TDT4137 - Exercise 4

Håkon Ødegård Løvdal

October 2015

Task a)

Step 1: Fuzzification

x_1 : distance and x_2 : delta

$\mu(x = \text{Perfect}) = 0.1$
$\mu(x = \text{Small}) = 0.6$

$\mu(x = \text{Stable}) = 0.3$
$\mu(x = \text{Growing}) = 0.4$

Step 2: Rule evaluation

OR = max, AND = min

```
IF distance is Small(0.6)
AND delta is Growing(0.4)
THEN action is None(0.4)
```

```
IF distance is Small(0.6)
AND delta is Stable(0.3)
THEN action is SlowDown(0.3)
```

```
IF distance is Perfect(0.1)
AND delta is Growing(0.4)
THEN action is SpeedUp(0.1)
```

```
IF distance is VeryBig(0.0)
AND (
    delta is NOT Growing(1- 0.4)(0.6)
    OR delta is NOT GrowingFast(1-0)(0)
    )
THEN action is FloorIt(0..0)
```

```
IF distance is VerySmall(0.0)
THEN action is BrakeHard(0.0)
```

Step 3: Aggregation of the rule outputs

- z is BrakeHard(0.0)
- z is SlowDown(0.3)
- z is None(0.4)
- z is SpeedUp(0.1)
- z is FloorIt(0.0)

Step 4: Defuzzification

$$
\begin{aligned}
COG &= \frac{\sum_{x=a}^{b} \mu_A(x)x}{\sum_{x=a}^{b} \mu_A(x)} \\
&= \frac{0.0\times(-10+-9+-8+-7)+0.3\times(-6+-5+-4+-3+-2)+0.4\times(-1+0+1+2)+0.1\times(3+4+5+6)+0.0\times(7+8+9+10)}{(0.0\times4)+(0.3\times5)+(0.1\times4)+(0.0\times4)} \\
&= \frac{-3.4}{3.5} \\
&= -0.9714
\end{aligned}
$$

Action is None

Task b)

Example output:

```
The robot chooses ANone with value -0.25
```

```python
from interface import *
from functools import partial as apply
from operator import itemgetter
import collections


class MamdaniReasoner(FuzzyReasoner):
    """
    A class object representing a Mamdani-reasoner for Fuzzy
    Reasoning
    """
    def __init__(self, *args, **kwargs):
        self.limits = args
        self.fuzzy_sets = kwargs
        self.range = range(-10, 11, 1)  # -10 to 10 with step 1
        self.rules = {k: None for k in kwargs['action_set'].keys()}
        self.rule_evaluation(*args)

    def rule_evaluation(self, crisp_x1, crisp_x2):
        dist = self.fuzzy_sets['distance_set']
        delta = self.fuzzy_sets['delta_set']
        rules = self.rules
```

```
23          rules['ANone'] = self.AND(dist['Small'](crisp_x1), delta['
    Growing'](crisp_x2))  # Rule 1 None
            rules['SlowDown'] = self.AND(dist['Small'](crisp_x1), delta
    ['Stable'](crisp_x2))  # Rule 2 SlowDown
25          rules['SpeedUp'] = self.AND(dist['Perfect'](crisp_x1),
    delta['Growing'](crisp_x2))  # Rule 3 SpeedUp
            rules['FloorIt'] = self.AND(dist['VeryBig'](crisp_x1), self
    .OR(
27              self.NOT(delta['Growing'](crisp_x1)), self.NOT(delta['
    GrowingFast'](crisp_x2))))  # Rule 4 FloorIt
            rules['BrakeHard'] = dist['VerySmall'](crisp_x1)
29
    def defuzzification(self):
31          action_set = self.fuzzy_sets['action_set']

33          upper_value, lower_value = 0, 0

35          for index in self.range:
                value, action, r = 0.0, None, index
37              for rule_function, aggregate_value in zip(action_set.
    keys(), self.rules.values()):
                    new_value = action_set[rule_function](index)
39                  #  print("Value for action: %s is %f" % (
    rule_function, new_value))
                    if new_value > value:
41                      action = rule_function
                        value = new_value
43                      r = index

45              #  print("Highest for (%d): VALUE: %f, ACTION: %s" % (r
    , value, action))
                upper_value += (r * self.rules[action])
47              lower_value += self.rules[action]

49          cog_value = upper_value / lower_value
            return self.get_action_name(cog_value), cog_value
51
    def get_action_name(self, value):
53          action_val = [(k, v(value)) for k, v in self.fuzzy_sets['
    action_set'].items()]
            return max(action_val, key=itemgetter(1))[0]
55
    @staticmethod
57  def AND(x, y):
        return min(x, y)
59
    @staticmethod
61  def OR(x, y):
        return max(x, y)
63
    @staticmethod
65  def NOT(x):
        return 1. - x
67

69 if __name__ == '__main__':
```

```
    # The following code is partially step 1
    # Create distance set
    distance_set = {
        'VerySmall': apply(MamdaniReasoner.reverse_grade, 1., 2.),
        'Small': apply(MamdaniReasoner.triangle, 1.5, 3., 4.5),
        'Perfect': apply(MamdaniReasoner.triangle, 3.5, 5., 6.5),
        'Big': apply(MamdaniReasoner.triangle, 5.5, 7., 8.5),
        'VeryBig': apply(MamdaniReasoner.grade, 7.5, 9.)
    }

    # Create delta set
    delta_set = {
        'ShrinkingFast': apply(MamdaniReasoner.reverse_grade, -4.,
    -2.5),
        'Shrinking': apply(MamdaniReasoner.triangle, -3.5, -2.,
    -.5),
        'Stable': apply(MamdaniReasoner.triangle, -1.5, 0., 1.5),
        'Growing': apply(MamdaniReasoner.triangle, .5, 2., 3.5),
        'GrowingFast': apply(MamdaniReasoner.grade, 2.5, 4.)
    }

    # Create action set
    actions = {
        'BrakeHard': apply(MamdaniReasoner.reverse_grade, -8., -5.)
    ,
        'SlowDown': apply(MamdaniReasoner.triangle, -7., -4., -1.),
        'ANone': apply(MamdaniReasoner.triangle, -3., 0., 3.),
        'SpeedUp': apply(MamdaniReasoner.triangle, 1., 4., 7.),
        'FloorIt': apply(MamdaniReasoner.grade, 5., 8.)
    }
    actions = collections.OrderedDict(sorted(actions.items()))

    mr = MamdaniReasoner(3.4, 1.4, distance_set=distance_set,
    delta_set=delta_set, action_set=actions)
    action_tuple = mr.defuzzification()
    print("The robot chooses %s with value %.2f" % action_tuple)
```

Listing 1: "mamdani.py"

```

class FuzzyReasoner(object):
    """
    The FuzzyReasoner implements the interface methods that a Fuzzy
     reasoner
    must implement
    """
    @property
    def AND(x, y):
        raise NotImplementedError

    @property
    def OR(x, y):
        raise NotImplementedError

    @property
    def NOT(x):
```

```
        raise NotImplementedError

    @staticmethod
    def triangle(x0, x1, x2, position=None, clip=1.):
        value = 0.0
        if x0 <= position <= x1:
            value = (position - x0) / (x1 - x0)
        elif x1 <= position <= x2:
            value = (x2 - position) / (x1 - x0)
        if value > clip:
            value = clip
        return value

    @staticmethod
    def grade(x0, x1, position=None, clip=1.):
        if position >= x1:
            value = 1.0
        elif position <= x0:
            value = 0.0
        else:
            value = (position - x0) / (x1 - x0)
        if value > clip:
            value = clip
        return value

    @staticmethod
    def reverse_grade(x0, x1, position=None, clip=1.):
        if position <= x0:
            value = 1.0
        elif position >= x1:
            value = 0.0
        else:
            value = (x1 - position) / (x1 - x0)
        if value > clip:
            value = clip
        return value
```

Listing 2: "interface.py"