```python
from collections import defaultdict
import numpy as np
from statistics import mean
from scipy.sparse import dok_matrix, csr_matrix, triu, tril
from random import randint

from tqdm.notebook import tqdm

"""
utils.py

Written by Sagar Kumar, April 2023

This is a Python script with useful functions which I have written and which are utilized throughout the scripts which
have been written for Problem 3, Homework 2 in NETS6116.

Some come from HW1.
"""


def count_faster(l):
    dic = {}

    for i in l:

        if i in dic.keys():
            dic[i] += 1

        else:
            dic[i] = 1

    return [(key, dic[key]) for key in dic.keys()]



def distributionBin(x: 'input data',
                    B: 'number of bins'
                    ):

    """
    :param x: (list) list of real-valued integers interpreted as the input data sampled from a presumed heavy-tail distribution
    :param B: (positive int) number of log-scaled bins to create a histogram from x

    distributionBin: Tuple, int -> Tuple, Tuple[Float]

    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-
    midpoints X and the binned
48      estimated distribution values Y.
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
        bin
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
        percolation
129     :param s: (positive int) total number of graphs created
130     :param sparse: (boolean) False maps output to dense numpy ndarrays, True maps them to
        scipy sparse csr_matrix
131
132     simpleSF: Int, Int, Bool -> Tuple[np.ndarray OR scipy.sparse.csr_matrix]
133
134     simpleSF takes a value for the size of the graph :n: and the number of samples :s: and
        percolates :s: scale-free graphs of
135     size :n: by randomly selecting an edge E := (i,j) and attaching new incoming nodes to both i and
         j. The output is a list of
```

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

```python
import random
from tqdm.notebook import tqdm
from HW2.utils import  distributionBin
from scipy.sparse import csr_matrix, triu, tril
import scipy.sparse
import numpy as np

"""
HyperNetworkModel.py

Written by Sagar Kumar, April 2023 for Problem 3 in Homework 2 in the course NETS6116.

All references to "HER" stand for Hypercanonical Erdos-Renyi Model as described in the homework and all references to
"HSCM" refer to the Hypersoft Configuration Model, as described in the homework.
"""

class HyperNetworkModel:

    def __init__(self,
                 n,
                 kmean,
                 gamma):
        self.n = n
        self.kmean = kmean
        self.gamma = gamma

        self.ensemble = list()

        """
        HyperNetworkModel is a class which holds the percolating functions for both the HER and HSCM models and
        supporting functions.

        :param n: [Int] Number of nodes
        :param kmean: [Float] scale parameter (m in the numpy documentation)
        :param gamma: [Float] shape (tail) parameter (a+1 in the numpy documentation)
        :param ensemble: [Tuple] List of adjacency matricies which sample the model
        """

    def paretoDistribution(self,
                           samples):

        """
        paretoDistribution() creates a list of n samples from a Pareto Distribution, as described in HW2 for NETS 6116.
        This is done using Numpy's random.pareto, which takes provides a Lomax distribution. Adding one and multiplying
        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
    below which correspond
65           to the necessary statistics for generation, and outputs the adjacency matrix as a scipy sparse
    matrix.
66           The latter two variables are fed into paretoDistribution() to sample the hidden variables
    which determine
67           a node's connection probabilities.
68
69          :return: [N x N Array]
70          """
71          # sampling n values from the distribution for the n nodes
72          distribution = self.paretoDistribution(samples=self.n)
73
74          row = list()
75          column = list()
76          data = list()
77
78          # iterating over each edge
79          for i in range(self.n):
80              xi = distribution[i]
81              for j in range(i+1, self.n):
82                  xj = distribution[j]
83                  pij = (1 + (self.kmean*self.n)/(xi*xj))**-1 # connection probability in HSCM
84
85                  r = random.random() # coin flip
86
87                  if r <= pij:
88                      row.append(i)
89                      column.append(j)
90                      data.append(1)
91                  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        """
104        HER() percolates the Hypercanonical ER model as described in HW2.
105
106        :return:
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
    value in the
120        # element is less than the probability sampled from the pareto distribution
121        M = (rand_m <= p)
122
123        # Graph is undirected, so lower triangle is discarded and upper triangle is reflected over the
    main diagonal
124        return triu(M) + tril(M.T)
125
126    def create_ensemble(self,
127                        model,
128                        num_graphs):
129        """
130        create_ensemble() samples to create [num_graphs] sample graphs, appending them to self.
    ensemble
131
132        :param model: [String] either "HER" or "HSCM"
133        :param num_graphs: [Int] number of sample graphs
134        :return: [N X N x num_graphs Array] ensemble of graphs
135        """
136
137        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         else:
147             raise ValueError("Graph model must be either HSCM or HER")
148 
149     def degree_distribution(self,
150                             binning,
151                             num_bins,
152                             graph=None):
153 
154         """
155         degree_distribution() calculates the degree distribution of the
156 
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
    of the object. If
160         graph is provided, it is run for that graph.
161         :return: Tuple[Array, Array] list of bin-midpoint x values, and a list of probability density
    values
162         """
163 
164         degrees = list()
165 
166         if graph is not None:
167             G = [graph]
168 
169         else:
170             G = self.ensemble
171 
172         for g in G:
173             # compressing ensemble by concatenating all degrees of all graphs into one degree
    sequence
174             degrees.extend(g.sum(axis=0).tolist()[0])
175 
176         if binning == "log":
177             dist_x, dist_y = distributionBin(degrees, num_bins)
178 
179         elif binning == "linear":
180             bins = np.linspace(min(degrees), max(degrees), num_bins)
181             dist_x = (bins[1:] + bins[:1])/2
182             dist_y, _ = np.histogram(degrees, bins=bins)
183 
```

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

```
#%%
import matplotlib.pyplot as plt
import random
from HW2.HyperNetworkModel import HyperNetworkModel
#%% md
# MAIN
#%%
n = 10**3
gamma = 3
k = 10

hscm = HyperNetworkModel(n, k, gamma)
her = HyperNetworkModel(n, k, gamma)

hscm.create_ensemble("HSCM", 10)
her.create_ensemble("HER", 10**4)
#%%
hscm_k, hscm_pk = hscm.degree_distribution('log', 25)
her_k, her_pk = her.degree_distribution('log', 25)

#%%
plt.plot(hscm_k, hscm_pk, 'o', label="HSCM", color='blue', alpha=0.8)
plt.plot(her_k, her_pk, 'o', label="HER", color='orange', alpha=0.8)
plt.legend()
plt.ylabel(r'P(X=k)')
plt.xlabel('Degree (k)')
plt.loglog()
plt.title("Ensemble Degree Distributions")
plt.savefig("ensemble_degree_distributions.pdf")
#%%
sample_hscm = hscm.ensemble[random.randint(0,9)]
sample_her = her.ensemble[random.randint(0, 10**4-1)]

sample_hscm_k, samples_hscm_pk = hscm.degree_distribution('log', 25, graph=sample_hscm)
sample_her_k, samples_her_pk = her.degree_distribution('linear', 25, graph=sample_her)

plt.plot(sample_hscm_k, samples_hscm_pk, 'o', label="HSCM")
plt.plot(sample_her_k, samples_her_pk, 'o', label="HER")
plt.legend()
plt.ylabel(r'P(X=k)')
plt.xlabel('Degree (k)')
plt.loglog()
plt.title("Sample Degree Distributions")
plt.savefig("sample_degree_distributions.pdf")
#%%

```