

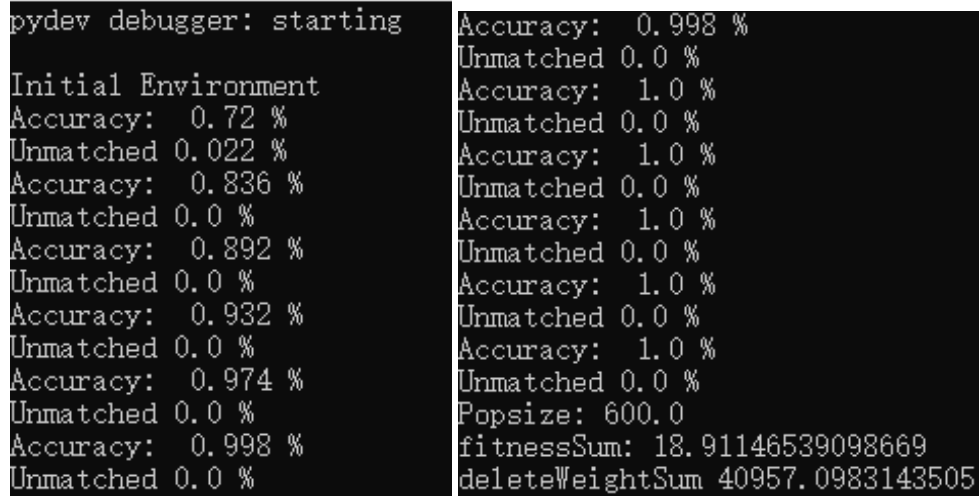
# Instruction

## 1 Abstract

The attached source code provides an XCS, a set of rule compaction algorithms and three visualization techniques. The code is based on Python 3.7, and Visual Studio 2017 is the recommended platform.

## 2 Train an XCS

Run the XCS.py, and when the processing completed, you can find the result in the "Result" file (the result is shown in Fig. 1, which is for the 6-bits Multiplexer problem).



```
pydev debugger: starting
Initial Environment
Accuracy: 0.72 %
Unmatched 0.022 %
Accuracy: 0.836 %
Unmatched 0.0 %
Accuracy: 0.892 %
Unmatched 0.0 %
Accuracy: 0.932 %
Unmatched 0.0 %
Accuracy: 0.974 %
Unmatched 0.0 %
Accuracy: 0.998 %
Unmatched 0.0 %
Accuracy: 0.998 %
Unmatched 0.0 %
Accuracy: 1.0 %
Unmatched 0.0 %
Accuracy: 1.0 %
Unmatched 0.0 %
Accuracy: 1.0 %
Unmatched 0.0 %
Accuracy: 1.0 %
Unmatched 0.0 %
Popsiz: 600.0
fitnessSum: 18.91146539098669
deleteWeightSum 40957.0983143505
```

Figure 1: results of running the XCS.py

### 2.1 Parameter Settings

Regarding training, all the parameters are needed to be set are in the XCS.py's `__init__` function. The provided XCS supports three boolean problems and four UCI problems.

Regarding boolean domains, you need set the `environment.type='b'`, then set the `ProblemId` with one of [0,1,2], here, 0= "Multiplexer", 1= "Carry", and 2= "Majority-On". Change the value of `AttributeSize` can set the length of problem, e.g., `environment.type='b'`, `ProblemId=0`, and `AttributeSize=6` means training a 6-bits Multiplexer problem.

Regarding UCI problems, you need set the `environment.type='r'`, then set the problem with one of [0,1,2,3], here, 0="Mushroom", 1="MushroomFull", 2="ZOO", and 3= "German". Note

"MushroomFull" contains the instances that have missing values, but "Mushroom" does not have missing value instances.

Note, Config.StandardXCS.py's standard\_XCS\_config class allows you to change some basic parameters of XCS. You can also write your own config class and set them in the XCS.py's `__init__` function (shown in Fig. 2) to allow the XCS to support different configs.

```
#load the setting for this XCS
self.config=standard_XCS_config()
```

Figure 2: Change here for supporting different XCS config document

The parameter iterations identifies the number of iterations that will be executed for training. Note the training process will not stop until the given iteration number is achieved.

```
#training iterations
self.iterations=6000

#how many iterations need to be tested
self.testRate=500

#initial the population pool
self.PopulationPool=self.operators.Initial_Population(self.env.AttributeNumber,self.env.NumberClasses)

#maximum number of population pool
self.MaxPop=600
```

Figure 3: Parameters for training

The parameter testRate identifies when to out print the training performance, e.g., if testRate=500, this means, after each 500 iterations, the system will output current training accuracy.

The parameter MaxPop identifies the maximum number of rules in the population (shown in Fig. 3).

## 2.2 Result

```
0 1 0 0 1 : 0 Numerosity: 3 Fitness: 0.16 Prediction: 1000.0 PredictionError: 0.03 Experience: 44.0 Accuracy: 1.0 ActionSize: 30.49 TimeStamp: 5747 DeleteWeight: 91.46
1 0 1 0 0 # : 0 Numerosity: 4 Fitness: 0.15 Prediction: 1000.0 PredictionError: 0.0 Experience: 98.0 Accuracy: 1.0 ActionSize: 38.46 TimeStamp: 5974 DeleteWeight: 153.86
0 1 0 1 # 0 : 0 Numerosity: 6 Fitness: 0.16 Prediction: 0.0 PredictionError: 0.0 Experience: 86.0 Accuracy: 1.0 ActionSize: 38.17 TimeStamp: 5914 DeleteWeight: 229.02
1 0 # 0 0 0 : 0 Numerosity: 1 Fitness: 0.03 Prediction: 1000.0 PredictionError: 0.0 Experience: 86.0 Accuracy: 1.0 ActionSize: 50.88 TimeStamp: 5974 DeleteWeight: 50.88
```

Figure 4: Trained result

When training is completed, you will get a trained population in your Result file, the name of your result is expected to be complete time \_ problem name. The result's format is a set of rules with the format of condition : action: Numerosity: value Fitness: value Prediction: value PredictionError: value Experience: value Accuracy: value ActionSize: value TimeStamp: value DeleteWeight: value (shown in Fig. 4).

## 3 Compaction

Run the CompactAgent.p. You need to give the system an address about your models' location, note these models are required to address the same problem. You also need to give an XCS.Operators, Environment, RepresentationSymbolistTernary, and standard\_XCS.config instances to the system, note the settings of these instances are expected to be the same of your

```

address="Compact\\6MUX"
ImplementList=[1, 2, 3, 4, 5, 6, 7, 8, 9]
ope=XCS_Operators()
config=standard_XCS_config()
env=Environment(0, 6, 1000, 'b')
rep=RepresentationSymbolistTernary(env, env, config)
CA=Compact_Agent_R(ope, env, env, rep, address, ImplementList)

```

Figure 5: This sample applying the CRA, CRA2, FU1, FU3, K1, PDRC, QRC, RCR, RCR2, and RCR3 to the 6-bits Multiplexer problem, you will find the result at the RCompacted file

training in the XCS.py. Lastly, you need set an implementList. ImplementList is the list inform which compaction algorithms will be invoked, i.e. [0:CRA, 1:CRA2, 2:FU1, 3:FU3 4:K1, 5:PDRC 6:QRC 7:RCR 8:RCR2 9:RCR3], for example, give a list [0,1,2], the system will execute the CRA, CRA2 and Fu1 for your models (shown in Fig. 5).

The results are expected with the format of condition: class prediction. As shown in Fig. 6.

```

1 1 # # # 1 : 1 1000.0
0 0 0 # # # : 0 1000.0
0 0 1 # # # : 1 1000.0
1 0 # # 1 # : 1 1000.0
1 1 # # # 0 : 0 1000.0

```

Figure 6: This sample applying the CRA, CRA2, FU1, FU3, K1, PDRC, QRC, RCR, RCR2, and RCR3 to the 6-bits Multiplexer problem, you will find the result at the RCompacted file

## 4 Visualization

```

title="6-bits Multiplexer"
env=Environment(0, 6, 1000, 'b')
Address="RCompacted\\19_38_45_6MultiplexerRCR.txt"
operator=Compacted_Model()

#R_AFIM=AFIM(Address, operator)
R_AFVM=AFVM(Address, operator)
#R_FIM=FIM(Address, operator)
png_Path="V_Result"
V=Visualization(png_Path)
print(R_AFVM.AFVM_Data)
print(R_AFVM.ActionList)
#V.DrewHeatMapSimple2D(title, R_FIM.FIM_Data)
V.Drew_MultiAction3D_Boolean(title, R_AFVM.AFVM_Data, R_AFVM.ActionList, 360)

```

Figure 7: This sample applying the Visualization technique (VFVM) to a compacted 6-bits Multiplexer problem, you will find the image at the VResult file. Note, instance of Environment need to be the same of training at XCS, operator must be an Compacted\_Model class's instance.

run the visualization.py, shown in Fig. 7. address is the location of where you store the target agent, note the agent is expected to be the products of the provided compaction algorithms.

You can change the AFVM instance to FIM instance or AFIM instance to obtain visualization results of FIM or AFIM (shown in Fig. 8). The expected visualization results are shown in Fig. 9.

```
R_AFIM=AFIM(Address,operator)
V.Drew_MultiAction3D_Boolean(title,R_AFIM.AFIM_Data,R_AFIM.ActionList,360)

R_FIM=FIM(Address,operator)
V.Drew_Simple3D_Boolean(title,R_FIM.FIM_Data,360)
```

Figure 8: To get the visualization results of FIM or AFIM.

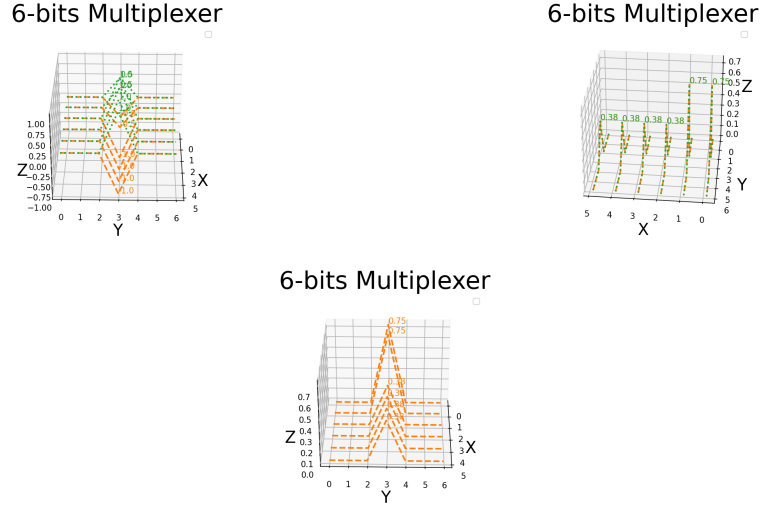


Figure 9: Results of AFVM, AFIM, FIM.

## 5 Extension

regarding extend the environment, for boolean domains, extend the BooleanEnvironment.py. In the `__init__` function, extend the problemName with your new problem. extend the RandomSample. regarding real domains, extend the RealEnvironment.py. In the `__init__` function, extend the problemName extend the Read function.