Assignment #4

Steven Porretta
Student # 100756494

Code can be found at `https://github.com/0xSteve/learning_automata_simulator`

1 Question 1

In this section we will examine some code snippets from the first question.

Listing 1: Testbench code for the Tsetlin.

```python
    c2 = 0.7
    c1 = 0.05
    for i in range(0, 7):
        print("c1 = " + str(c1) + ", c2 = " + str(c2) + ", N = 13.")
        a = la.Tsetlin(13, 2, [c1, c2])
        a.simulate(50, 30001)
        b = ala.Tsetlin.stationary_probability_analytic([c1, c2], 13)
        c = ala.Tsetlin.number_of_states_estimate([c1, c2])
        print("Tsetlin P1(infinity) = " + str(b) + "(Analytic)")
        print("Tsetlin P1(infinity) = " + str(a.action_average[0]) + "(Simulated)")
        print("Tsetlin # of states required = " + str(c) + "(Estimate)")
        c1 += 0.1
        c1 = round(c1, 2)
```

This excerpt of code generates the entire quantity of required code for this question. As can be seen in the following code-snippet.

Listing 2: Testbench output.

```
c1 = 0.05, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.9999999890725503(Analytic)
Tsetlin P1(infinity) = 1.0(Simulated)
Tsetlin # of states required = 3(Estimate)
c1 = 0.15, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.9999778874501014(Analytic)
Tsetlin P1(infinity) = 1.0(Simulated)
Tsetlin # of states required = 4(Estimate)
c1 = 0.25, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.9990142374252533(Analytic)
Tsetlin P1(infinity) = 0.999998000067(Simulated)
Tsetlin # of states required = 6(Estimate)
c1 = 0.35, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.9865794150962881(Analytic)
Tsetlin P1(infinity) = 0.999797340089(Simulated)
Tsetlin # of states required = 9(Estimate)
c1 = 0.45, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.9151468144874294(Analytic)
Tsetlin P1(infinity) = 0.980783973868(Simulated)
Tsetlin # of states required = 18(Estimate)
c1 = 0.55, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.7453193640776348(Analytic)
Tsetlin P1(infinity) = 0.786514449518(Simulated)
Tsetlin # of states required = 0(Estimate)
c1 = 0.65, c2 = 0.7, N = 13.
Tsetlin P1(infinity) = 0.5680008401435711(Analytic)
Tsetlin P1(infinity) = 0.570874970834(Simulated)
Tsetlin # of states required = 0(Estimate)
```

Now that it is seen working as one would expect, considering rounding errors from python 3.6, it is time to take a look at the useful snippets of code governing the functionality of the Tsetlin, and Krylov automata.

Listing 3: Tsetlin core code.

```
     def next_state_on_reward(self):
        '''Find the next state of the learner, given that the teacher
           rewarded.'''
        if (self.current_state mod (self.N / self.R) != 1):
            self.current_state -= 1

    def next_state_on_penalty(self):
        '''Find the next state of the learner, given that the teacher
           penalized.'''
        if(self.current_state mod (self.N / self.R) != 0):
            self.current_state += 1
        elif(self.current_state mod (self.N / self.R) == 0):
            # Don't really add states, just cycle through N, 2N, 4N, etc.
            if(self.current_state != self.N):
                a = (self.N / self.R) mod self.N
                self.current_state = a + self.current_state
            else:
                self.current_state = self.N / self.R

    # Determine the next state as the teacher.
    def environment_response(self):
        '''Determine the next state of the learner from the perspective
        of the teacher.'''
        response = uniform(0, 1)
        penalty_index = 1
        if(self.current_state <= self.n):
            self.actions[0] += 1
            penalty_index = 0
        else:
            self.actions[1] += 1

        if(response > self.c[penalty_index]):
            # Reward.
            self.next_state_on_reward()
        else:
            # Penalty.
            self.next_state_on_penalty()
```

The above is the core code of the Tsetlin machine, governing state translations and action choices. Essentially, whenever it is in the range 1 to N, it chooses action α_1 and α_2 otherwise. This code is essentially the same for the Krylov machine, which we will see in the next section.

2 Question 2

Listing 4: testbench for the Krylov 2-action.

```
    for i in range(0, 7):
    print("c1 = " + str(c1) + ", c2 = " + str(c2) + ", N = 13.")
    a = la.Tsetlin(13, 2, [c1/2, c2/2])
    a.simulate(50, 30001)
    b = ala.Tsetlin.stationary_probability_analytic([c1, c2], 13)
    c = ala.Tsetlin.number_of_states_estimate([c1, c2])
    d = la.Krylov(13, 2, [c1, c2])
    d.simulate(10, 50000)
    e = ala.Tsetlin.stationary_probability_analytic([c1, c2], 13)
    f = ala.Tsetlin.number_of_states_estimate([c1, c2])
    print("Tsetlin P1(infinity) = " + str(b) + "(Analytic)")
    print("Tsetlin P1(infinity) = " + str(a.action_average[0]) + "(Simulated)")
    print("Tsetlin # of states required = " + str(c) + "(Estimate)")
    print("Krylov P1(infinity) = " + str(e) + "(Analytic)")
    print("Krylov P1(infinity) = " + str(d.action_average[0]) + "(Simulated)")
    print("Krylov # of states required = " + str(f) + "(Estimate)")
    c1 += 0.1
    c1 = round(c1, 2)
```

As can be seen from the code, this test bench looks very similar to the test bench of question 1, however, note the c vector for the Tsetlin automaton is now $c_1/2$, $c_2/2$. As is expected, both automata behave in the same manner. as can be seen from the output code snippet.

Listing 5: testbench output for the Krylov 2-action.

```
    c1 = 0.05, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.9999999890725503(Analytic)
    Tsetlin P1(infinity) = 1.0(Simulated)
    Tsetlin # of states required = 3(Estimate)
    Krylov P1(infinity) = 0.9999999890725503(Analytic)
    Krylov P1(infinity) = 1.0(Simulated)
    Krylov # of states required = 3(Estimate)
    c1 = 0.15, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.9999778874501014(Analytic)
    Tsetlin P1(infinity) = 0.999999333356(Simulated)
    Tsetlin # of states required = 4(Estimate)
    Krylov P1(infinity) = 0.9999778874501014(Analytic)
    Krylov P1(infinity) = 1.0(Simulated)
    Krylov # of states required = 4(Estimate)
    c1 = 0.25, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.9990142374252533(Analytic)
    Tsetlin P1(infinity) = 1.0(Simulated)
    Tsetlin # of states required = 6(Estimate)
    Krylov P1(infinity) = 0.9990142374252533(Analytic)
    Krylov P1(infinity) = 1.0(Simulated)
    Krylov # of states required = 6(Estimate)
    c1 = 0.35, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.9865794150962881(Analytic)
    Tsetlin P1(infinity) = 1.0(Simulated)
    Tsetlin # of states required = 9(Estimate)
    Krylov P1(infinity) = 0.9865794150962881(Analytic)
    Krylov P1(infinity) = 1.0(Simulated)
    Krylov # of states required = 9(Estimate)
    c1 = 0.45, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.9151468144874294(Analytic)
    Tsetlin P1(infinity) = 1.0(Simulated)
    Tsetlin # of states required = 18(Estimate)
    Krylov P1(infinity) = 0.9151468144874294(Analytic)
    Krylov P1(infinity) = 1.0(Simulated)
    Krylov # of states required = 18(Estimate)
    c1 = 0.55, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.7453193640776348(Analytic)
    Tsetlin P1(infinity) = 0.983473884204(Simulated)
    Tsetlin # of states required = 0(Estimate)
    Krylov P1(infinity) = 0.7453193640776348(Analytic)
    Krylov P1(infinity) = 0.999998(Simulated)
    Krylov # of states required = 0(Estimate)
    c1 = 0.65, c2 = 0.7, N = 13.
    Tsetlin P1(infinity) = 0.5680008401435711(Analytic)
    Tsetlin P1(infinity) = 0.670734975501(Simulated)
    Tsetlin # of states required = 0(Estimate)
    Krylov P1(infinity) = 0.5680008401435711(Analytic)
    Krylov P1(infinity) = 0.870114(Simulated)
    Krylov # of states required = 0(Estimate)
```

Observing the code for the Krylov machine, one notices that most of the code is inherited from the Tsetlin machine, except the state translations. It is incredible, that the only major distinction is that a penalty is treated as a penalty with 50% probability and a success otherwise. Literally, all other code for the Krylov machine is inherited from the Tsetlin.

Listing 6: Krylov core code.

```python
def next_state_on_penalty(self):
        '''Find the next state of the learner, given that the teacher
           penalized.'''

        # If this number is greater than 0.5, then penalize the learner.
        is_penalty = uniform(0, 1)

        if(is_penalty >= 0.5):
            Tsetlin.next_state_on_penalty(self)
        else:
            Tsetlin.next_state_on_reward(self)
```

3 Question 3

Since the code snippets for the L_{R-I} automaton do not shed any insight into the operation of the machine, the test bench has been omitted from within, however it is available on github. This automaton was quite challenging.

First let us consider state changes in the L_{R-I}, since there really are no states, but instead just an interval, $\{0, 1\}$, of possibilities. Consider the following code-snippet.

Listing 7: State Translation in the L_{R-I} automaton.

```python
 def next_state_on_penalty(self):
        '''Do nothing, other than pick a action.'''
        self.last_action = self.action_index()

    def next_state_on_reward(self):
        '''increase probabilities by a factor of k.'''
        # so if action 1 is chosen increase by k*p1,
        # otherwise increase p1 by (1-k)p2.
        self.last_action = self.action_index()
        # print("the next action is " + str(self.last_action))
        if(self.last_action == 2):
            # Increase by kp1
            self.p[0] = self.p[0] * self.k
            self.p[1] = 1 - self.p[0]
        else:
            # increase by (1-k)p2
            self.p[0] = 1 - self.p[1] * self.k
            self.p[1] = 1 - self.p[0]
```

It can be seen from the code-snippet, above, that on penalty the probabilities are not updated. Only an action is updated, and the machine continues. The action is governed by the cumulative distribution function of the action probability vector, as follows:

Listing 8: Action selection in the L_{R-I} automaton.

```python
def find_action_distribution(self):
        action_distribution = []
        sigma = 0
        for i in range(len(self.p)):
            sigma += self.p[i]
            action_distribution.append(sigma)
        return action_distribution

    def action_index(self):
        is_action = uniform(0, 1)
        action_distribution = self.find_action_distribution()
        for i in range(len(action_distribution)):
            if(is_action < action_distribution[i]):
                if(i == 0):
                    self.act1 += 1
                    return 1
        self.act2 += 1
        return 2
```

Observing this code, it is clear that the state translation occurs in a manner which is desirable. The potential for error, stems from the simulation function, or the binary search implementation. It is important that we use the cumulative distribution to prevent biasing towards action, α_1, when we initialize the automaton with $P(0) = [0.5, 0.5]$, as seen from the above code-snippet.

Listing 9: Simulating the L_{R-I} automaton.

```
ef simulate(self, ensemble_size):
        '''Assume that the depth of the automaton is determined by k >= n
           depth of the automaton.'''
        for i in range(ensemble_size):
            self.p = [0.5, 0.5]
            self.last_action = self.action_index()
            # n = 0
            while(max(self.p) < 0.9):
                self.environment_response()
                self.n += 1
                # print(self.p)
                # print(self.last_action)
                # if(n == 10000):
                #     break
                # n += 1
            self.action_average = self.act1 / (self.act1 + self.act2)
```

It is possible that the difficulty with providing the appropriate $P_1(\infty)$ stems from this function, but stepping through this appears that the automaton correctly counts the average action. Note that the action counting is conducted at the moment the action is chosen in the *action_index*() function.

This leads to some perplexing results, that I believe are due to using objects in python 3.6. To avoid confusion the updates will be forked to the branched **newLRI** so that it does not confuse the grader, or appear to be a trick. The git log should show the updates as well, and will only merge once the corrections are in place, for the sake of my github portfolio and solving an interesting problem. The following is the current output for question 3:

Listing 10: Testing the L_{R-I} automaton.

```
c1 = 0.05 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 103172
c1 = 0.15 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 105609
c1 = 0.25 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 125542
c1 = 0.35 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 134469
c1 = 0.45 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 142272
c1 = 0.55 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 168961
c1 = 0.65 c2 = 0.7
Best lambda approaches: 0.9
Mean time to converge in steps = 195115
```

From the above, we see some good, and some bad. Of course, the L_{R-I} automaton should never produce a static value for λ_R. It is possible that the binary search is incorrect, but it is such a simple algorithm it is likely that there is average action negotiation errors due to a lack of encapsulation in python. It would be reasonable if the value of λ_R would decrease as c_1 approaches c_2, much in the same way that it is reasonable to see that the mean discrete time increases as c_1 approaches c_2.

4 Concluding Remarks

The Krylov and Tsetlin automata work as expected within the parameters of reasonable rounding error set out by languages like python. In the worst case, we can observe outlier data at times when running that correspond to a percent difference of $< 2\%$, which is to be expected of a language like python which is known for rounding issues with iterative arithmetic. From their output it can be observed that the Krylov and Tsetlin automata operate correctly.

However, this is not necessarily the case for the Linear reward inaction scheme. After analyzing the state translations for the L_{R-I} scheme it appears that the error is with the implementation of the counter responsible for counting $P_1(\infty)$ and possibly the search function.