```
#%% raw
!pip install powerlaw
#%%
import numpy as np
import math
import matplotlib.pyplot as plt
from collections import defaultdict
from statistics import mean
from scipy.sparse import dok_matrix, csr_matrix, triu, tril
from random import randint
import powerlaw as pl
import tail_estimation as te
from tqdm.notebook import tqdm
from utils import count_faster
#%% md
## Functions
#%%
################################################## Problem 2.1
 ##########################################################################
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
    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
    estimated distribution values Y.
    """

    xmin = min(x)
    xmax = max(x)

    # creating the B+1 bin edges
    bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)

    # using numpy's histogram function get distributions
    density, _ = np.histogram(x, bins=bin_edges, density=True)

    # obtaining bin midpoints for cleaner absciss
    log_be = np.log10(bin_edges)
    xout = 10**((log_be[1:] + log_be[:-1])/2)
```

```
46
47     return xout, density
48 #%%
49 ################################################## Problem 2.2
    ##########################################################################
50 def functionBin(x: 'preimage',
51                 y: 'function output for some function y | y_i=y(x)',
52                 B: 'number of bins'
53                ):
54
55     """
56     :param x: (list) list of real-valued numbers interpreted as the preimage of the function y(x)
57     :param y: (list) list of real-valued numbers interpreted as the values of function y applied to
    each input value in x
58     :param B: (positive int) number of log-scaled bins to create a histogram
59
60     functionBin: Tuple, Tuple, int -> Tuple, Tuple
61
62     functionBin takes an ordered list of sampled input values :x: and their corresponding outputs
    under some function :y: (
63     i.e. y(x)) and bins the input values into :B: log sized bins, averaging the output values within
    each bin. The output is
64     the bin-midpoints X and the binned outputs.
65
66     """
67
68     xmin = min(x)
69     xmax = max(x)
70
71     # creating the B+1 bin edges
72     bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)
73
74     # obtaining bin midpoints for cleaner absciss
75     log_be = np.log10(bin_edges)
76     bm = 10**((log_be[1:] + log_be[:-1])/2)
77
78     # creating (input, output) pairs and sorting by input value
79     fpairs = list(zip(x,y))
80     fpairs.sort(key=lambda x: x[0])
81
82     # creating (label, boundary) pairs using midpoint and right boundary for each bin
83     mid_redge_pairs = list(zip(bm, bin_edges[1:]))
84
85     # dictionary of values where key corresponds to the bin and values are those which fit in the bin

86     bin_out_list = defaultdict(list)
87
88     idx = 0
89     for mid, redge in tqdm(mid_redge_pairs, desc='Binning Function'):
```

```
 90         while (fpairs[idx][0] < redge) & (idx < len(fpairs)-1):
 91             bin_out_list[mid].append(fpairs[idx])
 92             idx += 1
 93 
 94     # adding the last value
 95     bin_out_list[list(bin_out_list.keys())[-1]].append(fpairs[-1])
 96 
 97     xout = list(bin_out_list.keys())
 98 
 99     # y value is the average of each bin
100     yout = [mean([i[1] for i in b]) for b in bin_out_list.values()]
101 
102     return xout, yout
103 #%%
104 ################################################## Problem 2.3
     ##########################################################################
105 def simpleSF(n: 'graph size' = 10**4,
106              s: 'number of samples' = 10,
107              sparse: 'choice of numpy vs. scipy sparse matrix' = False
108             ):
109 
110     """
111     :param n: (positive int) total number of nodes in the desired graph after completing its
    percolation
112     :param s: (positive int) total number of graphs created
113     :param sparse: (boolean) False maps output to dense numpy ndarrays, True maps them to
    scipy sparse csr_matrix
114 
115     simpleSF: Int, Int, Bool -> Tuple[np.ndarray OR scipy.sparse.csr_matrix]
116 
117     simpleSF takes a value for the size of the graph :n: and the number of samples :s: and
    percolates :s: scale-free graphs of
118     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
119     :s: (:n: x :n:) matrices which act as the adjacency matrix of each sample graph. Choosing scipy
    sparse matrix allows for
120     larger graph sizes.
121     """
122 
123     outlist = list()
124     pbar = tqdm(range(s), desc=f"Generating Samples")
125 
126     for i in pbar:
127 
128         # edgelist
129         edgelist = {(0,1)}
130 
131         # selecting a random edge and making the incident nodes the fruits of an incoming cherry
    to create a triangle
```

```
132         # process repeats until t == n
133         t = 2
134         while t < n:
135             m = randint(0, len(edgelist)-1)
136             i,j = list(edgelist)[m]
137             edgelist = edgelist | {(i, t), (j, t)}
138             t += 1
139 
140         # mapping corresponding matrix values for all pairs in edgelist from 0 -> 1
141         if sparse:
142             A = dok_matrix((n, n), dtype=int)
143             for e in edgelist:
144                 A[e] = 1
145 
146             # currently only have upper triangle, so making the matrix symmetric about the main
    diagonal
147             # triu and tril authomatically change it from dok_matrix to csr_matrix
148             A = triu(A) + tril(A.T)
149 
150         else:
151             A = np.zeros((n,n))
152             for e in edgelist:
153                 A[e] = 1
154 
155             # currently only have upper triangle, so making the matrix symmetric about the main
    diagonal
156             A = np.triu(A) + np.tril(A.T)
157 
158         outlist.append(A)
159 
160     return outlist
161 
162 
163 #%% md
164 ## Tests
165 #%%
166 import networkx as nx
167 import itertools
168 #%%
169 ################################################## Problem 2.4
     ##########################################################################
170 
171 # creating graph ensemble G
172 G = simpleSF(sparse=True)
173 
174 # setting number of bins
175 num_bins = 20
176 
177 # preparing the plots
```

```
178
179 fig, axs = plt.subplots(1, 3)
180
181 ################# plotting degree distribution ####################
182 degrees = list()
183 for g in G:
184     # compressing ensemble by concatenating all degrees of all graphs into one degree sequence
185     degrees.extend(g.sum(axis=0).tolist()[0])
186
187
188 # binning into :num_bins: bins using function from 2.1
189 dist_x, dist_y = distributionBin(degrees, num_bins)
190
191 # plotting
192 axs[0].loglog(dist_x, dist_y, 'o', color='brown')
193 axs[0].set_title('Degree Distribution')
194 axs[0].set_ylabel(r'$P[X=k]$')
195 axs[0].set_xlabel(r'$k$')
196
197 ################# plotting avg. clustering ####################
198
199 # using networkx to get average clustering and average neighbor degree
200 indexed_node_degrees = list()
201 indexed_clustering = list()
202 indexed_average_degree = list()
203
204 for g in G:
205     # this needs to be np.from_numpy_matrix if not sparse
206     graph = nx.from_scipy_sparse_array(g)
207     for node in graph.nodes:
208         indexed_node_degrees.append(graph.degree(node))
209         indexed_clustering.append(nx.clustering(graph, nodes=[node])[node])
210         indexed_average_degree.append(nx.average_neighbor_degree(graph, nodes=[node])[
    node])
211
212 # Using the function from 2.2 to plot these values
213 c_x, c_y = functionBin(indexed_node_degrees, indexed_clustering, num_bins)
214 kk_x, kk_y = functionBin(indexed_node_degrees, indexed_average_degree, num_bins)
215
216 # plotting clustering on a loglog scale
217 axs[1].loglog(c_x, c_y, 'o', color='orange')
218 axs[1].set_title('Average Clustering Coefficient')
219 axs[1].set_xlabel(r'$k$')
220 axs[1].set_ylabel(r'$\bar{c}(k)$')
221
222 # plotting average neighbor degree on a semi-log scale
223 axs[2].plot(kk_x, kk_y, 'o', color='green')
224 axs[2].set_xscale('log')
225 axs[2].set_title('Average Neighbor Degree')
```

```
226 axs[2].set_xlabel(r'$k$')
227 axs[2].set_ylabel(r'$\bar{k}_{nn}(k)$')
228
229 fig.set_size_inches(10, 4, forward=True)
230 fig.tight_layout()
231
232 fig.savefig('HW1_fig1_pk_ck_knnk.svg')
233 #%%
234 ## Powerlaw package fit
235
236 pl_fit = pl.Fit(degrees)
237 gamma_pl = pl_fit.power_law.alpha
238 print(f"Fit after x = {pl_fit.power_law.xmin}")
239 print(f"Estimated gamma: {gamma_pl}")
240 #%%
241 ## Hill, Moments, and Kernel estimators
242
243 # exporting the data to a dat file
244 import struct
245
246 # sending degree counts to a .dat file that can be used with the package
247 deg_data = count_faster(degrees)
248 deg_data.sort(key=lambda x: x[0])
249 deg_data = np.array(deg_data, dtype=np.int64)
250 np.savetxt('synthetic_scale_free_degree.dat', deg_data, fmt='%i')
251 #%%
252 !python tail_estimation.py synthetic_scale_free_degree.dat HW1_Estimator_plots.pdf
253 #%%
254
255 hill_gamma = 2.889644431436195
256 moments_gamma = 2.8602879353057453
257 kernel_gamma = 2.8648694044336103
258
259 # Plotting Distribution with all of its estimates
260 plt.loglog(dist_x, dist_y, 'o', color='brown')
261 plt.plot(dist_x, [k**(-1*pl_fit.power_law.alpha) for k in dist_x], linestyle='dashed', color='blue',
    label='Powerlaw pkg',
262          alpha=0.8)
263 plt.plot(dist_x, [k**(-1*hill_gamma) for k in dist_x], linestyle='dotted', color='green', label='Hill'
    , alpha=0.8)
264 plt.plot(dist_x, [k**(-1*moments_gamma) for k in dist_x], linestyle='dotted', color='purple',
    label='Moments', alpha=0.8)
265 plt.plot(dist_x, [k**(-1*kernel_gamma) for k in dist_x], linestyle='dotted', color='orange', label=
    'Kernel', alpha=0.8)
266 plt.legend()
267 plt.title('Binned Degree Distribution with Estimates')
268 plt.ylabel(r'$P[X=k]$')
269 plt.xlabel(r'$k$')
270 plt.tight_layout()
```

```
271
272 plt.savefig('HW1_fig2_degree_dist_estimates.svg')
273 #%%
274
275 #%%
276
```