/mountaincar-v0"
if tensorboard_dir:
    logdir = Path(f"{tensorboard_dir}/{datetime.now().strftime('%Y%m%d-%H%M%S')}")
    logdir.mkdir(parents=True, exist_ok=True)
    tensorboard_writer = tensorboardX.SummaryWriter(str(logdir))
else:
    tensorboard_writer = None

env = gym.make("MountainCarContinuous-v0")
runner = GymRunner(
    population_size=50,
    elite_size=1,
    n_iter=10,
    mutation_prob=0.9,
    crossover_prob=0.2,
    min_tree_height=1,
    max_tree_height=3,
    max_rules=4,
    whole_rule_prob=0.2,
)

runner.train(env, tensorboard_writer)
print(runner.best_str)
reward = runner.play(env)
print("Reward:", reward)
```