CS768 Learning with Graphs **2020-21 Sem I**

Lecture 7: September 13

Scribe: Kaveri Kale *Updated By: Niraj Mahajan* *Lecturer: Abir De*

1 Supervised Methods For Link Prediction

1.1 Outline

In this lecture we will see how to design and train a supervised link predictor. Following are the steps in designing a supervised link predictor:

- Model formulation
- Feature construction
- Loss function selection
- Training

1.2 Train Test Data Split

Consider a social network $G = (V, E)$ in which each edge $e = (u, v) \in E$ represents an interaction between u and v at a particular time t. We split them into training and testing set as follows:

1. Begin with a percentage η for the test-train split.
2. For every node u $\in$ V, assign η% edges connected to u to the test set and the remaining to the train set.
3. Repeat step 2 for all the non edges at u.
4. Note that the final percentage of test to train edges or non edges may not match η. This initial η will have to be tuned to finally obtain a certain η'

1.3 Supervised Predictor using Heuristic based scores

In the previous lecture we saw link prediction heuristics like Adamic Adar, Preferential Attachment, Common Neighbor which produce the score for an edge between nodes (u, v),

$$\begin{aligned} s_{AA}(u, v) &= AA(u, v) \\ s_{PA}(u, v) &= PA(u, v) \\ s_{CN}(u, v) &= CN(u, v) \end{aligned}$$

In unsupervised methods, this score is used to rank node pairs in the network. The top ranked pairs are predicted to be connected.

We first look at supervised methods, which take the scores from the above mentioned heuristics as features vectors, and perform a classification task.

$$\begin{aligned} s_W &= b + W_{AA}AA(u, v) + W_{CN}CN(u, v) + W_{PA}PA(u, v) \\ &= \mathbf{W}^T \begin{bmatrix} 1 \\ AA(u, v) \\ CN(u, v) \\ PA(u, v) \end{bmatrix} \end{aligned} \tag{1}$$

Henceforth, we refer to AA(u,v) as AA for simplicity (similarly PA, CN)
For each pair (u, v) we can define model,

$$s_w(u,v) = \mathbf{W}^T \begin{bmatrix} 1 \\ AA \\ CN \\ PA \end{bmatrix}_{(u,v)} \tag{2}$$

Here $s_W(u,v)$ is parameterized by W and it is stack value of,

$$\mathbf{W}^T = \begin{bmatrix} b \\ W_{AA} \\ W_{CN} \\ W_{PA} \end{bmatrix} \tag{3}$$

here b is the bias

1.4 Feature vectors from nodes

We can also define feature vectors of different nodes which can be the inherent properties of each node. Consider a graph which has two nodes u, v We define features for each node.

$$\mathbf{f_u} :- \begin{bmatrix} f_{u1} \\ f_{u2} \\ . \\ . \\ . \\ f_{un} \end{bmatrix} \ \& \ \mathbf{f_v} :- \begin{bmatrix} f_{v1} \\ f_{v2} \\ . \\ . \\ . \\ f_{vn} \end{bmatrix} \tag{4}$$

Here we can formulate the model,

$$similarity(u,v) = \Omega(\mathbf{f_u}, \mathbf{f_v}) \tag{5}$$

$$\tag{6}$$

where Ω is a similarity metric, like cosine similarity or rbf similarity

$$\Omega_{cosine}(X,Y) = \frac{X.Y}{||X||_2||Y||_2} \tag{7}$$

$$\Omega_{rbf}(X,Y) = e^{-\alpha(||X-Y||^2)} \tag{8}$$

Given a training graph $G_{Training} = (V, E_{Training})$ we can model $s_W(u,v)$ using both the feature similarities and the LP heuristic scores. Let the new feature vector for nodes (u,v) be f_{uv}

$$s_w(u,v) = \mathbf{W}^T \mathbf{f_{uv}} \tag{9}$$

$$s_w(u,v) = \mathbf{W}^T \begin{bmatrix} 1 \\ \Omega(f_u, f_v) \\ AA(u,v) \\ CN(u,v) \\ PA(u,v) \end{bmatrix} \tag{10}$$

For more complicated model simply use complex structure or dependency between two nodes using more "expressive" predictors.

$$s_\theta(u,v) = NN_\theta(\mathbf{f_{uv}}) \tag{11}$$

$$\tag{12}$$

Where NN is a neural network

1.5 Training the Network

We can implement any supervised approach like Linear Regression, Logistic Regression, etc to train our model. Following is an example of training on an SVM.

$$(u, v) \in E \; s_W(u, v) \geq 1 - \zeta_{u,v} \tag{13}$$
$$(u, v) \notin E \; s_W(u, v) \leq -1 + \zeta_{u,v} \tag{14}$$

where $\zeta_{u,v}$ is the slack parameter for the node pair (u,v)
The optimisation problem for an SVM is given by,

$$W_{opt} = \underset{W}{\operatorname{argmin}} \left\{ \frac{1}{2}\lambda ||W||^2 + \sum_{u,v} |\zeta_{u,v}| \right\} \tag{15}$$

subject to $Y_{(u,v)} s_W(u, v) \geq 1 - \zeta_{u,v}$

where Y is indicator of edge(+1) or non-edge(-1).

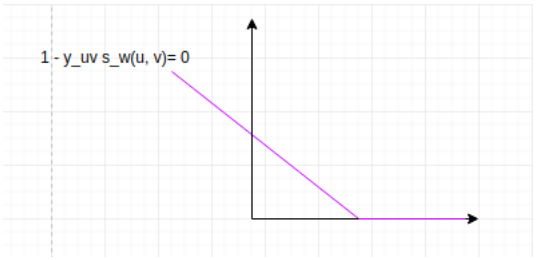

Figure 1: Hinge loss

Here we are trying to construct classifier which will classify edges and non edges.
Minimize the loss

$$\sum_{(u,v) \in Training} (1 - Y_{(u,v)} s_W(u, v))_+ = \sum_{(u,v) \in E_{Training}} (1 - s_W(u, v))_+ + \sum_{(u,v) \in NE_G / NE_{Test-set}} (1 + s_W(u, v))_+ \tag{16}$$

We can test our trained model on the test set using evaluation metrics like MAP and MRR.

Caveat : Most graphs are sparse, that is,

$$|E| << |V| \times |V| \tag{17}$$
$$|E| << |NE| \tag{18}$$

Since data is highly imbalanced, learning becomes biased towards the class with the higher cardinality. In our case, the predictor is bound to predict every test set edge/non-edge to be a non-edge
In such cases, we add an additional Δ to make up for this imbalance.
Consider $(u, v) \in E_{Training}$ and $(u^1, v^1) \notin E_{Training}$,
Loss is greater than 0 iff $s_W(u, v) < s_W(u^1, v^1) + \Delta$

Alternately, we can also scale (divide) the losses due to the edges and the non edges, by the number of edges and non edges in the graph to tackle such imbalanced datasets.