Appendix D Online Supplement

This section contains useful insights and examples that are helpful (but unnecessary) for understanding different sections of the book.

1 Section D.1 gives a working example for convergence concepts of spectral methods theory.
2 Section D.2 introduces the concept of unsupervised learning, and gives an overview of the `K-means` algorithm.
3 Section D.3 discusses the Bayes plugin classifier, which is a simple classifier that is useful for Signal Subnetworks.

D.1 Erdös-Rényi networks and the adjacency spectral embedding

The Erdös-Rényi (ER) model, introduced in Section 4.2 and Appendix A.3, represents the simplest example of a Random Dot Product Graph, where all edge probabilities are identical. Consider $\mathbf{A} \sim ER_n(p)$, for some probability p.

We can express the distribution of this network using a one-dimensional latent position matrix $\vec{x} = (x_1, \ldots, x_n)^\top$ (non-random in this case), with all entries equal to $\sqrt{p}$, yielding:

$$Pr(\mathbf{a}_{ij}) = x_i x_j = p.$$

Thus, we can think of $ER_n(p)$ as an RDPG with latent position distribution $F = \delta_{\sqrt{p}}$, where $\delta_{\sqrt{p}}$ is the Dirac delta distribution assigning positive probability only to the mass point $\sqrt{p}$. This representation reveals a non-identifiability in the model, as both $\vec{x}$ and $-\vec{x}$ result in the same edge probabilities. For practical purposes, the choice between these two representations is inconsequential.

The adjacency spectral embedding of $\mathbf{A}$ is given by $\widehat{\vec{\mathbf{x}}} = \texttt{ase}(\mathbf{A}, 1) = \sqrt{\widehat{\boldsymbol{\lambda}}}\widehat{\vec{\mathbf{v}}}$. The error in estimating $\vec{\mathbf{x}}$ from $\widehat{\vec{\mathbf{x}}}$ should decrease as the network size increases:

$$\max_{i \in [n]} |\widehat{\mathbf{x}}_i - \sqrt{p}\mathbf{w}_n| \leq \frac{c \log^2 n}{\sqrt{n}}.$$

Here, $\mathbf{w}_n$ functions as a one-dimensional orthogonal matrix accounting for the

model's non-identifiability. In other words, w_n equals either 1 or -1, reflecting the sign ambiguity of the singular vectors.

Let us examine how this works in practice. We begin by simulating 50 networks, with increasing numbers of nodes ranging from approximately 30 to 3000. We embed each network into a single dimension with `ase`, and then align the signs of the estimated latent positions with the true latent positions to address the non-identifiability issue using Orthogonal Procrustes (Section 7.3). The following code block requires approximately 3 minutes to execute on a laptop:

```
import numpy as np
from graspologic.simulations import er_np
from graspologic.embed import AdjacencySpectralEmbed

def orthogonal_align(Xhat, p=0.5):
   return -Xhat if ((Xhat*np.sqrt(p)).sum() < 0) else Xhat

p = 0.5
ns = np.round(10**np.linspace(1.5, 3.5, 5)).astype(int)
ase = AdjacencySpectralEmbed(n_components=1)

nrep = 50
As = [[er_np(n, p) for _ in range(nrep)] for n in ns]
Xhats_aligned = [[orthogonal_align(ase.fit_transform(A)) for A in An] for An in As]
```

Next, we compute the maximum difference between the estimated latent positions (after alignment) and $\sqrt{p}$ for each network. We divide this difference by $\log^2(n)$ and multiply by $\sqrt{n}$:

```
import pandas as pd

data = []
for n_idx, n in enumerate(ns):
   for j in range(50):
      data.extend([(Xhats_aligned[n_idx][j][i][0], i, n, j, np.sqrt(p)) for i in
           range(n)])

df = pd.DataFrame(data, columns=["Xhat", "i", "n", "j", "X"])
df["abs_diff"] = np.abs(df["Xhat"] - df["X"])

max_pernet = df.groupby(["n", "j"])["abs_diff"].max().reset_index()
max_pernet["norm_factor"] = np.log(max_pernet["n"])**2 / np.sqrt(max_pernet["n"])
max_pernet["norm_diff"] = max_pernet["abs_diff"] / max_pernet["norm_factor"]
```

Figure D.2 plots `max_pernet["norm_diff"]` as the average (solid line) and 95% probability interval (shaded ribbon) for each value of n. If there exists a constant value c such that, as n grows, the values consistently fall below this constant with increasing probability, we have empirically demonstrated the desired result. As observed, this value decreases rapidly as we increase the number of nodes. We could choose virtually any constant attained by this curve (for instance, 0.10). The plot demonstrates that as the number of nodes approaches infinity, the

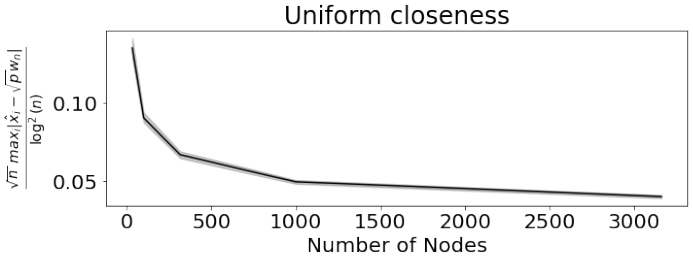

Figure D.1 A plot illustrating the uniform closeness between the estimates of $\vec{x}_i$ and $\vec{x} = \sqrt{p}$.

probability that the y-axis values are less than our chosen constant approaches 1, as desired.

We can also derive an expression for the asymptotic distribution of the difference between the true and estimated positions. Since F has only one point mass at $\sqrt{p}$, we can evaluate integrals and expectations with respect to this distribution:

$$\int_{\mathbb{R}^d} \Phi(z, \Sigma(\mathbf{x})), \mathrm{d}F(\mathbf{x}) = \Phi\left(z, \Sigma(\sqrt{p})\right).$$

We obtain the exact form of the covariance term $\Sigma(\mathbf{x})$ from the second moment matrix $\Delta = \mathbb{E}[\mathbf{x}_i^2] = p$:

$$\Sigma(\mathbf{x}) = \frac{1}{p^2}\mathbb{E}[p(\mathbf{x}\sqrt{p} - (\mathbf{x}\sqrt{p})^2)].$$

Combining these results, we find that the limiting distribution of the difference between $\widehat{\mathbf{x}}$ and $\mathbf{x}$ satisfies:

$$\sqrt{n}\left(\widehat{\mathbf{x}}_i - \sqrt{p}\right) \xrightarrow[n\to\infty]{} \mathcal{N}(0, 1-p) \qquad \text{for each node } i = 1, \ldots, n.$$

We implement this in Python as follows:

```
df_reduced = df[df["j"] == 0].copy()
df_reduced["limiting_factor"] = np.sqrt(df_reduced["n"]) * (df_reduced["Xhat"] -
    df_reduced["X"])
```

We show the distribution of the factors $\sqrt{n}(\hat{x}_i - \sqrt{p})$ for a single network for each value of n in Figure D.2 (black histograms). We plot these limiting factors against the pdf for an appropriate normal distribution with mean 0 and standard deviation $1-p$ (gray dashed line). As n grows, the limiting distribution of $\sqrt{n}(\hat{x}_i - \sqrt{p})$ approaches the desired normal distribution.

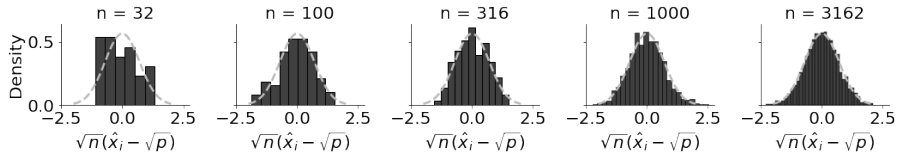

Figure D.2 A plot illustrating the density histograms of the limiting factors $\sqrt{n}(\hat{x}_i - \sqrt{p})$, as n increases (black histogram) against the density for the normal distribution with mean 0 and standard deviation $1 - p$ (gray dashed).

D.2 Unsupervised machine learning

To start this section off, we generate an example network with $K = 3$ communities, similar to the one we used in Section 6.1:

```
from graspologic.simulations import sbm
from graspologic.embed import AdjacencySpectralEmbed as ASE
import numpy as np

ns = [50, 40, 30]
B = [[0.6, 0.2, 0.2],
    [0.2, 0.6, 0.2],
    [0.2, 0.2, 0.6]]

np.random.seed(1234)
A = sbm(n=ns, p = B)

# the true community labels
z = [0 for i in range(0,ns[0])] + [1 for i in range(0, ns[1])] + [2 for i in range(0,
     ns[2])]
Xhat = ASE(n_components=3).fit(A).latent_left_
```

D.2.1 K-means clustering

K-means clustering allows us to learn about clusters in our dataset through the following process:

- Given: Estimates of latent positions $\vec{x}_i$ for each node i from 1 to n, and a number of clusters K.
- Output: Predicted labels z_i for each node i from 1 to n.

K-means is an unsupervised clustering technique that learns latent structure from data without prior information about node communities. It finds reasonable guesses for the "centers" of latent structure blobs in the dataset. The algorithm then assigns each point to its closest center to predict community labels.

To define "closest", we use the Euclidean distance introduced in Concept 5.3.10. For two points $\vec{x}_i$ and $\vec{x}_j$ in d dimensions, the Euclidean distance is:

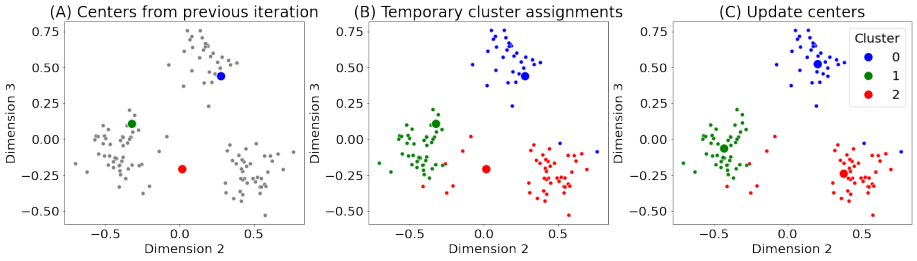

Figure D.1 **(A)** the centers for K-means from the previous iteration for each of the K clusters, **(B)** the cluster assignments for each point, **(C)** the updated centers, based on the cluster assignments.

$$\|\vec{x}_i - \vec{x}_j\|_2 = \sqrt{\sum_{l=1}^{d}(x_{il} - x_{jl})^2}$$

To illustrate the K-means process graphically, we examine a pairplot of the second and third dimensions:

```
from pandas import DataFrame
import seaborn as sns
import matplotlib.pyplot as plt

data = DataFrame({"Dimension 2" : Xhat[:,1], "Dimension 3" : Xhat[:,2]})
palette = {"0" : "blue", "1": "green", "2": "red"}
fig, ax = plt.subplots(1, 1, figsize=(6, 4))
sns.scatterplot(data=data, x="Dimension 2", y="Dimension 3", color="gray", ax=ax)
ax.set_title("Estimates of latent positions");
```

K-means requires initial center positions. While there are strategic methods for initialization (e.g., sklearn's `kmeans++`), we will use arbitrary starting locations for simplicity:

```
centers = np.array([[.5, .5], [-0.05, 0.05], [-0.05, -0.05]])
datcenters = DataFrame({"Dimension 2": centers[:,0], "Dimension 3": centers[:,1],
     "Cluster": ["0", "1","2"]})
```

Figure D.1(A) shows the initial center locations. We then identify which points are closest to each cluster center by computing the distance between each latent position estimate and the three centers. The smallest distance determines the point's cluster assignment:

```
from scipy.spatial import distance_matrix
distances = distance_matrix(Xhat[:,1:3], centers)
assignment = np.argmin(distances, axis=1)

data["Closest Center"] = assignment.astype(str)
```

Figure D.1(B) visualizes the point assignments. Some cluster centers are close to the center of blobs, but not always.

We now update our centers by taking the mean value (for each dimension) of the points which are assigned to that cluster:

```
centers = np.array([np.mean(Xhat[assignment == k,1:3], axis=0) for k in range(0, 3)])

datcenters = DataFrame({"Dimension 2": centers[:,0], "Dimension 3": centers[:,1],
     "Cluster": ["0", "1","2"]})
```

Figure D.1(C) illustrates the updated centers. The centers have shifted to the approximate midpoint of the points previously assigned to each center based on the old center positions. After one iteration, the centers appear well-positioned relative to the discernible data blobs.

For the second iteration, we treat the points as unlabeled again. We recompute the closest center for each point and update the centers accordingly:

```
distances = distance_matrix(Xhat[:,1:3], centers)
assignment = np.argmin(distances, axis=1)
centers_new = np.array([np.mean(Xhat[assignment == k,1:3], axis=0) for k in range(0, 3)])

data["Closest Center"] = assignment.astype(str)
```

This procedure repeats until the centers stabilize. Determining when to stop is an area of ongoing research, known as the *stopping criterion*. After two iterations, we typically observe fairly homogeneous blobs, with points from each of the three clusters assigned to the same cluster.

We can automate this process and produce predicted labels using `sklearn`:

```
from sklearn.cluster import KMeans

labels_kmeans = KMeans(n_clusters = 3, random_state=1234).fit_predict(Xhat)
```

To verify that we have identified the visually apparent blobs as uniform clusters, we can examine the pairs plot with the *predicted labels*:

```
from graspologic.plot import pairplot

_ = pairplot(Xhat, labels=labels_kmeans, title="Pairplot of embedding of $A$",
     legend_name="Predicted Cluster")
```

Figure D.2 illustrates the resulting K-means clustering. The discernible blobs from the latent positions are largely assigned to the same predicted clusters. We use these predicted cluster labels as the *predicted communities* for our Stochastic Block Model.

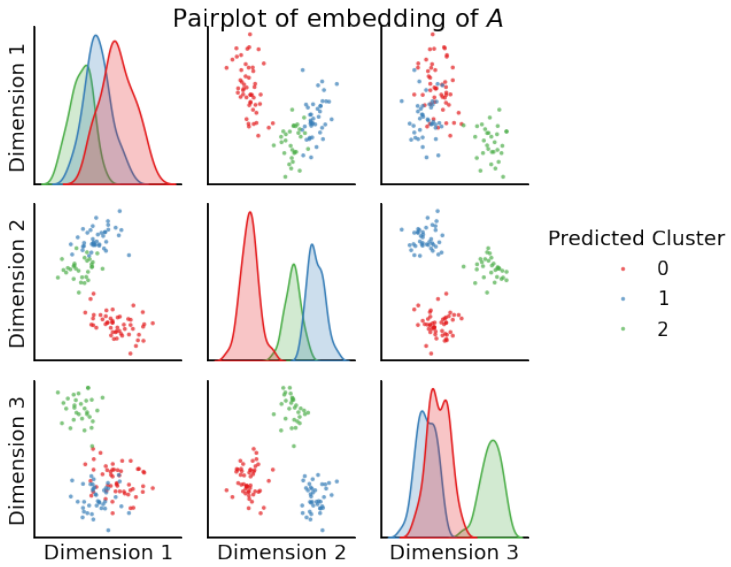

Figure D.2 A pairplot of the latent positions for the network, with points colored by their predicted cluster assignment (via K-means).

D.2.2 Evaluation of unsupervised learning techniques

Simulated data allows us to evaluate the quality of predicted community assignments against true community assignments. However, predicted labels may not align directly with true labels. True communities often impart meaningful information about the dataset (e.g., "School 1", "School 2", "School 3" rather than 0, 1, and 2). Since the clustering algorithm operates without prior knowledge of communities, predicted clusters do not necessarily correspond to original community names. For example, points in true community 0 might be assigned to predicted community 2, as the algorithm arbitrarily assigns cluster labels.

Visualizing prediction homogeneity with the confusion matrix

To address this limitation in evaluation, we employ a confusion matrix. A *confusion matrix* compares two sets of labels for a group of data points: one set of known true labels and another set of labels whose correspondence to the true labels is unknown. For L possible true label values and K possible predicted label values, the confusion matrix takes the form:

$$\begin{bmatrix} c_{11} & \dots & c_{1K} \\ \vdots & \ddots & \vdots \\ c_{L1} & \dots & c_{LK} \end{bmatrix},$$

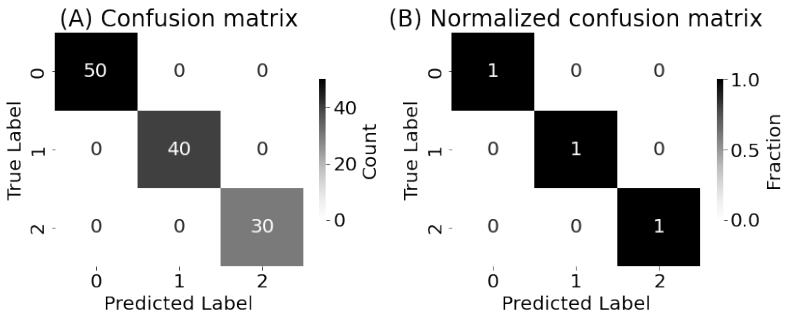

Figure D.3 **(A)** the confusion matrix, and **(B)** the normalized confusion matrix for a K-means clustering.

where each entry c_{lk} corresponds to the total number of points which are predicted to be in group l but have a true label of group k. We can compute and plot the confusion matrix using a heatmap:

```
from sklearn.metrics import confusion_matrix
# compute the confusion matrix between the true labels z
# and the predicted labels labels_kmeans
cf_matrix = confusion_matrix(z, labels_kmeans)
```

Figure D.3(A) illustrates the confusion matrix. An effective clustering typically results in a large proportion of nodes with a particular true label being assigned to the same predicted label. In this case, the predictions are homogeneous, with each predicted label corresponding to a single true label.

For a more generalizable approach, we normalize the confusion matrix by dividing the counts by the total number of points assigned to each true label:

```
cfm_norm = cf_matrix/cf_matrix.sum(axis=1)[:,None]
```

```
from graphbook_code import cmaps

fig, ax = plt.subplots(1,1, figsize=(6,4))
sns.heatmap(cfm_norm, cmap=cmaps["sequential"], ax=ax)
ax.set_title("Normalized confusion matrix")
ax.set_ylabel("True Label")
ax.set_xlabel("Predicted Label");
```

Figure D.3(B) demonstrates that all points in the normalized confusion matrix with a true label of 0 are assigned to predicted label 1, and similarly for the other rows.

Homogeneity of the confusion matrix and the adjusted rand index (ARI)

To assess the homogeneity of a predicted clustering relative to true labels, we use the Rand Index (RI). The RI examines all pairs of data points, considering whether their true labels and predicted labels are the same or different. We define two concepts:

1 Success: Two points with the same true labels have the same predicted labels, or two points with different true labels have different predicted labels. This indicates homogeneous alignment of the points with respect to true and predicted labels.
2 Failure: Two points with different true labels have the same predicted labels, or two points with the same true labels have different predicted labels. This indicates heterogeneous alignment of the points with respect to true and predicted labels.

The Rand Index is computed as the ratio of successes to total comparisons:

$$RI = \frac{\text{Successes}}{\text{Successes} + \text{Failures}}$$

The RI ranges from 0 to 1, with higher values indicating greater homogeneity between true and predicted labels.

However, the RI has a limitation when evaluating clustering with disproportionate true labels. Consider a dataset with 10 points, where 9 belong to class 1 and 1 belongs to class 2. If we blindly guess all points as class 1, we achieve an RI of 0.8 (72 successes out of 90 comparisons), which appears high despite learning nothing about the data structure.

Consider an extreme example with 10 points, where 9 belong to class 1 and 1 belongs to class 2. If we blindly guess all points as class 1, how does this affect the Rand Index (RI)?

We compare each of the 10 points with the other 9, resulting in 90 total comparisons. Let's analyze these comparisons:

1 When the first point is in class 1 (9 out of 10 cases):
   - Of the 9 remaining points, 8 will have the same true and predicted labels as the first point.
   - This yields $9 * 8 = 72$ successful comparisons.
2 When the first point is from class 2 (1 out of 10 cases):
   - All comparisons fail, as the other points have the same predicted label but different true labels.

Thus, the RI is $72/90 = 0.8$, a seemingly high value closer to the maximum of 1 than the minimum of 0.

To address this weakness in RI when dealing with unevenly distributed true labels, we use the Adjusted Rand Index (ARI) [1]. The ARI accounts for these imbalances in class distribution. We can easily compute the ARI using `sklearn`:

```
from sklearn.metrics import adjusted_rand_score

ari_kmeans = adjusted_rand_score(z, labels_kmeans)
```

```
print("ARI(predicted communities, true communities) = {}".format(ari_kmeans))
```

D.3 The Bayes plugin classifier

The Naive Bayes classifier [2] relies on class-conditional probabilities to determine which of Y possible classes a new sample belongs to, given features that are zeros and ones. This section explains the intuition behind this classifier, assuming a background in probability and statistics, particularly categorical random variables, statistical independence, and Bayes' theorem.

We begin by considering the probability of observing a particular sequence of edges in the signal subnetwork of the m^{th} network, given that the network is in class y. Let us denote this probability as:

$$Pr\left(\begin{array}{c}\text{observing } A^{(m)} \text{ given we assume } m \text{ is in class } y \\ \text{where the signal subnetwork is } \mathcal{S}\end{array}\right) = Pr(A^{(m)}|y_m = y; \mathcal{S})$$

Assuming independence of edges in the network, as in an independent-edge random network, we can express this probability as:

$$Pr(A^{(m)}|y_m = y; \mathcal{S}) = \prod_{i,j} Pr(a_{ij}^{(m)}|y_m = y; \mathcal{S})$$

Under the Signal Subnetwork model introduced in Section 4.10, each $A^{(m)}|y_m = y \sim IER_n\left(P^{(y)}\right)$ independently, and each $a_{ij}^{(m)}|y_m = y$ follows a Bernoulli distribution with probability $p_{ij}^{(y)}$. Thus, we can express the probability as:

$$Pr(A^{(m)}|y_m = y; \mathcal{S}) = \prod_{i,j} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1 - p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}} \tag{D.1}$$

We now apply Bayes' Theorem to compute the joint probability of observing both $A^{(m)}$ and y_m having the value y. This joint distribution can be expressed as:

$$Pr\left(\begin{array}{c}\text{observing } A^{(m)} \text{ and } m \text{ is in class } y \\ \text{where the signal subnetwork is } \mathcal{S}\end{array}\right) = Pr(A^{(m)}, y_m = y; \mathcal{S})$$

This expression represents the probability that the random network $\mathbf{A}^{(m)}$ takes

the value $A^{(m)}$ and the random class $\mathbf{y}_m$ has the value y simultaneously. Using Bayes' Theorem [3], we can write:

$$Pr(A^{(m)}|\mathbf{y}_m = y; \mathcal{S}) = \frac{Pr(A^{(m)}, \mathbf{y}_m = y; \mathcal{S})}{Pr(\mathbf{y}_m = y)}$$

Rearranging yields:

$$Pr(A^{(m)}, \mathbf{y}_m = y; \mathcal{S}) = Pr(A^{(m)}|\mathbf{y}_m = y; \mathcal{S}) Pr(\mathbf{y}_m = y)$$

Substituting the result from Equation (D.1):

$$Pr(A^{(m)}, \mathbf{y}_m = y; \mathcal{S}) = Pr(\mathbf{y}_m = y) \prod_{i,j} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1 - p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}}$$

Recall that in the signal subnetwork model, π_y represents the probability of our Y-sided die landing on class y, which is equivalent to the probability that the random class $\mathbf{y}_m$ takes the value y. Thus, $Pr(\mathbf{y}_m = y) = \pi_y$, and we can express the joint probability as:

$$Pr(A^{(m)}, \mathbf{y}_m = y; \mathcal{S}) = \pi_y \prod_{i,j} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1 - p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}} \tag{D.2}$$

The quantity we aim to calculate for our classifier is the posterior probability:

$$Pr\left(\begin{array}{c} m \text{ is in class } y \text{ given we observe } A^{(m)} \\ \text{where the signal subnetwork is } \mathcal{S} \end{array}\right) = Pr(\mathbf{y}_m = y|A^{(m)}; \mathcal{S})$$

This expression represents the probability that a given sample m is in class y, given that we observe the network $A^{(m)}$ with signal subnetwork $\mathcal{S}$.

Applying Bayes' Theorem once more, we can rewrite this expression as:

$$Pr(\mathbf{y}_m = y|A^{(m)}; \mathcal{S}) = \frac{Pr(A^{(m)}, \mathbf{y}_m = y; \mathcal{S})}{Pr(A^{(m)}; \mathcal{S})}$$

Using the expression from Equation (D.2):

$$Pr(\mathbf{y}_m = y|A^{(m)}; \mathcal{S}) = \frac{\pi_y \prod_{i,j} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1 - p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}}}{Pr(A^{(m)}; \mathcal{S})}$$

This posterior probability is particularly useful for network classification. When we encounter a new network and wish to assign it to a class, we aim to select the most probable class given the observed network. Thus, for a new network, we estimate its class as the one that maximizes this posterior probability:

$$\begin{aligned}\hat{y}_m &= \text{argmax}_{y\in\{1,\ldots,Y\}} Pr(\mathbf{y}_m = y|A^{(m)};\mathcal{S}) \\ &= \text{argmax}_{y\in\{1,\ldots,Y\}} \frac{\pi_y \prod_{i,j} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1-p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}}}{Pr(A^{(m)};\mathcal{S})}\end{aligned}$$

Observe that if $(i,j) \notin \mathcal{S}$, then $p_{ij}^{(y)} = p_{ij}^{(y')}$ for all possible classes y and y'. These terms in our product are irrelevant when finding the optimal $\hat{y}_m$, as they remain constant regardless of the class. Moreover, $Pr(A^{(m)};\mathcal{S})$ is not a function of the class and can be ignored (as it is a constant rescaling for every class). Thus, we simplify our expression to:

$$\hat{y}_m = \text{argmax}_{y\in\{1,\ldots,Y\}} \pi_y \prod_{(i,j)\in\mathcal{S}} \left(p_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1-p_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}} \tag{D.3}$$

This equation computes the probability that the class is y for all possible values of y (from 1 to Y), given the observed network. We then select the most plausible value as our prediction $\hat{y}_m$, ignoring components that do not depend on the class (edges not in the signal subnetwork and the unconditional probability $Pr(A^{(m)};\mathcal{S})$).

As $\mathcal{S}$, $\vec{\pi}$, and the probability matrices $P^{(1)},\ldots,P^{(Y)}$ are unknown, we employ a "plugin" classifier. We estimate the signal subnetwork $\mathcal{S}$ using methods outlined in Section 8.3 or Section 8.2 to produce $\widehat{\mathcal{S}}$. We then use the observed class vector $\vec{y}$ to estimate $\vec{\pi}$:

$$\hat{\pi}_y = \frac{M_y}{M}$$

where M is the total number of networks, and M_y is the number of networks where $y_m = y$. We estimate the probability entries as:

$$\hat{p}_{ij}^{(y)} = \begin{cases} \frac{1}{M_y}\sum_{m:y_m=y} a_{ij}^{(m)}, & (i,j)\in\widehat{\mathcal{S}} \\ \frac{1}{M}\sum_m a_{ij}^{(m)}, & (i,j)\notin\widehat{\mathcal{S}} \end{cases}$$

For each edge (i,j) in the estimated signal subnetwork $\widehat{\mathcal{S}}$, we compute the fraction of existing edges across all networks where $y_m = y$. Finally, we estimate the class for a new network $A^{(m)}$ by "plugging in" these values to the objective function in Equation (D.3):

$$\hat{y}_m = \text{argmax}_{y\in\{1,\ldots,Y\}} \hat{\pi}_y \prod_{(i,j)\in\widehat{\mathcal{S}}} \left(\hat{p}_{ij}^{(y)}\right)^{a_{ij}^{(m)}} \left(1-\hat{p}_{ij}^{(y)}\right)^{1-a_{ij}^{(m)}}$$

This equation represents the Bayes plugin classification rule.

Bibliography

[1] Rand WM. Objective Criteria for the Evaluation of Clustering Methods. J Am Stat Assoc. 1971 Dec;66(336):846–850.
[2] Hastie T, Tibshirani R, Friedman JH. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. New York, NY, USA: Springer; 2009.
[3] Joyce J. Bayes' Theorem; 2003. [Online; accessed 4. Feb. 2023].