1 Predicting Missing Links in Networks

Recall that most network data sets are *incomplete* in some way, either because of the way we measured them or because they are dynamic objects that change over time and we only get to observe them for part of their lifetime. When edges are the things that are missing, link prediction aims to identify those node pairs that are most likely to be missing links (or, future connections), based on their correlation with the links we do observe.

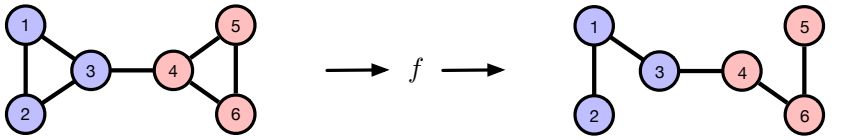
Link prediction serves three primary functions in network data science:

- *Suggesting missing or future connections*, because the observed network G° is known to represent an incomplete set of interactions.
- *Marshaling scarce resources* for edge discovery, in the setting where each pair of nodes i, j has to be measured directly, e.g., by a biological experiment or a social observation.
- *Comparing models*, by asking which model is better at "out of sample" predictions of edges. In this way, link prediction is akin to cross validation in machine learning, in which we divide the edge set E into a "training" set (observed links) and a "test" set (missing links), and then compare models in how well they can learn to predict the missing links based only on the observed links.[1]

In this setting, we assume that we have a complete account of a set of nodes V, but an incomplete account of the edges among them. Letting E denote the set of actual edges, and $E^\circ \subset E$ be the subset of them that we observe, we can define the observed network as $G^\circ = (V, E^\circ)$ and the actual network as $G = (V, E)$. Just as with missing node attributes, these is some *missingness function* f that chooses which subset of edges E° we actually observe. If there are some patterns within the observed edges E° that correlate with with the actual edges E, then we may be able to predict (better than baseline guessing) which of the observed unconnected pairs $Y = V \times V - E^\circ$ are in fact missing links $X = E - E^\circ$.

Here is a small example of this process of a missingness function f being applied to an actual graph G (on the left) to produce the observed network G° (on the right), where the removed lines are the missing links.

[1] Although this very analogy between link prediction and cross validation is commonly invoked, it is not mathematically exact. In the traditional setting where cross validation is well-understood mathematically, we assume we have iid draws of vectors from some underlying high-dimensional data generating process. Choosing a random partition of this set into two subgroups is mathematically well-defined because of the iid assumption. In networks, however, we lose the independence property, and a random partition of the edges may no longer be mathematically meaningful, e.g., when edges only ever appear in bundles or when edges perform some system-level function. Consider this example: what are the chances that a random 80% subset of the edges in your body's gene regulatory network be a functioning genome? Understanding how to do cross validation properly in a network setting is an active area of research.



actual $G = (V, E)$ observed $G^\circ = (V, E^\circ)$

The hardest form of link prediction is to make predictions using only the observed links E°, without leveraging any node attributes or metadata attached to the node, which may correlate with the existence of links, e.g., user demographics, species traits, or biological functions. When available, node attributes can sometimes make link prediction substantially easier, particularly in networks with very few edges available for analysis. Whether node attributes improve link prediction depends on whether the probability that a link exists correlates with the attributes of the node pair it connects, i.e., $\Pr((i,j) \in E \,|\, f(x_i, x_j))$, where $f()$ is a function of i and j's node attributes x_i and x_j. If the attributes x_i, x_j do not correlate with the existence of the link (i,j), then using node attributes will not improve link prediction accuracy. Hence, the hardest form of link prediction is precisely when node attributes are not correlated with links or when node attributes are absent entirely. The hard case is the problem setting we consider here, although when we get to advanced techniques, we will discuss how node attributes can be added into the models to improve predictions.

Mathematically, predicting missing links is a much harder problem than predicting missing attributes of either nodes or edges. With missing attributes, we start off knowing which attributes are missing $x_i = \emptyset$, but with missing links, we do not. Instead, our task is to sort among all the 0s of the adjacency matrix, which are the "non-edges" or unconnected pairs in the observed graph G°, to find those that should be 1s. Because most real-world networks are sparse, this task is like searching for $O(n)$ needles in a $\Theta(n^2)$ haystack. Hence, the baseline accuracy for guessing will be $O(1/n)$, making it extremely unlikely to guess correctly just by chance.

Link prediction methods operate by defining a *score* function over unconnected pairs $i, j \in Y$, and the better the predictor, the more likely it will assign a higher score to a pair i, j that is actually a missing link. The idea of guessing at random provides a simple baseline algorithm—if we can do better than this baseline, then we're getting somewhere—and this algorithm is equally likely to assign any particular score to each of the unconnected pairs.

The baseline predictor. In the absence of any information about f or about any patterns within E° and their relationship to the missing links in X, the simplest **baseline prediction** algorithm assigns a value that is independent of the graph G, meaning that every input pair i, j is equally

likely to receive a particular score. We can formalize this notion mathematically, as follows:

$$\text{if } i,j \in Y \quad \text{score}(i,j) = \text{Uniform}(0,1) \ . \tag{1}$$

That is, the baseline assigns a uniformly random score between 0 and 1 to each unconnected pair. If we apply this algorithm to the above example, and then *sort* the candidate pairs i,j by their score(i,j) values, we obtain a score table:

i	j	score(i,j)
1	5	r
1	4	r
1	6	r
2	3	r
2	6	r
2	4	r
2	5	r
3	5	r
3	6	r
4	5	r

where r is a stand-in for Uniform$(0,1)$, and the ordering is arbitrary among tied scores. (Note that in practice, ties must be broken randomly. Do you see why?) You can see the two missing edges $X = \{(2,3),(4,5)\}$ in the list, but there's nothing about their scores to suggest that they are any more or less likely to be the missing links than another pair.

If some correlation exists between the observed edges E° and the missing edges X, then we can likely improve considerably over this baseline. Just like with predicting missing node attributes, there are many ways we could do this. In practice, there are three approaches:

- **Topological predictors** are simple functions of the joint local network structure of the pair of nodes i,j being scored, e.g., their degrees, measures of overlapping neighborhoods, geodesic distance, and other measures that summarize a network's structure.
- **Model-based predictors** are a broad class of algorithms that rely on models of large-scale network structure, e.g., by decomposing the network into modules or communities, to make predictions about missing links.
- **Embedding-based predictors** are second broad class of algorithms that assign nodes to locations in a d-dimensional latent space in such a way that nodes are "close" in the embedded space if they are, or are likely to be, connected.

In this lecture, we'll focus primarily on topological predictors, and then briefly consider the remaining two in Section 2 before considering advanced techniques.

1.1 Topological predictors and local smoothing

Topological predictors use the structure of the graph itself, from the perspective of nodes i, j as a way to create a score function that may correlate with the missing links. Each such predictor is based on a specific assumption about how edges form. For instance, we might score a pair by the number of common neighbors (the more, the better), the number of shortest paths between the nodes, the product of their degrees, etc.[2]

Just as node attributes are often assortative (or, homophilous) in networks, edges themselves are often **assortative**, meaning that they tend to cluster together. This pattern is the idea behind the clustering coefficient, which measures the "local" density of edges. And, just as it did for node attributes, this pattern allows us to construct a kind of local smoothing algorithm for predicting missing links. Specifically, we can

predict a missing link to occur where it would cluster with other edges.

There are many ways to operationalize such an idea into a specific algorithm for predicting missing links. We will explore two. One uses the **Jaccard coefficient**, which quantifies the degree of overlap among two nodes' neighborhoods. Using the same notation as in the Lecture on predicting missing node attributes, let $\nu(i)$ return the set of neighbors of node i. The Jaccard coefficient is defined as

$$\text{Jaccard}(i,j) = \frac{|\nu(i) \cap \nu(j)|}{|\nu(i) \cup \nu(j)|} \ , \tag{2}$$

which measures the fraction of neighbors of either node that are neighbors of both nodes, i.e., the density of common neighbors. Two nodes with many common neighbors will have a higher Jaccard coefficient. In that case, adding an edge between i, j would then "close" each of the open triads i, k, j, which would increase both their local, and the global, clustering coefficient.

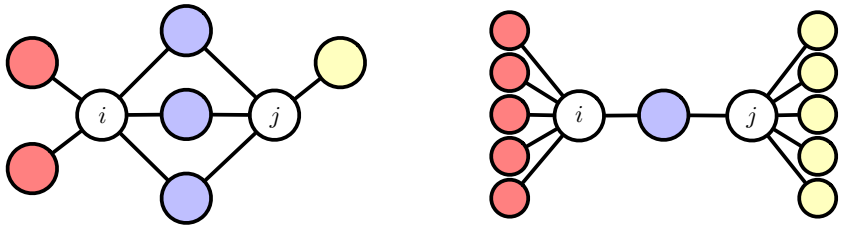
To turn this pairwise network measure into a prediction algorithm, we say

$$\text{score}(i,j) = \text{Jaccard}(i,j) + \text{Uniform}(0,\epsilon) \ , \tag{3}$$

where $\text{Uniform}(0,\epsilon)$ adds a small amount of noise in order to break ties randomly without changing the relative ordering outside those ties. (Do you see why this is necessary?)

Consider the following small examples. On the left, $\nu(i)$ returns 5 nodes, $\nu(j)$ returns 4 nodes,

[2] There are an enormous number of topological predictors for missing links. Examples include the number of common neighbors, shortest-path betweenness centrality, the Leicht–Holme–Newman index, personalized page rank, shortest path length, the mean neighbor entries within a low rank approximation, the Jaccard coefficient, the Adamic–Adar index, the resource allocation index, the dot product of columns i and j in a low rank approximation via a singular value decomposition, the local clustering coefficients of i and j, the average neighbor degrees, and any number of "centrality" measures.



and there are 3 common neighbors. Hence, the Jaccard$(i,j) = 0.50$, reflecting the relatively high proportion of total neighbors that i, j have in common. On the right, i, j have many fewer common neighbors, and their Jaccard coefficient is correspondingly smaller, only 0.091.

A second local topological predictor is the **degree product**, which embodies the idea we learned from our analysis of random graphs that nodes with high degrees are likely themselves to be connected, just by chance. We define this predictor as score$(i,j) = k_i k_j + \text{Uniform}(0, \epsilon)$. How would degree-product scores differ from those of the Jaccard coefficient for the above examples?

Let us return to the example network on the first page of this lecture, from which we removed two edges, and then apply the Jaccard and degree-product predictors to the observed network G'. Doing so produces the following score tables:

i	j	Jaccard score(i,j)
4	**5**	$1/2 + r$
2	**3**	$1/2 + r$
3	6	$1/3 + r$
1	4	$1/3 + r$
1	5	r
1	6	r
2	6	r
2	4	r
2	5	r
3	5	r

i	j	degree product score(i,j)
1	4	$4 + r$
1	6	$4 + r$
3	6	$4 + r$
1	5	$2 + r$
2	**3**	$2 + r$
2	6	$2 + r$
2	4	$2 + r$
3	5	$2 + r$
4	**5**	$2 + r$
2	5	$1 + r$

where the missing links $X = \{(2,3), (4,5)\}$ are highlighted, and links with the same score are ordered arbitrarily.

On the left, the Jaccard link predictor does very well at assigning these missing links high scores compared to the non-missing links, while on the right, the degree product seems very poor. These different behaviors illustrate a key point about link predictors: each predictor captures different types of correlations between observed and missing edges, and hence each will perform well on some inputs and poorly on others. In this case, the actual network has substantial local edge density, and so the Jaccard function does well. But the degree product predictor would perform better

if the network had much higher variance in its degree structure.

To quantify these differences in performance, compare them to the baseline, or understand how close they might be to an *optimal* predictor, we need a compact way of quantifying their performance.

1.2 Measuring performance: the AUC

Predicting missing links is a binary classification problem: every candidate $i, j \in Y$ is either a missing link or not.

To summarize the performance of a link prediction algorithm, we can use the **AUC** statistic,[3] [4] a context-agnostic measure of its ability to distinguish a missing link $i, j \in X$ (a true positive, or TP) from a non-edge $Y - X$ (a true negative, or TN). The AUC has two other attractive properties:

1. *scale invariance*, meaning it is not dependent on the scores themselves, and instead only depends on their relative values, and
2. *threshold invariance*, meaning it is a general measure of accuracy that doesn't require choosing a threshold on the scores for making predictions.

Mathematically, the AUC can be defined as

$$\text{AUC} = \Pr[\ \text{score(TP)} > \text{score(TN)}\ ] \quad . \tag{4}$$

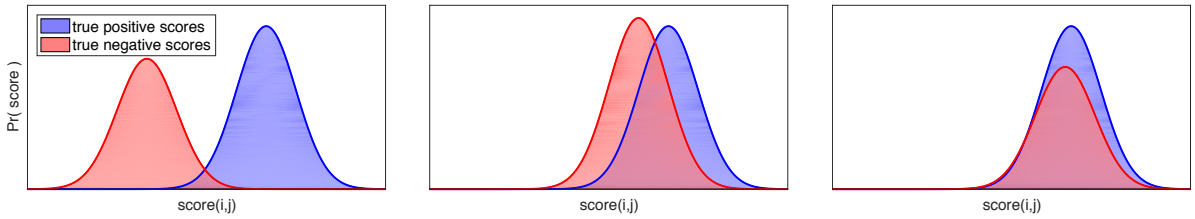
That is, if we choose a uniformly random missing edge $a, b \in X$ (TP) and a uniformly random non-edge $c, d \in Y - X$ (TN), the AUC is the probability that score(a, b) is higher than score(c, d).[5] The figure below illustrates this idea, for three hypothetical predictors. On the left, the predictor assigns scores in a way that the score distribution for true positives is well-separated from the score distribution of true negatives, which produces a high AUC value. In the middle example, the predictor assigns scores so that the distributions overlap more, which lowers the AUC. And, on the right, the predictor assigns scores so that the distributions are nearly the same, leading to a very low AUC.

If the algorithm is no better than guessing at random, then AUC $= 0.5$ and it is as equally likely that score(TP) $<$ score(TN) as it is that score(TP) $>$ score(TN). If that's the case, the probability

[3] AUC is short for "Area Under the Curve," where the "curve" here is the Receiver Operating Characteristic (ROC) curve. ROC curves were invented during World War 2 as a method for assessing the performance of radar at detecting enemy aircraft. Neato.

[4] There are other ways to quantify accuracy in this setting, each of which has a slightly different way of weighting the cost of different types of errors. Common choices include the F1-measure, which is the harmonic mean of the precision and recall, or the entire precision-recall curve. These alternatives are less context agnostic than the AUC, and hence may provide a better guide to performance in particular settings.

[5] Fun fact: the AUC is mathematically equivalent to the Mann-Whitney U test and to the "probability of superiority," which is a concept about statistical effect sizes.



AUC = 0.99 AUC = 0.76 AUC = 0.55

of the latter is 1/2, and hence that the AUC = 0.5. The baseline predictor of Eq. (1), which assigns a uniformly random value to every candidate, has exactly this behavior and thus has an AUC = 0.5. Hence, any algorithm with AUC > 0.5 does better than chance (the baseline). The maximum value of AUC = 1.0 is attained only when the algorithm assigns a higher score to every missing link (true positive) than it does to any non-edge (true negative).

In practice, we compute a link prediction algorithm's AUC by numerically integrating its corresponding "ROC" curve to get, literally, the area under the curve.[6] Formally, the ROC curve is a parametric plot of the *true positive rate* (TPR) and the *false positive rate* (FPR), as a function of prediction threshold. The idea is straightforward. Imagine drawing a line across one of our score tables just below the ℓth row, and then sliding it up, or down the table. This line represents a prediction "threshold": all the rows ℓ and above we predict to be missing links, while all rows $\ell+1$ and below we predict to be non-missing links. The TPR is the fraction of all the actual missing links that are in the predicted-to-be-missing set (at or above row ℓ), and the FPR is the fraction of all the actual non-missing links (non-edges) that are in that same set. Plotting TPR(ℓ) vs. FPR(ℓ) sweeps out the ROC curve on the unit square, starting at $(0,0)$ and ending at $(1,1)$.

To calculate the AUC, we begin by augmenting the score table with three additional columns: a column τ that gives the *actual* status of the ℓth row (pair i,j) as either a true positive ($\tau_\ell = 1$) or true negative ($\tau_\ell = 0$), and then two columns, one for TPR(ℓ) and one for FPR(ℓ). If a predictor performs well, then in the τ column, the 1s will tend to be located higher up in the table, while if it is no better than baseline, the 1s will be scattered uniformly at random within this column.[7]

Once the τ column has been added, we can fill in the TPR and FPR columns simultaneously in a single scan down the table. Let $\mathcal{T} = |X|$ be the total number of true positives (missing links; equal to the column sum of τ), and let $\mathcal{F} = |Y - X|$ be the total number of true negatives (non-edges;

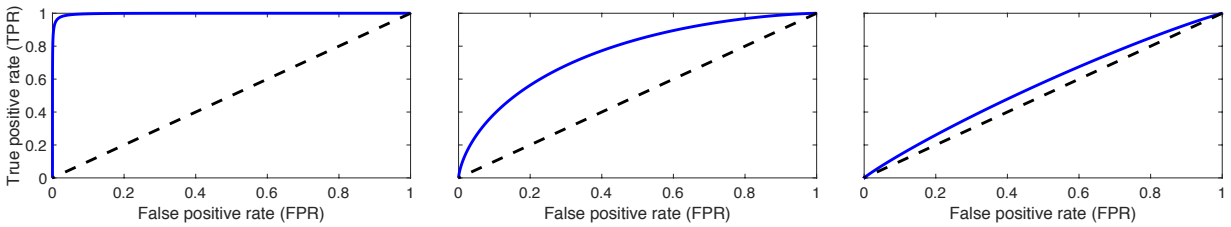
[6] We can also estimate the AUC via Monte Carlo, by evaluating Eq. (4) directly: sample many pairs of true positives and true negatives, and estimate the fraction of draws for which the former is assigned a higher score than the latter.

[7] If the 1s tend to be stacked at the bottom of the τ column, then the predictor is, in fact, pretty good at distinguishing 1s from 0s, it just gets it backwards. We can then construct a better-than-baseline predictor out of a worse-than-baseline predictor by simply doing the opposite of what it says (or by inverting its output scores).

equal to the length of the table minus the sum of τ). The TPR(ℓ) and FPR(ℓ) are then defined as

$$\text{TPR}(\ell) = \frac{1}{\mathcal{T}} \sum_{k=1}^{\ell} \tau_k \qquad \text{FPR}(\ell) = \frac{1}{\mathcal{F}} \sum_{k=1}^{\ell} 1 - \tau_k \quad . \tag{5}$$

Using the same three sets of hypothetical score distributions we saw above, we can now compute the corresponding TPR and FPR functions, and then plot them against each other to obtain the ROC curve for each case. As a reference, each plot also shows the TPR = FPR line, which represents the baseline predictor's performance.



AUC = 0.99 AUC = 0.76 AUC = 0.55

The area under each ROC curve is each hypothetical predictor's AUC, which we calculate by numerically integrating the ROC curve, using a simple box-rule approximation[8] technique, running down the elements of the TPR and FPR vectors:

$$\text{AUC} = \sum_{\ell=1}^{|Y|} \text{TPR}(\ell) \times [\text{FPR}(\ell) - \text{FPR}(\ell - 1)] \quad , \tag{6}$$

where we define FPR(0) = 0. Note that sometimes the area of a box we're adding up will be zero (do you see when this will happen?), and that's okay.

1.3 An example

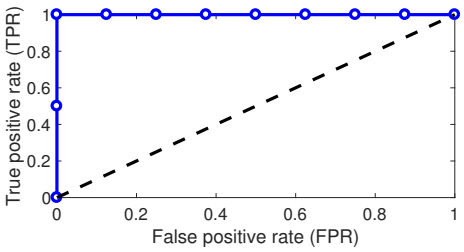
To see how this works in practice, let's consider again the Jaccard and degree-product link predictors from Section 1.1. The two score tables below now include the three new columns: the actual status τ, the corresponding true positive rates (TPR), and the corresponding false positive rates (FPR) for a threshold drawn between that row and the next. (Exercise: verify that the columns were calculated correctly.)

[8] Recall the box rule for numerical integration: given a function $f(x)$, a range $[x_s, x_t]$, and an increment Δx, the integral $\int_{x_s}^{x_t} f(x)\mathrm{d}x \approx \sum_{x=x_s}^{x_t} f(x) \times \Delta x$. The trapezoid rule is slightly more accurate, although the difference is negligible when Δx is small, as is the case in link prediction (do you see why?).

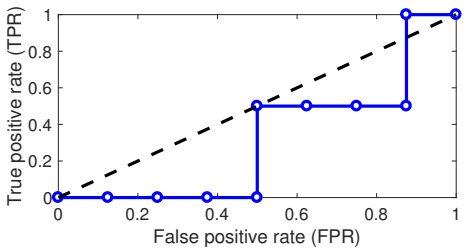
From these, we can plot the corresponding ROC curves, and apply Eq. (6) to calculate the AUC for each predictor. Unsurprisingly, the Jaccard predictor achieves perfect performance, with an AUC = 1.00. In this network, the degree-product predictor is in fact *worse* than guessing at random, producing an AUC = 0.31. This value is not as precise as it seems, however. Recall that the value r in the score column is a random variable. This means that the positions of the two true positives in the degree-product table will vary within the block of score$(i,j) = 2+r$ rows, and these lead to slightly different TPR and FPR functions. The right thing to do here is to compute an *average AUC*, when these ties are broken randomly, which produces a more reliable estimate of the AUC. The highest value possible, if the ties break luckily for the missing links is 0.62, while the lowest, is 0.12 (do you see why?). The average is AUC = 0.37.

i	j	τ_k	TPR$_\ell$	FPR$_\ell$	Jaccard score(i,j)
4	5	1	0.5	0.0	$1/2+r$
2	3	1	1.0	0.0	$1/2+r$
3	6	0	1.0	0.125	$1/3+r$
1	4	0	1.0	0.250	$1/3+r$
1	5	0	1.0	0.375	r
1	6	0	1.0	0.500	r
2	6	0	1.0	0.625	r
2	4	0	1.0	0.750	r
2	5	0	1.0	0.875	r
3	5	0	1.0	1.00	r

i	j	τ_k	TPR$_\ell$	FPR$_\ell$	degree product score(i,j)
1	4	0	0.0	0.125	$4+r$
1	6	0	0.0	0.250	$4+r$
3	6	0	0.0	0.375	$4+r$
1	5	0	0.0	0.500	$2+r$
2	3	1	0.5	0.500	$2+r$
2	6	0	0.5	0.625	$2+r$
2	4	0	0.5	0.750	$2+r$
3	5	0	0.5	0.875	$2+r$
4	5	1	1.0	0.875	$2+r$
2	5	0	1.0	1.000	$1+r$



Jaccard coefficient, AUC = 1.00



degree product, AUC = 0.31

2 Advanced link prediction methods

The basic methods described above are both simple unsupervised, topological predictors: both are simple functions of the topological structure around the target pair i,j. Dozens and potentially hundreds more such topological predictors have been defined for predicting missing links. Some have relatively straightforward interpretations, like the length of the shortest path ℓ_{ij} or

the Adamic-Adar index,[9] while others are considerably more abstract, like the i, j entry in a low rank approximation (LRA) via singular value decomposition of the adjacency matrix. In practical applications, if a particular individual predictor has a strong theoretical basis, e.g., we know that the underlying cause of edges existing is related to short paths, then it can be sufficient to consider it alone.

Modern approaches to link prediction, however, are typically more agnostic about the underlying causes of edges, and they instead recast the problem as a machine learning task. That is, we aim to learn some kind of model of $\Pr(i, j \,|\, \theta)$ for some parameters θ, and then use that model to predict missing links, expecting that missing links have higher likelihood under the learned model than non-edges. There are two common approaches in this direction:

- *Featurize the problem.* For each *pair* of nodes $i, j \in X$, generate feature vectors $\mathbf{x}_{ij}$ that quantify various aspects of the pair that we hope correlate with the existence of the edge. There are two main ways to generate $\mathbf{x}_{ij}$.

  - Topological features:
    Define $\mathbf{x}_{ij}$ as a list of node-level statistics like centrality measures, node degrees, local clustering; pair-level statistics like the Jaccard coefficient or degree product; and network-level statistics like the number of nodes, etc.; and even individual node attributes or functions of thereof.
  - Embedding features:
    Define $\mathbf{x}_{ij}$ by first using representation learning tools like neural embeddings, etc., to convert the observed graph G^o into a set of n vectors, one for each node, in a high-dimensional $\mathbb{R}^d$ space, with d dimensions, such that nodes that are connected are closer in this space than nodes that are not connected.[10] Graph neural networks and methods like DeepWalk or node2vec can produce these embeddings.

  Given the features $\mathbf{x}_{ij}$, we can then use a standard supervised learning algorithm to learn a predictive model $y = F(\mathbf{x}_{ij}, Y)$ over this vector space.

  Advantage: flexibility. One can define features in a wide variety of ways and rely on the supervised learning algorithm to figure out which features do and do not correlate with edges, and, it's easy to extend this model to add new features, including node attributes.

  Disadvantages: it requires a supervised learning step, and is thus more data intensive (works better on medium or larger networks) and offers less interpretability, particularly if we use embedding features, which obscure the underlying reasons for good predictions.

[9] The Adamic-Adar index is defined as the sum of reciprocals of the logarithm of the degrees of common neighbors of i, j: $\mathrm{AA}(i, j) = \sum_{u \in \nu(i) \cap \nu(j)} 1/\log(\nu(u))$.

[10] This assumptions is just local smoothing or homophily, in another guise, within the embedding or latent space.

- *Bayes the problem.* Define a Bayesian network model $\Pr(G, \mid \theta)$, sometimes called a probabilistic generative model, that specifies a conditional probability for each pair i, j being connected, and assume that the true network is an instance drawn from this graph distribution. Then, estimate the model's parameters using the observed graph, i.e., learn $\Pr(G^{\circ} \mid \hat{\theta})$, and score the $i, j \in X$ by their probability of connection under that model $\Pr(i \to j \mid \hat{\theta})$. The most popular forms of these models are variations of the stochastic block model (SBM), which we will learn about later in the semester.

  Advantages: interpretability and handling of uncertainty. The parameters θ of Bayesian network models are often directly related to theoretical assumptions about what drives the structure of the network, making it easier to learn as scientists why the model made certain predictions and not others, and because the method uses a conditional probability model, it naturally provides a quantitative measure of uncertainty about its predictions.

  Disadvantages: these models can be difficult to extend to include new types of information, like edge weights, node attributes, etc., and can require specialized estimation algorithms.

Both of these approaches are *global* prediction algorithms because they leverage information across the entire network to make each particular prediction. Hence, they end to be more computationally expensive than the basic topological predictors we learned about above. Within these broad categories, two are particularly popular.

Caveats. Although both families of algorithms can produce good predictions of missing links, the No Free Lunch Theorem in machine learning[11] largely guarantees that no one algorithm can be best across all input networks. Interestingly, recent work indicates that algorithm performance tends to vary strongly depending on the domain of the network: social networks are generally the easiest setting, with most modern algorithms performing well, while economic and biological networks are among the hardest.[12]

2.1 Model stacking for link prediction

Among the modern approaches, one technique called *model stacking* (also called stacked generalization) has the advantages of being both efficient and highly accurate. It runs well on commodity hardware, scales to large networks, and can preserve model interpretability. Model stacking is an example of the featurization family: it uses topological predictors to construct the features used in supervised learning. In general, stacking is an ensemble technique that learns to combine weakly predictive "base models" (topological predictors) to construct an optimal predictive distribution. In

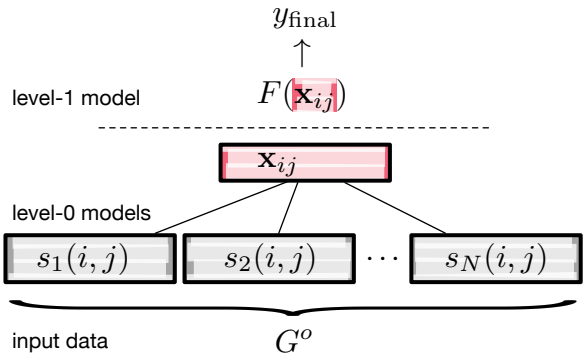
[11] See Wolpert and Macready, "No Free Lunch Theorems for optimization." *IEEE Transactions on Evolutionary Computation* **1**(1), 67–82 (1997)

[12] See Ghasemian et al., "Stacking Models for Nearly Optimal Link Prediction in Complex Networks." *Proc. Natl. Acad. Sci. USA* **117**(38), 23393–23400 (2020).

data applications, model stacking is often more accurate than techniques like Bayesian model averaging when true data generating process, of in our case the true missingness function f, is unknown.

At a high level. Model stacking is a two-level model. The first level trains a set of N base learners or level-0 models $\{s_1, s_2, \ldots, s_N\}$ on a common set of pairs X, and then "stacks" a meta-learner or level-1 model on top to learn to synthesize the level-0 outputs into a final prediction y. Level-0 models can be any supervised or unsupervised technique—here, they will be unsupervised topological predictors.

Each base learner s_ℓ maps an input $i, j \in X$ to a predicted output score $s_i(x) \in \mathbb{R}$. Applying these base learners to the candidate pairs $s_\ell(i, j)$ produces a feature vector $\mathbf{x}_{ij} = [s_1(x), s_2(x), \ldots, s_N(x)]$ that serves as training data for the meta-learner F. Hence, the level-1 model learns to predict the true target $y \in \mathbb{R}$ by combining various level-0 model predictions: $y_{\text{final}} = F(\mathbf{x}_{ij})$.



2.1.1 Defining level-0 and level-1 models

Specifying a stacking model for link prediction requires making several design choices. The most important pair of decisions is selecting the supervised learning algorithm for the level-1 predictor, and selecting the set of functions that constitute the level-0 predictors.

Choosing a level-1 model. Nearly any supervised machine learning algorithm can play the role of the level-1 model. For scientific applications where interpretability is important, we often prefer level-1 models with good interpretability, e.g., a random forest, logistic regression, or extreme gradient boosting (XGB). Random forests and XGB are among the most common choices because of their flexible functional forms. Both are ensemble methods themselves that construct a set of decision trees, but differ in how they do it. Random forests are a *bagging* technique that bootstraps the data for each tree and trains a decision tree on a random subset of features within each bootstrap. XBG is a *boosting* technique that trains the decisions trees in a sequence, in which for each new tree the lowest-scoring examples of the previous tree are upweighted, i.e., the next tree is trained

to correct the errors of the previous tree.

Choosing level-0 models. Nearly any function of a network's nodes, edges, and their respective attributes, can work as a level-0 predictor. The most common choices are precisely the unsupervised and often computationally lightweight topological functions we have learned about previously. At a high level, these functions come in three flavors: pair, node, and global:

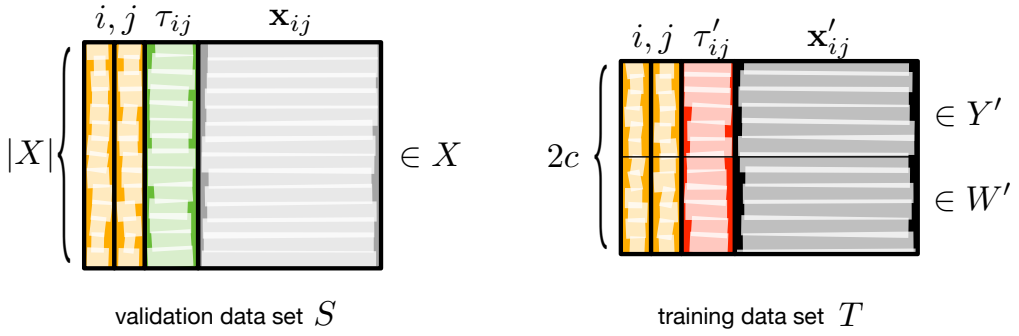
- *Pair features* are simple or complicated functions of the properties of the pair i, j. Examples include pairwise measures like the number of common neighbors $\mathrm{CN}(i, j)$, the degree product $\mathrm{DP}(i, j)$, the Jaccard coefficient $\mathrm{JC}(i, j)$, and the Adar-Adamic index $\mathrm{AA}(i, j)$. If a network has node attributes, any function of the pair of attributes can be included.
- *Node features* are properties of individual nodes, such as node-level summary statistics. Examples include a node's degree k_i, centrality measures like harmonic h_i, betweenness b_i, etc., local clustering coefficient C_i, etc. For each pair $i, j \in X$, we include in $\mathbf{x}_{ij}$ the pair of the node-level predictors for i and j. Node attributes can also be used as node-level predictors.
- *Global features* are network-level statistics like the number of nodes n or edges m, which can help contextualize other network features if the training step pools examples from multiple networks. If the training step uses only a single network, then these features cannot improve the level-1 model (do you see why?).

2.1.2 Training a stacked model

To train a stacked model, we use a two-step process, because we need to hold out one set of edges to evaluate the level-1 predictor's accuracy, and hold out another set of edges to train it. Here, we start by describing the training process at a high level, and then provide pseudocode for the full process at the end.

A validation data set. First, we derive the observed network $G' = (V, E')$ (previously called G^o), by 'observing' each edge $(i, j) \in E$ from the original network with probability α. This step defines two important 'observed' sets: $Y = E - E'$, the edges missing from G' (true positives), and $W = V \times V - E$, the non-edges in G (true negatives). The candidate missing edges in G' are then the set $X = Y \cup W$.

Then, we tabulate the *validation* data set S, which will be used to evaluate the level-1 model after training. Each row in the matrix S lists an unconnected pair $i, j \in X$ along with a binary variable indicating its ground truth status $\tau_{ij} = 1$ if $(i, j) \in Y$ and its corresponding feature vector $\mathbf{x}_{ij}$. We calculate the feature vector for i, j using the observed network G' by applying each of the level-0 predictors g_ℓ. Hence, S has the same structure as the table used in the basic link prediction section above, but now with multiple features for each pair $i, j \in X$.



Training the stacked model. Second, we derive the training network $G'' = (V, E'')$, by applying the 'observation' process to G', so that each edge $(i, j) \in E'$ from the observed network is included in the training network with probability α. (Hence, each edge in G appears in the training network G'' with probability α^2.) This step defines two training sets: $Y' = E' - E''$, the edges missing from G'' (true positives), and $W' = V \times V - E'$, the non-edges in G' (true negatives).

Then, we tabulate the *training* data set T, which will be used to train the level-1 model. In this step, we make the explicit choice to balance the positive and negative classes for training purposes, choosing an equal number c from Y' and W' to include in T. (A common choice is $c = 100$, or $c = 1000$, if n is large.) Each row in the matrix T lists an unconnected pair i, j along with a binary variable $\tau'_{ij} = 1$ if it was drawn (with replacement) from Y', and $\tau'_{ij} = 1$ if it was drawn from W', and its corresponding feature vector $\mathbf{x}'_{ij}$, calculated using the training network G''. Hence, the training data T has $2c$ rows and balanced classes, while the validation data S has $|X|$ rows and unbalanced classes.[13]

Finally, we train the level-1 model F on the training data T, using τ' as the prediction or dependent variable and the feature vectors $\mathbf{x}'_{ij}$ as the features or independent variables. Then, we evaluate the accuracy of F by applying it to the validation data S, using τ to score its performance at recovering missing links.

Complexities and caveats. Two notes about this process that make it different from a traditional machine learning setup.

- Edges are not independent in networks, which means that the division between validation

[13] If the level-1 model F requires hyperparameter optimization, we can generate a second training data set T' by sampling c pairs from Y' (with replacement) and c pairs from W' (with replacement, and then subdivide it as needed, e.g., 80-20 or 5-fold, to perform the hyperparameter optimization. Similarly, to generate more variability within the training data set, we can repeat the $G' \to G''$ observation process as many times as needed to generate additional data for training and tuning.

and training data is not ideal. Unlike vector data, when we remove an 'observation' from the actual network G to create the observed network G', we are changing the feature values that we will calculate, because we have changed an element of the adjacency matrix from $A_{ij} = 1$ to $A_{ij} = 0$, which may alter the output of any function of the graph g. Hence, the observed network is not an iid subsample of the original data—it is an altered form of the actual network, which includes misleading signals (the missing edges). The same applies for the relationship between the training network G'' and the observed network G'.

- Because real-world networks are sparse, the size of the true positive set can be quite small, particularly in the training data where edges are included with probability α^2. We improve the trained model by employing a class balancing step when we create the training data, by upsampling the true positives and downsampling the true negatives.

Finally, we note that the above process omits any hyperparameter tuning that model-1 might need. This step would be done using the training data T.

2.1.3 Stacked model pseudocode

From the above verbal description, we can now write down pseudocode for the validation and training data processes:

`function stacked-model`(G, γ):

1. `create validation data set` S
   (a) create the observed network $G' = (V, E')$ by observing each edge $(i, j) \in E$ with probability α.
   (b) define $Y = E - E'$ (true positives in G)
   (c) define $X = V \times V - E'$, the observed unconnected pairs set
   (d) for each pair $i, j \in X$, use G' to calculate the pair's feature vector $\mathbf{x}_{ij}$, using the N level-0 predictors
   (e) create a matrix S, with $|X|$ rows and $N + 3$ columns: columns 1-2 store the i, j node indices; column 3 stores the ground-truth variable τ_{ij}; and the remaining N columns store the feature vector $\mathbf{x}_{ij}$
2. `create training data set` T
   (a) create the training network $G'' = (V, E'')$ by selecting each edge $(i, j) \in E'$ with probability α.
   (b) define $Y' = E' - E''$ (true positives in G')
   (c) define $W' = V \times V - E'$ (true negatives in G')
   (d) select c pairs from Y' with replacement (upsampling)
   (e) select c pairs from W' with replacement (downsampling)
   (f) for each selected pair i, j, use G'' to calculate its feature vector $\mathbf{x}'_{ij}$, using the N level-0 predictors
   (g) create matrix T, with $2c$ rows and $N + 3$ columns: columns 1-2 store the i, j node indices; column 3 stores the truth variable τ'_{ij}; and the remaining N columns store the feature vector $\mathbf{x}'_{ij}$
3. `train and evaluate the model`
   (a) train the level-1 model F on T, learning to predict τ'_{ij} from $\mathbf{x}'_{ij}$
   (b) apply trained model F to validation data S, to make predictions y_{final} on all X unconnected pairs in G'
   (c) evaluate accuracy of y_{final} relative to τ_{ij}

2.1.4 Performance

[[under construction]]