



Complex Networks 2

Assignment 2

Problem 1

What does it mean for a network to be typical in a model?

For a network to be *typical* in a model, we assume it is meant to be (f, ϵ) -typical for some chosen graph-distance metric, $d : G^2 \rightarrow \mathbb{R}^2$, with threshold, $\bar{d} < \epsilon$, and sufficient statistics in the model, $\{f\}$, which are fixed and concentrated.

Describe typical networks in:

1. Canonical $G(n, p)$

The canonical $G(n, p)$ model is that which fixes an average degree $\bar{k} = np$. Thus, all graphs with average degree \bar{k} are typical in the canonical G_{np} model.

2. Microcanonical $G(n, p)$

All graphs with $m = n\bar{k} = p\binom{n}{2}$ edges are typical in the microcanonical G_{np} model.

3. Barabási-Albert Model $BA(n, m)$

There are no known rigorous notions of typicality in the BA Model $\backslash(\circ\bullet\circ)/$

4. The "simplest" model of scale-free networks

Because sufficient statistics for this model are unclear, there are no rigorous notions of typicality in this model either.

5. microcanonical configuration model $CM(\{k_1, k_2, \dots, k_n\})$

All labeled graphs with degree sequence $\{k_1, k_2, \dots, k_n\}$ are typical in the model.

6. soft (canonical) configuration model $SCM(\{\bar{k}_1, \bar{k}_2, \dots, \bar{k}_n\})$

All labeled graphs with expected degrees $\{\bar{k}_1, \bar{k}_2, \dots, \bar{k}_n\}$ are typical in the model.

Problem 2

(a) To prove that $P_f(G) = P^*(G)$, where $P^*(G)$ is the ensemble which maximizes ensemble entropy $S[P] = \sum_{G \in \mathbb{G}} P(G) \log P(G)$, we treat this as an optimization problem. The function to optimized being the entropy $S[P]$, with constraints coming from the normalization and definition of $\langle f_s(G) \rangle$ with Lagrange multipliers β and λ , respectively. To do this we maximize the functional

$$\mathcal{L} = - \sum_{\mathbb{G}} P(G) \log P(G) + \beta[1 - \sum_{\mathbb{G}} P(G)] + \sum_s \lambda_s [\sum_{\mathbb{G}} f_s(G)P(G) - \bar{f}_G] \quad (1)$$

We find the optima along $\frac{\partial \mathcal{L}}{\partial P(G_i)} = 0$ where $G \in \mathbb{G}$ is a single graph instance to be

$$P^*(G) = e^{-(1-\beta) - \sum_s \lambda_s f_s(G)} \quad (2)$$

The optima along the level set $\frac{\partial \mathcal{L}}{\partial \beta} = 0$ are then found defined as

$$\sum_{\mathbb{G}} P^*(G) = 1 \quad (3)$$

Inserting Eq. 2 into Eq. 3, we find arrive at the condition that

$$\sum_{\mathbb{G}} e^{-(1-\beta)-\sum_s \lambda_s f_s(G)} = 1 \implies e^{-(1-\beta)} = \frac{1}{e^{-\sum_s \lambda_s f_s(G)}} = Z_{\bar{f}}^{-1} \quad (4)$$

Finally, we can take the leftmost equivalence in 4 and plug it into 2 to get

$$P^*(G) = \frac{1}{e^{-\sum_s \lambda_s f_s(G)}} \cdot e^{-\sum_s \lambda_s f_s(G)} \quad (5)$$

□

(b) Using the definition of $Z_{\bar{f}}$ in the problem, and as concurrently derived in 4, we show that $\langle f_s(G) \rangle = -\frac{\partial \ln Z}{\partial \lambda_s}$ where $\langle \cdot \rangle$ is an average.

To begin, we use the chain rule as

$$\frac{\partial \ln Z}{\partial \lambda_s} = \frac{1}{Z} \frac{\partial Z}{\partial \lambda_s} \quad (6)$$

Opening up the summation over \mathbb{G} , and using the chain rule again we see that

$$\frac{\partial Z}{\partial \lambda_s} = e^{-\sum_s \lambda_s f_s(G_1)} \cdot \frac{\partial}{\partial \lambda_s} \left(-\sum_s \lambda_s f_s(G_1) \right) + \dots + e^{-\sum_s \lambda_s f_s(G_k)} \cdot \frac{\partial}{\partial \lambda_s} \left(-\sum_s \lambda_s f_s(G_k) \right) \quad (7)$$

where $k = ||\mathbb{G}||$. The derivative which remains simply evaluates to $\frac{\partial}{\partial \lambda_s} \left(-\sum_s \lambda_s f_s(G) \right) = -\sum_s f_s(G)$ and thus we're left with

$$\langle f_s(G) \rangle = \frac{\partial \ln Z}{\partial \lambda_s} = -\frac{1}{Z} \left(-\sum_{\mathbb{G}} f_s(G) e^{-\sum_s \lambda_s f_s(G)} \right) \quad (8)$$

The sum over s is dropped because the $\partial/\partial \lambda_s$ drops all terms, i in the sum where $i \neq s$. Finally, bringing $1/Z$ into the sum, we get

$$-\sum_{\mathbb{G}} \frac{1}{Z} e^{-\sum_s \lambda_s f_s(G)} f_s(G) = -\sum_{\mathbb{G}} P(G) f_s(G) \quad (9)$$

and thus,

$$\langle f_s(G) \rangle = -\frac{\partial \ln Z}{\partial \lambda_s} = \sum_{\mathbb{G}} P(G) f_s(G) \quad (10)$$

□

Problem 3

We begin by sampling from the Pareto Distribution described on the Homework. Using the Numpy python package, a bit of algebra shows that the shape parameter in their documentation, a , is equal to $\gamma - 1$. From there, it follows that what they call m is what we call x_- .

(a) See code attached.

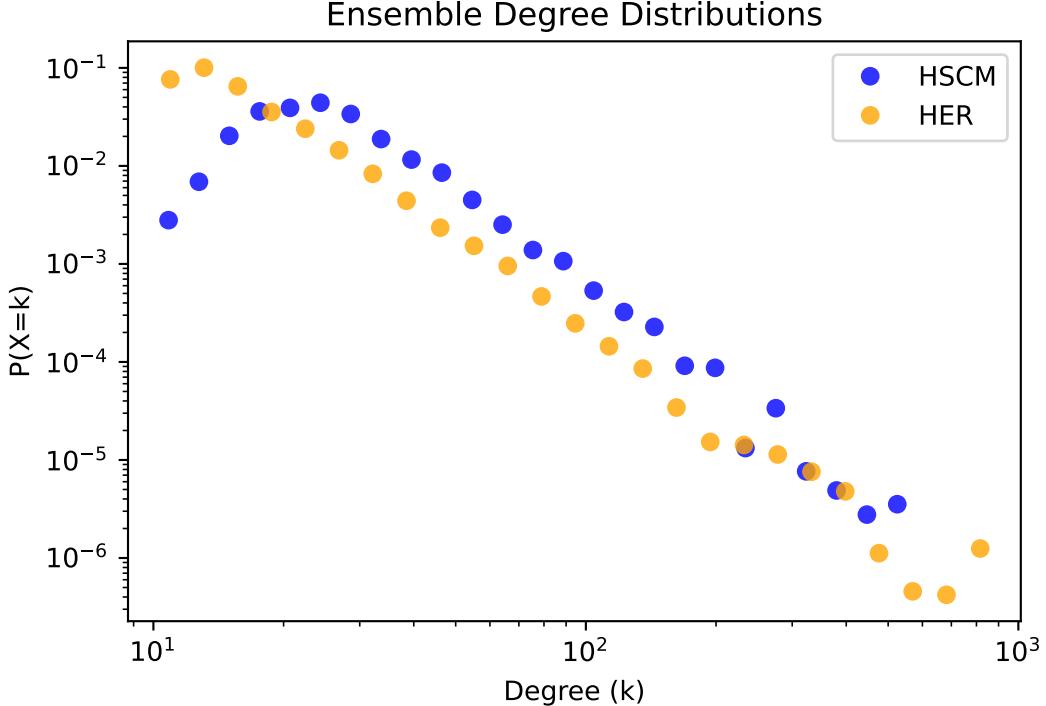


Figure 1: Degree distributions from the Hypersoft Configuration Model (HSCM), sampling all nodes from all $N_{HSCM} = 10$ graphs sampled from the model; and the Hypercanonical Erdős-RenyiModel (HER), sampling all nodes from all $N_{HER} = 10^4$ graphs sampled from the model. Both distributions were log-binned to normalize to 1.

(b, c) As shown in 1, the log-binned distributions of the two *ensembles*' degree distributions look very similar.

Both appear to show a clear power-law degree distribution with similar tail exponent. This is to be expected because by considering the degree distribution of all graphs in the ensemble, we are effectively sampling the same Pareto Distribution which defines $p_{k,\gamma}$. Thus, because the *p*-values for the *ensemble* of HER graphs came from the same Pareto Density as the *p*-values for *each edge in each graph* for the HSCM, it makes sense that the distributions align very well.

However, although *p* in a single HER model comes from the same Pareto Density, it is fixed. Thus, as seen in 2, the degree distribution of any *single* ER graph in the HER model is very different from that of any single graph in the HSCM model because in the HSCM graph, *every edge probability* is pulled from this Pareto Distribution, while in the ER graph, although *p* may be pulled from the Pareto, each edge is still coming from a Bernoulli Distribution with bias *p*. As such, instead of a power law distribution, we see a Binomial degree distribution for the sampled ER graph.

0.1 Appendix: Code

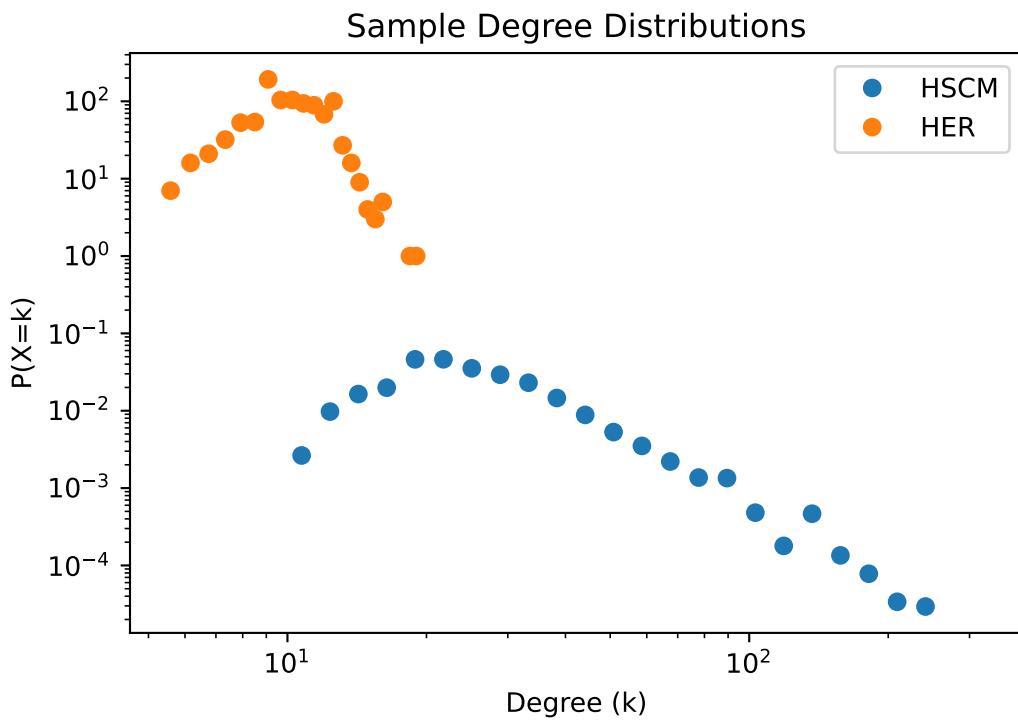


Figure 2: Degree distribution of a single, random sampled graph from both the HSCM and HER models. The HSCM distribution was log-binned, while the HER distribution was linearly binned. In an attempt to maintain the same number of points for both, the bin width of the HER graph renders probabilities greater than 1.

```

1  from collections import defaultdict
2  import numpy as np
3  from statistics import mean
4  from scipy.sparse import dok_matrix, csr_matrix, triu, tril
5  from random import randint
6
7  from tqdm.notebook import tqdm
8
9  """
10 utils.py
11
12 Written by Sagar Kumar, April 2023
13
14 This is a Python script with useful functions which I have written and which are utilized
   throughout the scripts which
15 have been written for Problem 3, Homework 2 in NETS6116.
16
17 Some come from HW1.
18 """
19
20
21 def count_faster(l):
22     dic = {}
23
24     for i in l:
25
26         if i in dic.keys():
27             dic[i] += 1
28
29         else:
30             dic[i] = 1
31
32     return [(key, dic[key]) for key in dic.keys()]
33
34
35
36 def distributionBin(x: 'input data',
37                      B: 'number of bins'
38                      ):
39
40 """
41 :param x: (list) list of real-valued integers interpreted as the input data sampled from a
   presumed heavy-tail distribution
42 :param B: (positive int) number of log-scaled bins to create a histogram from x
43
44 distributionBin: Tuple, int -> Tuple, Tuple[Float]
45
46 distributionBin takes a list of input data :x: assumed to be sampled from some heavy-tailed
   distribution and bins in :B: log

```

```

47 sized bins in such a way that the output function *integrates* to 1. The output is a list of bin-
48 midpoints X and the binned
49 """
50
51 xmin = min(x)
52 xmax = max(x)
53
54 # creating the B+1 bin edges
55 bin_edges = np.logspace(max(1,np.log10(xmin)), np.log10(xmax), num=B+1)
56
57 # using numpy's histogram function get distributions
58 density, _ = np.histogram(x, bins=bin_edges, density=True)
59
60 # obtaining bin midpoints for cleaner absciss
61 log_be = np.log10(bin_edges)
62 xout = 10**((log_be[1:] + log_be[:-1])/2)
63
64 return xout, density
65
66
67 def functionBin(x: 'preimage',
68                 y: 'function output for some function y | y_i=y(x)',
69                 B: 'number of bins'
70                 ):
71
72 """
73 :param x: (list) list of real-valued numbers interpreted as the preimage of the function y(x)
74 :param y: (list) list of real-valued numbers interpreted as the values of function y applied to each
    input value in x
75 :param B: (positive int) number of log-scaled bins to create a histogram
76
77 functionBin: Tuple, Tuple, int -> Tuple, Tuple
78
79 functionBin takes an ordered list of sampled input values :x: and their corresponding outputs
    under some function :y: (
80     i.e. y(x)) and bins the input values into :B: log sized bins, averaging the output values within
    each bin. The output is
81     the bin-midpoints X and the binned outputs.
82
83 """
84
85 xmin = min(x)
86 xmax = max(x)
87
88 # creating the B+1 bin edges
89 bin_edges = np.logspace(max(1,np.log10(xmin)), np.log10(xmax), num=B+1)
90
91 # obtaining bin midpoints for cleaner absciss

```

```

92     log_be = np.log10(bin_edges)
93     bm = 10**((log_be[1:] + log_be[:-1])/2)
94
95     # creating (input, output) pairs and sorting by input value
96     fpairs = list(zip(x,y))
97     fpairs.sort(key=lambda x: x[0])
98
99     # creating (label, boundary) pairs using midpoint and right boundary for each bin
100    mid_redge_pairs = list(zip(bm, bin_edges[1:]))
101
102    # dictionary of values where key corresponds to the bin and values are those which fit in the
103    bin_out_list = defaultdict(list)
104
105    idx = 0
106    for mid, redge in tqdm(mid_redge_pairs, desc='Binning Function'):
107        while (fpairs[idx][0] < redge) & (idx < len(fpairs)-1):
108            bin_out_list[mid].append(fpairs[idx])
109            idx += 1
110
111    # adding the last value
112    bin_out_list[list(bin_out_list.keys())[-1]].append(fpairs[-1])
113
114    xout = list(bin_out_list.keys())
115
116    # y value is the average of each bin
117    yout = [mean([i[1] for i in b]) for b in bin_out_list.values()]
118
119    return xout, yout
120
121
122 def simpleSF(n: 'graph size' = 10**4,
123             s: 'number of samples' = 10,
124             sparse: 'choice of numpy vs. scipy sparse matrix' = False
125             ):
126
127     """
128         :param n: (positive int) total number of nodes in the desired graph after completing its
129         percolation
130         :param s: (positive int) total number of graphs created
131         :param sparse: (boolean) False maps output to dense numpy ndarrays, True maps them to
132         scipy sparse csr_matrix
133
134         simpleSF takes a value for the size of the graph :n: and the number of samples :s: and
135         percolates :s: scale-free graphs of
136         size :n: by randomly selecting an edge E := (i,j) and attaching new incoming nodes to both i and
137         j. The output is a list of

```

File - C:\Users\szkum\NetSI\Complex_Networks_2\HW2\utils.py

```

136 :s: (:n: x :n:) matrices which act as the adjacency matrix of each sample graph. Choosing scipy
137 sparse matrix allows for
138 larger graph sizes.
139
140     outlist = list()
141     pbar = tqdm(range(s), desc="Generating Samples")
142
143     for i in pbar:
144
145         # edgelist
146         edgelist = {(0,1)}
147
148         # selecting a random edge and making the incident nodes the fruits of an incoming cherry to
149         # create a triangle
150         # process repeats until t == n
151         t = 2
152         while t < n:
153             m = randint(0, len(edgelist)-1)
154             i,j = list(edgelist)[m]
155             edgelist = edgelist | {(i, t), (j, t)}
156             t += 1
157
158         # mapping corresponding matrix values for all pairs in edgelist from 0 -> 1
159         if sparse:
160             A = dok_matrix((n, n), dtype=int)
161             for e in edgelist:
162                 A[e] = 1
163
164             # currently only have upper triangle, so making the matrix symmetric about the main
165             # diagonal
166             # triu and tril authomatically change it from dok_matrix to csr_matrix
167             A = triu(A) + tril(A.T)
168
169         else:
170             A = np.zeros((n,n))
171             for e in edgelist:
172                 A[e] = 1
173
174             # currently only have upper triangle, so making the matrix symmetric about the main
175             # diagonal
176             A = np.triu(A) + np.tril(A.T)
177
178         outlist.append(A)
179
180     return outlist

```

```

1 import random
2 from tqdm.notebook import tqdm
3 from HW2.utils import distributionBin
4 from scipy.sparse import csr_matrix, triu, tril
5 import scipy.sparse
6 import numpy as np
7
8 """
9 HyperNetworkModel.py
10
11 Written by Sagar Kumar, April 2023 for Problem 3 in Homework 2 in the course NETS6116.
12
13 All references to "HER" stand for Hypercanonical Erdos-Renyi Model as described in the
homework and all references to
14 "HSCM" refer to the Hypersoft Configuration Model, as described in the homework.
15 """
16
17 class HyperNetworkModel:
18
19     def __init__(self,
20                  n,
21                  kmean,
22                  gamma):
23         self.n = n
24         self.kmean = kmean
25         self.gamma = gamma
26
27         self.ensemble = list()
28
29 """
30     HyperNetworkModel is a class which holds the percolating functions for both the HER and
HSCM models and
31     supporting functions.
32
33     :param n: [Int] Number of nodes
34     :param kmean: [Float] scale parameter (m in the numpy documentation)
35     :param gamma: [Float] shape (tail) parameter (a+1 in the numpy documentation)
36     :param ensemble: [Tuple] List of adjacency matrices which sample the model
37 """
38
39     def paretoDistribution(self,
40                           samples):
41
42 """
43     paretoDistribution() creates a list of n samples from a Pareto Distribution, as described in
HW2 for NETS 6116.
44     This is done using Numpy's random.pareto, which takes provides a Lomax distribution. Adding
one and multiplying
45     by m=kmax, with a=gamma - 1, we obtain the Pareto Distribution described in the homework.

```

```

46
47     $p(\bar{k}, \gamma) = (\gamma - 1) (x_m)^{\gamma - 1} x^{-\gamma}$ where
48     $x_m = \frac{(\gamma - 2) \bar{k}}{\gamma - 1}$
49
50     :param samples: number of samples to take
51
52     :return: [1 x N Array] samples
53     """
54
55     s = (np.random.pareto(self.gamma-1, samples) + 1) * self.kmean
56
57
58     return s
59
60
61 def HSCM(self):
62
63     """
64     HSCM() percolates a hypersoft configuration graph model by taking in the three parameters
65     below which correspond
66         to the necessary statistics for generation, and outputs the adjacency matrix as a scipy sparse
67     matrix.
68         The latter two variables are fed into paretoDistribution() to sample the hidden variables
69     which determine
70         a node's connection probabilities.
71
72     :return: [N x N Array]
73     """
74
75     # sampling n values from the distribution for the n nodes
76     distribution = self.paretoDistribution(samples=self.n)
77
78     row = list()
79     column = list()
80     data = list()
81
82     # iterating over each edge
83     for i in range(self.n):
84         xi = distribution[i]
85         for j in range(i+1, self.n):
86             xj = distribution[j]
87             pij = (1 + (self.kmean*self.n)/(xi*xj))**-1 # connection probability in HSCM
88
89             r = random.random() # coin flip
90
91             if r <= pij:
92                 row.append(i)
93                 column.append(j)
94                 data.append(1)
95             else:

```

```

92     pass
93
94     # placing the values in a SciPy Sparse Matrix
95     M = csr_matrix((data, (row, column)), shape=(self.n, self.n))
96
97     # Only upper diagonal has been filled out, so we return a symmetrized version
98     return triu(M) + tril(M.T)
99
100
101 def HER(self):
102     """
103     HER() percolates the Hypercanonical ER model as described in HW2.
104     :return:
105     """
106
107
108
109     kappa = self.paretoDistribution(samples=1) # sampling the Pareto
110     p = min(1, kappa/self.n)
111
112     row = list()
113     column = list()
114     data = list()
115
116     # creating an NxN matrix filled with values (0,1)
117     rand_m = scipy.sparse.random(self.n, self.n, density=1, format='csr')
118
119     # Converting that matrix to a boolean matrix where values correspond to whether or not the
120     # value in the
121     # element is less than the probability sampled from the pareto distribution
122     M = (rand_m <= p)
123
124     # Graph is undirected, so lower triangle is discarded and upper triangle is reflected over the
125     # main diagonal
126     return triu(M) + tril(M.T)
127
128
129     def create_ensemble(self,
130                         model,
131                         num_graphs):
132         """
133         create_ensemble() samples to create [num_graphs] sample graphs, appending them to self.
134         ensemble
135
136         :param model: [String] either "HER" or "HSCM"
137         :param num_graphs: [Int] number of sample graphs
138         :return: [N X N x num_graphs Array] ensemble of graphs
139         """
140
141
142         if model == "HSCM":

```

```

138     for _ in tqdm(range(num_graphs), desc="Generating HSCM Ensemble: "):
139         m = self.HSCM()
140         self.ensemble.append(m)
141
142     elif model == "HER":
143         for _ in tqdm(range(num_graphs), desc="Generating HER Ensemble: "):
144             m = self.HER()
145             self.ensemble.append(m)
146
147     else:
148         raise ValueError("Graph model must be either HSCM or HER")
149
150     def degree_distribution(self,
151                           binning,
152                           num_bins,
153                           graph=None):
154
155         """
156         degree_distribution() calculates the degree distribution of the
157         :param binning: [String] either "log" or "linear"
158         :param num_bins: [Int] number of bins for the graph
159         :param graph: [Optional, Array] If graph is not none, this is run over the ensemble attribute
160         of the object. If
161         graph is provided, it is run for that graph.
162         :return: Tuple[Array, Array] list of bin-midpoint x values, and a list of probability density
163         values
164         """
165
166         degrees = list()
167
168         if graph is not None:
169             G = [graph]
170
171         else:
172             G = self.ensemble
173
174         for g in G:
175             # compressing ensemble by concatenating all degrees of all graphs into one degree
176             # sequence
177             degrees.extend(g.sum(axis=0).tolist()[0])
178
179         if binning == "log":
180             dist_x, dist_y = distributionBin(degrees, num_bins)
181
182         elif binning == "linear":
183             bins = np.linspace(min(degrees), max(degrees), num_bins)
184             dist_x = (bins[1:] + bins[:1])/2
185             dist_y, _ = np.histogram(degrees, bins=bins)

```

```
184     else:  
185         raise ValueError("Data binning must be either log or linear.")  
186  
187     return dist_x, dist_y  
188
```

```

1 #%%
2 import matplotlib.pyplot as plt
3 import random
4 from HW2.HyperNetworkModel import HyperNetworkModel
5 #%% md
6 # MAIN
7 #%%
8 n = 10**3
9 gamma = 3
10 k = 10
11
12 hscm = HyperNetworkModel(n, k, gamma)
13 her = HyperNetworkModel(n, k, gamma)
14
15 hscm.create_ensemble("HSCM", 10)
16 her.create_ensemble("HER", 10**4)
17 #%%
18 hscm_k, hscm_pk = hscm.degree_distribution('log', 25)
19 her_k, her_pk = her.degree_distribution('log', 25)
20
21 #%%
22 plt.plot(hscm_k, hscm_pk, 'o', label="HSCM", color='blue', alpha=0.8)
23 plt.plot(her_k, her_pk, 'o', label="HER", color='orange', alpha=0.8)
24 plt.legend()
25 plt.ylabel(r'P(X=k)')
26 plt.xlabel('Degree (k)')
27 plt.loglog()
28 plt.title("Ensemble Degree Distributions")
29 plt.savefig("ensemble_degree_distributions.pdf")
30 #%%
31 sample_hscm = hscm.ensemble[random.randint(0,9)]
32 sample_her = her.ensemble[random.randint(0, 10**4-1)]
33
34 sample_hscm_k, samples_hscm_pk = hscm.degree_distribution('log', 25, graph=sample_hscm)
35 sample_her_k, samples_her_pk = her.degree_distribution('linear', 25, graph=sample_her)
36
37 plt.plot(sample_hscm_k, samples_hscm_pk, 'o', label="HSCM")
38 plt.plot(sample_her_k, samples_her_pk, 'o', label="HER")
39 plt.legend()
40 plt.ylabel(r'P(X=k)')
41 plt.xlabel('Degree (k)')
42 plt.loglog()
43 plt.title("Sample Degree Distributions")
44 plt.savefig("sample_degree_distributions.pdf")
45 #%%
46

```