**DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Last time we introduced the node classification problem, which is a ‘toy’ problem in large scale network analytics with a variety of connections to clustering, link prediction, and recommendation systems.

In its simplest terms, we are given a graph, and only a random fraction of the labels of its nodes (which we assume are 0 and 1), and we want to label the rest of the vertices to be as accurate as possible. Here accuracy is measured in terms of $F_1$-score.

Today we will discuss recent ways of fitting this problem into more well understood machine learning pipelines. Specifically, we will discuss ways of converting the nodes into short vectors so that logistical regression can be applied.

### 1 Logistical Regression

One of the more mysterious aspects of the system that we discussed is the ‘self learning’ stage of the algorithms: after we have a label propagation / diffusion type algorithm, we go back using the training data to determine how we should initialize the ‘true’ labels in order to minimize error on the training data.

The hope is that because the training data was picked randomly, this choice of initialization also does well on the overall testing data.

We develop a model with this as the starting point instead: we assume instead that the $y$s are simply generated from a linear model on the vertices, $\bar{\theta}$:

$$y \approx A\theta$$

where $A$ is the adjacency matrix of the graph.

This can be solved using logistical regression, which takes the predictions through a logistical function. Say we have a value $z$, if we want it to be 1, the penalty we incur can be

$$\frac{1}{1 + e^z}$$

which tends to 0 as $z$ becomes negative. Similarly, if we want $z$ to be 1, the penalty we can incur is

$$\frac{e^z}{1 + e^z}.$$
The good thing about this function is that its value is always between 0 and 1: it’s a ‘smooth’ approximator to how many things we get wrong.

This gives an objective of the form:

\[
\sum_{u:y_u=1} \frac{1}{1 + \exp ((A\theta)_u^i)} + \sum_{u:y_u=0} \frac{\exp ((A\theta)_u^i)}{1 + \exp ((A\theta)_u^i)}
\]

Here the \( \theta \) is still on the vertices, but we can propagate gradients to and from it by multiplying by the matrix \( A \): this takes linear time.

With the right regulariation (by plugging in tools like LibSVM/LibLinear) training on such a model gives F1 scores close to 30% as well: about as good as the diffusion / node labeling approaches we discussed last time.

2 Smaller Models: Embeddings

This method also works well if one trains a model on a rank \( k \) approximation of \( A \) instead. This leads to the notion of graph embedding. Here the natural first step is to try matrix factorizations, or singular value decompositions. We will use SVD\((A,k)\) to denote the rank-\( k \) approximation of the matrix \( A \) (in Frobenius/spectral norm). Using this terminology, the embedding that we can start by trying is:

\[
M \leftarrow \text{SVD}(A,d),
\]

where \( d \) is typically set to either 64 or 128. Given this embedding, we can now learn a model on the smaller dimension, aka. \( \theta \in \mathbb{R}^d \). That is, we once again use logistical regression to fit

\[
y \approx M\theta.
\]

However, this only gives about 10% F1 score. As it turns out, it’s much better to take the truncated log of this matrix before taking the SVD. That is, taking

\[
M \leftarrow \text{trunc\_log}(m \cdot D^{-1}AD^{-1}).
\]

This is a way to discount the effects of the large entries, and gives a very significant increase in classification accuracy of this approach (from about 10% to upper 20%).

This in effect gives the Node2Vec embedding [GL16]. The reasons for taking logs can be interpreted in two ways, neither of which I find satisfying:

1. Geometrically: the \( M \) matrix puts each vertex into \( d \)-dimensional space, once we get such points, the \( \theta \) vector is a direction: the value \( \langle M_u, \theta \rangle \) tells one how far along the direction of \( \theta \) does the embedded point of \( u \) lie.

2. Logistical regression penalizes dot products exponentially, aka. \( \langle \theta, M_u \rangle \) gets taken to \( \exp \langle \theta, M_u \rangle \) in the objective function. So we want the underlying embedding to directly preserve the objective value.
Neither of these interpretations fully explain why the accuracy jumps significantly when taking logs. However, they do suggest that this model can also be directly used to measure similarity of two vertices. The dot product

\[ \langle M_{u,:}, M_{v,:} \rangle \]
serves as a measure of how similar \( u \) and \( v \) are. This can be used directly for the task of link prediction: predict whether two nodes are likely to become friends.

3 Twiddling Entries and Adding Edges

If we’re really sure that \( A \) is the right matrix, we can use it to simplify itself. To start with, we can add in all two-step walks to get:

\[ M_2 \leftarrow \text{TRUNC}\_\text{LOG} \left( m \cdot \left( D^{-1} A D^{-1} \right) + \left( D^{-1} A D^{-1} \right)^2 \right) \]

increases the classification accuracy by about 5%. It corresponds to add in all (weighted) triangles in the graph, and is appealing because for graphs with millions of edges, adding in triangles still keeps the total number of edges at a manageable level (usually around a hundred million to a billion). It has served as the starting point of distributed graph embedding packages [ZXTQ19].

Naturally, one can add even longer walks to form \( M_t \):

\[ M_t \leftarrow \text{TRUNC}\_\text{LOG} \left( \alpha \cdot \left( D^{-1} A D^{-1} \right) + \left( D^{-1} A D^{-1} \right)^2 + ... D^{-1} \left( A D^{-1} \right)^T \right) \]

where \( \alpha \) is a hyper-parameter. This method was motivated by the inclusion of \( k \)-grams, a common technique used in analyzing text. It was proposed initially as a random walk based strategy [PARSL4], and referred to as DeepWalk. The intuition is:

1. When processing text, it’s very useful to consider sequences of multiple words, aka. tri-grams or higher.
2. Vertices on short walks are akin to \( k \)-grams, sampling a number of them from a graph turns it into a ‘bag of words’.
3. The occurrences of pairs of words in such \( k \)-grams then gives a matrix, which we can then factorize.

Such connections are formalized in [QDM+18], which shows that most graph embedding methods can be written in terms of matrix operators involving \( D \) and \( A \). It observed that in general, as the value \( t \) increases from 2 to 10, one gets about an additional 1% F1-score per increase in \( t \), topping out at about 35%.

After just triangles though, this matrix gets very large. So a variety of ideas from randomized numerical linear algebra are then used to reduce the number of entries in it. Two very useful ones are:
1. Use random projections to speed up the matrix factorization stage \[W^{+14}\].

2. Sample a smaller number of random walks, instead of putting down all the walks. This was analyzed in \[QDM^{+19}\].

4 Additional Comments

The most mysterious aspect of all graph embedding methods are the use of the truncated logarithm: I have yet to see a good theoretical explanation of it.

**Open Problem.** *Theoretically justify the use of the truncated logarithm in graph embeddings*

Aside from that, there is also the question of whether directly doing label propagation, such as with semi-supervised learning, or diffusion, can obtain the same type of accuracy as graph embedding methods.

Here the issue is that the regularizer used in LibLinear has a very large effect on the resulting classification accuracy: it’s often significantly better to train a sparse \(\theta\) vector. This can be considered from the perspective of graphs having a few authority vertices. Understanding the role of this regularizer is necessary before making any tweaks to the embedding framework for node classification.

References


