In [1]:

```
import networkx as nx
import igraph
#import community
import copy
import seaborn as sns
import numpy as np
%matplotlib inline
#from infomap import infomap
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from collections import Counter
import math
from itertools import combinations
from itertools import product
from random import shuffle
import pandas as pd
from copy import deepcopy
```

In [2]:

```
# Funcion que paso Agus
def clusterize(nx_Graph, method="infomap"):
    """
    Calcula el agrupamiento en comunidades de un grafo.

    In:
        nx_Graph: grafo de networkx
        method: metodo de clustering, puede ser: "infomap", "fastgreedy", "eigenvector",
"louvain", "edge_betweenness","label_prop", "walktrap", ""

    Out:
        labels_dict: diccionario de nodo : a label al cluster al que pertenece.
    """
    if method == "edge_betweenness":
        nx_Graph = max(nx.connected_component_subgraphs(nx_Graph), key=len)#se queda con la compone
nte más grande.
        print("AVISO: restringiendo a la componente connexa más grade. De otro modo falla el
algoritmo de detección de comunidades edge_betweenness.")

    isdirected = nx.is_directed(nx_Graph)
    np_adj_list = nx.to_numpy_matrix(nx_Graph)
    g = igraph.Graph.Weighted_Adjacency(np_adj_list.tolist(),mode=igraph.ADJ_UPPER)

    if method=="infomap":
        labels = g.community_infomap(edge_weights="weight").membership
    if method=="label_prop":
        labels = g.community_label_propagation(weights="weight").membership
    if method=="fastgreedy":
        labels = g.community_fastgreedy(weights="weight").as_clustering().membership
    if method=="eigenvector":
        labels = g.community_leading_eigenvector(weights="weight").membership
    if method=="louvain":
        labels = g.community_multilevel(weights="weight").membership
    if method=="edge_betweenness":
        labels = g.community_edge_betweenness(weights="weight", directed=isdirected).as_clustering(
).membership
    if method=="walktrap":
        labels = g.community_walktrap(weights="weight").as_clustering().membership

    label_dict = {node:label for node,label in zip(nx_Graph.nodes(), labels)}
    return label_dict
```

In [3]:

```
# Varias de visualizaciones
def drawNetwork(G):
    # position map
    pos = nx.spring_layout(G)
    # community ids
    communities = [v for k,v in nx.get_node_attributes(G, 'community').items()]
```

```
    # print(communities)
    numCommunities = max(communities) + 1
    # color map from http://colorbrewer2.org/
    cmapLight = colors.ListedColormap(['#a6cee3', '#b2df8a', '#fb9a99', '#fdbf6f', '#cab2d6'],
                                      'indexed', numCommunities)
    cmapDark = colors.ListedColormap(['#1f78b4', '#33a02c', '#e31a1c', '#ff7f00', '#6a3d9a'],
                                     'indexed', numCommunities)

    # Draw edges
    nx.draw_networkx_edges(G, pos)

    # Draw nodes
    nodeCollection = nx.draw_networkx_nodes(G,
        pos = pos,
        node_color = communities,
        cmap = cmapLight
    )
    # Set node border color to the darker shade
    darkColors = [cmapDark(v) for v in communities]
    nodeCollection.set_edgecolor(darkColors)

    # Draw node labels
    for n in G.nodes():
        plt.annotate(n,
            xy = pos[n],
            textcoords = 'offset points',
            horizontalalignment = 'center',
            verticalalignment = 'center',
            xytext = [0, 0],
            color = cmapDark(nx.get_node_attributes(G, 'community')[n])
        )

    plt.axis('off')
    plt.show()

def community_layout(g, partition):
    """
    Compute the layout for a modular graph.


    Arguments:
    ----------
    g -- networkx.Graph or networkx.DiGraph instance
        graph to plot

    partition -- dict mapping int node -> int community
        graph partitions


    Returns:
    --------
    pos -- dict mapping int node -> (float x, float y)
        node positions

    """

    pos_communities = _position_communities(g, partition, scale=3.)

    pos_nodes = _position_nodes(g, partition, scale=1.)

    # combine positions
    pos = dict()
    for node in g.nodes():
        pos[node] = pos_communities[node] + pos_nodes[node]

    return pos


def _position_communities(g, partition, **kwargs):

    # create a weighted graph, in which each node corresponds to a community,
    # and each edge weight to the number of edges between communities
    between_community_edges = _find_between_community_edges(g, partition)

    communities = set(partition.values())
    hypergraph = nx.DiGraph()
    hypergraph.add_nodes_from(communities)
```

```
    for (ci, cj), edges in between_community_edges.items():
        hypergraph.add_edge(ci, cj, weight=len(edges))

    # find layout for communities
    pos_communities = nx.spring_layout(hypergraph, **kwargs)

    # set node positions to position of community
    pos = dict()
    for node, community in partition.items():
        pos[node] = pos_communities[community]

    return pos

def _find_between_community_edges(g, partition):

    edges = dict()

    for (ni, nj) in g.edges():
        ci = partition[ni]
        cj = partition[nj]

        if ci != cj:
            try:
                edges[(ci, cj)] += [(ni, nj)]
            except KeyError:
                edges[(ci, cj)] = [(ni, nj)]

    return edges

# Otra variante
def _position_nodes(g, partition, **kwargs):
    """
    Positions nodes within communities.
    """

    communities = dict()
    for node, community in partition.items():
        try:
            communities[community] += [node]
        except KeyError:
            communities[community] = [node]

    pos = dict()
    for ci, nodes in communities.items():
        subgraph = g.subgraph(nodes)
        pos_subgraph = nx.spring_layout(subgraph, **kwargs)
        pos.update(pos_subgraph)

    return pos
```

In [4]:

```
def silhouette(red, comus):
    numcomus = len(set(comus.values()))
    nodosxcomu = [list(comus.values()).count(i) for i in set(comus.values())]
    d = nx.shortest_path(red)
    # Distancia media de cada nodo a cada comunidad.
    avgd = dict([(nodo, [0]*numcomus) for nodo in red.nodes()])

    for i, nodoi in enumerate(red.nodes()):
        for j,nodoj in enumerate(red.nodes()):
            if (j>i):
                avgd[nodoi][comus[nodoj]] += (len(d[nodoi][nodoj])-1)/(nodosxcomu[comus[nodoj]]-(co
us[nodoi]==comus[nodoj]))
                avgd[nodoj][comus[nodoi]] += (len(d[nodoj][nodoi])-1)/(nodosxcomu[comus[nodoi]]-(co
us[nodoj]==comus[nodoi]))

    # Estas son las tiras a, b y s que aparecen en el paper.
    # a = distancia media de cada nodo a todos los nodos de su comunidad.
    # b = mínima de avgd (sin contar a su propia comunidad).
    # s = silhouette de cada nodo.
    # avgs = silhouette promedio de la partición entera.
    a = dict([(nodo,avgd[nodo][comus[nodo]]) for nodo in red.nodes()])
    b = dict([(nodo,min(avgd[nodo][:comus[nodo]]+avgd[nodo][comus[nodo]+1:])) for nodo in red.nodes
()])
    s = dict([(nodo,(b[nodo]-a[nodo])/max(b[nodo],a[nodo])) for nodo in red.nodes()])
```

```
    avgs = np.mean(list(s.values()))

    return avgs, s
```

In [5]:

```
def modularidad(red, comus):
    numcomus = len(set(comus.values()))
    #nodosxcomu = [list(comus.values()).count(i) for i in set(comus.values())]
    #nodosxcomu = dict([(com, nodo) for nodo,com in comus])
    M = 0
    L = red.number_of_edges()
    Lc = [0]*numcomus
    kc = [0]*numcomus

    for edge in red.edges():
        if comus[edge[0]] == comus[edge[1]]:
            Lc[comus[edge[0]]] += 1

    for grado in red.degree():
        kc[comus[grado[0]]] += grado[1]

    for c in range(0,numcomus):
        M += Lc[c]/L - (kc[c]/(2*L))**2
    return M
```

In [55]:

```
def mod_distrib(red, comus, n):
    #print (r.degree())
    L = red.number_of_edges()
    M = [0]*n
    for i in range(n):
        nx.double_edge_swap(red,10*L,1000*L)
        M[i] = modularidad(red,comus)
    return ((np.mean(M),np.std(M,ddof=1)))

def mod_distrib_silu(red, comus, n):
    #print (r.degree())
    L = red.number_of_edges()
    M = [0]*n
    for i in range(n):
        nx.double_edge_swap(red,10*L,1000*L)
        M[i] = silhouette(red,comus)
    return ((np.mean(M),np.std(M,ddof=1)))
```

Cargamos los datos y el sexo

In [7]:

```
red_delf = nx.read_gml('./Datos/dolphins.gml')
gen_delf = open('./Datos/dolphinsGender.txt').readlines()

sex_delf = []
for i in range(len(gen_delf)):
    a = gen_delf[i].rstrip('\n').split('\t')
    sex_delf.append(a)

def atributoNodos(r, alist, atributo):

    for idx, nodo in enumerate(np.array(alist).transpose()[0]):
        r.nodes[nodo][atributo] = np.array(alist).transpose()[1][idx]

atributoNodos(red_delf, sex_delf, 'gender')
```

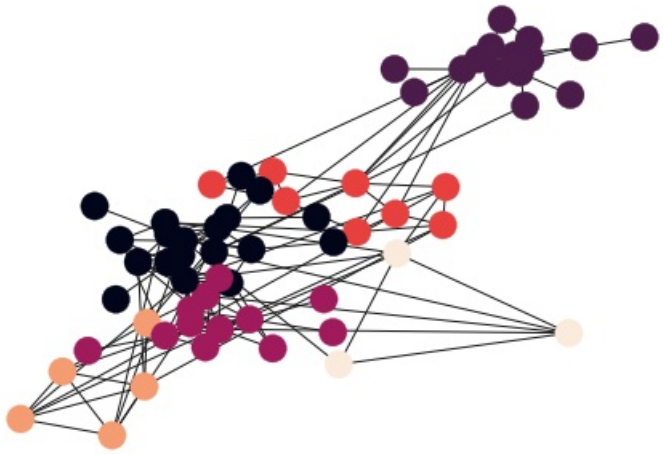
In [8]:

```
# Pequeña observación: Hay delfines de los cuales no conocemos el sexo
# tenemos que decidir que vamos a hacer con eso --> borrarlos?
# ¿Considerarlo como otra categoría?
```

Punto A

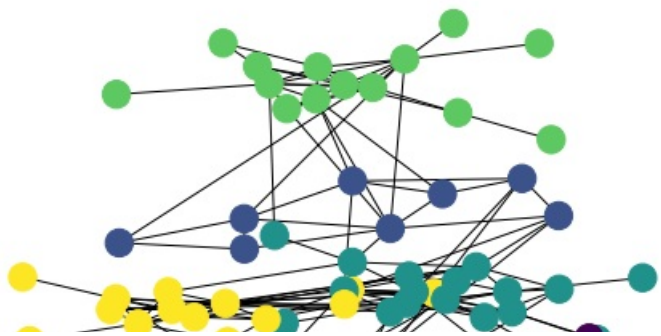
In [39]:

```
# Infomap
comus_infomap = clusterize(red_delf, "infomap") #List
nx.set_node_attributes(red_delf, name='community', values=comus_infomap)
#drawNetwork(red_delf)

pos = community_layout(red_delf,comus_infomap)
nx.draw(red_delf, pos, node_color=list(comus_infomap.values()))
plt.show()
```



In [10]:

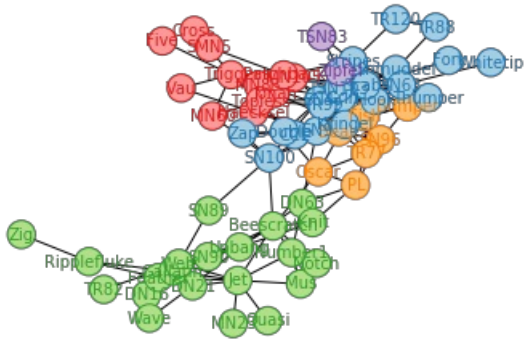
```
# Louvain
comus_louvain = clusterize(red_delf, "louvain") #List
nx.set_node_attributes(red_delf, name='community', values=comus_louvain)
drawNetwork(red_delf)

pos = community_layout(red_delf,comus_louvain)
nx.draw(red_delf, pos, node_color=list(comus_louvain.values()))
plt.show()
```

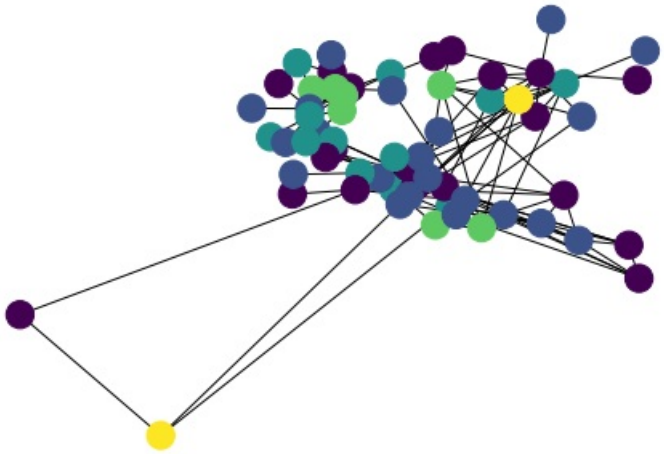


In [11]:

```
# Edge-betweenness
comus_edgeb = clusterize(red_delf, "edge_betweenness") #List
nx.set_node_attributes(red_delf, name='community', values=comus_edgeb)
drawNetwork(red_delf)

pos = community_layout(red_delf,comus_edgeb)
nx.draw(red_delf, pos, node_color=list(comus_edgeb.values()))
plt.show()
```

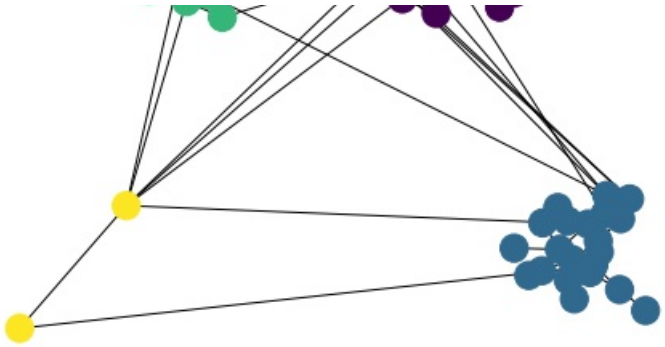
AVISO: restringiendo a la componente connexa más grade. De otro modo falla el algoritmo de detección de comunidades edge_betweenness.





In [12]:

```
# Fastgreedy
comus_fg = clusterize(red_delf, "fastgreedy") #List
nx.set_node_attributes(red_delf, name='community', values=comus_fg)
#drawNetwork(red_delf)

pos = community_layout(red_delf,comus_fg)
nx.draw(red_delf, pos, node_color=list(comus_fg.values()))
plt.show()
```



Punto B

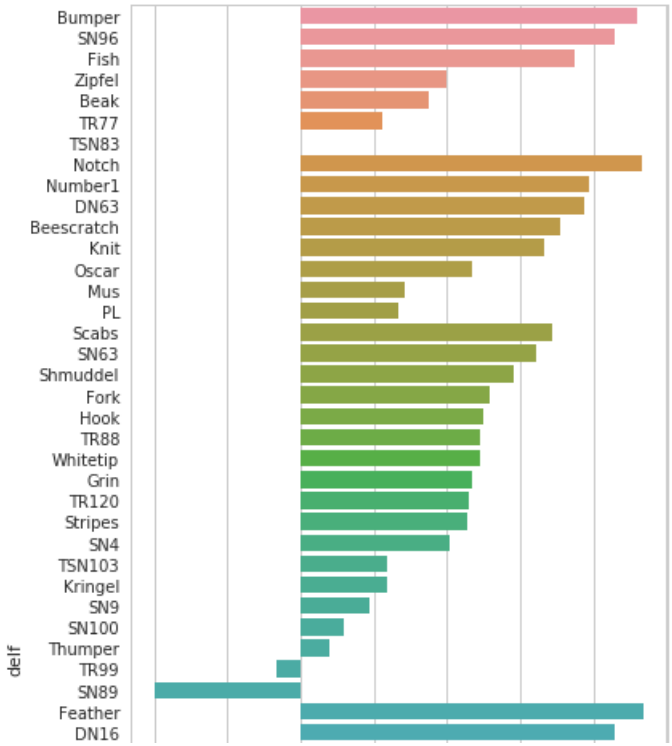
In [13]:

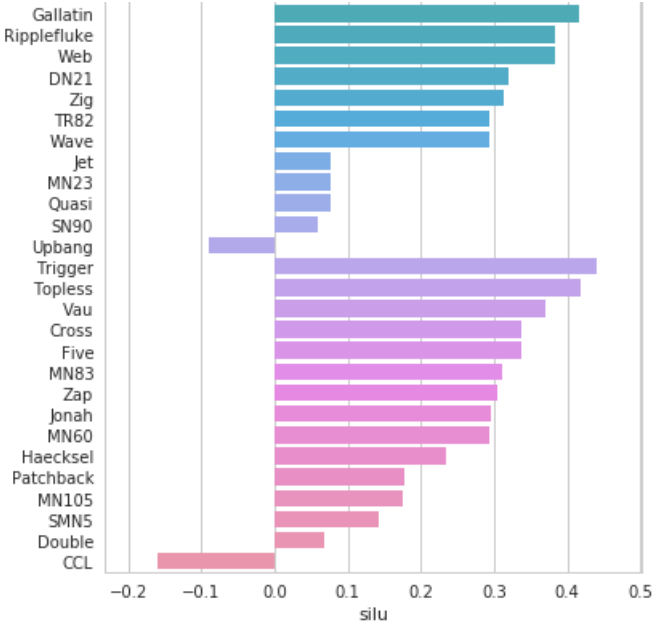
```
def grafico_silouhette(red, comus):
    s = silhouette(red,comus)[1]
    ds = [s, comus]
    sil = {}
    for nodo in s.keys():
        sil[nodo] = tuple(d[nodo] for d in ds)
    df_silu = pd.DataFrame(sil).T
    df_silu.reset_index(level=0, inplace=True)
    df_silu.columns = ['delf','silu','comu']
    df_silu.sort_values(['comu','silu'], ascending=[True, False], inplace=True)
    sns.set(style="whitegrid")

    f, ax = plt.subplots(figsize=(6, 15))
    sns.set_color_codes("pastel")
    sns.barplot(x="silu", y="delf", data=df_silu, label="Silouhette")

    return None
```

In [58]:

```
# Louvain
grafico_silouhette(red_delf, comus_louvain)
print("Silhouette Louvain:",silhouette(red_delf,comus_louvain)[0])
```

```
Silhouette Louvain: 0.233909844451
```

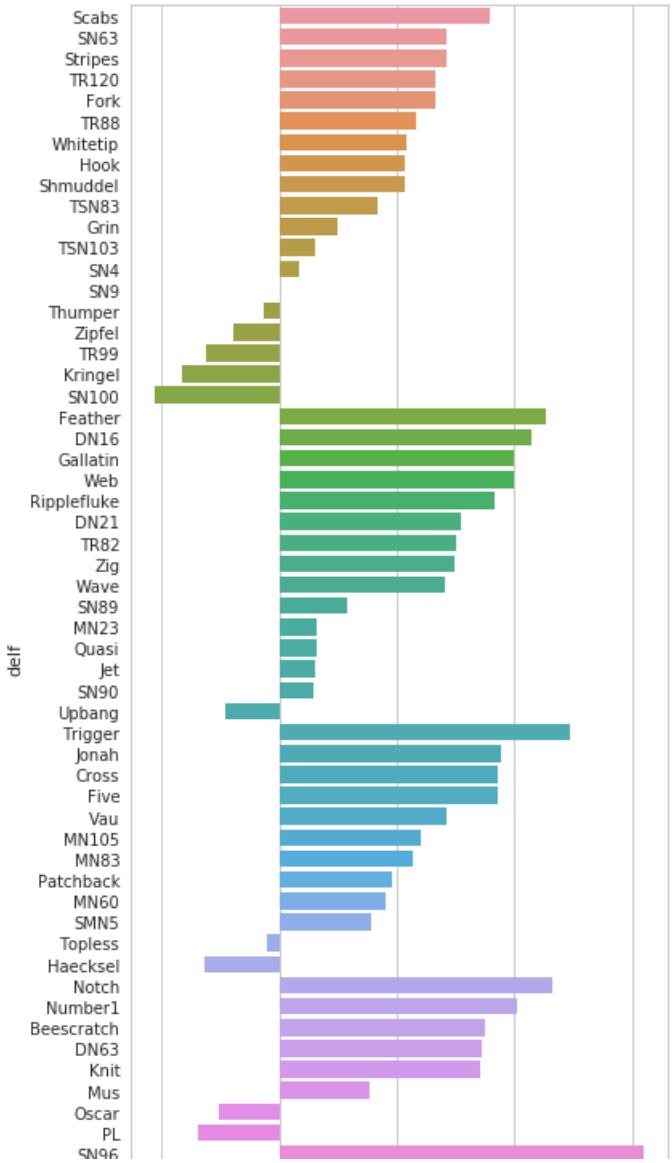


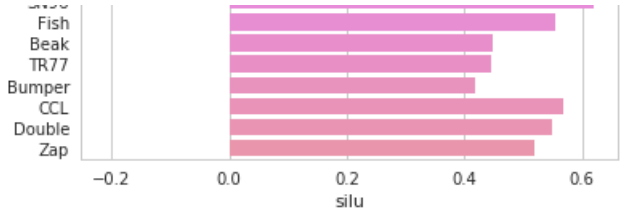


In [59]:

```
# Infomap
grafico_silouhette(red_delf, comus_infomap)
print("Silhouette Infomap:",silhouette(red_delf,comus_infomap)[0])
```

```
Silhouette Infomap: 0.230484183401
```

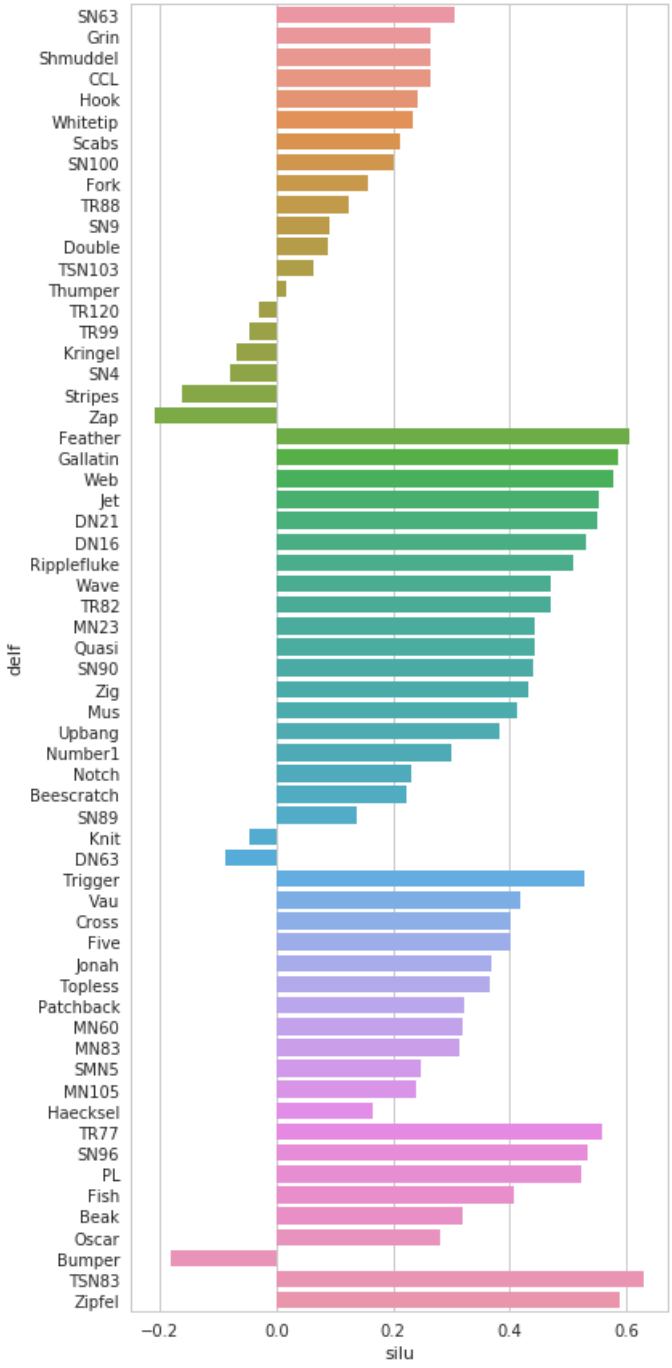




In [60]:

```
# Edge-Betweenness
grafico_silouhette(red_delf, comus_edgeb)
print("Silhouette Edge Betweenness:",silhouette(red_delf,comus_edgeb)[0])
```

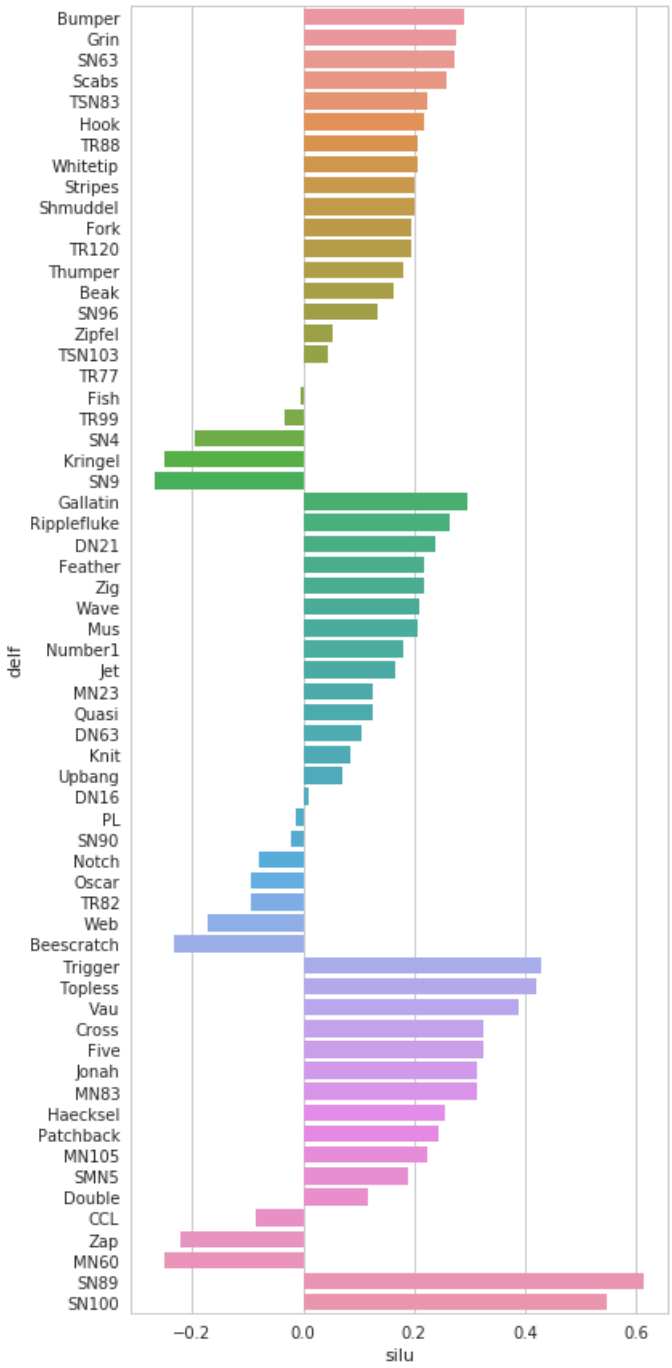
```
Silhouette Edge Betweenness: 0.287627332428
```



In [61]:

```
# Fast Greedy
grafico_silouhette(red_delf, comus_fg)
print("Silhouette Fast Greedy:",silhouette(red_delf,comus_fg)[0])
```

```
Silhouette Fast Greedy: 0.137954632797
```



In [18]:

```
print("Modularidad Infomap:",modularidad(red_delf,comus_infomap))
print("Modularidad Louvain:",modularidad(red_delf,comus_louvain))
print("Modularidad Edge Betweenness:",modularidad(red_delf,comus_edgeb))
print("Modularidad Fast Greedy:",modularidad(red_delf,comus_fg))
```

```
Modularidad Infomap: 0.5277283335311103
Modularidad Louvain: 0.5185317036509632
Modularidad Edge Betweenness: 0.5193821446936434
Modularidad Fast Greedy: 0.4954906847039278
```

In [56]:

```
red_delf_randomizada = red_delf.copy()
dist_infomap = mod_distrib(red_delf_randomizada,comus_infomap,100)
dist_louvain = mod_distrib(red_delf_randomizada,comus_louvain,100)
dist_edgeb = mod_distrib(red_delf_randomizada,comus_edgeb,100)
dist_fg = mod_distrib(red_delf_randomizada,comus_fg,100)
```

```
print("Modularidad Infomap para red randomizada (media,std):",dist_infomap)
print("Modularidad Louvain para red randomizada:",dist_louvain)
print("Modularidad Edge Betweenness para red randomizada:",dist_edgeb)
print("Modularidad Fast Greedy para red randomizada:",dist_fg)
```

```
Modularidad Infomap para red randomizada (media,std): (-0.020094141845654823,
0.025181044572207864)
Modularidad Louvain para red randomizada: (-0.016499742889917318, 0.028158656125250259)
Modularidad Edge Betweenness para red randomizada: (-0.021498358451010603, 0.032280412212296751)
Modularidad Fast Greedy para red randomizada: (-0.010861516553933803, 0.034880139544242017)
```

Algunas conclusiones sobre la modularidad de la red:

Las redes randomizadas tienen valores de modularidad mucho menores, cercanos a 0, que la red estudiada para cualquiera de los algoritmos. Por lo tanto, esto parace indicarnos que la red es modular.

Punto C

In [20]:

```
# Funciones auxiliares

def poblacionComus(particion):
# IN: Una partición en forma de diccionario.
# OUT: Un diccionario donde las keys son las comunidades y los values son el
# número de miembros de cada comunidad.
    values = particion.values()
    return Counter(values)


def pares(listas):
    pares_set = set()
    for t in combinations(listas, 2):
        for par in product(*t):
            pares_set.add(frozenset(par))
    return pares_set


def intersect(l1, l2):
# Intersección entre dos listas.
    l3 = [value for value in l1 if value in l2]
    return l3
```

In [21]:

```
# Información Mutua

def informacionMutua(p1,p2):
# IN: Dos particiones, en forma de diccionarios.
# OUT: Información mutua normalizada.

    # En estas líneas calculo las probabilidades, para cada partición, de que
    # un nodo elegido al azar de la red pertenezca a una cierta comunidad.
    # En las listas probs_p1 y probs_p2, están dichas probabilidades.
    probs_p1 = []
    probs_p2 = []
    prob_conj = []
    comus1 = poblacionComus(p1)
    comus2 = poblacionComus(p2)
    n = len(p1)

    for comu in sorted(comus1.keys()):
        prob = comus1[comu]/n
        probs_p1.append(prob)
    for comu in sorted(comus2.keys()):
        prob = comus2[comu]/n
        probs_p2.append(prob)

    # En estas lineas hago dos listas de listas. Cada lista, tiene adentro una
    # lista con los miembros de cada comunidad.
    comus1_list = []
```

```
    comus2_list = []
    for comu in sorted(comus1.keys()):
        miembros_comu = []
        for nodo in p1.keys():
            if p1[nodo] == comu:
                miembros_comu.append(nodo)
        comus1_list.append(miembros_comu)

    for comu in sorted(comus2.keys()):
        miembros_comu = []
        for nodo in p2.keys():
            if p2[nodo] == comu:
                miembros_comu.append(nodo)
        comus2_list.append(miembros_comu)

    # En estas líneas calculo la probabilidad conjunta pij de que un dado nodo
    # pertenezca a la comunidad i de la primera partición y a la comunidad j de
    # la segunda partición. Estas probabilidades están en la lista de listas
    # (o también podemos llamarle Matriz de coaparición normalizada), prob_conj.
    for pobi_list in comus1_list:
        pij = []
        for pobj_list in comus2_list:
            pij.append(len(intersect(pobi_list,pobj_list))/n)
        prob_conj.append(pij)

    # En las líneas que quedan, hago el cálculo de la información mutua, y
    # entropías de Shannon (me encanta decirlo) para devolver la información
    # mutua normalizada.
    I = 0
    for i in range(len(comus1.keys())):
        for j in range(len(comus2.keys())):
            if prob_conj[i][j] == 0:
                I += 0
            else:
                I += prob_conj[i][j]*math.log(prob_conj[i][j]/(probs_p1[i]*probs_p2[j]),2)

    H1 = 0
    for i in range(len(comus1.keys())):
        H1 += -probs_p1[i]*math.log(probs_p1[i],2)

    H2 = 0
    for i in range(len(comus2.keys())):
        H2 += -probs_p2[i]*math.log(probs_p2[i],2)

    return 2*I/(H1 + H2)
```

In [22]:

```
# Presición

def presicion(p1,p2):
    comus1 = poblacionComus(p1)
    comus2 = poblacionComus(p2)
    comus1_list = []
    comus2_list = []
    n = len(p1)

    for comu in sorted(comus1.keys()):
        miembros_comu = []
        for nodo in p1.keys():
            if p1[nodo] == comu:
                miembros_comu.append(nodo)
        comus1_list.append(miembros_comu)

    for comu in sorted(comus2.keys()):
        miembros_comu = []
        for nodo in p2.keys():
            if p2[nodo] == comu:
                miembros_comu.append(nodo)
        comus2_list.append(miembros_comu)

    pares_comus1 = []
    for comu in comus1_list:
        pares1 = set()
        for par in combinations(comu,2):
            pares1.add(frozenset(par))
        pares_comus1.append(pares1)
```

```
        pares_comus1.append(pares1)

    a11 = 0
    a00 = 0
    for set_pares in pares_comus1:
        for comu in comus2_list:
            for par in combinations(comu,2):
                if set(par) in set_pares:
                    a11 += 1

    np1 = pares(comus1_list)
    np2 = pares(comus2_list)

    for par1 in np1:
        if par1 in np2:
            a00 += 1

    return (a11 + a00)/(n*(n-1)/2)
```

In [23]:

```
particiones = ['Fast-greedy', 'Edge-betweenness', 'Infomap', 'Louvain']

tabla_infomutua = pd.DataFrame(np.nan, columns = particiones, index = particiones)
tabla_presicion = pd.DataFrame(np.nan, columns = particiones, index = particiones)

tabla_infomutua.loc['Fast-greedy']['Fast-greedy'] = informacionMutua(comus_fg, comus_fg)
tabla_infomutua.loc['Edge-betweenness']['Edge-betweenness'] = informacionMutua(comus_edgeb, comus_e
dgeb)
tabla_infomutua.loc['Infomap']['Infomap'] = informacionMutua(comus_infomap, comus_infomap)
tabla_infomutua.loc['Louvain']['Louvain'] = informacionMutua(comus_louvain, comus_louvain)
tabla_infomutua.loc['Louvain']['Fast-greedy'] = informacionMutua(comus_louvain, comus_fg)
tabla_infomutua.loc['Fast-greedy']['Louvain'] = informacionMutua(comus_fg, comus_louvain)
tabla_infomutua.loc['Louvain']['Edge-betweenness'] = informacionMutua(comus_louvain, comus_edgeb)
tabla_infomutua.loc['Edge-betweenness']['Louvain'] = informacionMutua(comus_edgeb, comus_louvain)
tabla_infomutua.loc['Louvain']['Infomap'] = informacionMutua(comus_louvain, comus_infomap)
tabla_infomutua.loc['Infomap']['Louvain'] = informacionMutua(comus_louvain, comus_infomap)
tabla_infomutua.loc['Edge-betweenness']['Fast-greedy'] = informacionMutua(comus_edgeb, comus_fg)
tabla_infomutua.loc['Fast-greedy']['Edge-betweenness'] = informacionMutua(comus_edgeb, comus_fg)
tabla_infomutua.loc['Edge-betweenness']['Infomap'] = informacionMutua(comus_edgeb, comus_infomap)
tabla_infomutua.loc['Infomap']['Edge-betweenness'] = informacionMutua(comus_edgeb, comus_infomap)
tabla_infomutua.loc['Fast-greedy']['Infomap'] = informacionMutua(comus_fg, comus_infomap)
tabla_infomutua.loc['Infomap']['Fast-greedy'] = informacionMutua(comus_fg, comus_infomap)

tabla_presicion.loc['Fast-greedy']['Fast-greedy'] = presicion(comus_fg, comus_fg)
tabla_presicion.loc['Edge-betweenness']['Edge-betweenness'] = presicion(comus_edgeb, comus_edgeb)
tabla_presicion.loc['Infomap']['Infomap'] = presicion(comus_infomap, comus_infomap)
tabla_presicion.loc['Louvain']['Louvain'] = presicion(comus_louvain, comus_louvain)
tabla_presicion.loc['Louvain']['Fast-greedy'] = presicion(comus_louvain, comus_fg)
tabla_presicion.loc['Fast-greedy']['Louvain'] = presicion(comus_fg, comus_louvain)
tabla_presicion.loc['Louvain']['Edge-betweenness'] = presicion(comus_louvain, comus_edgeb)
tabla_presicion.loc['Edge-betweenness']['Louvain'] = presicion(comus_edgeb, comus_louvain)
tabla_presicion.loc['Louvain']['Infomap'] = presicion(comus_louvain, comus_infomap)
tabla_presicion.loc['Infomap']['Louvain'] = presicion(comus_louvain, comus_infomap)
tabla_presicion.loc['Edge-betweenness']['Fast-greedy'] = presicion(comus_edgeb, comus_fg)
tabla_presicion.loc['Fast-greedy']['Edge-betweenness'] = presicion(comus_edgeb, comus_fg)
tabla_presicion.loc['Edge-betweenness']['Infomap'] = presicion(comus_edgeb, comus_infomap)
tabla_presicion.loc['Infomap']['Edge-betweenness'] = presicion(comus_edgeb, comus_infomap)
tabla_presicion.loc['Fast-greedy']['Infomap'] = presicion(comus_fg, comus_infomap)
tabla_presicion.loc['Infomap']['Fast-greedy'] = presicion(comus_fg, comus_infomap)
```

En las siguientes tablas puede apreciarse el acuerdo entre las particiones hechas con los distintos algoritmos de clusterización de la red. En la primera de ellas se ve el acuerdo entre las particiones según el observable de información mutua y en la segunda según el observable de presición.

In [24]:

```
tabla_infomutua
```

Out[24]:

	Fast-greedy	Edge-betweenness	Infomap	Louvain

	Fast-greedy	Edge-betweenness	Infomap	Louvain
Fast-greedy	1.000000	0.662148	0.796621	0.794842
Edge-betweenness	0.662148	1.000000	0.873041	0.732946
Infomap	0.796621	0.873041	1.000000	0.742574
Louvain	0.794842	0.732946	0.742574	1.000000

In [25]:

```
tabla_presicion
```

Out[25]:

	Fast-greedy	Edge-betweenness	Infomap	Louvain
Fast-greedy	1.000000	0.843469	0.903755	0.864622
Edge-betweenness	0.843469	1.000000	0.936542	0.873083
Infomap	0.903755	0.936542	1.000000	0.882602
Louvain	0.864622	0.873083	0.882602	1.000000

Se puede ver que en líneas generales, exceptuando el caso de la comparación entre la partición de Edge-betweenness y Fast-greedy mediante el método de información mutua, hay un buen acuerdo entre las particiones, evaluándolo mediante los dos métodos. En particular, mediante el método de presición parece haber más acuerdo entre las particiones.

Punto D

A continuación definimos las funciones que van a ser necesarias para el analizar la relación entre el género de los delfines y la estructura en comunidades de las diferentes particiones.

In [26]:

```
def atributoNodos(r, alist, atributo):
# Toma como argumentos una red, una lista de listas, donde cada una de ellas
# indica el atributo que se le va a asignar a cada nodo de la red, y el
# atributo que uno quiere asignar. Devuelve la red con ese atributo ya asociado
# a cada nodo.
    for idx, nodo in enumerate(np.array(alist).transpose()[0]):
        r.nodes[nodo][atributo] = np.array(alist).transpose()[1][idx]
    return


def generoAzar(r):
# Toma una red donde sus nodos tienen el atributo "genero" y lo distribuye al
# azar.
# Estaría bueno generalizarlo después para cualquier atributo.
    ng = contadorGenero(r)
    n = list(r.nodes)
    shuffle(n)
    ra = deepcopy(r)
    for i in range(ng['m']):
        ra.nodes[n[i]]['gender'] = 'm'
    for i in range(ng['f'], ng['m']+ng['f']):
         ra.nodes[n[i]]['gender'] = 'f'
    for i in range(ng['m']+ng['f'], ng['m']+ng['f']+ng['NA']):
         ra.nodes[n[i]]['gender'] = "NA"
    return ra


def contadorGenero(r): #Generalizarlo para cualquier atributo
    gen_red = dict()
    a = list(nx.get_node_attributes(r, 'gender').values())
    gen_red['m'] = a.count('m')
    gen_red['f'] = a.count('f')
    gen_red['NA'] = a.count('NA')
    return gen_red


def combinatorio(n,r):
```

```
    f = math.factorial
    return f(n) / f(r) / f(n-r)
```

In [27]:

```
def poblacionAtributoComus(red, particion):
# IN: Una partición en forma de diccionario.
# OUT: Un diccionario donde las keys son las comunidades y los values son el
# número de miembros de cada comunidad.
    c = [{'m': 0, 'f': 0, 'NA': 0} for i in set(particion.values())]
    for nodo, comu in particion.items():
        c[comu]['m'] += red.nodes[nodo]['gender'] == 'm'
        c[comu]['f'] += red.nodes[nodo]['gender'] == 'f'
        c[comu]['NA'] += red.nodes[nodo]['gender'] == 'NA'
    return c


def generoAzarComus(red, particion, iters):
    lista = []
    for i in range(iters):
        rr = generoAzar(red)
        lista.append(np.array(poblacionAtributoComus(rr,particion)))
    #lista = np.swapaxes(np.array(lista),0,1)
    return lista

def datosGenComu(red, particion, iters):
    dat0 = generoAzarComus(red, particion, iters)
    dat = []
    for comu in range(len(set(particion.values()))):
        gencomu = []
        for i in range(iters):
            gencomu.append((dat0[i][comu]['m'], dat0[i][comu]['f']))
        dat.append(gencomu)
    return dat
```

In [28]:

```
# HISTOGRAMAS
# Con esta funcion se crean las listas de datos aleatorios que voy a necesitar
# para hacer los histogramas, una lista de listas para los machos y una para
# las hembras. Dentro de la lista de machos (hembras) hay listas donde están
# los números de machos (hembras) que se obtuvieron en las iteraciones, para
# una dada comunidad.

def listasGenComu(red, particion, iters):
    dat0 = datosGenComu(red, particion, iters)
    dat_machos = []
    dat_hembras = []
    for comu in range(len(set(particion.values()))):
        mcomu = []
        hcomu = []
        for i in range(iters):
            mcomu.append(dat0[comu][i][0])
            hcomu.append(dat0[comu][i][1])
        dat_machos.append(mcomu)
        dat_hembras.append(hcomu)
    return dat_machos, dat_hembras
```

In [29]:

```
# TEST DE FISHER

def hipergeometrica(N, r, k, m):
    p = combinatorio(k, m)*combinatorio(N-k, r-m)/combinatorio(N, r)
    return p

# Esta función hace el test de Fisher para todas las comunidades de una dada
# partición. Hay que darle el atributo dicotómico para que haga el test según
# esa variable. Por ej: Si el atributo es 'f', me va a decir, para cada
# comunidad, qué tanta probabilidad hay de obtener el número de hembras que
# tengo, o uno más grande, asumiendo que están distribuídos al azar (esto es el
# p-value). Si esta probabilidad es muy chica, significa que debe existir una
# correlación entre esa comunidad y la cantidad de hembras que hay en ella y que
# la hipótesis de que ese número viene del azar es poco probable de que sea
```

```
# verdadera.
def testFisherParticion(red, particion, atributo):
    poblaciones = poblacionComus(particion)
    distGenerosComus = poblacionAtributoComus(red, particion)
    N = red.number_of_nodes()
    k = contadorGenero(red)[atributo]
    pval_comus = []
    for comu in poblaciones.keys():
        suma = 0
        r = poblaciones[comu]
        m = distGenerosComus[comu][atributo]
        for i in range(m, r+1):
            suma += hipergeometrica(N, r, k, i)
            #print(suma)
            #print(N-k, r-i)
        pval_comus.append(suma)
    return pval_comus
```

In [46]:

```
# Datos para los histogramas:
machosAzar_louvain = listasGenComu(red_delf, comus_louvain, 5000)[0]
machosAzar_infomap = listasGenComu(red_delf, comus_infomap, 5000)[0]
machosAzar_edgeb = listasGenComu(red_delf, comus_edgeb, 5000)[0]
machosAzar_fg = listasGenComu(red_delf, comus_fg, 5000)[0]

hembrasAzar_louvain = listasGenComu(red_delf, comus_louvain, 5000)[1]
hembrasAzar_infomap = listasGenComu(red_delf, comus_infomap, 5000)[1]
hembrasAzar_edgeb = listasGenComu(red_delf, comus_edgeb, 5000)[1]
hembrasAzar_fg = listasGenComu(red_delf, comus_fg, 5000)[1]
```

En primer se realizaron cliclos de 5000 iteraciones en donde, en cada una de ellas, se distribuyeron aleatoriamente los géneros de los delfines a través de la red, conservándose la etiqueta de cada delfín de la pertenencia a una determinada comunidad, para una dada partición. En cada uno de estos ciclos se registraron datos de los porcentajes de machos, hembras e indefinidos en cada comunidad y con estos se confeccionaron histogramas.

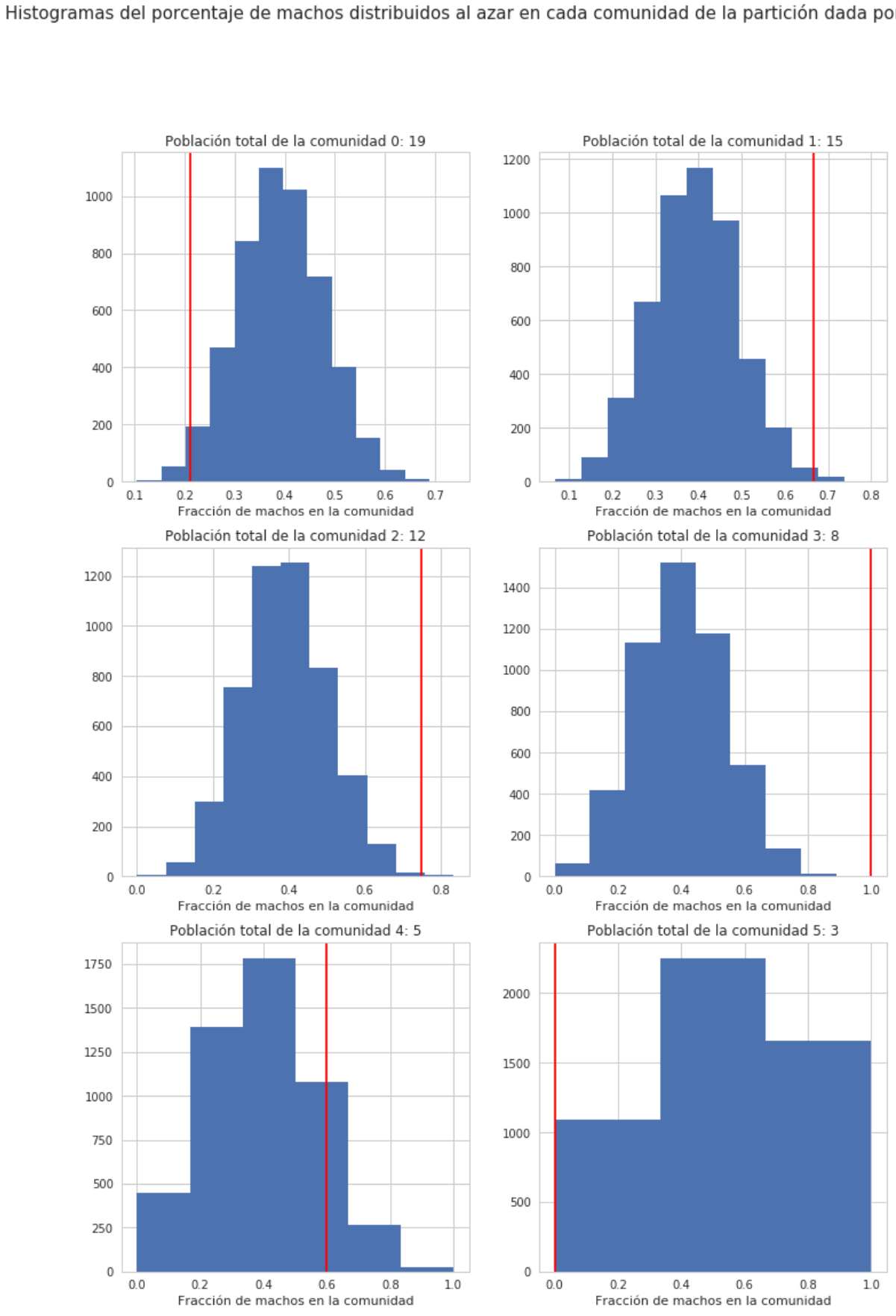
Los histogramas de porcentajes de machos en cada comunidad obtenida por infomap se muestran a continuación, junto con una línea roja en cada uno de ellos, indicando el valor del porcentaje de machos de la comunidad con la distribución de géneros empírica.

In [47]:

```
# Histogramas infomap

generos_infomap = poblacionAtributoComus(red_delf, comus_infomap)

fig = plt.figure(figsize=(12,18))
numcomus = len(set(comus_infomap.values()))

axs = [None]*numcomus

for i in range(numcomus):
    axs[i] = fig.add_subplot(320+i+1)
"""ax1 = fig.add_subplot(321)
ax2 = fig.add_subplot(322)
ax3 = fig.add_subplot(323)
ax4 = fig.add_subplot(324)
ax5 = fig.add_subplot(325)
ax6 = fig.add_subplot(326)"""

print(numcomus)
#axs_info = [ax1, ax2, ax3, ax4, ax5, ax6]
bineo_infomap_machos = [13, 12, 11, 9, 6, 3]
bineo_aux = bineo_infomap_machos[:numcomus+1]
axvlines_info = []

for i in range(len(generos_infomap)):
    axvlines_info.append(generos_infomap[i]['m'])

for i in range(numcomus):
    axs[i].hist(np.array(machosAzar_infomap[i])/poblacionComus(comus_infomap)[i], bins = bineo_aux[
i])
    axs[i].set_title('Población total de la comunidad {0}: {1}'.format(i,
```

```
poblacionComus(comus_infomap)[i]))
    axs[i].axvline(axvlines_info[i]/poblacionComus(comus_infomap)[i], c='red')
    axs[i].set_xlabel('Fracción de machos en la comunidad')

fig.suptitle('Histogramas del porcentaje de machos distribuidos al azar en cada comunidad de la pa
rtición dada por Infomap', fontsize=15)
plt.show()
```

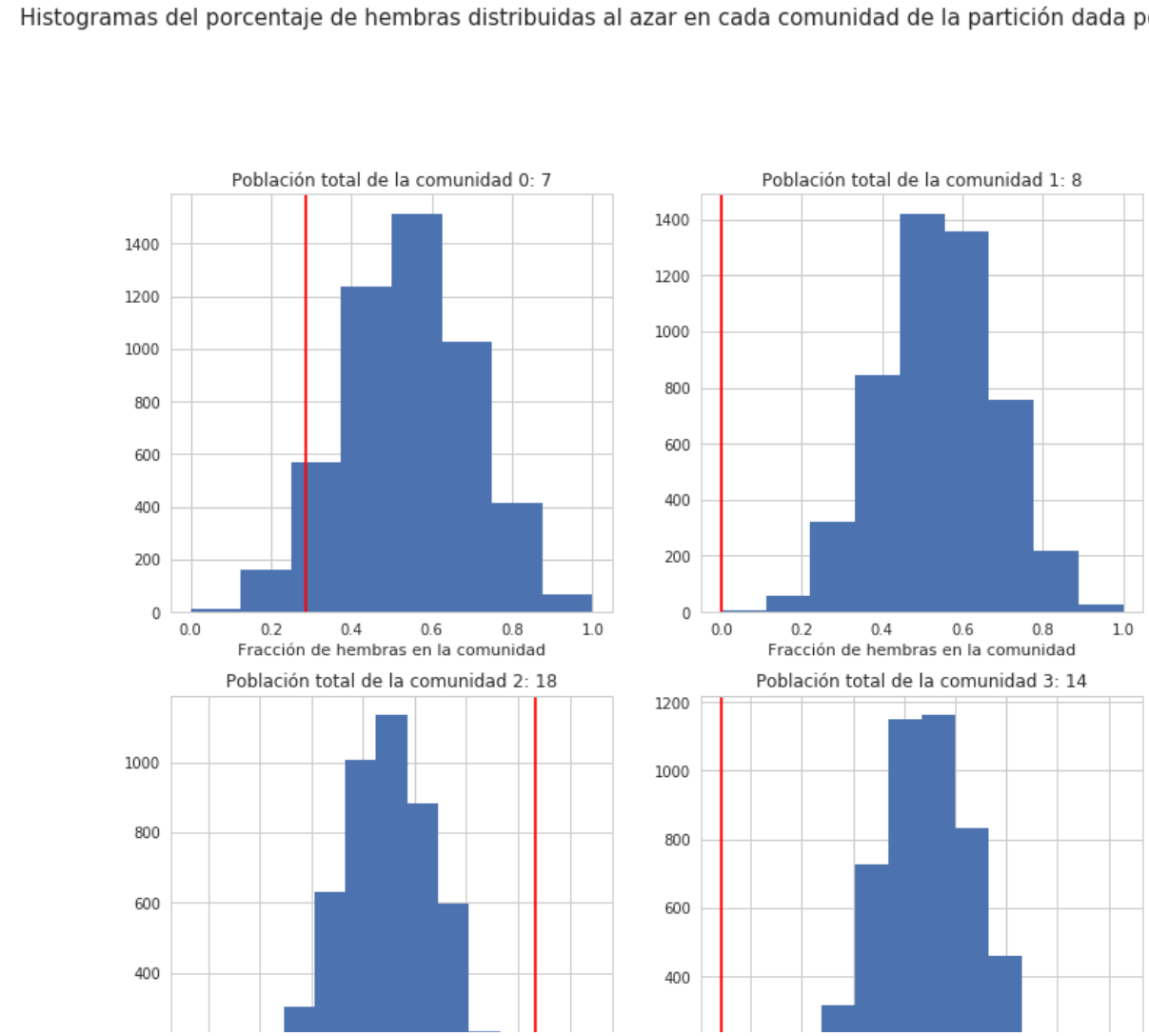
6

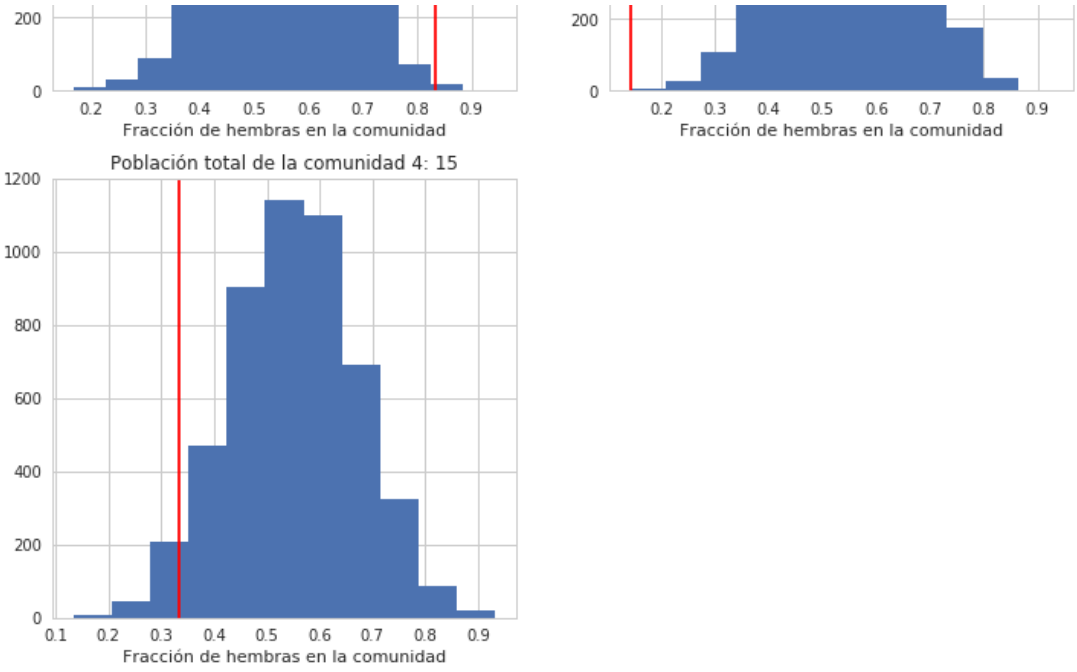


Histogramas del porcentaje de hembras para las comunidades obtenidas mediante Louvain:

In [62]:

```
# Histogramas Louvain

generos_louvain = poblacionAtributoComus(red_delf, comus_louvain)

fig = plt.figure(figsize=(12,18))
ax1 = fig.add_subplot(321)
ax2 = fig.add_subplot(322)
ax3 = fig.add_subplot(323)
ax4 = fig.add_subplot(324)
ax5 = fig.add_subplot(325)


axs_lou = [ax1, ax2, ax3, ax4, ax5]
bineo_louvain_hembras = [8, 9, 13, 12, 11]

axvlines_lou = []
for i in range(len(generos_louvain)):
    axvlines_lou.append(generos_louvain[i]['f'])

for i in range(len(axs_lou)):
    axs_lou[i].hist(np.array(hembrasAzar_louvain[i])/poblacionComus(comus_louvain)[i], bins = bineo
_louvain_hembras[i])
    axs_lou[i].set_title('Población total de la comunidad {0}: {1}'.format(i, poblacionComus(comus_
louvain)[i]))
    axs_lou[i].axvline(axvlines_lou[i]/poblacionComus(comus_louvain)[i], c='red')
    axs_lou[i].set_xlabel('Fracción de hembras en la comunidad')

fig.suptitle('Histogramas del porcentaje de hembras distribuidas al azar en cada comunidad de la p
artición dada por Louvain', fontsize=15)
plt.show()
```

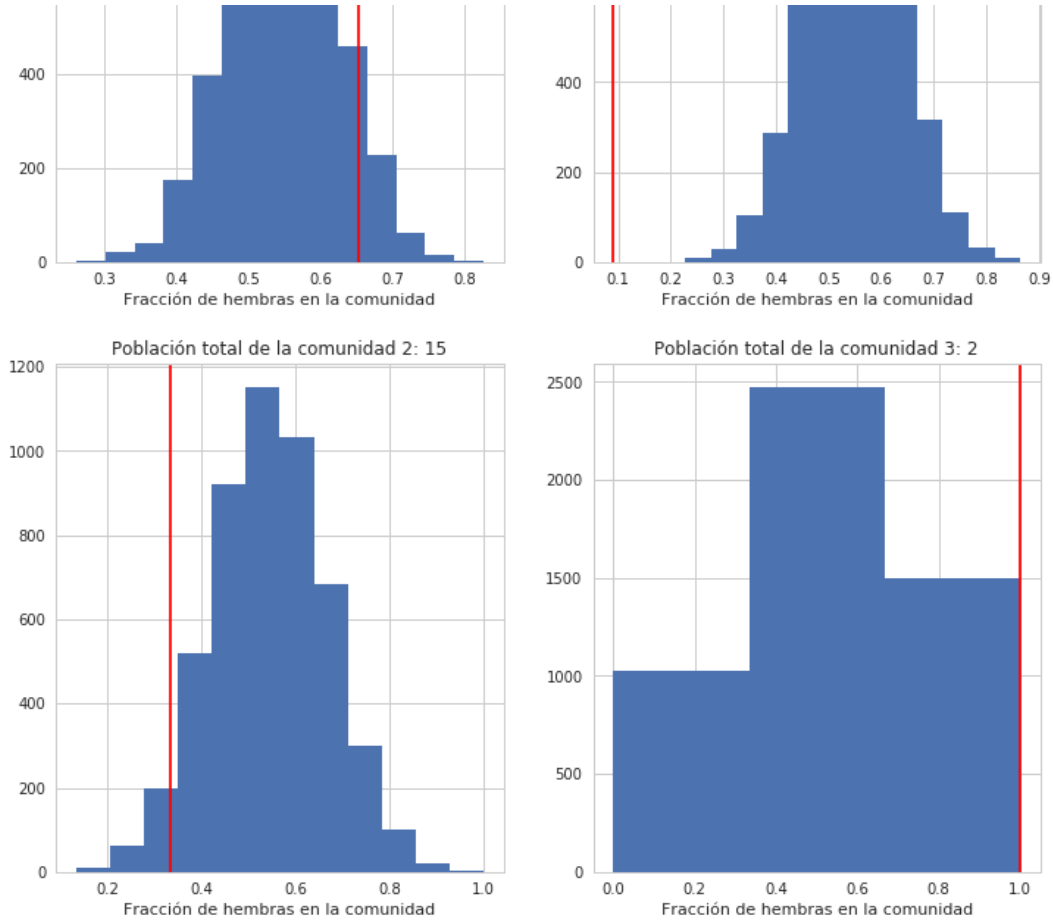




Histogramas del porcentaje de hembras para las comunidades obtenidas mediante Fast-Greedy:

In [63]:

```
# Histogramas Fast-Greedy

generos_fg = poblacionAtributoComus(red_delf, comus_fg)

fig = plt.figure(figsize=(12,14))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)


axs_fg = [ax1, ax2, ax3, ax4]
bineo_fg_hembras = [14, 13, 12, 3]

axvlines_fg = []
for i in range(len(generos_fg)):
    axvlines_fg.append(generos_fg[i]['f'])

for i in range(len(axs_fg)):
    axs_fg[i].hist(np.array(hembrasAzar_fg[i])/poblacionComus(comus_fg)[i], bins = bineo_fg_hembras
[i])
    axs_fg[i].set_title('Población total de la comunidad {0}: {1}'.format(i, poblacionComus(comus_f
g)[i]))
    axs_fg[i].axvline(axvlines_fg[i]/poblacionComus(comus_fg)[i], c='red')
    axs_fg[i].set_xlabel('Fracción de hembras en la comunidad')

fig.suptitle('Histogramas del porcentaje de hembras distribuidas al azar en cada comunidad de la p
artición dada por Fast-Greedy', fontsize=15)
plt.show()
```

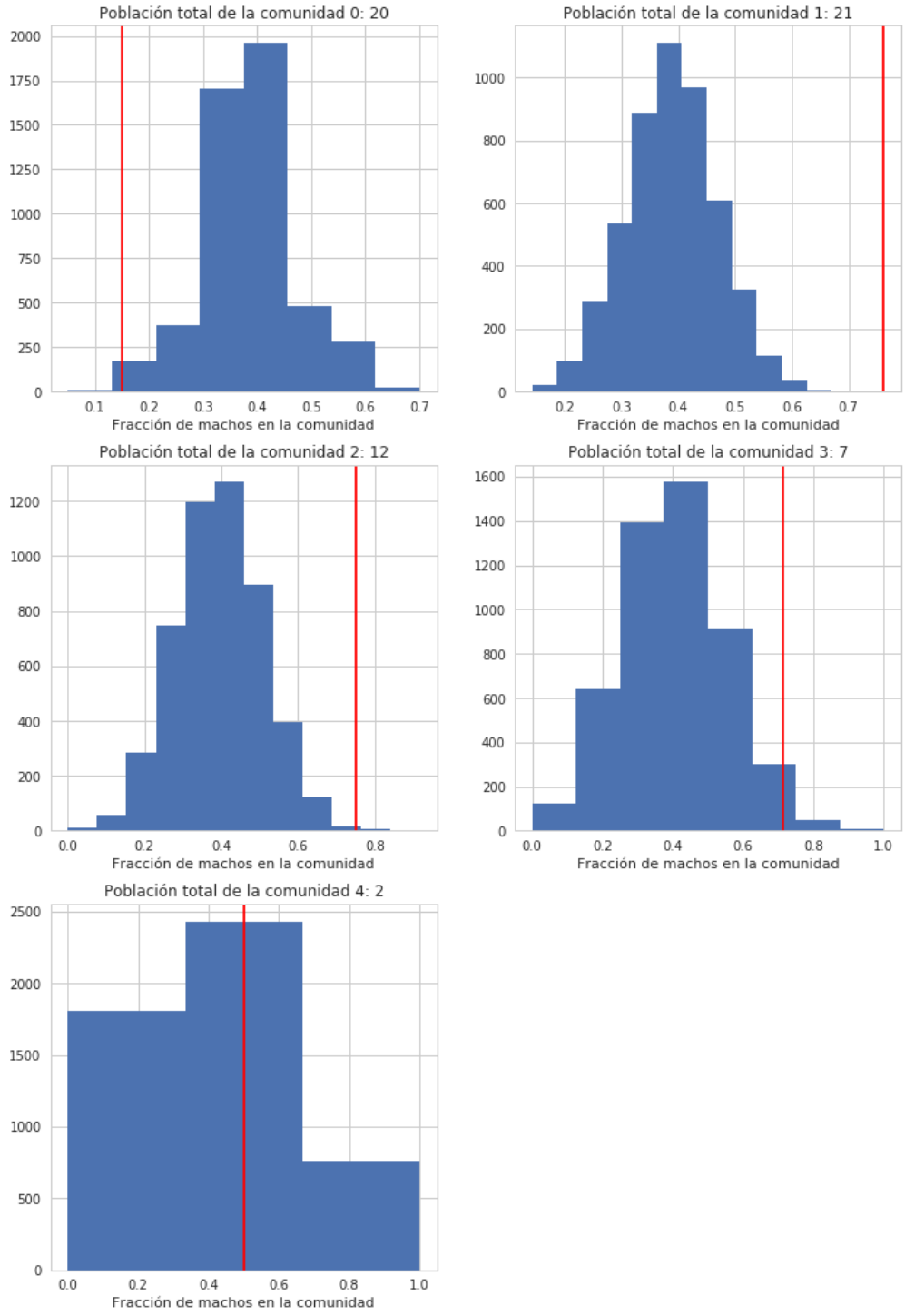


Histogramas del porcentaje de hembras para las comunidades obtenidas mediante Edge-betweenness:

In [66]:

```
# Histogramas Edge-Betwenness

generos_edgeb = poblacionAtributoComus(red_delf, comus_edgeb)

fig = plt.figure(figsize=(12,18))
ax1 = fig.add_subplot(321)
ax2 = fig.add_subplot(322)
ax3 = fig.add_subplot(323)
ax4 = fig.add_subplot(324)
ax5 = fig.add_subplot(325)


axs_edgeb = [ax1, ax2, ax3, ax4, ax5]
bineo_edgeb_machos = [8, 13, 12, 8, 3]

axvlines_edgeb = []
for i in range(len(generos_edgeb)):
    axvlines_edgeb.append(generos_edgeb[i]['m'])

for i in range(len(axs_edgeb)):
    axs_edgeb[i].hist(np.array(machosAzar_edgeb[i])/poblacionComus(comus_edgeb)[i], bins = bineo_ed
geb_machos[i])
    axs_edgeb[i].set_title('Población total de la comunidad {0}: {1}'.format(i, poblacionComus(comu
s_edgeb)[i]))
    axs_edgeb[i].axvline(axvlines_edgeb[i]/poblacionComus(comus_edgeb)[i], c='red')
    axs_edgeb[i].set_xlabel('Fracción de machos en la comunidad')

fig.suptitle('Histogramas del porcentaje de machos distribuidos al azar en cada comunidad de la pa
rtición dada por Edge-Betweenness', fontsize=15)
plt.show()
```

Histogramas del porcentaje de machos distribuidos al azar en cada comunidad de la partición dada por Edge-Betweenness



Luego, se realizó un Test exacto de Fisher para cada comunidad de cada partición, con las variables dicotómicas "macho" o "hembra", dependiendo del caso.

Los p-values para cada comunidad de infomap:

In [51]:

```
fisher_infomap_machos = testFisherParticion(red_delf, comus_infomap, 'm')
for i in range(len(fisher_infomap_machos)):
    print('El p-value de la comunidad', i,'es', fisher_infomap_machos[i])
```

```
El p-value de la comunidad 0 es 0.5932212661964871
El p-value de la comunidad 1 es 0.00536991269504687
El p-value de la comunidad 2 es 1.0
El p-value de la comunidad 3 es 0.10645349185322082
El p-value de la comunidad 4 es 0.22495995266098842
El p-value de la comunidad 5 es 0.999954567931268
```

Los p-values para cada comunidad de Louvain:

In [52]:

```
fisher_louvain_hembras = testFisherParticion(red_delf, comus_louvain, 'f')
for i in range(len(fisher_louvain_hembras)):
    print('El p-value de la comunidad', i,'es', fisher_louvain_hembras[i])
```

```
El p-value de la comunidad 0 es 0.8396152558753652
El p-value de la comunidad 1 es 1.0
El p-value de la comunidad 2 es 0.7849768092214894
El p-value de la comunidad 3 es 0.9951983068072966
El p-value de la comunidad 4 es 6.2519838427336695e-06
```

Los p-values para cada comunidad de Edge-Betweenness:

In [53]:

```
fisher_edgeb_machos = testFisherParticion(red_delf, comus_edgeb, 'm')
for i in range(len(fisher_edgeb_machos)):
    print('El p-value de la comunidad', i,'es', fisher_edgeb_machos[i])
```

```
El p-value de la comunidad 0 es 0.9999991744935438
El p-value de la comunidad 1 es 0.014791081862983273
El p-value de la comunidad 2 es 0.10645349185322082
El p-value de la comunidad 3 es 0.3013803491492142
El p-value de la comunidad 4 es 0.8001057641459546
```

Los p-value para cada comunidad de Fast-Greedy:

In [54]:

```
fisher_fg_hembras = testFisherParticion(red_delf, comus_fg, 'f')
for i in range(len(fisher_fg_hembras)):
    print('El p-value de la comunidad', i,'es', fisher_fg_hembras[i])
```

```
El p-value de la comunidad 0 es 0.0012441631483900758
El p-value de la comunidad 1 es 0.9999792223153197
El p-value de la comunidad 2 es 0.7849768092214894
El p-value de la comunidad 3 es 0.14595452141723952
```

Puede verse que, tanto mediante el análisis de los histogramas obtenidos por una redistribución aleatoria de géneros como mediante el cálculo del p-value de cada comunidad mediante un Test de Fisher, existe una correlación entre la estructura de muchas de las comunidades con el género. Esto puede observarse en el gran número de histogramas en donde la línea roja cae en las colas de los mismos, indicando una sobrerepresentación o una sobrepresentación, dependiendo cada caso. Al mismo tiempo, muchos de los test arrojaron p-values muy pequeños, indicando una sobrerepresentación, o muy grandes, indicando subrepresentación.