Question 1

Consider Figure 3 in node2vec: Scalable Feature Learning for Networks. Our goal for this problem is to (roughly) reproduce this figure.

Implementation of Node2Vec from given repository

```
import networkx as nx
from scipy import stats
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from collections import deque as que
import copy
from numpy.random import randint
import random
import math
import operator

from functools import reduce
from itertools import combinations_with_replacement

import numpy as np
import pkg_resources
from gensim.models import KeyedVectors
from tqdm import tqdm
from sklearn.cluster import KMeans
import os
import random
from collections import defaultdict
from abc import ABC, abstractmethod
import gensim
import pkg_resources
from joblib import Parallel, delayed
from tqdm.auto import tqdm

#from .parallel import parallel_generate_walks


def parallel_generate_walks(d_graph: dict, global_walk_length: int, num_walks: int, cpu_nu
                            sampling_strategy: dict = None, num_walks_key: str = None, wal
                            neighbors_key: str = None, probabilities_key: str = None, firs
                            quiet: bool = False) -> list:
    """
    Generates the random walks which will be used as the skip-gram input.

    :return: List of walks. Each walk is a list of nodes.
    """

    walks = list()
```

```
if not quiet:
    pbar = tqdm(total=num_walks, desc='Generating walks (CPU: {})'.format(cpu_num))

for n_walk in range(num_walks):

    # Update progress bar
    if not quiet:
        pbar.update(1)

    # Shuffle the nodes
    shuffled_nodes = list(d_graph.keys())
    random.shuffle(shuffled_nodes)

    # Start a random walk from every node
    for source in shuffled_nodes:

        # Skip nodes with specific num_walks
        if source in sampling_strategy and \
                num_walks_key in sampling_strategy[source] and \
                sampling_strategy[source][num_walks_key] <= n_walk:
            continue

        # Start walk
        walk = [source]

        # Calculate walk length
        if source in sampling_strategy:
            walk_length = sampling_strategy[source].get(walk_length_key, global_walk_l
        else:
            walk_length = global_walk_length

        # Perform walk
        while len(walk) < walk_length:

            walk_options = d_graph[walk[-1]].get(neighbors_key, None)

            # Skip dead end nodes
            if not walk_options:
                break

            if len(walk) == 1:  # For the first step
                probabilities = d_graph[walk[-1]][first_travel_key]
                walk_to = random.choices(walk_options, weights=probabilities)[0]
            else:
                probabilities = d_graph[walk[-1]][probabilities_key][walk[-2]]
                walk_to = random.choices(walk_options, weights=probabilities)[0]

            walk.append(walk_to)

        walk = list(map(str, walk))  # Convert all to strings

        walks.append(walk)

if not quiet:
```

```
        pbar.close()

    return walks


class Node2Vec:
    FIRST_TRAVEL_KEY = 'first_travel_key'
    PROBABILITIES_KEY = 'probabilities'
    NEIGHBORS_KEY = 'neighbors'
    WEIGHT_KEY = 'weight'
    NUM_WALKS_KEY = 'num_walks'
    WALK_LENGTH_KEY = 'walk_length'
    P_KEY = 'p'
    Q_KEY = 'q'

    def __init__(self, graph: nx.Graph, dimensions: int = 128, walk_length: int = 80, num_
                 q: float = 1, weight_key: str = 'weight', workers: int = 1, sampling_stra
                 quiet: bool = False, temp_folder: str = None, seed: int = None):
        """
        Initiates the Node2Vec object, precomputes walking probabilities and generates the

        :param graph: Input graph
        :param dimensions: Embedding dimensions (default: 128)
        :param walk_length: Number of nodes in each walk (default: 80)
        :param num_walks: Number of walks per node (default: 10)
        :param p: Return hyper parameter (default: 1)
        :param q: Inout parameter (default: 1)
        :param weight_key: On weighted graphs, this is the key for the weight attribute (d
        :param workers: Number of workers for parallel execution (default: 1)
        :param sampling_strategy: Node specific sampling strategies, supports setting node
        :param seed: Seed for the random number generator.
        Use these keys exactly. If not set, will use the global ones which were passed on
        :param temp_folder: Path to folder with enough space to hold the memory map of sel
        """

        self.graph = graph
        self.dimensions = dimensions
        self.walk_length = walk_length
        self.num_walks = num_walks
        self.p = p
        self.q = q
        self.weight_key = weight_key
        self.workers = workers
        self.quiet = quiet
        self.d_graph = defaultdict(dict)

        if sampling_strategy is None:
            self.sampling_strategy = {}
        else:
            self.sampling_strategy = sampling_strategy

        self.temp_folder, self.require = None, None
        if temp_folder:
            if not os.path.isdir(temp_folder):
```

```
            raise NotADirectoryError("temp_folder does not exist or is not a directory

        self.temp_folder = temp_folder
        self.require = "sharedmem"

    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)

    self._precompute_probabilities()
    self.walks = self._generate_walks()

def _precompute_probabilities(self):
    """
    Precomputes transition probabilities for each node.
    """

    d_graph = self.d_graph

    nodes_generator = self.graph.nodes() if self.quiet \
        else tqdm(self.graph.nodes(), desc='Computing transition probabilities')

    for source in nodes_generator:

        # Init probabilities dict for first travel
        if self.PROBABILITIES_KEY not in d_graph[source]:
            d_graph[source][self.PROBABILITIES_KEY] = dict()

        for current_node in self.graph.neighbors(source):

            # Init probabilities dict
            if self.PROBABILITIES_KEY not in d_graph[current_node]:
                d_graph[current_node][self.PROBABILITIES_KEY] = dict()

            unnormalized_weights = list()
            d_neighbors = list()

            # Calculate unnormalized weights
            for destination in self.graph.neighbors(current_node):

                p = self.sampling_strategy[current_node].get(self.P_KEY,
                                                             self.p) if current_node i
                q = self.sampling_strategy[current_node].get(self.Q_KEY,
                                                             self.q) if current_node i

                if destination == source:  # Backwards probability
                    ss_weight = self.graph[current_node][destination].get(self.weight_
                elif destination in self.graph[source]:  # If the neighbor is connecte
                    ss_weight = self.graph[current_node][destination].get(self.weight_
                else:
                    ss_weight = self.graph[current_node][destination].get(self.weight_

                # Assign the unnormalized sampling strategy weight, normalize during r
                unnormalized_weights.append(ss_weight)
                d_neighbors.append(destination)
```

```
            # Normalize
            unnormalized_weights = np.array(unnormalized_weights)
            d_graph[current_node][self.PROBABILITIES_KEY][
                source] = unnormalized_weights / unnormalized_weights.sum()

        # Calculate first_travel weights for source
        first_travel_weights = []

        for destination in self.graph.neighbors(source):
            first_travel_weights.append(self.graph[source][destination].get(self.weigh

        first_travel_weights = np.array(first_travel_weights)
        d_graph[source][self.FIRST_TRAVEL_KEY] = first_travel_weights / first_travel_w

        # Save neighbors
        d_graph[source][self.NEIGHBORS_KEY] = list(self.graph.neighbors(source))

def _generate_walks(self) -> list:
    """
    Generates the random walks which will be used as the skip-gram input.
    :return: List of walks. Each walk is a list of nodes.
    """

    flatten = lambda l: [item for sublist in l for item in sublist]

    # Split num_walks for each worker
    num_walks_lists = np.array_split(range(self.num_walks), self.workers)

    walk_results = Parallel(n_jobs=self.workers, temp_folder=self.temp_folder, require
        delayed(parallel_generate_walks)(self.d_graph,
                                         self.walk_length,
                                         len(num_walks),
                                         idx,
                                         self.sampling_strategy,
                                         self.NUM_WALKS_KEY,
                                         self.WALK_LENGTH_KEY,
                                         self.NEIGHBORS_KEY,
                                         self.PROBABILITIES_KEY,
                                         self.FIRST_TRAVEL_KEY,
                                         self.quiet) for
        idx, num_walks
        in enumerate(num_walks_lists, 1))

    walks = flatten(walk_results)

    return walks

def fit(self, **skip_gram_params) -> gensim.models.Word2Vec:
    """
    Creates the embeddings using gensim's Word2Vec.
    :param skip_gram_params: Parameters for gensim.models.Word2Vec - do not supply 'si
        taken from the Node2Vec 'dimensions' parameter
    :type skip_gram_params: dict
    :return: A gensim word2vec model
```

```
        """

        if 'workers' not in skip_gram_params:
            skip_gram_params['workers'] = self.workers

        # Figure out gensim version, naming of output dimensions changed from size to vect
        gensim_version = pkg_resources.get_distribution("gensim").version
        size = 'size' if gensim_version < '4.0.0' else 'vector_size'
        if size not in skip_gram_params:
            skip_gram_params[size] = self.dimensions

        if 'sg' not in skip_gram_params:
            skip_gram_params['sg'] = 1

        return gensim.models.Word2Vec(self.walks, **skip_gram_params)

class EdgeEmbedder(ABC):
    INDEX_MAPPING_KEY = 'index2word' if pkg_resources.get_distribution("gensim").version <

    def __init__(self, keyed_vectors: KeyedVectors, quiet: bool = False):
        """
        :param keyed_vectors: KeyedVectors containing nodes and embeddings to calculate ed
        """

        self.kv = keyed_vectors
        self.quiet = quiet

    @abstractmethod
    def _embed(self, edge: tuple) -> np.ndarray:
        """
        Abstract method for implementing the embedding method
        :param edge: tuple of two nodes
        :return: Edge embedding
        """
        pass

    def __getitem__(self, edge) -> np.ndarray:
        if not isinstance(edge, tuple) or not len(edge) == 2:
            raise ValueError('edge must be a tuple of two nodes')

        if edge[0] not in getattr(self.kv, self.INDEX_MAPPING_KEY):
            raise KeyError('node {} does not exist in given KeyedVectors'.format(edge[0]))

        if edge[1] not in getattr(self.kv, self.INDEX_MAPPING_KEY):
            raise KeyError('node {} does not exist in given KeyedVectors'.format(edge[1]))

        return self._embed(edge)

    def as_keyed_vectors(self) -> KeyedVectors:
        """
        Generated a KeyedVectors instance with all the possible edge embeddings
        :return: Edge embeddings
        """

        edge_generator = combinations_with_replacement(getattr(self.kv, self.INDEX_MAPPING
```

```
        if not self.quiet:
            vocab_size = len(getattr(self.kv, self.INDEX_MAPPING_KEY))
            total_size = reduce(lambda x, y: x * y, range(1, vocab_size + 2)) / \
                         (2 * reduce(lambda x, y: x * y, range(1, vocab_size)))

            edge_generator = tqdm(edge_generator, desc='Generating edge features', total=t

        # Generate features
        tokens = []
        features = []
        for edge in edge_generator:
            token = str(tuple(sorted(edge)))
            embedding = self._embed(edge)

            tokens.append(token)
            features.append(embedding)

        # Build KV instance
        edge_kv = KeyedVectors(vector_size=self.kv.vector_size)
        if pkg_resources.get_distribution("gensim").version < '4.0.0':
            edge_kv.add(
                entities=tokens,
                weights=features)
        else:
            edge_kv.add_vectors(
                keys=tokens,
                weights=features)

        return edge_kv


class AverageEmbedder(EdgeEmbedder):
    """
    Average node features
    """

    def _embed(self, edge: tuple):
        return (self.kv[edge[0]] + self.kv[edge[1]]) / 2


class HadamardEmbedder(EdgeEmbedder):
    """
    Hadamard product node features
    """

    def _embed(self, edge: tuple):
        return self.kv[edge[0]] * self.kv[edge[1]]


class WeightedL1Embedder(EdgeEmbedder):
    """
    Weighted L1 node features
    """
```

```
    def _embed(self, edge: tuple):
        return np.abs(self.kv[edge[0]] - self.kv[edge[1]])


class WeightedL2Embedder(EdgeEmbedder):
    """
    Weighted L2 node features
    """

    def _embed(self, edge: tuple):
        return (self.kv[edge[0]] - self.kv[edge[1]]) ** 2
```

• Download the Les Mis´erables coapperance network and read it into NetworkX (the read_gml function should be useful here).

• Use the parameters detailed in section 4.1 of node2vec: Scalable Feature Learning for Networks to construct two embeddings of this network.

```
# FILES
EMBEDDING_FILENAME_COMM = './embeddings_comm.emb'
EMBEDDING_MODEL_FILENAME_COMM = './embeddings_comm.model'

EMBEDDING_FILENAME_STRUCT_EQ = './embeddings_struct_eq.emb'
EMBEDDING_MODEL_FILENAME_STRUCT_EQ = './embeddings_struct_eq.model'

graph = nx.read_gml('/content/lesmis.gml',label='id')
graph = nx.Graph(graph)
graph.remove_edges_from(nx.selfloop_edges(graph))

print(graph.number_of_edges())
print(graph.number_of_nodes())

# Communities
node2vec_comm = Node2Vec(graph, dimensions=16, walk_length=30, num_walks=100, workers=1, p
model_comm = node2vec_comm.fit(window=10, min_count=1, batch_words=4)  # Any keywords acce

#Structual Equivalence
node2vec_struct_eq = Node2Vec(graph, dimensions=16, walk_length=3, num_walks=100, workers=
model_struct_eq = node2vec_struct_eq.fit(window=3, min_count=1, batch_words=3)  # Any keyw

# Save embeddings for later use
model_comm.wv.save_word2vec_format(EMBEDDING_FILENAME_COMM)

# Save model for later use
model_comm.save(EMBEDDING_MODEL_FILENAME_COMM)

# Save embeddings for later use
model_struct_eq.wv.save_word2vec_format(EMBEDDING_FILENAME_STRUCT_EQ)

# Save model for later use
model_struct_eq.save(EMBEDDING_MODEL_FILENAME_STRUCT_EQ)
```

```
254
77
```

```
Computing transition probabilities:                    77/77 [00:00<00:00,
100%                                                   875.70it/s]
Generating walks (CPU: 1):                             100/100 [00:01<00:00,
100%                                                   63.06it/s]
```

.

.

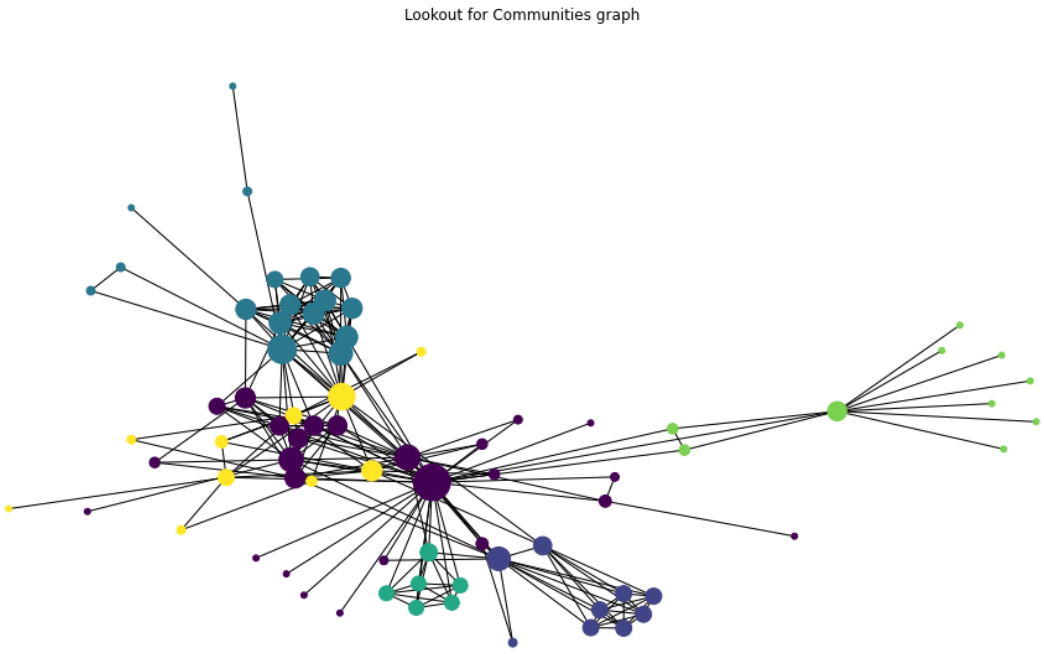
.

.

• Apply k-means clustering with k = 6 on the embedding created with p = 1 and q = 0.5 (looking for communities). This should give you six groups of nodes with different labels.

• Finally, use NetworkX to generate two visualizations of this network with nodes in the same cluster given the same color. Your visualizations do not need to look exactly like those in the paper. You should, however, be able to pick out similar groups of nodes.

```
nodes_comm = pd.read_csv('/content/embeddings_comm.emb', skiprows=range(0, 1), sep=' ', he
#nodes_comm = pd.read_csv('/content/embeddings_struct_eq.emb', skiprows=range(0, 1), sep='
nodes_comm = nodes_comm.sort_values(by=0).reset_index(drop=True)
cluster_comm = KMeans(n_clusters=6)
cluster_comm.fit(nodes_comm.iloc[:, 1:])
cluster_comm.labels_
pos_comm = nx.drawing.layout.spring_layout(graph)
#pos_comm=nx.draw(graph)
plt.figure(figsize=(18,10))
ax = plt.gca()
ax.set_title('Lookout for Communities graph')
d = dict(graph.degree)
nx.draw_networkx_edges(graph, pos=pos_comm, alpha=0.1)
nx.draw(graph, pos=pos_comm, with_labels=False, nodelist=sorted(graph.nodes()),
        node_color=cluster_comm.labels_, node_size=[v*24 for v in d.values()], ax=ax)
```



.

• Apply k-means clustering with k = 3 on the embedding with p = 1 and q = 2 (looking for structural equivalence). This should give you three groups of nodes with different labels.

• Finally, use NetworkX to generate two visualizations of this network with nodes in the same cluster given the same color. Your visualizations do not need to look exactly like those in the paper. You should, however, be able to pick out similar groups of nodes.

```
nodes_struct_eq = pd.read_csv('/content/embeddings_struct_eq.emb', skiprows=range(0, 1), s
#nodes_struct_eq = pd.read_csv('/content/embeddings_comm.emb', skiprows=range(0, 1), sep='
nodes_struct_eq = nodes_struct_eq.sort_values(by=0).reset_index(drop=True)
cluster_struct_eq = KMeans(n_clusters=3)
cluster_struct_eq.fit(nodes_struct_eq.iloc[:, 1:])
cluster_struct_eq.labels_
pos_struct_eq = nx.drawing.layout.spring_layout(graph)
plt.figure(figsize=(18,10))
ax = plt.gca()
ax.set_title('Structural Equivalence graph')
d = dict(graph.degree)
nx.draw_networkx_edges(graph, pos=[v for v in pos_struct_eq.values()], alpha=0.1)
nx.draw(graph, pos=pos_struct_eq, with_labels=False, nodelist=sorted(graph.nodes()),
        node_color=cluster_struct_eq.labels_, node_size=[v*24 for v in d.values()], ax=ax)
```

Structural Equivalence graph

Question 5

Generate a figure showing the average harmonic centrality of nodes in Erd¨os-R´enyi random graphs with 1000 nodes and p ranging from 0.05 to 0.95 in increments of 0.05. For each value of p you only need to consider a single graph but take an average over all nodes in the graph. In particular, values of p will be on the x-axis and average harmonic centralities should be on the y-axis. You may use the harmonic_centrality function available in NetworkX

Answer

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

Function to plot the mean hormanic centrality on Y axis vs Probablity(p) on X-axis

```
# plot the mean hormanic centrality on Y axis vs Probablity(p) on X-axis
def plot_graph_linear_log_axis(plot_data_x,plot_data_y,linearLog):
 # create plot on linear axis
 fig, ax = plt.subplots(figsize=(12, 6))
 ax.scatter(plot_data_x, plot_data_y)
 ax.set_title('Mean hormanic centrality Vs Probablity - Linear Scale')
 ax.set_xlabel('Probablity')
 ax.set_ylabel('Mean hormanic centrality')
 if linearLog:
  plt.yscale('log')
  plt.xscale('log')
  ax.set_title('Mean hormanic centrality Vs Probablity- Log Scale')
 plt.show()
```

Function to find avgrage hormonic centrality

```
#Function to find avgrage hormonic centrality
def find_avg_hormanic_centrality(G,plot_data_y):
 #calculate average neighbor degree
 avg_harmonic_centrality = nx.harmonic_centrality(G)
 mean_harmonic_centrality = 0
 n = len(G.nodes)
 for val in avg_harmonic_centrality:
  mean_harmonic_centrality += avg_harmonic_centrality[val]
  mean_harmonic_centrality /= n
```

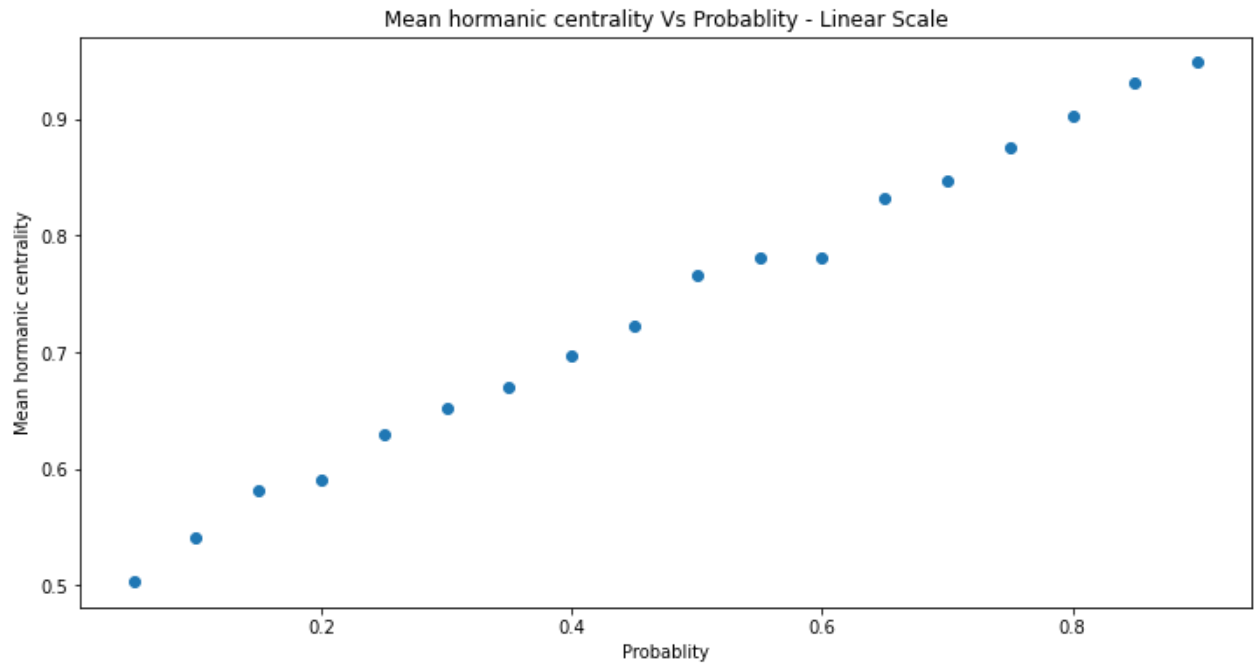
```
 plot_data_y.append(mean_harmonic_centrality)
 return plot_data_y
```

Find average harmonic centrality of nodes in Erd¨os-R´enyi random graphs with 1000 nodes and p ranging from 0.05 to 0.95 in increments of 0.05.

```
#Find average harmonic centrality of nodes in Erd¨os-R´enyi random graphs
#with 1000 nodes and p ranging from 0.05 to 0.95 in increments of 0.05.
plot_data_x = []
plot_data_y = []
for i in np.arange(0.05, 0.95, 0.05):
  plot_data_x.append(i)
  G= nx.erdos_renyi_graph(1000,i)
  plot_data_y = find_avg_hormanic_centrality(G,plot_data_y)
```

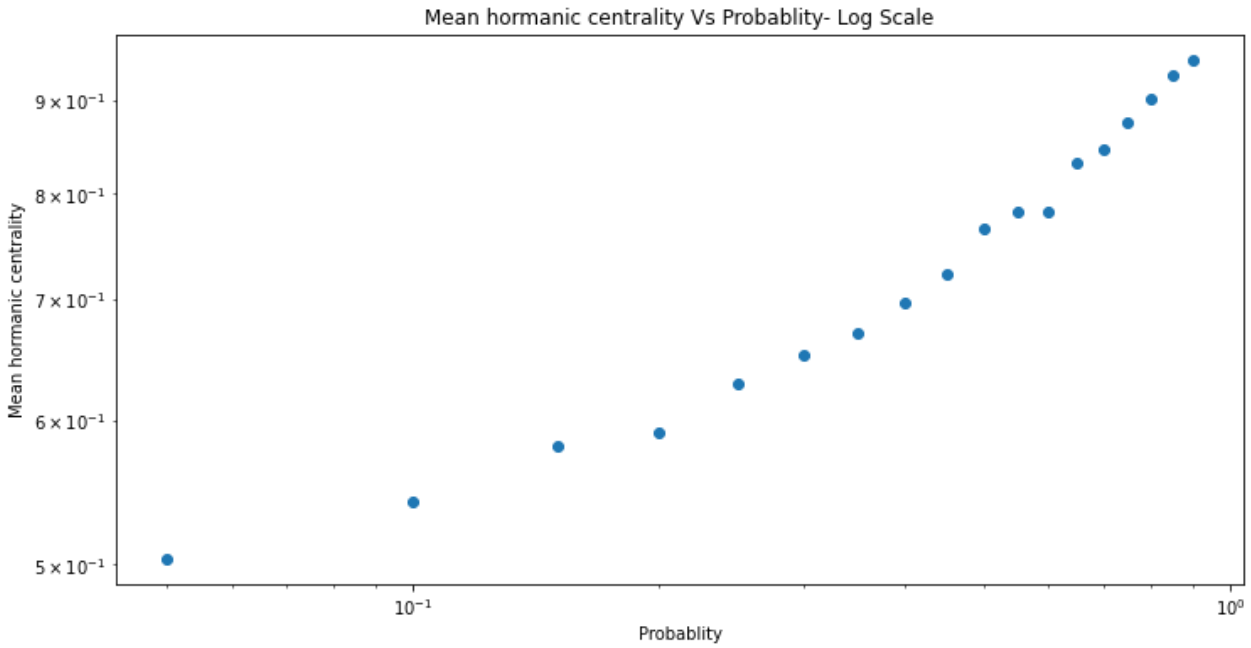
Plot the mean hormanic centrality on Y axis vs probablity(p) on X-axis with linear scale

```
plot_graph_linear_log_axis(plot_data_x,plot_data_y,False)
```



Plot the mean hormanic centrality on Y axis vs probablity(p) on X-axis with log scale

```
plot_graph_linear_log_axis(plot_data_x,plot_data_y,True)
```


Mean hormanic centrality Vs Probablity- Log Scale
Mean hormanic centrality
Probablity
9×10^{-1}
8×10^{-1}
7×10^{-1}
6×10^{-1}
5×10^{-1}
10^{-1}
10^{0}

.

.

Q2. (The K¨onigsberg Problem from section 2.12 of Network Science) Which of the icons in image 2.19 can be drawn without raising your pencil from the paper, and without drawing any line more than once? Why?

Answer:

It isn't possible to solve the bridge problem if there are four vertices with an odd degree. According to Euler's proof, we could only solve it if either all the vertices in the graph were even, or if only two of the vertices were odd.

So, in the images shown in figure 2.12

Fig (a) --> This figure has total 4 nodes,

Two nodes in a graph are having degree of 2 (Even Degree)

Two nodes in a graph are having degree of 3 (Odd Degree)

Hence, it **can be drawn** using without raising your pencil from the paper, and without drawing any line more than once

fig (b) --> This figure has 5 nodes

Four nodes in a graph are having degree of 3 (Odd Degree)

One node in a graph is having degree of 4 (Even Degree)

Hence, it **cannot be drawn** using without raising your pencil from the paper, and without drawing any line more than once

fig (c) --> This figure has 12 nodes where 6 nodes are having degree 4 (even) and 6 nodes are having degree 2 (even)

Six nodes in a graph are having degree of 4 (Even Degree)

Six nodes in a graph are having degree of 2 (Even Degree)

There are no nodes with odd degree

Hence, **it can be drawn** using without raising your pencil from the paper, and without drawing any line more than once

fig (d) --> This figure has 6 nodes

Two nodes in a graph are having degree of 2 (Even Degree)

Two nodes in a graph are having degree of 4 (Even Degree)

One node in a graph is having degree of 1 (Odd Degree)

One node in a graph is having degree of 3 (Odd Degree)

The graph contains 4 nodes having even degree and 2 nodes having odd degree

Hence, it **can be drawn** using without raising your pencil from the paper, and without drawing any line more than once

Q3. What is the probability that a random node has degree 95 in $G_{1000,0.1}$?

Answer:

The formula to find the probability that a given vertex is of degree k is

$$\text{Prob}(k) = \binom{n-1}{K} P^k (1-P)^{n-1-k}$$

By substituting the values in formula,

where, p = 0.1 n=1000 k=95

$$\text{Prob}(k) = (999!/95!\,(999-95)!)\,(0.1)^{95}\,(1-0.1)^{(999-95)}$$

$$= (999!/95! * (904)!)\,(0.1)^{95}\,(0.9)^{(904)}$$

$$= (8.69 * 10^{134}) * (10^{-95}) * (4.32 * 10^{-42}) = 37.54 * 10^{-3}$$

$$= 3.75 * 10^{-2}$$

Hence the probability that a random node has degree 95 in $G_{1000,0.1}$ is **$3.75 * 10^{-2}$**

Q4. What are the four regimes associated with the connectedness of Erd¨os-R´enyi random graphs? What features do we observe in graphs belonging to each regime?

Answer:

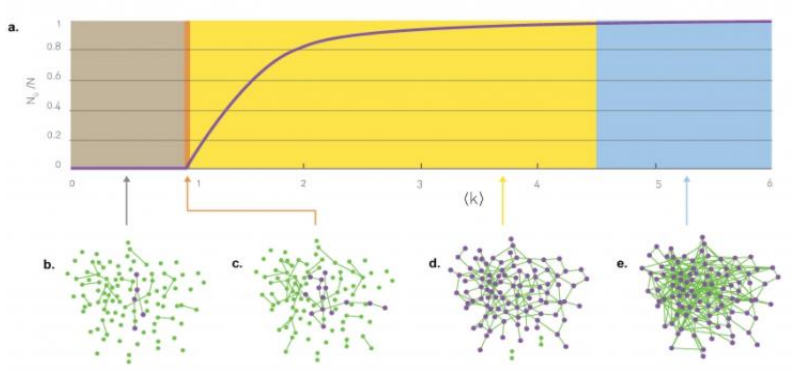
Below are the regimes associated with Erd¨os-R´enyi random graphs

1) Subcritical Regime: $0 < \langle k \rangle < 1$ ($p < 1/N$) (Figure b)

2) Critical Point: $\langle k \rangle = 1$ ($p = 1/N$). (Figure c)

3) Supercritical Regime: $\langle k \rangle > 1$ ($p > 1/N$). (Figure d)

4) Connected Regime: $\langle k \rangle > \ln N$ ($p > \ln N/N$). (Figure e)



Subcritical Regime: 0 ‹ ‹k› ‹ 1 (p ‹ 1/N)

- For ‹k› = 0 the network consists of N isolated nodes. Increasing ‹k› means that we are adding N‹k› = pN(N-1)/2 links to the network.
- Given that ‹k› ‹ 1, we have only a small number of links in this regime, hence we mainly observe tiny clusters
- We can designate at any moment the largest cluster to be the giant component.
- In this regime the relative size of the largest cluster, N_G/N, remains zero. The reason is that for ‹k› ‹ 1 the largest cluster is a tree with size $N_G \sim \ln N$, hence its size increases much slower than the size of the network. Therefore $N_G/N \simeq \ln N/N \to 0$ in the $N \to \infty$ limit.
- In summary, in the subcritical regime the network consists of numerous tiny components, whose size follows the exponential distribution. Hence these components have comparable sizes, lacking a clear winner that we could designate as a giant component.

Critical Point: ‹k› = 1 (p = 1/N).

- The critical point separates the regime where there is not yet a giant component (‹k› ‹ 1) from the regime where there is one (‹k› › 1).
- At this point the relative size of the largest component is still zero. The size of the largest component is $N_G \sim N^{2/3}$.
- N_G grows much slower than the network's size, so its relative size decreases as $N_G/N \sim N^{-1/3}$ in the $N \to \infty$ limit.
- In summary, at the critical point most nodes are located in numerous small components, The power law form indicates that components of different sizes coexist. These numerous small components are mainly trees, while the giant component may contain loops.

Supercritical Regime: ‹k› › 1 (p › 1/N).

- This regime has the most relevance to real systems, as for the first time we have a giant component that looks like a network.
- In the vicinity of the critical point the size of the giant component varies as $(N_G/N) \sim \langle k \rangle - 1$ or $N_G \sim (p - pc)N$ where pc is given by $pc = 1/N - 1 \approx 1/N$.
- In other words, the giant component contains a finite fraction of the nodes. The further we move from the critical point; a larger fraction of nodes will belong to it.
- In summary in the supercritical regime numerous isolated components coexist with the giant component. These small components are trees, while the giant component contains loops and cycles. The supercritical regime lasts until all nodes are absorbed by the giant component.

Connected Regime: ‹k› › lnN (p › lnN/N).

- For sufficiently large p the giant component absorbs all nodes and components, hence $N_G \simeq N$.
- In the absence of isolated nodes, the network becomes connected. The average degree at which this happens depends on N where $\langle k \rangle = \ln N$
- In summary, the random network model predicts that the emergence of a network is not a smooth, gradual process. The isolated nodes and tiny components observed for small ‹k› collapse into a giant component through a phase transition. As we vary ‹k› we encounter four topologically distinct regimes.

Q6. What is the approximate degree distribution of a scale-free network? In other words what is *P*(deg(*v*) = *k*) approximately?

Answer:

Scale-free networks are a type of network characterized by the presence of large hubs. It is one with a power-law degree distribution. For an undirected network, we can just write the degree distribution as

$$P_{deg}(k) \propto k^{-\gamma}$$

where γ is exponent.

This form of $P_{deg}(k)$ decays slowly as the degree k increases

As node degrees are positive integers, k = 0, 1, 2, ..., the discrete formalism provides the probability pk that a node has exactly k links

$P_k = C\, k^{-\gamma}$

The constant C is determined by the normalization condition

$$\sum_{k=1}^{\infty} p_k = 1$$

$$C \sum_{k=1}^{\infty} k^{-\gamma} = 1$$

$$C = \frac{1}{\sum_{k=1}^{\infty} k^{-\gamma}} = \frac{1}{\zeta(\gamma)}$$

where ζ (γ) is the Riemann-zeta function. Thus, for k > 0 the discrete power-law distribution has the form

$$p_k = \frac{k^{-\gamma}}{\zeta(\gamma)}$$

Here, P_k diverges at k=0. If needed, we can separately specify P_0, representing the fraction of nodes that have no links to other nodes. In that case the calculation of C in constant equation needs to incorporate P_0.

Q 7. What are the four regimes associated with the degree exponent of scale-free networks? What are distinguishing features of networks in each regime?

Answer:

Below are the four regimes associated with the degree exponent of scale-free networks

Anomalous Regime (γ ≤ 2):

- For values of γ < 2, the value of 1/(γ-1) is greater than 1.

$$K_{max} = K_{min}\, N^{1/(\gamma-1)}$$

- This implies that the largest hub should have a degree greater than N.
- for γ = 2 the degree of the biggest hub grows linearly with the system size, i.e. $K_{max} \sim N$.
- This forces the network into a hub and spoke configuration in which all nodes are close to each other because they all connect to the same central hub.

- In this regime the average path length does not depend on N.
- For this, the hub should have self-loops and/or multiple links between the hub and the other nodes.
- Neither of these are common in real networks. Hence, it is very rare to find real networks whose degree distribution fit to a power law with $\gamma <= 2$.

Ultra-Small world or Scale-Free Regime ($2 < \gamma < 3$):

- In this regime the average distance increases as lnlnN, a significantly slower growth than the lnN derived for random networks.
- We call networks in this regime ultra-small, as the hubs radically reduce the path length.
- They do so by linking to a large number of small-degree nodes and creating short distances between them
- The first moment is finite, whereas the second and higher moments diverge as $N \rightarrow \infty$.
- K_{max} grows with the size of the network with exponent $1/(\gamma-1)$ of value less than 1.

Critical Point ($\gamma = 3$)

- This value is of theoretical interest, as the second moment of the degree distribution does not diverge any longer.
- Therefore, $\gamma = 3$ the critical point.
- At this critical point the lnN dependence encountered for random networks returns.
- The calculations indicate the presence of a double logarithmic correction, which shrinks the distances compared to a random network of similar size.

Small World Regime ($\gamma > 3$):

- In this regime $\langle k^2 \rangle$ is finite and the average distance follows the small world result derived for random networks.
- Hubs continue to be present, for $\gamma > 3$ they are not sufficiently large and numerous to have a significant impact on the distance between the nodes.
- The more pronounced the hubs are, the more effectively they shrink the distances between nodes.
- It shows the scaling of the average path length for scale-free networks with different γ.
- For large γ, the degree distribution decays sufficiently fast to make the hubs smaller and less numerous (characteristic of random networks) uncharacteristic of real networks.

Q8. Clearly describe the algorithm behind the Barab´asi-Albert model. Think about parameters and starting conditions. What are the two main features of this model that lead to scale-free networks?

Answer:

- The recognition that growth and preferential attachment coexist in real networks has inspired a minimal model called the Barabási-Albert model, which can generate scale-free networks. Also known as the BA model or the scale-free model,
- We start with m0 nodes, the links between which are chosen arbitrarily, as long as each node has at least one link.
- The two main feature of Barabasi Albert Model are Growth and Preferential attachment
    - Growth
    - Preferential attachment

It is defined as below:

- **Growth** means that the number of nodes in the network increases over time.
- **Preferential attachment** means that the more connected a node is, the more likely it is to receive new links. Nodes with a higher degree have a stronger ability to grab links added to the network.

Algorithm

- The network begins with an initial connected network of m0 nodes.
- New nodes are added to the network one at a time.
- Each new node is connected to m<=m0 existing nodes with a probability that is proportional to the number of links that the existing nodes already have.
- Formally, the probability π (k_i) that a link of the new node connects to node i depends on the degree k_i as

$$\Pi(k_i) = \frac{k_i}{\sum_j k_j}$$

Where k_i = degree of node

k_j = sum is made over all pre-existing nodes j (i.e. the denominator results in twice the current number of edges in the network).

- Heavily linked nodes "hubs" tend to quickly accumulate even more links, while nodes with only a few links are unlikely to be chosen as the destination for a new link.
- The new nodes have a "preference" to attach themselves to the already heavily linked nodes.
- In summary, the Barabási-Albert model indicates that two simple mechanisms, **growth and preferential attachment**, are responsible for the emergence of scale-free networks. The origin of the power law and the associated hubs is a rich-gets-richer phenomenon induced by the coexistence of these two ingredients

Q9. What key characteristic of scale-free networks allow information and disease to spread very quickly over these structures? Give an intuitive answer and an answer referring to the characteristic time of a spread and the gamma parameter associated with scale-free networks.

Answer:

- Scale free networks are characterized by the existence of hubs which are individuals with an unusually large number of connections, whereas most of the elements have very few links.
- There are different factors that characterize a Scale free network, such as the number of connections an individual has. These factors can alter the behavior of the virus in the scale-free network.
- One other factor is the assortativeness of the network. The individuals in an assortative network are connected preferentially to other similar individuals.
- This similarity can be expressed by the number of connections, which in a simplified way means that in an assortative network the hubs will be connected to other hubs, while in a disassortative network the hubs are not connected to each other.
- This influences on the spread of disease in the network.

Scale-free Network with $\gamma \geq 3$

- If the contract network on which the disease spreads is scale-free with degree exponent $\gamma \geq 3$, both $\langle k \rangle$ and $\langle k^2 \rangle$ are finite.
- Consequently, the characteristic time for the spread of the disease is also finite and the spreading dynamics is similar to the behavior predicted for a random network but with an altered characteristic time for the spread of the disease

Scale-free Networks with γ≤3

- For $\gamma < 3$ in the $N \to \infty$ limit $\langle k^2 \rangle \to \infty$ hence characteristic time for the spread of the disease predicts to zero.
- In other words, the spread of a disease on a scale-free network is instantaneous. This is perhaps the most unexpected prediction of network epidemics.

- In Summary, In a scale-free network the hubs are the first to be infected, as through the many links they have, they are very likely to be in contact with an infected node.
- Once a hub becomes infected, It "broadcasts" the disease to the rest of the network, turning into a super-spreader.