pyalcs Documentation

Release 1.4

Norbert Kozlowski

Dec 16, 2019

Contents:

CHAPTER 1

Anticipatory Learning Classifier Systems (ALCS)

ALCS are is an extension to basic LCS compromising the notation of anticipations. Doing that the systems predominantly are able to **anticipate perceptual consequences of actions** independent of a reinforcement predictions.

ALCS are able to form complete anticipatory representation (build environment model) which allows faster and intelligent adaptation of behaviour or problem classification.

CHAPTER 2

Guide

2.1 Installation

Use a **Python 3.7** environment for development.

2.1.1 Creating environment with Conda (example)

Having a Conda distribution (i.e. Anacoda, Minicoda etc) create environment like:

```
conda create --name pyalcs python=3.7
```

Then activate it with:

```
source activate pyalcs
```

2.1.2 Dependencies

You should be fine with:

```
pip install -r requirements.txt
pip install -r requirements-integrations.txt --upgrade
```

In case of troubles see `Dockerfile` and `.travis.yml` how the project is built from scratch.

2.1.3 Launching example integrations

I assuming you are inside the virtual environment created before. In order to run the integrations from the console you need to specify Python PATH to use the currently checked-out version of `pyalcs` library:

```
PYTHONPATH=<PATH_TO_MODULE> python examples/acs2/maze/acs2_in_maze.py
```

2.1.4 Interactive notebooks

Start the Jupyter notebook locally with:

```
make notebook
```

Open the browser at `localhost:8888` and examine files inside `notebooks/` directory.

You might also want to install some extensions:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter contrib nbextension install
```

2.2 ACS2

ACS2 is derived from the original ACS framework. The most important change is that it embodies genetic generalization mechanism. Implementation is based on *"An Algorithmic Description of ACS2"* by Martin V. Butz and Wolfgang Stolzmann.

2.2.1 Original code

The original author's code is located in `assets/original` directory.

However it was written in 2001 when C++ was quite different than now. For that reason a slightly changed version (syntax) working on nowadays compilers can be found in `assets/ACS2`.

To compile the sources type (inside `assets/ACS2`):

```
make
```

And to run it:

```
./acs2++.out <environment>
```

For example:

```
./acs2++.out Envs/Maze4.txt
```

2.2.2 Maze statistics

After running an example integration, say `acs2_in_maze.py`, here's what the output tells you:

Agent stats

See `lcs.agents.acs2.ACS2`

- `population`: number of classifiers in the population
- `numerosity`: sum of numerosities of all classifiers in the population

- `reliable`: number of reliable classifiers in the population
- `fitness`: average classifier fitness in the population
- `trial`: trial number
- `steps`: number of steps in this trial
- `total_steps`: number of steps in all trials so far

Environment stats

There are currently no environment statistics for maze environment.

Performance stats

- `knowledge`: As defined in `examples.acs2.maze.utils.calculate_performance()`: If any of the reliable classifiers successfully predicts a transition, we say that the transition is anticipated correctly. This is a percentage of correctly anticipated transitions among all possible transitions.

2.3 rACS

rACS (*"Real-valued anticipatory classifier system"*) is an extension to ACS2 handling real-value input.

2.3.1 Real-value representation

Real values from range $x \in [0, 1]$ are encoded with predefined resolution using the following class

UBR

2.3.2 Changes

The following is and *in-progress* list of all modifications made to the ACS2:

Representation

- `Don't care` and `pass-through` symbols (in ACS2 `'#'`) are represented as fully ranged UBR.
- `complement_marks()` works if there is no previous value in the set

Classifier

- `specialize()` creates a fixed, narrow UBR like `UBR(4, 4)`. Later on during another processes it can be generalized more.
- `is_more_general` looks at average `cover_ratio`

Condition

- `cover_ratio` function,
- `does_match_condition` - analyzes incorporation (when doing subsumption)

Mark

- Mark holds a set of encoded perception values that holds bad states for classifier

Effect

- `is_specializable` looks inside range.

Components

- ALP implemented
- Custom mutation in GA (condition and effect)

Thoughts

- Maybe effect could return just encoded value, instead of UBR...
- Specificity/generality should measure how wide is the UBR
- Maybe `u_max` should hold information how specific condition should be (not just wildcards but spread)
- In the end of ALP phase we should perform something like classifier merge
- There is still very aggressive‘‘generalize‘‘ function in Condition part.
- favour most general condition and least general effect
- majority voting for best action
- mutation should not check for lower/upper bounds. Let it be random (alleles might swap)
- covering should add some random noise
- mutation can shrink/broad both condition / effect

2.4 Jupyter notebooks

2.4.1 Description of ACS2

History

ACS (1997)

First ALCS (ACS) was developed by Stolzman in 1997 (with puts an additional anticipatory part in each classifier). It is a new kind of classifier system that learn by using the cognitive mechanism of anticipatory behavioral control (introduced in cognitive psychology by Hoffman).

Anticipatory Behavioral Control

In 1992 Hoffman formulated a learning mechanism with basic assumption that a decisive factor for purposive behavior is the anticipation of the behavior consequences. Behavior consequences usually depend on the situation in which the behavior is executed. So it is necessary to learn in which situation `S` which behavior `R` (reaction) leads to which effects `E`.

CXCS (2000)

Tomlison and Bull (2000) published a cooperate learning classifier system (CXCS) in which cooperations between rules allow anticipatory proceses.

YACS (2001)

Another ALCS with an explicit anticipatory part YACS, has been published in Gerard and Siguad (2001).

ACS2

ACS2 (derived from ACS) is intendet to create a solution that is *complete*, *accurate* and *maximally general*. Major differences between ACS and ACS2: - ACS2 evolves explicit rules for situation-action tuples in which no changes occurs (a *pass-through-symbol* in E part **requires** a change in value), - ACS2's ALP (specialization pressure) and GA (generalization pressure) processes are improved, - starts with initially empty population of classifiers, - modifications are made on the whole action set (not just on the executed classifier),

Knowledge representation

Knowledge in an ACS2 is represented by a population of classifiers. Each classifier represents a *condition-action-effect* that anticipates the model *state* resulting from the execution of the *action* given the specified *conditions*.

A classifier in ACS2 always specifes a complete resulting state.

It consists of the following main components: - *condition part (C)* - specifies the set of situations in which a classifier is applicable, - *action part (A)* - proposes an available action, - *effect part (E)* - anticipates the effects of the proposed action in the specific conditions, - *quality (q)* - measures the accuracy of the anticipated results, $q \in [0, 1]$, - *reward prediction (r)* - estimates the reward encountered after the execution of action A in condition C, $r \in R$ - *immediate reward prediction (ir)* - estimates the *direct* reinforcement encountered after execution of action A in condition C, $ir \in R$

Classifier **fitness score** is calculated using quality and reward - $fitness(cl) = cl.q \cdot cl.r$

The *condition* and *effect* part consist of the values perceived from the environment and # symbols (i.e. $C, E \in \{l_1, l_2, \ldots, l_m, \#\}^L$.

A #-symbol in the: - condition part is called *"don't care"* and denotes that the classifier matches any value in this attribute, - effect part is called *"pass-through"* specifies that the classifier anticipates that the value of this attribute will not change after the execution of the specified action

Non pass-through symbols in E anticipate the change of the particular attribute to the specified value (in contrast to ACS in which a non pass-through symbol did not require a change in value).

Additionally each classifier compromises: - *Mark (M)* - records the values of each attribute of all situations in which the classifer did not anticipate correctly sometimes, - *GA timestamp (:math:'t_{ga}')* - timestamp when GA was last applied, - *ALP timestamp (:math:'t_{alp}')* - timestamp when ALP was last applied, - *application average (aav)* - estimates the frequency a classifier is updated (i.e. part of an action set), - *experience counter (exp)* - counts the

number of applications, - *numerosity (num)* - denotes the number of micro-classifers this macroclassifier represents (one classifier may represent many identical micro-classifier)

Agent interaction

ACS2 interacts *autonomously* with an environment.

In a *behavioral act* at a certain time t, the agent perceives a situation $\sigma(t) = \{l_1, l_2, \ldots, l_m\}^L$, where:

- m denotes the number of possible values of each environmental attribute (or feature),
- $l_1, l_2, \ldots, l_n$ denote the different possible values for each attribute,
- L denotes the string length.

  Note that each attribute can only take **discrete** values.

The system can act upon the environment with an action $\alpha(t) = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, where:

- n specifies the number of different possible actions in in the environment,
- $\alpha_1, \alpha_2, \ldots, \alpha_n$ denote the different possible actions

After the execution of an action, the environment provides a scalar reinforcement value $\rho(t) \in \mathbb{R}$

Environmental Model

By interacting with the environemnt the ACS2 learns about it's structure. Usually the agent starts without any prior knowledge. Initially new classifiers are mainly generated by a *covering* mechanizm in ALP. Later the ALP generates specialized clasifiers while the GG tries to introduce some genetic generalization.

Figure below presents the interaction with greater details. 1. After the **perception of the current situation** $\sigma(t)$, ACS2 forms a **match set** `[M]` comprising all classifiers in the population `[P]` whose conditions are satisfied in $\sigma(t)$, 2. ACS2 **chooses an action** $\alpha(t)$ according to some strategy (see below), 3. With respect to the chosen action, an **action set** `[A]` is generated that consist of all classifiers in `[M]` that specify the chosen action $\alpha(t)$, 4. After the execution of $\alpha(t)$ **classifier parameters are updated** by ALP and RL. New classifiers might be added or deleted due to the ALP and GG (see below).

Action selection plays an important role in building the environmental model. Aproaches for dealing with exploration/explotation trade-off can be used. It's worth mentioning that different approach should be taken for either stationary and non-stationary environments.

Literature: - *'"The exploration-exploitation dilemma for adaptive agents" <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.5740&rep=rep1&type=pdf>'__ ~* Rejeb (exploration/exploitation dillema in XCS) - *"Biasing Exploration in an Anticipatory Classifier System" ~* Butz (action delay bias, knowledge array bias) - *"Action-Planning in Anticipatory Classifier Systems" ~* Butz, Stolzmann (action planning) - *"'Should I stay or should I go? How the human brain manages the trade-off between exploitation and exploration <http://rstb.royalsocietypublishing.org/content/362/1481/933.short>'__"* (discussing the aspect of making decisons what examined in depth, gittins index, soft-max) - *"ACS2 Model Exploitation" ~* Butz (all techniques above)

Fact: Humans show a greater tendency to explore when there is more time left in a task

Aim is to explore unknown regions in a more directed way. In each time step the exploration can be executed with ϵ probability:

- random (default), > randomly select an action (might be the best or totally bad one),
- action delay bias, > the execution of actions which have been executed quite long ago (according to t_{alp}) in a given situation promises the detection of new situation-action-effect relations.

- knowledge array bias > calculates the averaged quality for each action from the match-set and selectes the worst one.
- action delay & knowledge array bias > action is chosen randomly using two approaches
- gittins index (NYI) > measure the reward that can be obtained taking into consideration the terminating condition (used in bandit problems). Restricted only to stationary problems. More info here
- soft-max (NYI) > the drawback of random action selection is that it chooses equally amongs all possible actions. Soft-max enables the possibility to vary action probabilities as a graded function of estimated value. More information about this probabilistic approach here
- expected and unexpected uncertanity (NYI) > biological approach by analysing ACh levels and NE signals
- best (default). > selects a classifier with highest fitness value - $q \cdot r$ from the match set `[M]`)
- action planning > Knows where the goal is and tries to find the shortes paths with known classifiers. It's also a good idea to explore unknown paths (not only this to goal).
- other techniques might use `ir` value (currently not used)

Anticipatory Learning Process (ALP)

The ALP was originally derived from the cognitive theory of anticipatory behavioral control (Hoffman, 1993).

It compares the anticipation of each classifier in an action set with the real next situation $\sigma(t+1)$.

The process results in the evaluation and specialization of the anticipatory model in ACS2.

> ALP first update classifier parameters. Next an offspring is generated and inaccuate classifiers are removed.
>
> The quality q of a generated classifier is set to the parent's value but never lower than 0.5 since the new classifier is supposedly better than the parental classifier. The reward prediction r inherits the value of its parent.

The execution of an action is accompanied by the formation of the action set, which represents the anticipations of the real next situation. Thus, ACS2 satisfies the first point of Hoffmann's theory of anticipatory behavioral control which states that *any behavioral act or response (R) is accompanied with an anticipation of its effects*. Moreover, *the comparison of the anticipation of each classifier in [A] can be compared to a continuous comparison of anticipations with real next situations* as stated in Hoffmann's second point. The third and fourth point address the consequences of the comparison and are realized in the distinction of an unexpected case and an expected case.

Parameter updates

The following parameters are updated in the **following order** - quality (q), mark (M), application average (aav), ALP timestamp (t_{alp}) and the experience counter (exp).

Quality

The quality q is updated according to the classifiers anticipation. If the classifier predicted correctly, the quality is increased using the following (Widrow-Hoff delta rule) formula:

$$q \leftarrow q + \beta(1 - q)$$

Otherwise it is decreased:

$In the equation, : math : `\beta \in [0, 1]` denotes the * *learning rate * *of ACS2. The smaller the learning rate, the more passive ACS$

Mark

Situation $\sigma(t) = (\sigma_1, \ldots, \sigma_L)$ is added to the mark $M = (m_i, \ldots, m_L)$ if the classifier did not anticipate correctly. In this case $:nbsphinx-math:*forall*_i m_i = m_i \cup‘: $nbsphinx - math : sigma‘_i\}$.

Application average

Parameter is updated using the *“moyenne adaptive modifiee”* technique as introduced in Venturini (1994).

$The technique assures the fast adaptation of : math : ‘aav‘once the classifier is introduced and later assures a continues update a c$

ALP timestamp

The ALP time stamp is set to the current time t recording the last parameter update in the ALP.

$$cl.t_{alp} \leftarrow t$$

Experience

Increment experience by 1

$$cl.exp \leftarrow cl.exp + 1$$

Classifier generation and deletion

The ALP generates specialized offspring and/or deletes *inaccurate* classifiers.

> Inaccurate classifiers are classifiers whose quality is lower than the inaccuracy threshold θ_i. When the quality of a classifiers falls below θ_i after an update, it is deleted.

More specialized classifiers are generated in two ways.

1. An **expected case**, in which a classifier anticipated the correct outcome, a classifier *might* be generated if the mark M differs from the situation $\sigma(t)$ in some attributes, i.e. $\exists_{i,j} l_j \in m_i \wedge l_j \neq \sigma_i$. Since the mark specifies the characteristics of situations in which a classifier did not work correctly, a **difference indicates that the specific position might be important to distinguish the correct and wrong outcome case**. Thus, the ALP generates an offspring whose conditions are further specialized. If there are unique differences in the mark compared to the current situation, i.e. $\exists_i \sigma_i \notin m_i$, then one of the unique difference is specialized in the offspring. However, if there are only positions that differ but σ_i is always part of m_i, i.e. $\forall_i \sigma_i \in m_i$, then all differing positions are specialized. > The number of specialized positions in the conditions that are not specialized in the effects is limited to u_{max}.

In other words: - the strength of the classifier is increased, - if there is a mark, then a new classifier is formed that is more specific in the `C` and `E` parts, trying to exclude the possibility to be chosen in the state(s) described by mark

2. In an **unexpected case**, a classifier did not anticipate the correct outcome (one or more predicted changes are incorrect or when one or more components change that were predicted to stay the same).

In this case a classifier: - is marked by the situation $\sigma(t)$, - has decreased quality: $cl.q \leftarrow cl.q - \beta \cdot cl.q$

In difference with the ACS all classifiers from the action set will become marked in this case.

Also, an offspring classifier may be generated, if the effect part of the old classifier can be further specialized (by changing pass-through symbols to specific values) to specify the perceived outcome correctly. All positions in condition and effect part are specialized that change from $\sigma(t)$ to $\sigma(t+1)$.

Therefore attributes whose value changed in the environment but are anticipated to stay the same are specified in condition and effect part of the offspring.

In other words: - the classifier gets a mark, - the strength of the classifier is decreased, - if possible, a new classifier is generated that is more specific than the old one and that anticipates the environment correctly.

A classifier is also generated if there was no other classifier in the actual action set `[A]` that anticipated the effect correctly. In this case, a **covering classifier** is generated that is specialized in all attributes in condition and effect part that changed from $\sigma(t)$ to $\sigma(t+1)$. > The covering method was not applied in ACS since in ACS a completely general classifiers was always present for each action.

The attributes of the Mark M of the covering classifier are initially empty. Quality q is set to 0.5 as well as the reward prediction r, while the immediate reward prediction ir as well as the application average avv are set to 0. The time stamps are set to the current time t.

$$cl.M \leftarrow \emptyset$$
$$cl.q \leftarrow 0.5$$
$$cl.r \leftarrow 0.5$$
$$cl.ir \leftarrow 0$$
$$cl.aav \leftarrow 0$$
$$cl.t_{alp} \leftarrow t$$
$$cl.t_{ga} \leftarrow t$$

$It is important to mention that both adding and removing classifiers * * does not influence the ongoing ALP application * *.$

Genetic Generalization (GG)

While the ALP specializes classifiers in a quite competent way, over-specializations can occur sometimes as studied in (Butz, 2001). Since the over-specialization cases can be caused by various circumstances, a *genetic generalization* (GG) mechanism was applied that, interacting with the ALP, results in the evolution of a complete, accurate, and maximally general model.

The mechanism starts after applying ALP module, and looks as follow:

1. **Determine if GG should be applied**. GG is applied if the average time since last GG application in the current action set `[A]` is larger than the threshold θ_{ga}.

$$t - \frac{\sum_{cl \in [A]} cl.t_{ga} cl.num}{\sum_{cl \in [A]} cl.num} > \theta_{ga}$$

2. If the mechanism is applied update the application time for all classifiers $\forall_{cl \in [A]} cl.t_{ga} \leftarrow t$,
3. Select two classifiers using *"roulette-wheel selection"*, with respect to their qualities q.

4. Reproduce two classifiers, by removing marks and halving the qualities.

$$cl_1.M \leftarrow \emptyset$$
$$cl_2.M \leftarrow \emptyset$$
$$cl_1.q \leftarrow \frac{cl_1.q}{2}$$
$$cl_2.q \leftarrow \frac{cl_2.q}{2}$$

5. Mutate classifiers, by applying a *"generalizing mutation"*. >Generalizing mutation is only mutating **specified** attributes in the condition part C back to don't care # symbols. A specialized attribute is generalized with a probability μ. Moreover, conditions of the offspring are crossed applying **two-point crossover** with a probability of χ. In the case of a crossover application, quality, reward prediction, and immediate reward prediction are averaged over the offspring.

6. Insert new offspring classifiers into `[P]` and `[A]` > If a generated offspring already exists in the population, the offspring classifier is discarded and if the existing classifier is not marked its numerosity is increased by one.

$$cl.M \leftarrow \emptyset$$
$$cl.num \leftarrow cl.num + 1$$

The GG mechanism also applies a **deletion procedure** inside the action set. If an action set `[A]` exceeds the action set size threshold θ_{as}, excess classifiers are deleted in `[A]`. The procedure applies a modified *"tournament selection"* process in which the classifier with the significant lowest quality, or the classifier with the highest specificity is deleted. Thus, deletion causes the extinction of low-quality as well as over-specialized classifiers.

Aproximatelly **a third of the action set size** takes part in the tournament.

Reinforcement Learning (RL)

RL approach in ACS2 adapts the *Q-learning* (Watkins, 1989; Watkins & Dayan, 1992) idea (away from the traditional bucket brigade algorithm).

An environmental model needs to be specific enough to represent a proper behavioral policy (rewards) in the model. If this is not the case, *"model aliasing"* might take place.

> In *"model aliasing"* a model is completely accurate in terms of anticipations but over-general with respect to reinforcement.

In order to learn an optimal behavioral policy in ACS2, parameters r and ir are continuously updated after executing an action and obtaining next environmental perception and reward - $\rho(t)$.

$$r \leftarrow r + \beta(\rho(t) + \gamma \max_{cl \in [M](t+1) \wedge cl.E \neq \{\#\}^L} (cl.q \cdot cl.r) - r)$$
$$ir \leftarrow ir + \beta(\rho(t) - ir)$$

As before $\beta \in [0, 1]$ denotes the *"learning rate"* biasing the parameters more or less towards recently encountered reward. $\gamma \in [0, 1)$ denotes the *"discount factor"*.

> The values of r and ir consequently specify an average of the resulting reward after the execution of action A over all possible situations of the environment in which the classifier is applicable.

Other aspects

Subsumption

If an offspring classifier was generated (regardless if by ALP or GG), the set is searched for a subsuming classifier.

The offspring is subsumed if a classifier: - has more general condition part, - has identical effect part, - is reliable (its quality is higher than the threshold θ_r), - is not marked, - is experienced (its experience counter exp is higher than the threshold θ_{exp}).

If there are more than one possible subsumer, the syntactically maximally general subsumer is chosen. In the case of a draw, the subsumer is chosen at random from the maximally general ones. If a subsumer was found, the offspring is discarded and either quality or numerosity is increased dependent on if the offspring was generated by ALP or GG, respectively.

Interaction of ALP and GG.

Several distinct studies in various environments revealed that the interaction of ALP and GG is able to evolve a complete, accurate, and maximally general model in various environments in a competent way (see e.g. Butz, Goldberg, and Stolzmann, 2000, Butz, 2001). The basic idea behind the interacting model learning processes is that the specialization process extracts as much information as possible from the encountered environment continuously specializing over-general classifiers. The GG mechanism, on the other hand, randomly generalizes exploiting the power of a genetic algorithm where no more additional information is available from the environment. The ALP ensures diversity and prevents the loss of information of a particular niche in the environment. Only GG generates identical classifiers and causes convergence in the population.

Sources

- *"Anticipatory Classifier Systems"* - Wolfgang Stolzmann
- *"Biasing Exploration in an Anticipatory Learning Classifier System"* - Martin V. Butz

Other resources

- ACS-tutorial (presentation)

2.4.2 ACS2 in Maze

This notebook presents how to integrate ACS2 algorithm with maze environment (using OpenAI Gym interface).

Begin with attaching required dependencies. Because most of the work is by now done locally no PIP modules are used (just pure OS paths)

[1]:
```
# General
from __future__ import unicode_literals

%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# To avoid Type3 fonts in generated pdf file
matplotlib.rcParams['pdf.fonttype'] = 42
matplotlib.rcParams['ps.fonttype'] = 42

# Logger
import logging
logging.basicConfig(level=logging.WARN)

# ALCS + Custom environments
import sys, os
sys.path.append(os.path.abspath('../'))

# Enable automatic module reload
%load_ext autoreload
%autoreload 2

# Load PyALCS module
from lcs.agents.acs2 import ACS2, Configuration, ClassifiersList

# Load environments
import gym
import gym_maze
```

Environment - Maze

We are going to look at provided mazes. Their names starts with "*Maze...*" or "*Woods...*" so see what is possible to load:

[2]:
```
# Custom function for obtaining available environments
filter_envs = lambda env: env.id.startswith("Maze") or env.id.startswith("Woods")

all_envs = [env for env in gym.envs.registry.all()]
maze_envs = [env for env in all_envs if filter_envs(env)]

for env in maze_envs:
    print("Maze ID: [{}], non-deterministic: [{}], trials: [{}]".format(
            env.id, env.nondeterministic, env.trials))
```

```
Maze ID: [MazeF1-v0], non-deterministic: [False], trials: [100]
Maze ID: [MazeF2-v0], non-deterministic: [False], trials: [100]
Maze ID: [MazeF3-v0], non-deterministic: [False], trials: [100]
Maze ID: [MazeF4-v0], non-deterministic: [True], trials: [100]
Maze ID: [Maze4-v0], non-deterministic: [False], trials: [100]
Maze ID: [Maze5-v0], non-deterministic: [False], trials: [100]
Maze ID: [Maze6-v0], non-deterministic: [True], trials: [100]
Maze ID: [Woods1-v0], non-deterministic: [False], trials: [100]
Maze ID: [Woods14-v0], non-deterministic: [False], trials: [100]
```

Let's see how it looks in action. First we are going to initialize new environment using `gym.make()` instruction from OpenAI Gym.

[3]:
```
#MAZE = "Woods14-v0"
MAZE = "Maze5-v0"

# Initialize environment
```

(continues on next page)

(continued from previous page)

```
maze = gym.make(MAZE)

# Reset it, by putting an agent into random position
situation = maze.reset()

# Render the state in ASCII
maze.render()
```

```
      $




  A
```

The `reset()` function puts an agent into random position (on path inside maze) returning current perception.

> The perception consists of 8 values representing N, NE, E, SE, S, SW, W, NW directions. It outputs 0 for the path, 1 for the wall and 9 for the reward.

[4]:
```
# Show current agents perception
situation
```

[4]:
```
('1', '0', '0', '1', '1', '1', '0', '0')
```

We can interact with the environment by performing actions.

> Agent can perform 8 actions - moving into different directions.

To do so use `step(action)` function. It will return couple interesting information: - new state perception, - reward for executing move (ie. finding the reward) - is the trial finish, - debug data

[5]:
```
ACTION = 0 # Move N

# Execute action
state, reward, done, _ = maze.step(ACTION)

# Show new state
print(f"New state: {state}, reward: {reward}, is done: {done}")

# Render the env one more time after executing step
maze.render()
```

```
New state: ('1', '0', '0', '1', '1', '1', '0', '0'), reward: 0, is done: False

      $




  A
```

Agent - ACS2

First provide a helper method for calculating obtained knowledge

[6]:
```
def _maze_knowledge(population, environment) -> float:
    transitions = environment.env.get_all_possible_transitions()

    # Take into consideration only reliable classifiers
    reliable_classifiers = [c for c in population if c.is_reliable()]

    # Count how many transitions are anticipated correctly
    nr_correct = 0

    # For all possible destinations from each path cell
    for start, action, end in transitions:

        p0 = environment.env.maze.perception(*start)
        p1 = environment.env.maze.perception(*end)

        if any([True for cl in reliable_classifiers
                if cl.predicts_successfully(p0, action, p1)]):
            nr_correct += 1

    return nr_correct / len(transitions) * 100.0
```

[7]:
```
from lcs.metrics import population_metrics

def _maze_metrics(pop, env):
    metrics = {
        'knowledge': _maze_knowledge(pop, env)
    }

    # Add basic population metrics
    metrics.update(population_metrics(pop, env))

    return metrics
```

Exploration phase

[8]:
```
CLASSIFIER_LENGTH=8
NUMBER_OF_POSSIBLE_ACTIONS=8

# Define agent's default configuration
cfg = Configuration(
    classifier_length=CLASSIFIER_LENGTH,
    number_of_possible_actions=NUMBER_OF_POSSIBLE_ACTIONS,
    metrics_trial_frequency=1,
    user_metrics_collector_fcn=_maze_metrics)

# Define agent
agent = ACS2(cfg)
```

[9]:
```
%%time
population, metrics = agent.explore(maze, 100)
```

```
CPU times: user 5.19 s, sys: 11.7 ms, total: 5.2 s
Wall time: 5.22 s
```

We can take a sneak peek into a created list of classifiers. Let's have a look at top 10:

[10]:
```
population.sort(key=lambda cl: -cl.fitness)

for cl in population[:10]:
    print("{!r} \tq: {:.2f} \tr: {:.2f} \tir: {:.2f}".format(cl, cl.q, cl.r, cl.ir))
```

```
9####010 0 1####101          (empty)                q: 0.963 r: 884.0  ir: 884.0  f:␣
↪851.7  exp: 41  tga: 645   talp: 2817  tav: 46.2   num: 1    q: 0.96         r: 884.
↪02       ir: 884.02
9#1##010 0 1####101          (empty)                q: 0.921 r: 809.5  ir: 806.3  f:␣
↪745.3  exp: 31  tga: 1682  talp: 2817  tav: 35.7   num: 1     q: 0.92         r:␣
↪809.54       ir: 806.29
##901### 2 ##110###          (empty)                q: 0.875 r: 762.2  ir: 762.2  f:␣
↪666.8  exp: 27  tga: 590   talp: 2875  tav: 79.1   num: 1  q: 0.87         r: 762.
↪17       ir: 762.17
011###01 0 9#####10          (empty)                q: 0.989 r: 563.2  ir: 0.0    f:␣
↪557.2  exp: 40  tga: 951   talp: 2817  tav: 43.4   num: 1    q: 0.99         r: 563.
↪17       ir: 0.00
01##0#01 0 9#####10          (empty)                q: 0.976 r: 563.2  ir: 0.0    f:␣
↪549.5  exp: 41  tga: 949   talp: 2817  tav: 42.7   num: 1    q: 0.98         r: 563.
↪17       ir: 0.00
01110001 0 9#####10          (empty)                q: 0.972 r: 563.0  ir: 0.0    f:␣
↪547.1  exp: 41  tga: 949   talp: 2817  tav: 41.3   num: 1       q: 0.97         r:␣
↪563.01       ir: 0.00
0#1##001 0 9#####10          (empty)                q: 0.953 r: 553.8  ir: 0.0    f:␣
↪527.6  exp: 32  tga: 1769  talp: 2817  tav: 32.5   num: 1    q: 0.95         r: 553.
↪76       ir: 0.00
1000#101 1 9111#010          (empty)                q: 0.942 r: 347.8  ir: 0.0    f:␣
↪327.6  exp: 14  tga: 644   talp: 2795  tav: 1.46e+02 num: 1        q: 0.94
↪r: 347.78       ir: 0.00
1#0110## 2 ##90#1##          (empty)                q: 0.958 r: 290.4  ir: 0.0    f:␣
↪278.1  exp: 22  tga: 1168  talp: 2874  tav: 76.9   num: 1    q: 0.96         r: 290.
↪39       ir: 0.00
11011001 2 ##90#1##          (empty)                q: 0.846 r: 290.3  ir: 0.0    f:␣
↪245.7  exp: 22  tga: 1168  talp: 2874  tav: 77.0   num: 1       q: 0.85         r:␣
↪290.33       ir: 0.00
```

Exploitation

Now we can either reuse our previous agent or initialize it one more time passing the initial population of classifiers as *apriori* knowledge.

[11]:
```
# Reinitialize agent using defined configuration and population
agent = ACS2(cfg, population)
```

[12]:
```
%%time
population, metrics = agent.exploit(maze, 1)
```

```
CPU times: user 33.3 ms, sys: 16 µs, total: 33.4 ms
Wall time: 33.4 ms
```

[13]:
```
metrics[-1]
```

[13]:
```
{'trial': 0,
 'steps_in_trial': 2,
 'reward': 1000,
 'knowledge': 24.65753424657534,
 'population': 419,
 'numerosity': 419,
 'reliable': 78}
```

Experiments

[14]:
```
def parse_metrics_to_df(explore_metrics, exploit_metrics):
    def extract_details(row):
        row['trial'] = row['trial']
        row['steps'] = row['steps_in_trial']
        row['numerosity'] = row['numerosity']
        row['reliable'] = row['reliable']
        row['knowledge'] = row['knowledge']
        return row

    # Load both metrics into data frame
    explore_df = pd.DataFrame(explore_metrics)
    exploit_df = pd.DataFrame(exploit_metrics)

    # Mark them with specific phase
    explore_df['phase'] = 'explore'
    exploit_df['phase'] = 'exploit'

    # Extract details
    explore_df = explore_df.apply(extract_details, axis=1)
    exploit_df = exploit_df.apply(extract_details, axis=1)

    # Adjuts exploit trial counter
    exploit_df['trial'] = exploit_df.apply(lambda r: r['trial']+len(explore_df),␣
↪axis=1)

    # Concatenate both dataframes
    df = pd.concat([explore_df, exploit_df])
    df.set_index('trial', inplace=True)

    return df
```

For various mazes visualize - classifiers / reliable classifiers for steps - optimal policy - steps (exploration | exploitation) - knowledge - parameters setting

[15]:
```
def find_best_classifier(population, situation, cfg):
    match_set = population.form_match_set(situation)
    anticipated_change_cls = [cl for cl in match_set if cl.does_anticipate_change()]

    if (len(anticipated_change_cls) > 0):
        return max(anticipated_change_cls, key=lambda cl: cl.fitness)

    return None

def build_fitness_matrix(env, population, cfg):
```

(continues on next page)

(continued from previous page)

```
    original = env.env.maze.matrix
    fitness = original.copy()

    # Think about more 'functional' way of doing this
    for index, x in np.ndenumerate(original):
        # Path - best classfier fitness
        if x == 0:
            perception = env.env.maze.perception(index[1], index[0])
            best_cl = find_best_classifier(population, perception, cfg)
            if best_cl:
                fitness[index] = best_cl.fitness
            else:
                fitness[index] = -1

        # Wall - fitness = 0
        if x == 1:
            fitness[index] = 0

        # Reward - inf fitness
        if x == 9:
            fitness[index] = fitness.max () + 500

    return fitness

def build_action_matrix(env, population, cfg):
    ACTION_LOOKUP = {
        0: u'↑', 1: u'', 2: u'→', 3: u'',
        4: u'↓', 5: u'', 6: u'←', 7: u''
    }

    original = env.env.maze.matrix
    action = original.copy().astype(str)

    # Think about more 'functional' way of doing this
    for index, x in np.ndenumerate(original):
        # Path - best classfier fitness
        if x == 0:
            perception = env.env.maze.perception(index[1], index[0])
            best_cl = find_best_classifier(population, perception, cfg)
            if best_cl:
                action[index] = ACTION_LOOKUP[best_cl.action]
            else:
                action[index] = '?'

        # Wall - fitness = 0
        if x == 1:
            action[index] = '\#'

        # Reward - inf fitness
        if x == 9:
            action[index] = 'R'

    return action
```

Plotting functions and settings

[16]:
```
# Plot constants
TITLE_TEXT_SIZE=24
AXIS_TEXT_SIZE=18
LEGEND_TEXT_SIZE=16
```

[17]:
```
def plot_policy(env, agent, cfg, ax=None):
    if ax is None:
        ax = plt.gca()

    ax.set_aspect("equal")

    # Handy variables
    maze_countours = maze.env.maze.matrix
    max_x = env.env.maze.max_x
    max_y = env.env.maze.max_y

    fitness_matrix = build_fitness_matrix(env, agent.population, cfg)
    action_matrix = build_action_matrix(env, agent.population, cfg)

    # Render maze as image
    plt.imshow(fitness_matrix, interpolation='nearest', cmap='Reds', aspect='auto',
           extent=[0, max_x, max_y, 0])


    # Add labels to each cell
    for (y,x), val in np.ndenumerate(action_matrix):
        plt.text(x+0.4, y+0.5, "${}$".format(val))

    ax.set_title("Policy", fontsize=TITLE_TEXT_SIZE)
    ax.set_xlabel('x', fontsize=AXIS_TEXT_SIZE)
    ax.set_ylabel('y', fontsize=AXIS_TEXT_SIZE)

    ax.set_xlim(0, max_x)
    ax.set_ylim(max_y, 0)

    ax.set_xticks(range(0, max_x))
    ax.set_yticks(range(0, max_y))

    ax.grid(True)
```

[18]:
```
def plot_knowledge(df, ax=None):
    if ax is None:
        ax = plt.gca()

    explore_df = df.query("phase == 'explore'")
    exploit_df = df.query("phase == 'exploit'")

    explore_df['knowledge'].plot(ax=ax, c='blue')
    exploit_df['knowledge'].plot(ax=ax, c='red')
    ax.axvline(x=len(explore_df), c='black', linestyle='dashed')

    ax.set_title("Achieved knowledge", fontsize=TITLE_TEXT_SIZE)
    ax.set_xlabel("Trial", fontsize=AXIS_TEXT_SIZE)
    ax.set_ylabel("Knowledge [%]", fontsize=AXIS_TEXT_SIZE)
    ax.set_ylim([0, 105])
```

[19]:
```
def plot_steps(df, ax=None):
    if ax is None:
        ax = plt.gca()

    explore_df = df.query("phase == 'explore'")
    exploit_df = df.query("phase == 'exploit'")

    explore_df['steps'].plot(ax=ax, c='blue', linewidth=.5)
    exploit_df['steps'].plot(ax=ax, c='red', linewidth=0.5)
    ax.axvline(x=len(explore_df), c='black', linestyle='dashed')

    ax.set_title("Steps", fontsize=TITLE_TEXT_SIZE)
    ax.set_xlabel("Trial", fontsize=AXIS_TEXT_SIZE)
    ax.set_ylabel("Steps", fontsize=AXIS_TEXT_SIZE)
```

[20]:
```
def plot_classifiers(df, ax=None):
    if ax is None:
        ax = plt.gca()

    explore_df = df.query("phase == 'explore'")
    exploit_df = df.query("phase == 'exploit'")

    df['numerosity'].plot(ax=ax, c='blue')
    df['reliable'].plot(ax=ax, c='red')

    ax.axvline(x=len(explore_df), c='black', linestyle='dashed')

    ax.set_title("Classifiers", fontsize=TITLE_TEXT_SIZE)
    ax.set_xlabel("Trial", fontsize=AXIS_TEXT_SIZE)
    ax.set_ylabel("Classifiers", fontsize=AXIS_TEXT_SIZE)
    ax.legend(fontsize=LEGEND_TEXT_SIZE)
```

[21]:
```
def plot_performance(agent, maze, metrics_df, cfg, env_name):
    plt.figure(figsize=(13, 10), dpi=100)
    plt.suptitle(f'ACS2 Performance in {env_name} environment', fontsize=32)

    ax1 = plt.subplot(221)
    plot_policy(maze, agent, cfg, ax1)

    ax2 = plt.subplot(222)
    plot_knowledge(metrics_df, ax2)

    ax3 = plt.subplot(223)
    plot_classifiers(metrics_df, ax3)

    ax4 = plt.subplot(224)
    plot_steps(metrics_df, ax4)

    plt.subplots_adjust(top=0.86, wspace=0.3, hspace=0.3)
```

Maze5

[22]:
```
%%time

# define environment
```

(continues on next page)

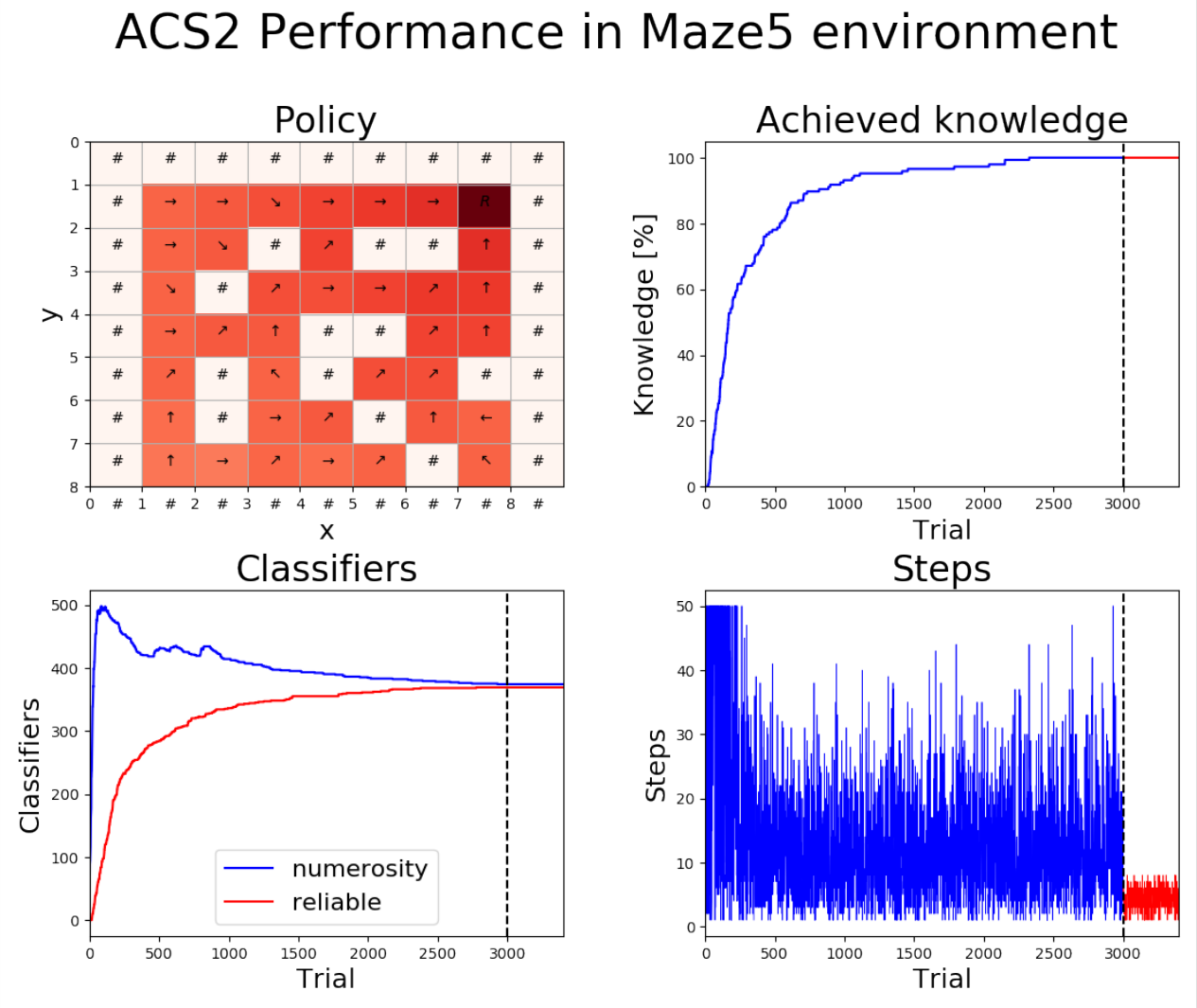
(continued from previous page)

```
maze5 = gym.make('Maze5-v0')

# explore
agent_maze5 = ACS2(cfg)
population_maze5_explore, metrics_maze5_explore = agent_maze5.explore(maze5, 3000)

# exploit
agent_maze5 = ACS2(cfg, population_maze5_explore)
_, metrics_maze5_exploit = agent_maze5.exploit(maze5, 400)
```

```
CPU times: user 7min 2s, sys: 876 ms, total: 7min 3s
Wall time: 7min 4s
```

[23]:
```
maze5_metrics_df = parse_metrics_to_df(metrics_maze5_explore, metrics_maze5_exploit)
```

[24]:
```
plot_performance(agent_maze5, maze5, maze5_metrics_df, cfg, 'Maze5')
```



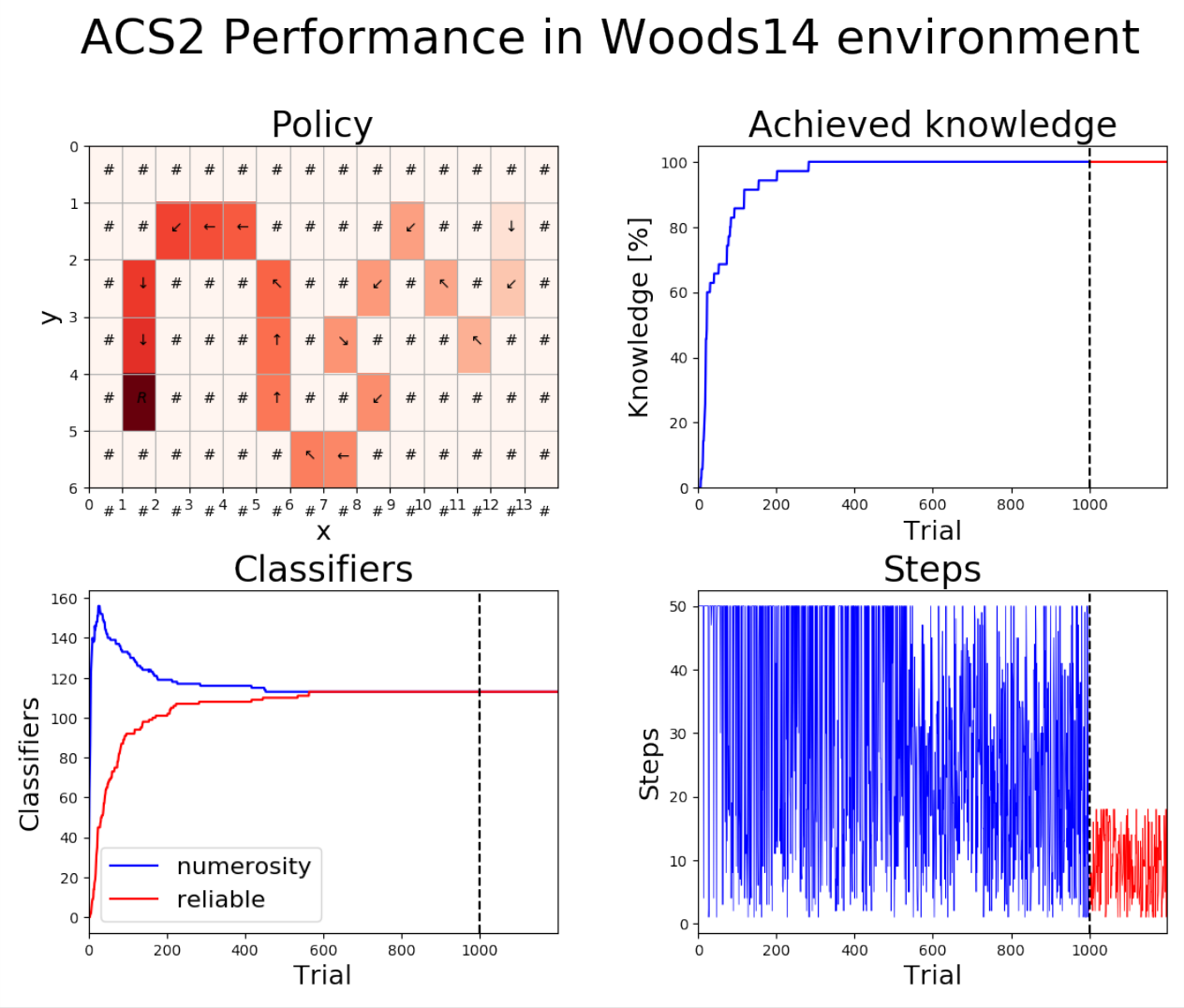
Woods14

[25]:
```
%%time

# define environment
woods14 = gym.make('Woods14-v0')

# explore
agent_woods14 = ACS2(cfg)
population_woods14_explore, metrics_woods14_explore = agent_woods14.explore(woods14,␣
↪1000)

# exploit
agent_woods14 = ACS2(cfg, population_woods14_explore)
_, metrics_woods14_exploit = agent_woods14.exploit(woods14, 200)
```

```
CPU times: user 32.1 s, sys: 62.5 ms, total: 32.2 s
Wall time: 32.3 s
```

[26]:
```
woods14_metrics_df = parse_metrics_to_df(metrics_woods14_explore, metrics_woods14_
↪exploit)
```

[27]:
```
plot_performance(agent_woods14, woods14, woods14_metrics_df, cfg, 'Woods14')
```



[1]:
```
%matplotlib inline

import matplotlib.pyplot as plt

import gym
from gym.envs.registration import register
```

2.4.3 ACS2 in Frozen Lake

About the environment > The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

[2]:
```
fl_env = gym.make('FrozenLake-v0')

# Reset the state
state = fl_env.reset()
```

(continues on next page)

(continued from previous page)

```
# Render the environment
fl_env.render()
```

```
SFFF
FHFH
FFFH
HFFG
```

Each state might get following possible values: `{S, F, H, G}` which, refers to

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

In case of interacting with environment agent cant perform 4 action which map as follow: - 0 - left - 1 - down - 2 - right - 3 - up

> FrozenLake-v0 defines "solving" as getting average reward of 0.78 over 100 consecutive trials.

We will also define a second version of the same environment but with `slippery=False` parameters. That make it more deterministic.

[3]:
```
register(
    id='FrozenLakeNotSlippery-v0',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'map_name': '4x4', 'is_slippery': False},
    max_episode_steps=100,
    reward_threshold=0.78,  # optimum = .8196
)

fl_ns_env = gym.make('FrozenLakeNotSlippery-v0')

# Reset the state
state = fl_ns_env.reset()

# Render the environment
fl_ns_env.render()
```

```
SFFF
FHFH
FFFH
HFFG
```

ACS2

[4]:
```
# Import PyALCS code from local path
import sys, os
sys.path.append(os.path.abspath('../'))

from lcs.agents import EnvironmentAdapter
from lcs.agents.acs2 import ACS2, Configuration
```

(continues on next page)

(continued from previous page)

```
# Enable automatic module reload
%load_ext autoreload
%autoreload 2

CLASSIFIER_LENGTH = 16  # Because we are operating in 4x4 grid
POSSIBLE_ACTIONS = fl_env.action_space.n  # 4
```

Encoding perception

The only information back from the environment is the current agent position (not it's perception). Therefore our agent task will be to predicit where it will land after executing each action.

To do so we will represent state as a one-hot encoded vector.

[5]:
```
class FrozenLakeAdapter(EnvironmentAdapter):
    @classmethod
    def to_genotype(cls, phenotype):
        genotype = ['0' for i in range(CLASSIFIER_LENGTH)]
        genotype[phenotype] = 'X'
        return ''.join(genotype)
```

`X` corresponds to current agent position. State 4 is encoded as follows:

[6]:
```
FrozenLakeAdapter().to_genotype(4)
```

[6]:
```
'0000X00000000000'
```

Environment metrics

We will also need a function for evaluating if agent finished succesfuly a trial

[7]:
```
from lcs.metrics import population_metrics


# We assume if the final state was with number 15 that the algorithm found the reward.
↪ Otherwise not
def fl_metrics(pop, env):
    metrics = {
        'found_reward': env.env.s == 15,
    }

    # Add basic population metrics
    metrics.update(population_metrics(pop, env))

    return metrics
```

Performance evaluation

[8]:
```
def print_performance(population, metrics):
    population.sort(key=lambda cl: -cl.fitness)
    population_count = len(population)
```

(continues on next page)

(continued from previous page)

```
    reliable_count = len([cl for cl in population if cl.is_reliable()])
    successful_trials = sum(m['found_reward'] for m in metrics)

    print("Number of classifiers: {}".format(population_count))
    print("Number of reliable classifiers: {}".format(reliable_count))
    print("Percentage of successul trials: {:.2f}%".format(successful_trials /␣
↪EXPLOIT_TRIALS  * 100))
    print("\nTop 10 classifiers:")
    for cl in population[:10]:
        print("{!r} \tq: {:.2f} \tr: {:.2f} \tir: {:.2f} \texp: {}".format(cl, cl.q,␣
↪cl.r, cl.ir, cl.exp))
```

[9]:
```
def plot_success_trials(metrics, ax=None):
    if ax is None:
        ax = plt.gca()

    trials = [m['trial'] for m in metrics]
    success = [m['found_reward'] for m in metrics]

    ax.plot(trials, success)
    ax.set_title("Successful Trials")
    ax.set_xlabel("Trial")
    ax.set_ylabel("Agent found reward")
```

[10]:
```
def plot_population(metrics, ax=None):
    if ax is None:
        ax = plt.gca()

    trials = [m['trial'] for m in metrics]

    population_size = [m['numerosity'] for m in metrics]
    reliable_size = [m['reliable'] for m in metrics]

    ax.plot(trials, population_size, 'b', label='all')
    ax.plot(trials, reliable_size, 'r', label='reliable')

    ax.set_title("Population size")
    ax.set_xlabel("Trial")
    ax.set_ylabel("Number of macroclassifiers")
    ax.legend(loc='best')
```

[11]:
```
def plot_performance(metrics):
    plt.figure(figsize=(13, 10), dpi=100)
    plt.suptitle('Performance Visualization')

    ax1 = plt.subplot(221)
    plot_success_trials(metrics, ax1)

    ax2 = plt.subplot(222)
    plot_population(metrics, ax2)

    plt.show()
```

Default ACS2 configuration

Right now we are ready to configure the ACS2 agent providing some defaults

[12]:
```
cfg = Configuration(
    classifier_length=CLASSIFIER_LENGTH,
    number_of_possible_actions=POSSIBLE_ACTIONS,
    environment_adapter=FrozenLakeAdapter(),
    metrics_trial_frequency=1,
    user_metrics_collector_fcn=fl_metrics,
    theta_i=0.3,
    epsilon=0.7)

print(cfg)
```

```
ACS2Configuration:
        - Classifier length: [16]
        - Number of possible actions: [4]
        - Classifier wildcard: [#]
        - Environment adapter function: [<__main__.FrozenLakeAdapter object at␣
↪0x117bd0b00>]
        - Fitness function: [None]
        - Do GA: [False]
        - Do subsumption: [True]
        - Do Action Planning: [False]
        - Beta: [0.05]
        - ...
        - Epsilon: [0.7]
        - U_max: [100000]
```

Experiments

[13]:
```
EXPLORE_TRIALS = 2000
EXPLOIT_TRIALS = 100


def perform_experiment(cfg, env):
    # explore phase
    agent = ACS2(cfg)
    population_explore, metrics_explore = agent.explore(env, EXPLORE_TRIALS)

    # exploit phase, reinitialize agent with population above
    agent = ACS2(cfg, population=population_explore)
    population_exploit, metrics_exploit = agent.exploit(env, EXPLOIT_TRIALS)

    return (population_explore, metrics_explore), (population_exploit, metrics_
↪exploit)
```

FrozenLake-v0 environment (baseline)

[14]:
```
%%time
explore_results, exploit_results = perform_experiment(cfg, fl_env)
```

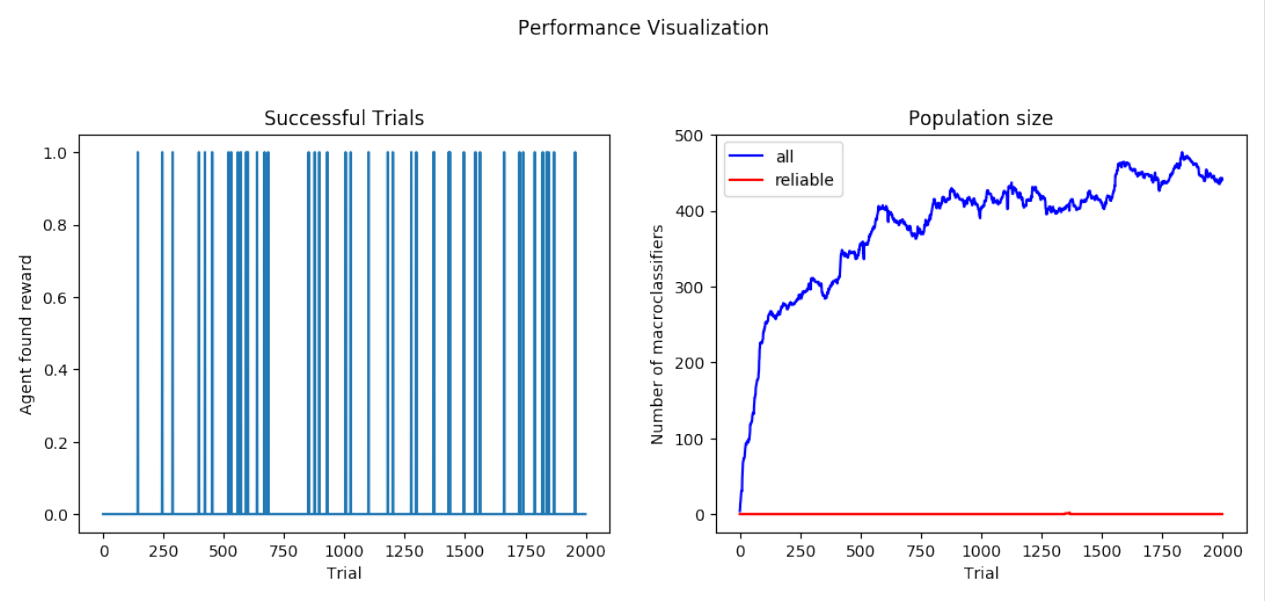
```
CPU times: user 46.1 s, sys: 138 ms, total: 46.2 s
Wall time: 46.3 s
```

Learn some behaviour during exploration phase

[15]:
```
# exploration
print_performance(explore_results[0], explore_results[1])
```

```
Number of classifiers: 441
Number of reliable classifiers: 0
Percentage of successul trials: 42.00%

Top 10 classifiers:
###############X0 1 ###############0X (0000000000000##)    q: 0.602 r: 0.3693 ir: 0.
↪2825 f: 0.2224 exp: 41  tga: 1105  talp: 15265 tav: 2.99e+02 num: 1    q: 0.60
↪    r: 0.37         ir: 0.28         exp: 41
#0#############X0 1 ###############0X (0#000000000000##)    q: 0.577 r: 0.3633 ir: 0.
↪2765 f: 0.2096 exp: 32  tga: 3961  talp: 15265 tav: 3.26e+02 num: 1    q: 0.58
↪    r: 0.36         ir: 0.28         exp: 32
#0############0X# 3 #############X0# (0#0000000000##0)    q: 0.471 r: 0.3723 ir: 0.
↪3615 f: 0.1753 exp: 43  tga: 3660  talp: 14982 tav: 2.96e+02 num: 1    q: 0.47
↪    r: 0.37         ir: 0.36         exp: 43
###############X0 3 ###############0X (0000000000000##)    q: 0.442 r: 0.3876 ir: 0.
↪377  f: 0.1715 exp: 45  tga: 2040  talp: 14982 tav: 3.13e+02 num: 1    q: 0.44
↪    r: 0.39         ir: 0.38         exp: 45
#############0X# 3 #############X0# (00000000000##0)    q: 0.446 r: 0.3829 ir: 0.
↪3723 f: 0.1709 exp: 45  tga: 3117  talp: 14982 tav: 2.98e+02 num: 1    q: 0.45
↪    r: 0.38         ir: 0.37         exp: 45
#############0X# 1 #############X0# (00000000000##0)    q: 0.452 r: 0.3676 ir: 0.
↪2807 f: 0.1661 exp: 33  tga: 3251  talp: 15265 tav: 3.33e+02 num: 1    q: 0.45
↪    r: 0.37         ir: 0.28         exp: 33
#0############0X# 1 #############X0# (0#000000000##0)    q: 0.451 r: 0.3633 ir: 0.
↪2765 f: 0.1639 exp: 33  tga: 3251  talp: 15265 tav: 3.26e+02 num: 1    q: 0.45
↪    r: 0.36         ir: 0.28         exp: 33
###############X# 1 ################ (0000000000000#0)    q: 0.368 r: 0.3676 ir: 0.
↪2807 f: 0.1351 exp: 40  tga: 1748  talp: 15265 tav: 2.93e+02 num: 1  q: 0.37
↪ r: 0.37         ir: 0.28        exp: 40
#0#############X# 1 ################ (0#000000000000#0)    q: 0.355 r: 0.3633 ir: 0.
↪2765 f: 0.129  exp: 36  tga: 3119  talp: 15265 tav: 2.97e+02 num: 1  q: 0.36
↪ r: 0.36         ir: 0.28        exp: 36
##########0###X# 2 ##########X###0# (0000000000#000#0)    q: 0.412 r: 0.2099 ir: 0.
↪1545 f: 0.0865 exp: 18  tga: 631   talp: 14255 tav: 7.17e+02 num: 1    q: 0.41
↪    r: 0.21         ir: 0.15         exp: 18
```

[16]:
```
plot_performance(explore_results[1])
```



Metrics from exploitation

[17]:
```
# exploitation
print_performance(exploit_results[0], exploit_results[1])
```

```
Number of classifiers: 441
Number of reliable classifiers: 0
Percentage of successul trials: 6.00%

Top 10 classifiers:
###############X0 1 ###############0X (0000000000000##)    q: 0.602 r: 0.3693 ir: 0.
↪2825 f: 0.2224 exp: 41  tga: 1105  talp: 15265 tav: 2.99e+02 num: 1    q: 0.60
↪    r: 0.37         ir: 0.28        exp: 41
#0#############X0 1 ###############0X (0#000000000000##)    q: 0.577 r: 0.3633 ir: 0.
↪2765 f: 0.2096 exp: 32  tga: 3961  talp: 15265 tav: 3.26e+02 num: 1    q: 0.58
↪    r: 0.36         ir: 0.28        exp: 32
#0############0X# 3 #############X0# (0#00000000000##0)    q: 0.471 r: 0.3723 ir: 0.
↪3615 f: 0.1753 exp: 43  tga: 3660  talp: 14982 tav: 2.96e+02 num: 1    q: 0.47
↪    r: 0.37         ir: 0.36        exp: 43
###############X0 3 ###############0X (0000000000000##)    q: 0.442 r: 0.3876 ir: 0.
↪377  f: 0.1715 exp: 45  tga: 2040  talp: 14982 tav: 3.13e+02 num: 1    q: 0.44
↪    r: 0.39         ir: 0.38        exp: 45
##############0X# 3 ##############X0# (000000000000##0)    q: 0.446 r: 0.3829 ir: 0.
↪3723 f: 0.1709 exp: 45  tga: 3117  talp: 14982 tav: 2.98e+02 num: 1    q: 0.45
↪    r: 0.38         ir: 0.37        exp: 45
##############0X# 1 ##############X0# (000000000000##0)    q: 0.452 r: 0.3676 ir: 0.
↪2807 f: 0.1661 exp: 33  tga: 3251  talp: 15265 tav: 3.33e+02 num: 1    q: 0.45
↪    r: 0.37         ir: 0.28        exp: 33
#0############0X# 1 ##############X0# (0#00000000000##0)    q: 0.451 r: 0.3633 ir: 0.
↪2765 f: 0.1639 exp: 33  tga: 3251  talp: 15265 tav: 3.26e+02 num: 1    q: 0.45
↪    r: 0.36         ir: 0.28        exp: 33
###############X# 1 ################ (000000000000000#0)    q: 0.368 r: 0.3676 ir: 0.
↪2807 f: 0.1351 exp: 40  tga: 1748  talp: 15265 tav: 2.93e+02 num: 1  q: 0.37
↪ r: 0.37         ir: 0.28        exp: 40
#0#############X# 1 ################ (0#000000000000#0)    q: 0.355 r: 0.3633 ir: 0.
↪2765 f: 0.129  exp: 36  tga: 3119  talp: 15265 tav: 2.97e+02 num: 1  q: 0.36
↪ r: 0.36         ir: 0.28        exp: 36
```

(continues on next page)

(continued from previous page)

```
##########0###X# 2 ##########X###0# (0000000000#000#0)    q: 0.412 r: 0.2099 ir: 0.
↪1545 f: 0.0865 exp: 18  tga: 631   talp: 14255 tav: 7.17e+02 num: 1    q: 0.41
↪   r: 0.21         ir: 0.15        exp: 18
```

FrozenLakeNotSlippery-v0 environment

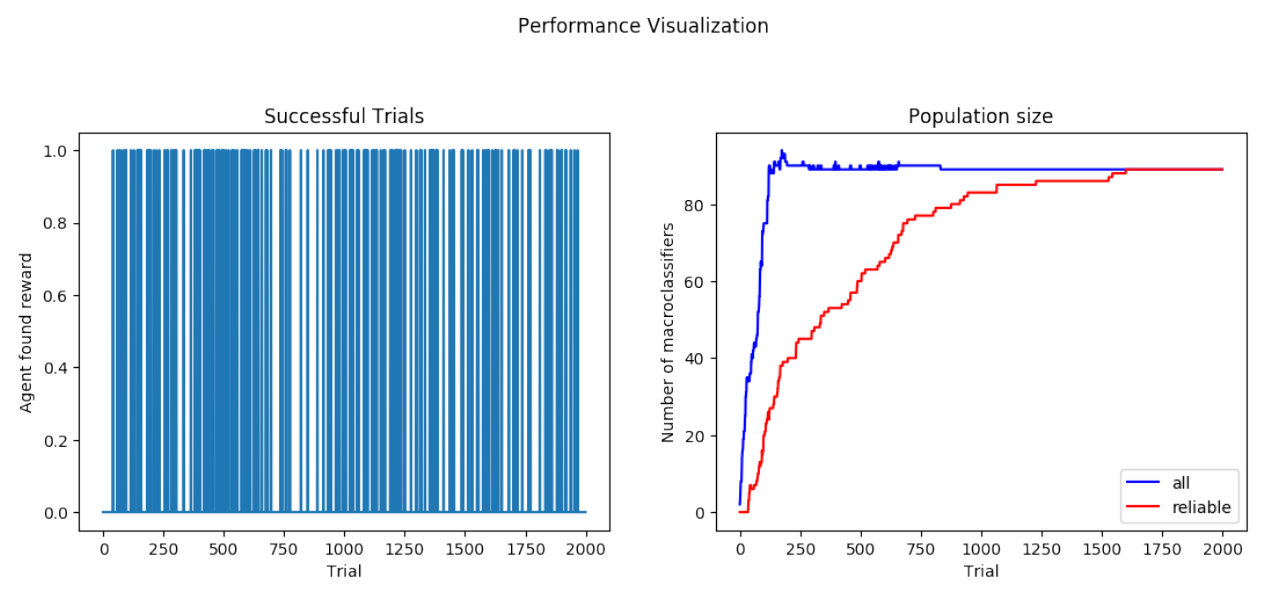
[18]:
```
%%time
explore_results_2, exploit_results_2 = perform_experiment(cfg, fl_ns_env)
```

```
CPU times: user 9.99 s, sys: 136 ms, total: 10.1 s
Wall time: 10.2 s
```

[19]:
```
# exploration
print_performance(explore_results_2[0], explore_results_2[1])
```

```
Number of classifiers: 89
Number of reliable classifiers: 89
Percentage of successul trials: 192.00%

Top 10 classifiers:
###############X0 2 ###############0X (empty)               q: 1.0   r: 1.0    ir: 1.0
↪   f: 1.0    exp: 191 tga: 237   talp: 14842 tav: 79.9   num: 1        q: 1.00
↪   r: 1.00         ir: 1.00        exp: 191
##########X###0# 1 ##########0###X# (empty)               q: 1.0   r: 0.95   ir: 0.0
↪   f: 0.95   exp: 245 tga: 324   talp: 15005 tav: 73.1   num: 1        q: 1.00
↪   r: 0.95         ir: 0.00        exp: 245
##############X0# 2 ##############0X# (empty)               q: 0.999 r: 0.9459 ir: 0.0
↪   f: 0.9453 exp: 130 tga: 236   talp: 14842 tav: 1.38e+02 num: 1      q: 1.00
↪   r: 0.95         ir: 0.00        exp: 130
###############X# 1 ################ (empty)               q: 0.997 r: 0.9274 ir: 0.0
↪   f: 0.9245 exp: 87  tga: 572   talp: 15006 tav: 2.14e+02 num: 1   q: 1.00
↪r: 0.93         ir: 0.00        exp: 87
#########X0##### 2 #########0X##### (empty)               q: 1.0   r: 0.9025 ir: 0.0
↪   f: 0.9025 exp: 399 tga: 117   talp: 15035 tav: 50.2   num: 1        q: 1.00
↪   r: 0.90         ir: 0.00        exp: 399
######X###0##### 1 ######0###X##### (empty)               q: 1.0   r: 0.8974 ir: 0.0
↪   f: 0.8971 exp: 137 tga: 26    talp: 14979 tav: 89.2   num: 1        q: 1.00
↪   r: 0.90         ir: 0.00        exp: 137
#########X###0## 1 #########0###X## (empty)               q: 1.0   r: 0.8892 ir: 0.0
↪   f: 0.8892 exp: 168 tga: 115   talp: 14994 tav: 93.6   num: 1        q: 1.00
↪   r: 0.89         ir: 0.00        exp: 168
####0#####0###X# 3 ##########X###0# (empty)               q: 0.997 r: 0.8828 ir: 0.0
↪   f: 0.8797 exp: 98  tga: 485   talp: 14543 tav: 1.62e+02 num: 1       q: 1.00
↪    r: 0.88         ir: 0.00        exp: 98
##########0###X# 3 ##########X###0# (empty)               q: 0.997 r: 0.8828 ir: 0.0
↪   f: 0.8797 exp: 97  tga: 485   talp: 14543 tav: 1.62e+02 num: 1      q: 1.00
↪   r: 0.88         ir: 0.00        exp: 97
########X0###### 2 ########0X###### (empty)               q: 1.0   r: 0.8573 ir: 0.0
↪   f: 0.8573 exp: 740 tga: 114   talp: 15034 tav: 28.7   num: 1        q: 1.00
↪   r: 0.86         ir: 0.00        exp: 740
```

[20]:
```
plot_performance(explore_results_2[1])
```



[21]:
```
# exploitation
print_performance(exploit_results_2[0], exploit_results_2[1])
```

```
Number of classifiers: 89
Number of reliable classifiers: 89
Percentage of successul trials: 100.00%

Top 10 classifiers:
###############X0 2 ##############0X (empty)               q: 1.0   r: 1.0    ir: 1.0
↪   f: 1.0    exp: 191 tga: 237   talp: 14842 tav: 79.9   num: 1        q: 1.00
↪   r: 1.00         ir: 1.00        exp: 191
##########X###0# 1 ##########0###X# (empty)               q: 1.0   r: 0.95   ir: 0.0
↪   f: 0.95   exp: 245 tga: 324   talp: 15005 tav: 73.1   num: 1        q: 1.00
↪   r: 0.95         ir: 0.00        exp: 245
##############X0# 2 ##############0X# (empty)               q: 0.999 r: 0.9459 ir: 0.0
↪   f: 0.9453 exp: 130 tga: 236   talp: 14842 tav: 1.38e+02 num: 1      q: 1.00
↪   r: 0.95         ir: 0.00        exp: 130
###############X# 1 ################ (empty)               q: 0.997 r: 0.9274 ir: 0.0
↪   f: 0.9245 exp: 87  tga: 572   talp: 15006 tav: 2.14e+02 num: 1   q: 1.00
↪r: 0.93         ir: 0.00        exp: 87
#########X0##### 2 #########0X##### (empty)               q: 1.0   r: 0.9025 ir: 0.0
↪   f: 0.9025 exp: 399 tga: 117   talp: 15035 tav: 50.2   num: 1        q: 1.00
↪   r: 0.90         ir: 0.00        exp: 399
######X###0##### 1 ######0###X##### (empty)               q: 1.0   r: 0.8974 ir: 0.0
↪   f: 0.8971 exp: 137 tga: 26    talp: 14979 tav: 89.2   num: 1        q: 1.00
↪   r: 0.90         ir: 0.00        exp: 137
#########X###0## 1 #########0###X## (empty)               q: 1.0   r: 0.8892 ir: 0.0
↪   f: 0.8892 exp: 168 tga: 115   talp: 14994 tav: 93.6   num: 1        q: 1.00
↪   r: 0.89         ir: 0.00        exp: 168
####0####0###X# 3 #########X###0# (empty)               q: 0.997 r: 0.8828 ir: 0.0
↪   f: 0.8797 exp: 98  tga: 485   talp: 14543 tav: 1.62e+02 num: 1       q: 1.00
↪    r: 0.88         ir: 0.00        exp: 98
##########0###X# 3 #########X###0# (empty)               q: 0.997 r: 0.8828 ir: 0.0
↪   f: 0.8797 exp: 97  tga: 485   talp: 14543 tav: 1.62e+02 num: 1      q: 1.00
↪   r: 0.88         ir: 0.00        exp: 97
########X0###### 2 ########0X###### (empty)               q: 1.0   r: 0.8573 ir: 0.0
↪   f: 0.8573 exp: 740 tga: 114   talp: 15034 tav: 28.7   num: 1        q: 1.00
↪   r: 0.86         ir: 0.00        exp: 740
```

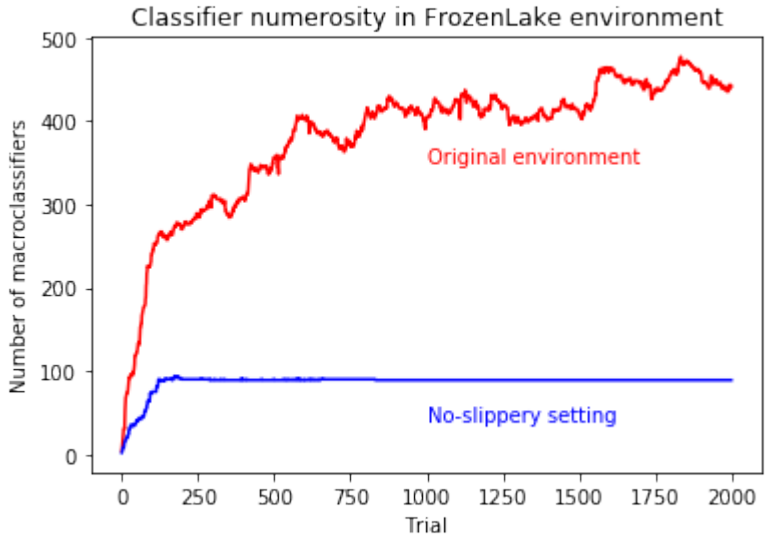
(continues on next page)

(continued from previous page)

Comparison

[22]:
```
def plot_population(metrics, ax=None):
    if ax is None:
        ax = plt.gca()

    trials = [m['trial'] for m in metrics]

    population_size = [m['numerosity'] for m in metrics]
    reliable_size = [m['reliable'] for m in metrics]

    ax.plot(trials, population_size, 'b', label='all')
    ax.plot(trials, reliable_size, 'r', label='reliable')

    ax.set_title("Population size")
    ax.set_xlabel("Trial")
    ax.set_ylabel("Number of macroclassifiers")
    ax.legend(loc='best')
```

[23]:
```
original = explore_results[1]
modified = explore_results_2[1]

ax = plt.gca()

trials = [m['trial'] for m in original]

original_numerosity = [m['numerosity'] for m in original]
modified_numerosity = [m['numerosity'] for m in modified]

ax.plot(trials, original_numerosity, 'r')
ax.text(1000, 350, "Original environment", color='r')

ax.plot(trials, modified_numerosity, 'b')
ax.text(1000, 40, 'No-slippery setting', color='b')


ax.set_title('Classifier numerosity in FrozenLake environment')
ax.set_xlabel('Trial')
ax.set_ylabel('Number of macroclassifiers')

plt.show()
```


Classifier numerosity in FrozenLake environment
Number of macroclassifiers
Trial
Original environment
No-slippery setting
0
100
200
300
400
500
0
250
500
750
1000
1250
1500
1750
2000