# BrunoFBessa_5881890_P5_codigo

May 24, 2021

## 0.1 SFI5904 - Redes Complexas

Projeto Prático 5: Análise e comparação de tipos de modelos Primeiro Semestre de 2021

Docente: Luciano da Fontoura Costa (luciano@ifsc.usp.br)

Estudante: Bruno F. Bessa (num. 5881890, bruno.fernandes.oliveira@usp.br) Universidade de São Paulo, São Carlos, Brasil.

Escopo do projeto:

Comparar e discutir contidamente os 4 modelos implementados usando as propriedades (medidas) respectivamente obtidas. Considerar os histogramas de frequência relativa obtidos.

[93]:
```
# Importação de bibliotecas necessárias para o processamento e visualização

import random
import numpy as np
import pandas as pd
import scipy
import math
import networkx as nx
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
import secrets
from IPython import display

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

[3]:
```
# Definição das redes

class Point():

    """
    Define os pontos (nodos) do grafo.
    A rede de Erdos-Renyi será formada por um conjunto de objetos desta classe.
    """
```

```
    def __init__(self, index: str) -> None:
        self.index = index
        self.neighbors = []

    def __repr__(self) -> None:
        return repr(self.index)


def erdos_renyi(N: int,
                p: float,
                plot: bool = True,
                file_name: str = None) -> nx.classes.graph.Graph:

    """
    Define as conexões (i,j) = (j,i) para todos os pares de pontos com base em␣
 ↪um
    evento medida aleatória para probabilidade p, recebida como parâmetro na␣
 ↪construção da rede.
    """

    G = nx.Graph()
    nodes = [Point(i) for i in range(N)]
    edges = [(i, j) for i in range (N) for j in range(i) if random.random() < p]

    # Configura e adiciona arestas (edges) na rede
    for (i, j) in edges:
        nodes[i].neighbors.append(nodes[j])
        nodes[j].neighbors.append(nodes[i])
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # Para calcularmos medidas de distância precisaremos remover nós não␣
 ↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    # Opção de visualização da rede gerada (não utilizar para séries grandes de␣
 ↪experimentos)
    if plot:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
 ↪with_labels=False)
```

```
        plt.suptitle("Erdos-Renyi Network (N={}, p={})".format(N, p),␣
↪fontsize=15)
        plt.show(fig_net)
    if file_name != None:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Erdos-Renyi Network (N={}, p={})".format(N, p),␣
↪fontsize=15)
        plt.savefig("images/"+file_name)
        plt.close(fig_net)

    return G


class SpatialPoint():

    """
    Define os pontos do grafo em um plano 2d.
    As redes espaciais serão formadas por um conjunto de objetos desta classe.
    """

    def __init__(self, index: str, box_size: int) -> None:
        self.index = index
        self.x = random.uniform(0, 1) * box_size
        self.y = random.uniform(0, 1) * box_size

    def get_coordinates(self) -> np.array:
        return np.array([(self.x, self.y)])

    def get_distance(self, other) -> float:
        p1_coord = self.get_coordinates()
        p2_coord = other.get_coordinates()

        dist = scipy.spatial.distance.cdist(p1_coord, p2_coord, 'euclidean')
        return dist[0][0]

    def __repr__(self) -> None:
        return repr([(self.x, self.y)])


def spatial_network_voronoi(N: int,
                            box_size: int = 1,
                            plot: bool = True,
                            file_name: str = None) -> nx.classes.graph.Graph:

    """
```

```
    Define as conexões (i,j) = (j,i) para todos os pares de pontos se eles␣
↪possuem fronteiras adjacentes
    nas células de Voronoi.

    """

    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))

    # Cálculo das fronteiras de Voronoi:
    vor = scipy.spatial.Voronoi(points2d)

    # Criação da rede baseada em céluas adjacentes
    edges = vor.ridge_points
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        fig.suptitle("Células de Voronoi e Rede para {} pontos espaciais␣
↪aleatórios".format(N), fontsize=15)
        scipy.spatial.voronoi_plot_2d(vor, ax1)
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
        plt.show()

    if file_name != None:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        fig.suptitle("Células de Voronoi e Rede para {} pontos espaciais␣
↪aleatórios".format(N), fontsize=15)
        scipy.spatial.voronoi_plot_2d(vor, ax1)
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
```

```
        plt.savefig("images/"+file_name)
        plt.close(fig)

    return G


def spatial_network_radius(N: int,
                           box_size: int = 1,
                           radius = 0.3,
                           plot: bool = True,
                           file_name: str = None) -> nx.classes.graph.Graph:

    """
    Define uma rede em que as conexões entre os nós aleatoriamente distribuidos␣
↪no espaço
    são dadas pela distância menor ou igual a um raio definido.
    """

    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))


    # Para cada par de nós (i,j) a aresta criada se distância(i,j)<=radius
    edges = [(node_i.index, node_j.index) for node_i in spatial_points for␣
↪node_j in spatial_points if node_i.get_distance(node_j) > 0 and node_i.
↪get_distance(node_j) <= radius]
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        plt.suptitle("{} pontos aleatórios conectados em uma rede se distância␣
↪entre si é <={}".format(N,radius))
```

```
        ax1.scatter([x[0] for x in points2d], [y[1] for y in points2d])
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
        plt.show()

    if file_name != None:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        plt.suptitle("{} pontos aleatórios conectados em uma rede se distância␣
↪entre si é <={}".format(N,radius))
        ax1.scatter([x[0] for x in points2d], [y[1] for y in points2d])
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
        plt.savefig("images/"+file_name)
        plt.close(fig)

    return G

def spatial_network_waxman(N: int,
                           box_size: int = 1,
                           alpha = 0.3,
                           plot: bool = True,
                           file_name: str = None) -> nx.classes.graph.Graph:
    """
    Dedine uma rede em que os pontos espaciais aleatórios são conectados na␣
↪ocorrência de um
    evento aleatório de probabilidade regulada por um parâmetro alpha e da␣
↪distância.
    """
    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))


    # Os pontos (i,j) são conectados se um evento aleatório de probabilidade p␣
↪é < exp^(distância(i,j)/alpha)
    edges = [(node_i.index, node_j.index) for node_i in spatial_points for␣
↪node_j in spatial_points if random.random() < np.exp(-1*node_i.
↪get_distance(node_j)/alpha)]
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
```

```
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        plt.suptitle("{} pontos aleatórios e a rede de Waxman com alpha={}".
↪format(N,alpha))
        ax1.scatter([x[0] for x in points2d], [y[1] for y in points2d])
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
        plt.show()

    if file_name != None:
        fig, (ax1, ax2) = plt.subplots(1, 2)
        plt.suptitle("{} pontos aleatórios e a rede de Waxman com alpha={}".
↪format(N,alpha))
        ax1.scatter([x[0] for x in points2d], [y[1] for y in points2d])
        pos = nx.spring_layout(G)
        nx.draw(G, pos, node_color='w', node_size=1, with_labels=False, ax=ax2)
        plt.savefig("images/"+file_name)
        plt.close(fig)

    return G

def regular_reticulated(N: int,
                        plot: bool = True,
                        file_name: str = None) -> nx.classes.graph.Graph:
    """
    Define uma rede reticulada regular com as propriedades:
     - cada nodo é conectado aos seus vizinhos mais próximos (horizontal e␣
↪verticalmente)
     - as bordas da rede são ligadas entre si

     Dada a escolha de representação dos nodos como posições em uma matriz,␣
↪podemos usar
     a simetria da matriz para referenciar todos os pontos como elementos de␣
↪uma única lista:

     A_ij = L[i+j*L]

     Por exmeplo, se N=10, haverá uma rede reticulada de 3x3 com arestas dadas␣
↪por:
```

```
    list_edges = [ (0,1), (1,2), (2,0),
                   (3,4), (4,5), (5,3),
                   (6,7), (7,8), (8,6),
                   (0,3), (3,6), (6,0),
                   (1,4), (4,7), (7,1),
                   (2,5), (5,8), (8,2)]
    """

    G = nx.Graph()
    L = math.floor(N**(0.5))

    # Ligações na horizontal
    for i in range(L*L):
        x = i % L
        y = i // L
        if x+1 < L and y < L:
            G.add_edge(i, i+1)
        else:
            G.add_edge(i, L*y)
    # Ligações na vertical
    for i in range(L*L):
        y = i % L
        x = i // L
        if x+1 < L and y < L:
            G.add_edge(i, i+L)
        else:
            G.add_edge(i, y)

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede reticulada gerada com {} nodos.".format(N),␣
↪fontsize=15)
        plt.show(fig_net)

    if file_name != None:
        pos = nx.spring_layout(G)
```

```
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede reticulada gerada com {} nodos.".format(N),␣
↪fontsize=15)
        plt.savefig("images/"+file_name)
        plt.close(fig_net)

    return G

def reconnect_regular(G: nx.classes.graph.Graph,
                      p: float,
                      plot: bool = True,
                      file_name: str = None) -> nx.classes.graph.Graph:
    """
    Dada uma rede regular, para cada conexão avalia com probabilidade
    p'< p (dado como parâmetro)
    a remoção da aresta e criação de nova conexão com um nó aleatório da rede.
    """
    N = len(G)
    nodes = list(G.nodes())
    for node in nodes:
        edges_node = list(G.edges(node))
        for edge in edges_node:
            if random.random() < p:
                random_node = secrets.choice(nodes)
                new_edge = (node, random_node)
                # Remove selected edge. Add new edge
                G.remove_edge(list(edge)[0], list(edge)[-1])
                G.add_edge(node, random_node)

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede de {} reconectada com prob. {}".format(N, p))
        plt.show(fig_net)

    if file_name != None:
```

```
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede de {} reconectada com prob. {}".format(N, p))
        plt.savefig("images/"+file_name)
        plt.close(fig)

    return G

def random_graph(N_random:int = 10) -> nx.classes.graph.Graph:
    """
    Define um grafo de N nodos e conexões aleatórias entre eles.
    """

    G = nx.Graph()
    for node in range(N_random+1):
        G.add_node(node)

    nodes = list(G.nodes())
    for node in nodes:
        random_node = secrets.choice(nodes)
        G.add_edge(node, random_node)

    return G

def ba_graph(N: int,
             m: int = 3,
             plot: bool = True,
             file_name: str = None) -> nx.classes.graph.Graph:
    """
    Define a rede de Barbási-Albert acrescentando a uma rede aleatória novos␣
↪nodos.
    Os nodos acrescentados são inseridos conforme a lógica de "associação␣
↪preferencial",
    com chance maior de serem ligados a nós com grau elevado.
    """

    G = random_graph()
    dict_degree = dict(G.degree())
    list_k_nodes = []
    for k_value, k_freq in dict_degree.items():
        for freq in range(k_freq):
            list_k_nodes.append(k_value)

    for node in range(len(G), N-1):
        for conn in range(m):
            random_node = secrets.choice(list_k_nodes)
```

```
            G.add_edge(node, random_node)

    # Para calcularmos medidas de distância precisaremos remover nós não␣
↪conectados
    # No trecho abaixo mantemos somente o maior componente conctado da rede.
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    if plot:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede Barási-Albert com {} nodos, m={}".format(N, m),␣
↪fontsize=15)
        plt.show(fig_net)

    if file_name != None:
        pos = nx.spring_layout(G)
        fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
↪with_labels=False)
        plt.suptitle("Rede Barási-Albert com {} nodos, m={}".format(N, m),␣
↪fontsize=15)
        plt.savefig("images/"+file_name)
        plt.close(fig_net)

    return G
```

[4]:
```
# Definições de medidas para as redes

def avg_shortest_path(G: nx.classes.graph.Graph) -> float:
    """
    Percorre todos os nodos do grafo e para cada um deles verifica o menor␣
↪caminho até todos os demais.
    Retorna a média desses valoeres.
    Disclaimer: this function uses shortest_path_length build in function from␣
↪NetworkX library.
    """
    dict_shortest_paths = nx.shortest_path_length(G)
    node_path_avg = []
    for node, paths in dict_shortest_paths:
        node_path_avg.append(sum(paths.values())/len(G.nodes()))

    return sum(node_path_avg)/len(node_path_avg)
```

```
def degree_distribution(G: nx.classes.graph.Graph) -> list:

    """
    Retorna a lista de valores de grau (k) para todos os nós da rede.
    """

    dict_degree = dict(G.degree())
    list_k = []
    for node, k_value in dict_degree.items():
        list_k.append(k_value)

    return list_k

def clustering_coef_distribution(G: nx.classes.graph.Graph) ->list:

    """
    Retorna a lista de valores de cluster coefficient (cc) para todos os nós da␣
 ↪rede.

    """

    list_cc_nodes = []
    for node in G.nodes():
        list_cc_nodes.append(nx.clustering(G, node))

    return list_cc_nodes

def spl_distribution(G: nx.classes.graph.Graph) ->list:
    """
    Retorna a lista de valores de shortest path length (spl) para todos os nós␣
 ↪da rede.

    """

    N = len(G)
    if nx.is_connected(G) == True:
        distance_matrix = np.zeros(shape=(N,N))
        diameter = nx.diameter(G)
        slp_values = []
        for i in np.arange(0,N):
            for j in np.arange(i+1, N):
                if(i != j):
                    aux = nx.shortest_path(G,i,j)
                    dij = len(aux)-1
                    distance_matrix[i][j] = dij
```

```
                    distance_matrix[j][i] = dij
                    slp_values.append(dij)
        return slp_values
    else:
        pass

def shannon_entropy(G: nx.classes.graph.Graph) ->float:

    """
    Calcula a entropia de Shannon para um grafo G recebido como parâmetro.
    """
    list_k = degree_distribution(G)
    min_k = np.min(list_k)
    max_k = np.max(list_k)

    k_values= np.arange(0,max_k+1)
    k_prob = np.zeros(max_k+1)
    for k in list_k:
        k_prob[k] = k_prob[k] + 1
    k_prob = k_prob/sum(k_prob)

    H = 0
    for p in k_prob:
        if(p > 0):
            H = H - p*math.log(p, 2)
    return H

def distribution_plot(list_values: list,
                      plot_title: str = "Histograma de densidade",
                      var_name: str = "Variável",
                      file_name: str = None) -> None:
    """
    Produz histgrama de uma medida recebida na forma de lista.
    """

    avg_value = np.mean(list_values)
    var_value = np.var(list_values)

    fig, ax = plt.subplots()
    n, bins, patches = ax.hist(list_values, density=True)
    ax.set_xlabel(var_name)
    ax.set_ylabel("Densidade de probabilidade")
    ax.set_title("{} de {}: média={:.2f}, var={:.2f}".format(plot_title,
                                                            var_name,
                                                            avg_value,
                                                            var_value),
                                                            fontsize=15)
```

```
    plt.show(True)
    if file_name != None:
        fig.savefig("images/"+file_name)

def correlation_plot(x: list,
                     y: list,
                     x_label: str = "x",
                     y_label: str = "y",
                     file_name: str = None) -> None:
    """
    Produz gráfico de dispersão de duas variáveis x e y recebidas na forma de␣
 ↪listas.
    Calcula correlação de Pearson e Spearman para x e y.
    """

    pearson_corr = np.corrcoef(x, y)[0,1]
    spearman_corr, spearman_pval = scipy.stats.spearmanr(x, y)

    fig, ax = plt.subplots()
    ax.scatter(x, y)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.set_title("Dispersão de {} e {}: Pearson: {:.2f}, Spearman: {:.2f}␣
 ↪(p-val: {:.3f})".format(x_label,
                                                                          ␣
 ↪           y_label,
                                                                          ␣
 ↪           pearson_corr,
                                                                          ␣
 ↪           spearman_corr,
                                                                          ␣
 ↪           spearman_pval),
                                                                          ␣
 ↪           fontsize=15)
    plt.show(True)
    if file_name != None:
        fig.savefig("images/"+file_name)

def simple_plot2d(x: list,
                  y: list,
                  x_label: str = "x",
                  y_label: str = "y",
                  file_name: str = None) -> None:
    """
    Produz gráfico simples com associação entre suas variáveis x e y recebidas␣
 ↪na forma de listas.
```

```
    """
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.set_title("Dispersão de {} e {}".format(x_label, y_label, fontsize=15))
    plt.show(True)
    if file_name != None:
        fig.savefig("images/"+file_name)
```

Vamos calcular as métricas para um conjunto de redes e compará-las, fixando o número de nós e grau médio.

[56]:
```
N = 500
sample = 30
list_columns = ["network_model", "avg_k", "var_k", "avg_cc", "var_cc",␣
 ↪"avg_spl", "shannon_entr"]
df = pd.DataFrame(columns=list_columns)

def cal_metrics(G: nx.classes.graph.Graph, network_model: str):
    degree_dist = degree_distribution(G)
    cc_dist = clustering_coef_distribution(G)
    spl_dist = spl_distribution(G)
    network_model = network_model
    avg_k = np.mean(degree_dist)
    var_k = np.var(degree_dist)
    avg_cc = np.mean(cc_dist)
    var_cc = np.var(cc_dist)
    avg_spl = avg_shortest_path(G)
    shannon_entr = shannon_entropy(G)

    return [network_model, avg_k, var_k, avg_cc, var_cc, avg_spl, shannon_entr]

for i in range(sample+1):
    G = erdos_renyi(N, 0.0081, False)
    list_insert_values = cal_metrics(G, "erdosrenyi")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = spatial_network_voronoi(N, 1, False)
    list_insert_values = cal_metrics(G, "spatial_voronoi")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = spatial_network_radius(N, 1, 0.0488, False)
    list_insert_values = cal_metrics(G, "spatial_radius")
    df.loc[len(df)]=list_insert_values
```

```
for i in range(sample+1):
    G = spatial_network_waxman(N, 1, 0.0275, False)
    list_insert_values = cal_metrics(G, "spatial_waxman")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = regular_reticulated(N, False)
    G = reconnect_regular(G, 0, False)
    list_insert_values = cal_metrics(G, "ws_reticulated_p_baixo")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = regular_reticulated(N, False)
    G = reconnect_regular(G, 0.2, False)
    list_insert_values = cal_metrics(G, "ws_reticulated_p_critico")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = regular_reticulated(N, False)
    G = reconnect_regular(G, 1, False)
    list_insert_values = cal_metrics(G, "ws_reticulated_p_alto")
    df.loc[len(df)]=list_insert_values

for i in range(sample+1):
    G = ba_graph(500, 2, False)
    list_insert_values = cal_metrics(G, "barbasi_albert")
    df.loc[len(df)]=list_insert_values
```

[108]:
```
#df.to_csv("dataframe_redes_complexas.csv")
```

[94]:
```
features = ["avg_k", "var_k", "avg_cc", "var_cc", "avg_spl", "shannon_entr"]
targets = ["erdosrenyi",
           "spatial_voronoi",
           "spatial_radius",
           "spatial_waxman",
           "ws_reticulated_p_baixo",
           "ws_reticulated_p_critico",
           "ws_reticulated_p_alto",
           "barbasi_albert"]
```

[96]:
```
x = df.loc[:, features].values
y = df.loc[:,['network_model']].values
x = StandardScaler().fit_transform(x)

pca_proj=PCA(n_components=2)
principalComponents=pca_proj.fit_transform(x)
```

```
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['PC1', 'PC2'])
finalDf = pd.concat([principalDf, df[['network_model']]], axis = 1)
```

[ ]:
```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('PCA', fontsize = 20)

colors = ["blue", "orange", "green", "red", "purple", "brown", "pink", "olive"]
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['network_model'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'PC1']
               , finalDf.loc[indicesToKeep, 'PC2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```