Autoencoder-Based Community Detection

Alex Rambasek

July 30, 2021

1 Static Networks

1.1 CiteSeer

The CiteSeer [1] dataset is a citation network of 3,312 scientific publications divided into 6 classes. Also, each node is associated with a 3,703-dimensional word vector, indicating the absence/presence of the respective words from the dictionary. For this dataset, I didn't have much success using anything more than a single-layer autoencoder (I'd love to be proven wrong however) with dimensionality reduction 3312-512.

1.2 Cora

Cora [2] is a citation network similar to CiteSeer. The network has 2,708 publications with 7 classes and 1433-dimensional word vectors. Unlike CiteSeer, I did use a stacked autoencoder architecture with dimensions 2708-512-256-128. The associated Jupyter notebook also has some code for ensembling autoencoders trained with different graph representations, following an approach similar to that described in [3].

1.3 Football/Karate

Both the American College Football [4] and Zachary's Karate Club [5] networks are toy networks, useful as a sanity check for different model architectures and techniques. The Football network has 35 teams grouped into different conferences, where the edges represent games played between teams. Karate Club is a 34-node social network with 2 communities, representing a schism that occurred in a karate club. Anything more than a single-layer autoencoder for these networks is overkill.

1.4 LFR

The Lancichinetti–Fortunato–Radicchi (LFR) [6] network is a synthetic benchmark network for static community detection, where the node degrees and community sizes are distributed according to a power law. The `networkx` Python module has methods for generating LFR networks.

2 Dynamic Networks

2.1 DANCer

The Dynamic Attributed Network with Community Structure Generator (DANCer) [7] is a useful tool for generating synthetic networks with community structure that mantain important properties of real-world networks, such as homophily and small world. The software can be downloaded here, where you can also read about the user-defined parameters. The program saves the created snapshots as `.graph` files, along with a PDF of a visualization of the evolving community structure. There is code in `graph_creation.ipynb` to convert to `networkx`.

One of the most important parameters for graph creation is "Nb. Rep.," or the number of "representative" nodes for community. Basically, a representative node is the author's way of saying that the node is a particularly good exemplar of core community characteristics. The higher this parameters is, the more uniform communities will be, and detection will be easier. A low value more often than not results in poor model performance.

A critique I have of this generator is that new "communities" of a very few number of vertices will arise often. I use quotations here because the community structure of, say, 5 nodes out of a graph of 1,000 is likely ill-defined. I experimented with RDyn [8] as an alternative, but a major issue with this generator is that you can only specify the expected degree and not the minimum degree of vertices, so snapshots are often not connected.

2.2 RealityMining

The MIT RealityMining [9] dataset is a dynamic proximity network constructed from 94 students and staff at MIT over a year. Test subjects would carry around a cell phone with Bluetooth at all times, and whenever two phones were in close enough proximity to ping each other, that interaction would be recorded. This data was aggregated into 46 time steps, where each snapshot represents a week [10]. The authors of [10] *a posteriori* determined $k = 2$ to be the most fitting number of communities. Since there is no ground-truth community information associated with this data, it is suggested that NMI is taken between the clusterings of different time steps to measure community stability. The preprocessed graph snapshots can be found at `/graphs/realitymining.p`. Due to time constraints, I never got around to training deep autoencoders on the individual time steps, but from my experience this is very promising.

3 Autoencoders

3.1 Deep autoencoders

I have found that the benefit of deepening the model is highly variant across different graphs, so as of yet I have no substitute for experimenting both ways and following up with whatever is most promising. However, I have gotten very poor results across the board by trying to train the entire deep autoencoder with one backwards pass. Training with Hinton's greedy unsupervised layer-wise training method [11] is the only method I recommend. Most of my

experimentation involved doing this manually; training the second autoencoder on the latent space embedding produced by the first autoencoder, and so on. I've recently added code to `Autoencoder.py` to do this automatically, but this has not been tested rigorously and I would advise caution while using.

3.2 Sparsity

The vast majority of autoencoders I train are sparse; i.e., the activity of the hidden layer is forced to be mostly near zero. The `SparseRegularizer` class in `Autoencoder.py` uses Kullback-Leibler (KL) divergence for this purpose. I usually choose ρ in the range $(0.01, 0.05)$. I tend to keep the scalar multiplier β at unity, but the optimal choice warrants more experimentation.

3.3 Graph Representation

Below are several graph representations that I use most frequently. Let $\theta(i)$ denotes the set of vertices adjacent to vertex v_i, and l_{ij} denotes the shortest path between vertices v_i and v_j.

- *Modularity* $\mathbf{B} = [b_{ij}]^{n \times n}$, $b_{ij} = a_{ij} - \frac{d_{ii} d_{jj}}{2m}$
- *Similarity* $\mathbf{S} = [s_{ij}]^{n \times n}$, $s_{ij} = \frac{\theta(i) \cap \theta(j)}{\theta(i) + \theta(j)}$
- *Probability Transition* $\mathbf{T}_k = [t_{ij}]^{n \times n}$, $\mathbf{T}_k = (\prod_{i=1}^{k} \mathbf{D^{-1}}) \cdot \mathbf{A}$
- *Proximity* $\mathbf{P} = [p_{ij}]^{n \times n}$, $p_{ij} = exp(\frac{2 - l_{ij}}{2})$

Modularity is the most popular of these and works well overall, but I have occasionally noticed an uptick in performance when substituting similarity. The Gaussian kernel-based proximity matrix shows promise as well. The probability transition matrix is very peculiar in the sense that the results are very promising, but the NMI difference between clustering with the raw input matrix and the learned latent space embedding is generally not significant. My suspicion is that the construction of the matrix in and of itself resembles a random walk-based community detection method like WalkTrap [12]. I generally set $k = 4$ per the recommendation of the authors.

3.4 Temporal Smoothness

As a refresher, temporal smoothness constraints force the autoencoder at time t to obtain a latent space representation close to that of time $t - 1$. In `Autoencoder.py`, there are two primary forms of temporal smoothness for dynamic community detection. The first involves directly penalizing latent space distance, which I find works the best. Essentially, a subspace distance is taken between the orthonormal projections of the latent space representation. I primarily use the chordal distance, but other measures are defined on the associated Grassmanian. The second method indirectly constrains the latent spaces by penalizing the resulting K-means clusterings from being divergent. The success of this method is sporadic and generally worse than the former, but more experimentation is in order before any definite conclusions can be made.

References

[1] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[2] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, p. 93, Sep. 2008.

[3] R. Xu, Y. Che, X. Wang, J. Hu, and Y. Xie, "Stacked autoencoder-based community detection method via an ensemble clustering framework," *Information Sciences*, vol. 526, pp. 151–165, 2020.

[4] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[5] W. W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of Anthropological Research*, vol. 33, no. 4, pp. 452–473, 1977.

[6] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, Oct 2008.

[7] C. Largeron, P. N. Mougel, O. Benyahia, and O. R. Zaïane, "Dancer: dynamic attributed networks with community structure generation," *Knowledge and Information Systems*, vol. 53, pp. 109–151, Oct 2017.

[8] G. Rossetti, "RDyn: graph benchmark handling community dynamics," *Journal of Complex Networks*, vol. 5, pp. 893–912, 07 2017.

[9] N. Eagle and A. (Sandy) Pentland, "Reality mining: sensing complex social systems," *Personal and Ubiquitous Computing*, vol. 10, pp. 255–268, May 2006.

[10] A. Karaaslanlı and S. Aviyente, "Community detection in dynamic networks: Equivalence between stochastic blockmodels and evolutionary spectral clustering," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 7, pp. 130–143, 2021.

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, pp. 1527–1554, 07 2006.

[12] P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Computer and Information Sciences - ISCIS 2005* (p. Yolum, T. Güngör, F. Gürgen, and C. Özturan, eds.), (Berlin, Heidelberg), pp. 284–293, Springer Berlin Heidelberg, 2005.