STAT6011/7611/6111/3317
COMPUTATIONAL STATISTICS (2016 Fall)
Assignment 4

Kei Ikegami (u3535947)

November 22, 2016

1

The code is below. I denote the 'Fair' by 0 and 'Loaded' by 1 in this code.

Listing 1: problem 1

```
1  import numpy as np
2
3  def vitervi(transition):
4      tosses = "25132634421246336665655356145665236655613263456214432352132634461435421"
5      prob_loaded = {"1": 1/12, "2": 1/12, "3": 1/12, "4": 1/12, "5":1/3, "6":1/3}
6      prob_fair = {"1":1/6, "2":1/6, "3":1/6, "4":1/6, "5":1/6, "6":1/6}
7      score_loaded = np.ones(len(tosses))
8      score_fair = np.ones(len(tosses))
9      score_loaded[0] = 0.5 * prob_loaded[tosses[0]]
10     score_fair[0] = 0.5 * prob_fair[tosses[0]]
11
12     for i in range(1,len(tosses)):
13         score_loaded[i] = max(score_loaded[i−1] * transition[1,1], score_fair[i−1] * transition[1,0]) * prob_loaded[tosses[i
                ]]
14         score_fair[i] = max(score_loaded[i−1] * transition[0,1], score_fair[i−1] * transition[0,0]) * prob_fair[tosses[i]]
15
16     state_estimation = np.ones(len(tosses))
17     for n, j in enumerate(score_loaded − score_fair):
18         if j < 0:
19             state_estimation[n] = 0
20
21     return(state_estimation)
22
23  transition1 = np.array([[0.8, 0.3], [0.2, 0.7]])
24  transition2 = np.array([[0.9, 0.15], [0.1, 0.85]])
25  transition3 = np.array([[0.95, 0.05], [0.6, 0.4]])
26  transition4 = np.array([[0.5, 0.5], [0.5, 0.5]])
27
28  vitervi(transition1)
29  vitervi(transition2)
30  vitervi(transition3)
31  vitervi(transition4)
```

And the results are as follows.

(a) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

(b) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

(c) [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

(d) [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

2

```
1  import numpy as np
2  from scipy.special import eval_hermite
3  from scipy.special import hermite
4  from scipy.special import h_roots
5  from scipy.special import he_roots
6  from scipy.stats import norm
7
8  def GH(m):
9      points = h_roots(m)[0]
10     weights = h_roots(m)[1]
11     return sum([weights[i] * np.exp(points[i]**2) * norm.pdf(points[i]) for i in range(len(points))])
12
13 def GH_weights(m):
14     points = h_roots(m)[0]
15     weights = h_roots(m)[1]
16     return [weights[i] * np.exp(points[i]**2) for i in range(len(points))]
17
18 a = [5, 10, 20, 30]
19
20 # approximation result
21 for m in a:
22     print(GH(m))
23
24 # weights
25 for m in a:
26     print(GH_weights(m))
27
28 # points
29 for m in a:
30     print(h_roots(m)[0])
```

In the below question, I mean the weights by the values multiplied by weight functions.

2.1

I prove the integral of standard normal probability distribution function is equal to 1.

2.2

I use the above code to compute the approximation of the integral.

When the number of nodes is 5,
the approximation result is 0.99684628222456162,
the nodes are [-2.02018287, -0.95857246, 0. , 0.95857246, 2.02018287],
the weights are [1.1814886255359833, 0.98658099675142807, 0.94530872048294179, 0.98658099675142807, 1.1814886255359833]

When the number of nodes is 10,
the approximation result is 0.99998763906433263
the nodes are [-3.43615912, -2.53273167, -1.75668365, -1.03661083, -0.34290133, 0.34290133, 1.03661083, 1.75668365, 2.53273167, 3.43615912],
and the weights are [1.0254516913657519, 0.82066612640481784, 0.74144193194356545, 0.70329632310490586, 0.6870818539512733, 0.6870818539512733, 0.70329632310490586, 0.74144193194356545, 0.82066612640481784, 1.0254516913657519]

When the number of nodes is 20,
the approximation result is 0.99999999979803234,
the nodes are [-5.38748089, -4.60368245, -3.94476404, -3.34785457, -2.78880606, -2.254974 , -1.73853771, -1.23407622, -0.73747373, -0.24534071, 0.24534071, 0.73747373, 1.23407622, 1.73853771, 2.254974 , 2.78880606, 3.34785457, 3.94476404, 4.60368245, 5.38748089],
and the weights are [0.89859196145317, 0.70433296117692357, 0.62227869619138665, 0.57526244285250083, 0.54485174236452072, 0.52408035094855054, 0.50967902711745705, 0.49992087133628998, 0.4938433852720529,

0.49092150066674595, 0.49092150066674595, 0.4938433852720529, 0.49992087133628998, 0.50967902711745705, 0.52408035094855054, 0.54485174236452072, 0.57526244285250083, 0.62227869619138665, 0.70433296117692357, 0.89859196145317]

When the number of nodes is 30,
the approximation result is 0.99999999999999634,
the nodes are [-6.86334529, -6.13827922, -5.53314715, -4.98891897, -4.48305536, -4.0039086 , -3.54444387, -3.09997053, -2.66713212, -2.24339147, -1.82674114, -1.4155278 , -1.00833827, -0.60392106, -0.20112858, 0.20112858, 0.60392106, 1.00833827, 1.4155278 , 1.82674114, 2.24339147, 2.66713212, 3.09997053, 3.54444387, 4.0039086 , 4.48305536, 4.98891897, 5.53314715, 6.13827922, 6.86334529],
and the weights are [0.83424747101269592, 0.64909798155433118, 0.56940269194957616, 0.52252568933130883, 0.49105799583287552, 0.4683748125647248, 0.451321035991189, 0.43817702265268194, 0.42791806293273793, 0.41989500373682354, 0.41367936361113872, 0.40898157500353133, 0.40560512332568432, 0.40341981692480389, 0.40234606670190304, 0.40234606670190304, 0.40341981692480389, 0.40560512332568432, 0.40898157500353133, 0.41367936361113872, 0.41989500373682354, 0.42791806293273793, 0.43817702265268194, 0.451321035991189, 0.4683748125647248, 0.49105799583287552, 0.52252568933130883, 0.56940269194957616, 0.64909798155433118, 0.83424747101269592].

3

Listing 3: problem 3

```
1  import numpy as np
2  from scipy.special import p_roots
3  from scipy.special import u_roots
4  from scipy.special import t_roots
5
6  def f(x):
7      return (x**9) / np.sqrt(x**2 + 1)
8
9  def GL(m, a, b):
10     points = p_roots(m)[0]
11     weights = p_roots(m)[1]
12     return (b-a)*sum([weights[i]*f((b-a)*points[i]/2 + (b+a)/2) for i in range(len(points))])/2
13
14 def Cheby1(m, a, b):
15     points = t_roots(m)[0]
16     weights = t_roots(m)[1]
17     return (b-a)*sum([weights[i]*np.sqrt(1-(points[i])**2) * f((b-a)*points[i]/2 + (b+a)/2) for i in range(len(points
           ))])/2
18
19 def Cheby2(m, a, b):
20     points = u_roots(m)[0]
21     weights = u_roots(m)[1]
22     return (b-a)*sum([(weights[i]/np.sqrt(1-(points[i])**2)) * f((b-a)*points[i]/2 + (b+a)/2) for i in range(len(
           points))])/2
23
24 a = [5, 10]
25
26 for m in a:
27     # approximation results
28     print(GL(m))
29     print(Cheby1(m))
30     print(Cheby2(m))
31
32     # nodes
33     print(p_roots(m)[0])
34     print(t_roots(m)[0])
35     print(u_roots(m)[0])
36
37     # weights
38     print(p_roots(m)[1])
39     print(t_roots(m)[1])
40     print(u_roots(m)[1])
```

In the below questions, I mean the weights by the values not multiplied by the weight functions.

3.1

When I use Legendre polynomial as orthogonal polynomial,
the approximation result is -49.506283813990549,
the nodes are [-0.90617985, -0.53846931, 0. , 0.53846931, 0.90617985],
the weights are [0.23692689, 0.47862867, 0.56888889, 0.47862867, 0.23692689].

When I use Chebyshev type 1 polynomial as orthogonal polynomial,
the approximation result is -57.161045874594777,
the nodes are [-9.51056516e-01, -5.87785252e-01, 6.12323400e-17, 5.87785252e-01, 9.51056516e-01,]
the weights are [0.62831853, 0.62831853, 0.62831853, 0.62831853, 0.62831853].

When I use Chebyshev type 2 polynomial as orthogonal polynomial,
the approximation result is -40.789633611622008,
the nodes are [-8.66025404e-01, -5.00000000e-01, 6.12323400e-17, 5.00000000e-01, 8.66025404e-01]
the weights are [0.13089969, 0.39269908, 0.52359878, 0.39269908, 0.13089969].

3.2

When I use Legendre polynomial as orthogonal polynomial,
the approximation result is -49.493963006199031,
the nodes are [-0.97390653, -0.86506337, -0.67940957, -0.43339539, -0.14887434, 0.14887434, 0.43339539, 0.67940957, 0.86506337, 0.97390653],
the weights are [0.06667134, 0.14945135, 0.21908636, 0.26926672, 0.29552422, 0.29552422, 0.26926672, 0.21908636, 0.14945135, 0.06667134].

When I use Chebyshev type 1 polynomial as orthogonal polynomial,
the approximation result is -50.987455239343021,
the nodes are [-0.98768834, -0.89100652, -0.70710678, -0.4539905 , -0.15643447, 0.15643447, 0.4539905 , 0.70710678, 0.89100652, 0.98768834]
the weights are [0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927, 0.31415927].

When I use Chebyshev type 2 polynomial as orthogonal polynomial,
the approximation result is -47.101342989695162,
the nodes are [-0.95949297, -0.84125353, -0.65486073, -0.41541501, -0.14231484, 0.14231484, 0.41541501, 0.65486073, 0.84125353, 0.95949297]
the weights are [0.02266894, 0.08347854, 0.16312218, 0.23631356, 0.27981494, 0.27981494, 0.23631356, 0.16312218, 0.08347854, 0.02266894].

4

Listing 4: problem 4

```
1  import numpy as np
2  from scipy.stats import binom
3
4  n = 25
5
6  iteration = 100
7  values = np.ones((iteration, 3))
8  initial_value = [1/3, 1/3, 1/3]
9  values[0, :] = initial_value
10
11 for ite in range(1, iteration):
12     q_a = (2*values[ite−1, 2]) / (2*values[ite−1, 2] + values[ite−1, 0])
13     q_b = (2*values[ite−1, 2]) / (2*values[ite−1, 2] + values[ite−1, 1])
14     under = sum([binom.pmf(i, n, q_a) * binom.pmf(j, n, q_b) * (3*n − i) for i in range(n+1) for j in range(n+1)])
15     upper = sum([binom.pmf(i, n, q_a) * binom.pmf(j, n, q_b) * (2*n + i + j) for i in range(n+1) for j in range(n+1)
              ])
16     values[ite, 0] = 1/(2+(upper/under))
17     values[ite, 1] = 1/(2+(upper/under))
```

4.1

4.2

By using the above code, I get the estimation result after 100 iterations.

The result is $[p_a, p_b, p_o] = [0.28021178, 0.28021178, 0.43957643]$.

5

Listing 5: problem 5

```
1   # data generate
2   import numpy as np
3
4   I = 100
5   J = 2
6   beta_0 = −1
7   beta_1 = 1
8   sigma_u = 0.5
9   sigma_eps = 1
10
11  np.random.seed(12345)
12
13  value_x = np.random.normal(0, 1, (I, J))
14  u = np.random.normal(0, sigma_u, I)
15  value_u = np.ones((I, J))
16  value_u[:, 0] = u
17  value_u[:, 1] = u
18  value_eps = np.random.normal(0, sigma_eps, (I, J))
19  value_y = beta_0 + beta_1 * value_x + value_u + value_eps
20
21
22  # implement EM algorithm
23  # para = (beta_0, beta_1, sigma_eps, sigma_u)
24  # sigma is variance
25  iteration = 500
26  XY = sum(sum(value_y * value_x))
27  X = sum(sum(value_x))
28  Y = sum(sum(value_y))
29  X_2 = sum(sum(value_x * value_x))
30
31  def xE(E_t):
32      return sum(sum(value_x * np.array([E_t, E_t]).T))
33
34  def eps_right(E_t, beta0, beta1):
35      return (sum(sum((value_y − beta0−beta1∗value_x)∗∗2)) + J * sum(E_t∗∗2) − 2∗sum(sum((value_y − beta0 −
            beta1∗value_x) * np.array([E_t, E_t]).T)))/(I∗J)
36
37  initial = [0, 0, 5, 5]
38  E_t = np.ones(I)
39  estimation = np.ones((iteration, 4))
40  estimation[0, :] = initial
41
42  for ite in range(1,iteration):
43      u = estimation[ite−1, 3]
44      eps = estimation[ite−1, 2]
45      beta0 = estimation[ite−1, 0]
46      beta1 = estimation[ite−1, 1]
47      for i in range(I):
48          E_t[i] = u∗sum([value_y[i, j] − beta0 − beta1∗value_x[i, j] for j in range(J)]) / (J∗u + eps)
49      V_t = eps∗u/(J∗u + eps)
50      estimation[ite, 1] = (I∗J ∗XY − Y∗X − J * (I∗xE(E_t) − sum(E_t)∗X))/(I∗J∗X_2 − X∗∗2)
51      estimation[ite, 0] = (Y−beta1∗X − J∗ sum(E_t))/(I∗J)
52      estimation[ite, 2] = V_t + eps_right(E_t, estimation[ite−1, 0], estimation[ite−1, 1])
53      estimation[ite, 3] = V_t + sum(E_t∗∗2)/I
54
55  # change the last variance into standard deviation
56  estimation[:, 2] = np.sqrt(estimation[:, 2])
```

```
57  estimation[:, 3] = np.sqrt(estimation[:, 3])
58
59  # compute the bias and std
60  bias = np.average(estimation, axis = 0) − [−1, 1, 1, 0.5]
61  std = np.std(estimation, axis = 0)
62  print("bias␣:", bias)
63  print("std␣:", std)
```