Link Prediction Via Algorithm Selection

Owen Kutzscher

Boulder, Colorado

owenkutzscher@gmail.com github.com/owenkutzscher

Biological Networks CSCI 3352

Introduction

Within networks, link prediction is extremely useful. Large scale networks are being applied to many impactful domains including social network analysis, biological network analysis, and cybersecurity. Within real world networks, it is difficult and often impossible to get all of the data to complete the graph. This is where link prediction comes in.

Link prediction algorithms help to fill in missing edges in a network. There are many types of link prediction algorithms, and different structural configurations of graphs cause different edge prediction algorithms to perform significantly better than others making it difficult to know which algorithm to use for a graph.

What if there was a way to use these many link prediction methods simultaneously? In this paper I will explore what happens when we train a machine learning model to pick, among a set of edge prediction algorithms (predictors), the best predictor to use for any potential missing edge in a graph. The machine learning model will pick which predictor to use on an edge by edge basis, taking as input information about the graph the edge came from, as well as the data from all the predictors it has at its disposal to choose from.

This project is heavily inspired by "Stacking models for nearly optimal link prediction in complex networks"[4], this study is concerned with combining many models to get a more optimal prediction, and my study is focused on picking the best predictor of the available predictors.

Methods

Overview

There are five distinct parts of this project: Data, Edge Prediction Algorithms, Data Normalization, Model Training, and Visualizing Model Performance.

The *Data* section involves downloading graph data.

Edge Prediction Algorithms involves running many edge prediction algorithms on the graph data.

Data Normalization involves normalizing the data so every edge prediction algorithm produces a score between 0 and 1.

Model Training involves training several machine learning models on the normalized data. The models will be trained to guess the 'best' predictor for any given edge.

Finally there is the *Visualizing Model Performance* section where we will plot an ROC plot to compare ML-model performance to each predictor's performance, and a bar chart to visualize each of the model's average AUCs compared to the predictors.

All these sections will be run with several different alpha values. Alpha represents the proportion of edges hidden from the predictors and models, a smaller alpha value indicates that a smaller portion of edges will be visible to the predictors and models.

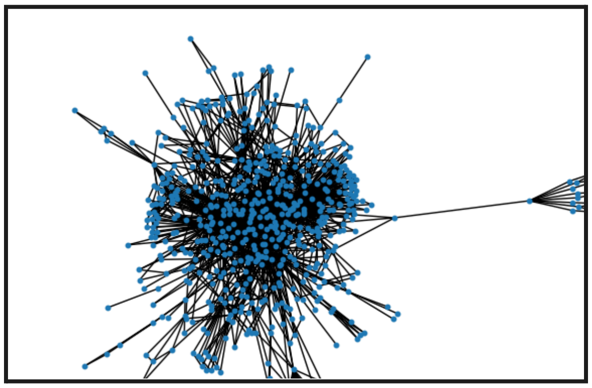
Data

There are three sets of graph data we will use to train the models. Metabolic networks, protein interaction networks, and tissue networks. During model training the models are trained and tested on graphs from the same dataset.

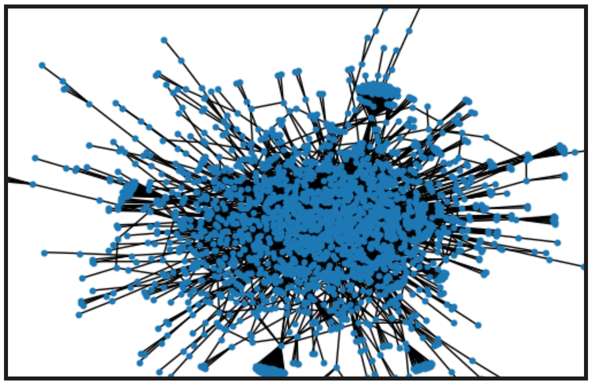
The data is sourced from the study that my study is inspired by: "Stacking models for nearly optimal link prediction in complex networks"[4]. This study sourced their data from ICON. [2]

The metabolic, protein interaction, and tissue networks respectively have 26, 20 and 18 graphs within each of them, an average of 673, 882, 305 nodes per graph, and an average of 1648, 1797, and 403 edges per graph.

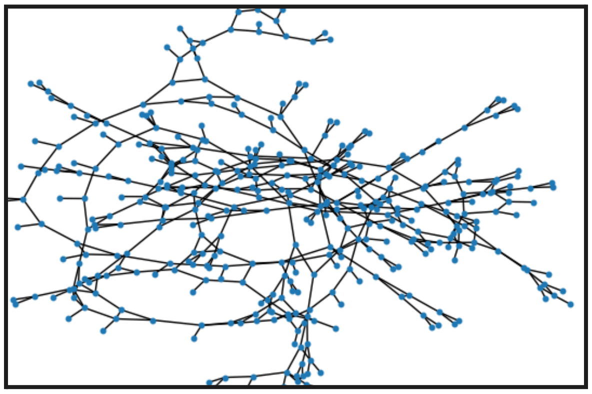
In the spirit of *scientific paper* I am obligated to include at least one rediculogram. I will show a rediculogram from each dataset. Note how the Tissue graph looks different than the others. This will be interesting to note later..



Sample Graph: Metabolic



Sample Graph: Protein Interaction



Sample Graph: Tissue

Before running edge prediction algorithms on the graphs, we must create the set of all potential missing edges. For the largest alpha value of 0.9, this creates about 200 million potential missing edges per graph.

Edge Prediction Algorithms

Now that the missing edges have been created we can run edge prediction algorithms on them. There are five edge prediction algorithms which will be used: jaccard coefficient, adamic adar index, geodesic distance, common neighbors, and degree product.

We will also create an additional data set called 'edge metadata'. Later this dataset will be used to train the models. It contains the raw scores from the predictors along with additional info to assist the models in finding patterns in the data. The additional graph level metadata is: node count, edge count, density, average degree, node count, clustering coefficient.

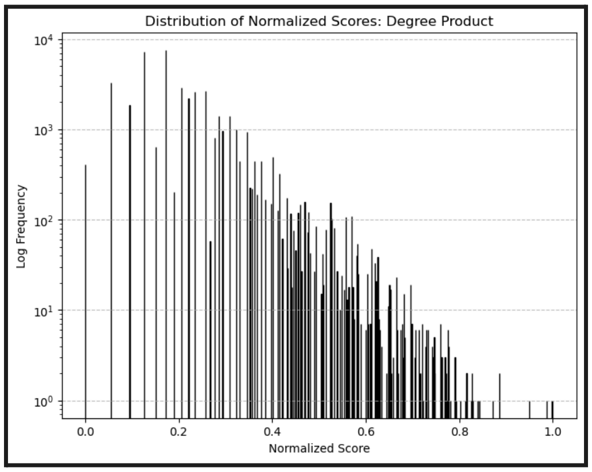
Data Normalization

<u>Apply Noise</u>
Before normalizing the data we apply a small amount of noise, 1e-10, to break ties when we later measure model and predictor performance with ROC curves in the Visualizing Model Performance section.
<u>Normalization</u>

We normalize the data to be between 0 and 1. The specific normalization techniques applied to each predictor are as follows:

- Jaccard coefficient - min-max scaling
- Adamic adar index - log scaling normalization (log(1 + x) / log(1 + max))

- Geodesic distance - inverse scaling (1 / 1+x)
- Common neighbors - log scaling (same as above)
- Degree product - min-max scaling



Sample Data Distribution After Normalization: Degree Product, Y-Axix Log Scaled

Model Training

<u>Creating Train and Test Sets</u>

Before training the model we need to create features (X) and labels (y) for the model to be trained and tested on.

The training set is created by simply using 'edge metadata' which was created in the Edge Prediction Algorithms section.

The test set is created using a 'most correct predictor array, where for each potential edge we mark which predictor was 'most correct'. If the edge in question is a non-edge, store the index of the predictor with the lowest score as the 'most correct'.

Each edge has edge metadata in the train set, and the index of the most correct predictor in the test set associated with it.

Since there are vastly more nonedges than real-edges, it is necessary to reduce the number of non-edges in the training dataset. Using the entire dataset of real and non edges causes the models to perform poorly. After some experimentation, I found including about 1% of the non-edge data yielded the best model performance. This number may appear very small, but remember that the graphs have an average of about 1300 edges, while the number of potential edges (which consist of almost entirely non edges) is closer to 200,000. Reducing this to 2,000 non edges works wonders for model performance.

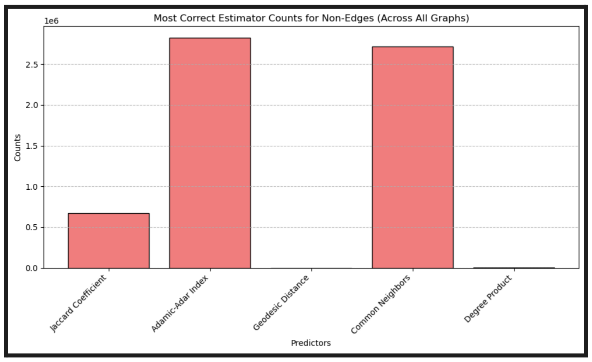
<u>Randomize Data Ordering</u>

The data's order should be randomized. This helps with model training, and later for validating the results are not purely by chance.

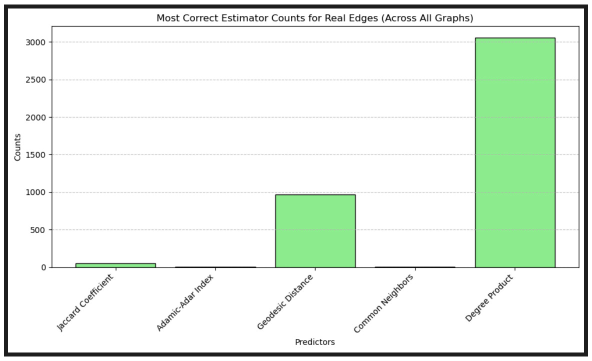
Up until this point the potential edge data has been stored as so: Each graph has its own array of its respective potential edges.

To determine the train and test sets, we randomly select about 80% of the graphs to have their edge sets used in the 'X' set, and then let the remaining 20% of graphs be used for the 'y' set.

Finally, for the 'X' set, it is necessary to get rid of the nested array structure the data currently exists in. We take all the 'edge-most correct predictor' pairs of train/test data from each graph and put them into one large list. Lastly we randomize the order of this list to improve training.



Most Correct Predictor Counts for Non-Edges. Dominant predictors are: jaccard coefficient, adamic adar, and common neighbors



Most Correct Predictor Counts for Non-Edges. Dominant predictors are: Geodesic distance and degree product

<u>Model Training</u>

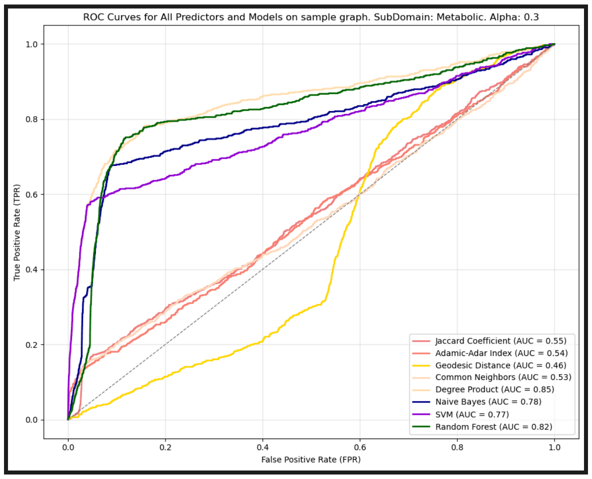
Finally we train several models. Naive bayesian classifier, SVM, and random forest.

Visualizing Model Performance

<u>ROC curves</u>

The ROC curves will be created in the same way they were created in homework 5

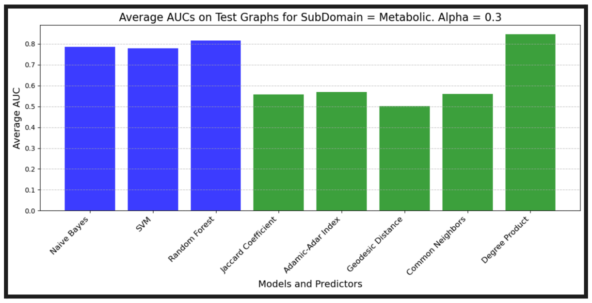
The ROC plots display the performance of each of the predictors and each of the ML models.



Sample ROC plot, ML models in darker colors

Average AUC

To create the average AUC plot we average the AUC after testing the models and predictors on each graph in the 'y'. The averages will then be displayed as a bar plot.

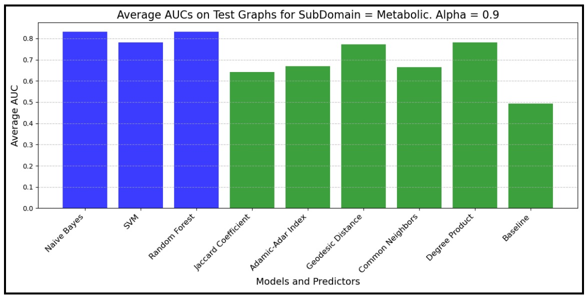


Sample AUC Bar Chart, ML Models in Blue

Discussion

Key Findings

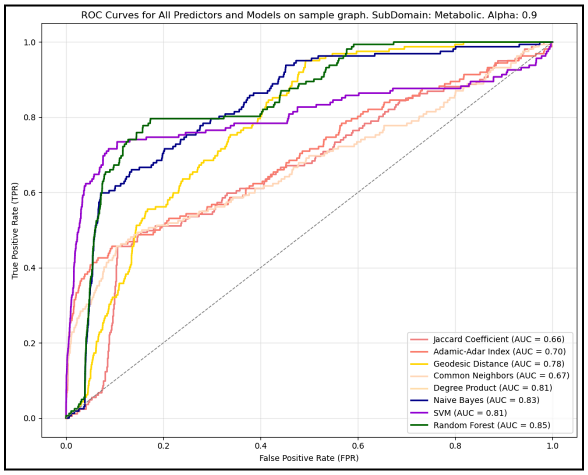
In general the ML models did great! At their best they had better performance than any of the predictors.

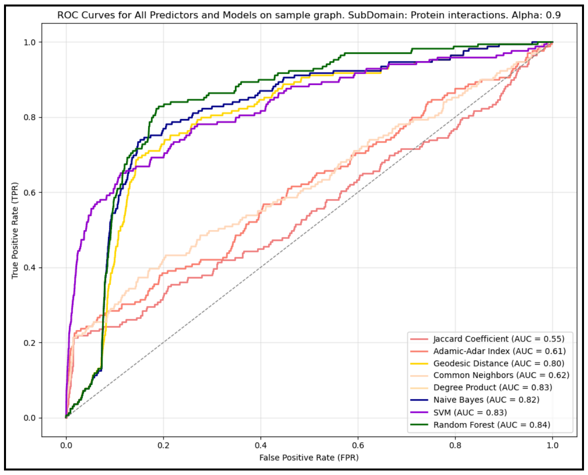


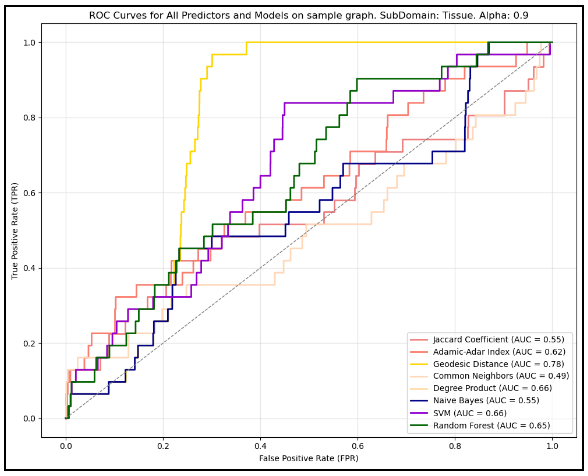
Bar plot of average AUC for models (blue) and predictors (green). Alpha value of 0.9. Subdomain: metabolic

Tissue networks were the most difficult for the models to predict missing edges on. The models even had worse performance than the predictors on tissue networks!

I would guess this has to do with the odd structure of tissue networks (seen in the rediculograms), and them having significantly less edges than the other networks, causing less training data.

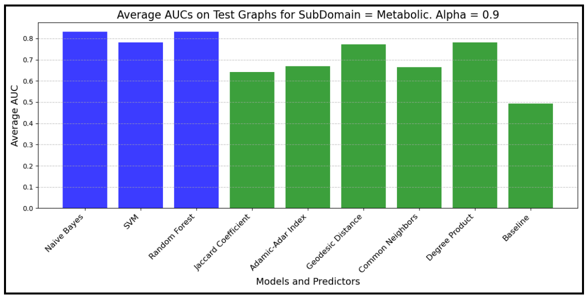


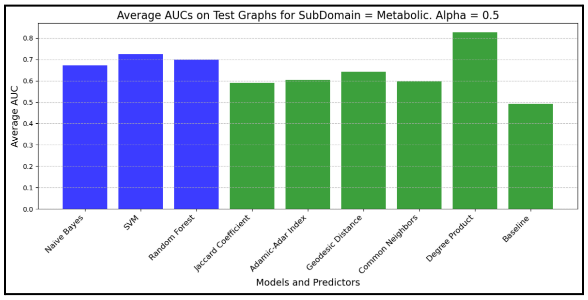


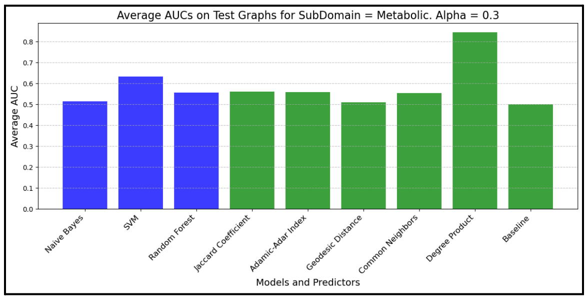


ROC curves for metabolic, protein interaction, tissue networks respectively. Alpha = 0.9. ML models in darker colors: Purple, green, blue

Another key finding was that alpha values have a very significant effect on model performance. Model performance decreases as a direct result of alpha decreasing. This makes sense, a smaller alpha value means less knowledge about the graph's current edges, so the predictors perform worse and as a result the models have a harder time generalizing

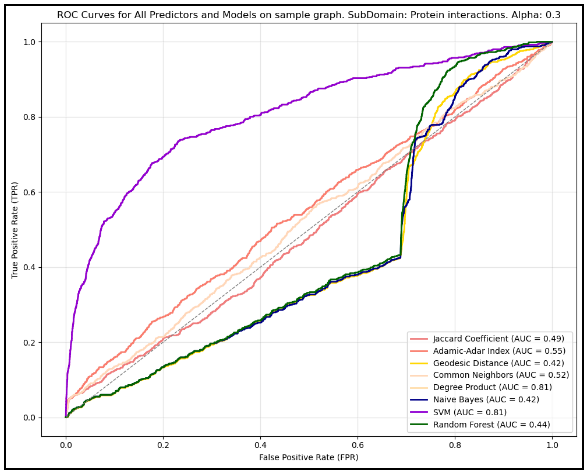






Bar plots of average AUC for models (blue) and predictors (green) for alpha values of 0.9, 0.5, 0.3 respectively.

Another interesting finding was how with smaller alpha values and the protein interaction graphs, SVM is better than all other models. It essentially learned to guess degree product every time. Degree product has the best score so this is not a bad strategy.



ROC curve all predictors, protein interaction graphs, alpha = 0.3

Future Research Directions

In the future I would try out everything with more predictors, and more ML models. More predictors should help the ML models better generalize what the 'best predictor' would be for any given edge.

I would also tighten up code so that the runtime is not so long and more insights can be extracted from the data and models. For example, finding the point where model performance starts to drop off due to a small alpha value.

<u>Bibliography</u>

[1] A. Clauset, "Lecture Notes 4: Predicting Missing Data in Networks," CSCI 3352 Lecture Notes, University of Colorado Boulder, Fall 2024.

[2] A. Ghasemian, "Optimal Link Prediction," GitHub repository. [Online]. Available: https://github.com/Aghasemian/OptimalLinkPrediction. [Accessed: Dec. 6, 2024].

[3] CSCI 3352, "Homework 5," University of Colorado Boulder, Fall 2024.

[4] A. Ghasemian, P. Zhang, A. Clauset, C. Moore, and L. Peel, "Evaluating overfit and underfit in models of network community structure," *Proceedings of the National Academy of Sciences*, vol. 116, no. 12, pp. 5295–5300, Mar. 2019. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.1914950117. [Accessed: Dec. 6, 2024].