```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.
```

```

        $p(\bar k, \gamma) = (\gamma - 1) (x_m)^{\gamma - 1} x^{-\gamma}$ where
         $x_m = \frac{(\gamma - 2) \bar k}{\gamma - 1}$

         :param samples: number of samples to take

        :return: [1 x N Array] samples
        """

        s = (np.random.pareto(self.gamma-1, samples) + 1) * self.kmean


        return s


    def HSCM(self):

        """
        HSCM() percolates a hypersoft configuration graph model by taking in the three parameters below which correspond
         to the necessary statistics for generation, and outputs the adjacency matrix as a scipy sparse matrix.
         The latter two variables are fed into paretoDistribution() to sample the hidden variables which determine
         a node's connection probabilities.

        :return: [N x N Array]
        """
        # sampling n values from the distribution for the n nodes
        distribution = self.paretoDistribution(samples=self.n)

        row = list()
        column = list()
        data = list()

        # iterating over each edge
        for i in range(self.n):
            xi = distribution[i]
            for j in range(i+1, self.n):
                xj = distribution[j]
                pij = (1 + (self.kmean*self.n)/(xi*xj))**-1 # connection probability in HSCM

                r = random.random() # coin flip

                if r <= pij:
                    row.append(i)
                    column.append(j)
                    data.append(1)
                else:
```

```python
                pass

        # placing the values in a SciPy Sparse Matrix
        M = csr_matrix((data, (row, column)), shape=(self.n, self.n))

        # Only upper diagonal has been filled out, so we return a symmetrized version
        return triu(M) + tril(M.T)


    def HER(self):

        """
        HER() percolates the Hypercanonical ER model as described in HW2.

        :return:
        """

        kappa = self.paretoDistribution(samples=1) # sampling the Pareto
        p = min(1, kappa/self.n)

        row = list()
        column = list()
        data = list()

        # creating an NxN matrix filled with values (0,1)
        rand_m = scipy.sparse.random(self.n, self.n, density=1, format='csr')

        # Converting that matrix to a boolean matrix where values correspond to whether or not the value in the
        # element is less than the probability sampled from the pareto distribution
        M = (rand_m <= p)

        # Graph is undirected, so lower triangle is discarded and upper triangle is reflected over the main diagonal
        return triu(M) + tril(M.T)

    def create_ensemble(self,
                        model,
                        num_graphs):
        """
        create_ensemble() samples to create [num_graphs] sample graphs, appending them to self.ensemble

        :param model: [String] either "HER" or "HSCM"
        :param num_graphs: [Int] number of sample graphs
        :return: [N X N x num_graphs Array] ensemble of graphs
        """

        if model == "HSCM":
```

```python
            for _ in tqdm(range(num_graphs), desc="Generating HSCM Ensemble: "):
                m = self.HSCM()
                self.ensemble.append(m)

        elif model == "HER":
            for _ in tqdm(range(num_graphs), desc="Generating HER Ensemble: "):
                m = self.HER()
                self.ensemble.append(m)
        else:
            raise ValueError("Graph model must be either HSCM or HER")

    def degree_distribution(self,
                            binning,
                            num_bins,
                            graph=None):

        """
        degree_distribution() calculates the degree distribution of the

        :param binning: [String] either "log" or "linear"
        :param num_bins: [Int] number of bins for the graph
        :param graph: [Optional, Array] If graph is not none, this is run over the ensemble attribute of the object. If
        graph is provided, it is run for that graph.
        :return: Tuple[Array, Array] list of bin-midpoint x values, and a list of probability density values
        """

        degrees = list()

        if graph is not None:
            G = [graph]

        else:
            G = self.ensemble

        for g in G:
            # compressing ensemble by concatenating all degrees of all graphs into one degree sequence
            degrees.extend(g.sum(axis=0).tolist()[0])

        if binning == "log":
            dist_x, dist_y = distributionBin(degrees, num_bins)

        elif binning == "linear":
            bins = np.linspace(min(degrees), max(degrees), num_bins)
            dist_x = (bins[1:] + bins[:1])/2
            dist_y, _ = np.histogram(degrees, bins=bins)

```

```
    else:
        raise ValueError("Data binning must be either log or linear.")

    return dist_x, dist_y

```