参赛队员姓名：白行健

中学：北京师范大学附属实验中学国际部

省份：北京市

国家/地区：中华人民共和国

指导教师姓名：王鸿伟

论文题目：基于自适应性图卷积神经网络的暴力用户检测

基于自适应性图卷积神经网络的暴力用户检测

白行健

北京师范大学附属实验中学

摘要

近年来，互联网和在线社交网络已经成为每个人生活中至关重要的一部分。然而，随着社交网络的发展和扩大，我们面临一个严重的问题：网络暴力用户。这些网络暴力用户在互联网上传播负面言论，宣扬暴力，攻击或威胁他人。由于他们对其它用户带来的严重危害，我们有必要寻找一种有效的方法来检测暴力用户。

考虑到检测网络暴力用户的任务对方法的可扩展性和准确性的要求较高，我们将机器学习作为该任务的解决方案。在机器学习的框架中，暴力用户检测属于二元分类问题，且每个用户的类别（即是否属于暴力用户）依赖于两个重要的因素：用户的个人特征（包括其人口统计学特征及其以前发帖的内容）和该用户周围的局部社交网络结构（即该用户的邻居）。

最近，图卷积神经网络（Graph Convolutional Neural Network，简称GCN）引起了研究人员的关注。GCN是一种神经网络模型，可以解决图中半监督式的节点分类问题。由于GCN在模型中同时考虑了节点的属性和图的结构信息作为输入，GCN可以自然地适用于社交网络中的暴力用户检测问题。但是，传统的GCN未能考虑社交网络中的一个重要因素：边的权重（edge weights）。具体来说，在社交网络中，一个用户更有可能被其熟悉的用户或与其相似的用户的影响，因此他们之间的关系应该在社交网络中具有更大的权重。传统的GCN没有考虑这一点，因此无法准确地刻画社交网络的结构信息，从而影响了预测结果的准确性。

本文提出了一种新的自适应图卷积神经网络模型（Adaptive Graph Convolutional Neural Networks，简称AdaGCN），在传统的GCN模型的基础上进行了改进和创新。在新模型中，边权被设置为可训练的变量，这允许模型自适应性地学习用户之间关系的权重。一个值得关注的问题是虽然可训练的边权提升了模型的能力，但是参数的增加会导致模型更难训练并可能发生过拟合。为了解决这个问题，本文引入标签平滑假设（Label Smoothness Assumption），即在社交网络上相邻的两个用户更有可能会有相同的标签（即他们更有可能同时为暴力用户或同时为正常用户）。本文使用标签平滑假设对边权的训练施加了额外的监督。具体来说，本文引入标签传播算法（Label Propagation Algorithm，简称LPA），并设计了丢一损失（the leave-one-out loss）作为标签平滑假设的具体实现，从而实现了和GCN模型的自然结合。

本文在Manoel Horta Ribeiro等人收集的数据集上应用了AdaGCN模型。该数据集包含了10万余名Twitter用户和200余万条社交关系，其中大约5千名用户被标注了是否为暴力用户。实验结果表明，AdaGCN的AUC得分为0.80，F1得分为0.47，得分高于所有对比方法，包括传统的GCN模型，图注意力网络 (GAT)，标签传播算法（LPA），支持向量机（SVM）等等。此外，AdaGCN模型的结果具有最低的标准差，这表明AdaGCN模型具有很强的稳定性。

在线社交平台可以利用本文提出的方法来更好地评估、检测暴力用户，防止暴力用户伤害他人并传播仇恨言论。同时，自适应图卷积神经网络模型也可以用来评估不同类型的暴力言论造成的社会影响。技术旨在为人类带来便利和幸福，但没有任何东西是尽善尽美的。暴力言论和暴力用户正是互联网技术无意中带来的问题。我们希望用技术的方法缓解这个问题，给所有用户提供一个干净、友好的互联网世界。

关键字：网络暴力用户检测，图卷积神经网络，标签传播算法。

Hateful User Detection with Adaptive Graph Convolutional Neural Networks

Xingjian Bai
The Experimental High School Attached to BNU

Abstract

The Internet and online social networks have become a vital part of people's lives in the recent decade. However, as online social networks expand, a serious problem is brought in front of us: *hateful users*. These people spread negative speech, promote violence, and attack or threaten others on the Internet. With their hazardous interference in the judgement of the online public, it becomes increasingly necessary to find an effective way to detect online hateful users.

The requirements of scalability and accuracy in hateful user detection prompt us to seek solutions from the perspective of machine learning, by which the problem can be formulated as a binary classification task. In general, whether a user is hateful or not largely depends on two factors: *his/her profile* (including his/her demographic features and content from previous posts) and *his/her neighbors in the social network*.

Recently, *Graph Convolutional Neural Networks* (GCNs) have drawn a lot of attention of researchers. GCNs are a type of neural network models that address the problem of semi-supervised node classification in graphs. Typical GCN models take both node features and graph structure into consideration for node classification, and thus are naturally applicable to the task of hateful user detection. However, existing GCN models fail to consider one essential factor: *the weights of edges* in the social network. Specifically, a user is more likely to be affected by neighbors that he/she is familiar with or shares high similarity with, so the edges between them should have larger weights. But existing GCN models treat edges in the network unweighted, which cannot fully capture the structural information and leads to sub-optimal performance.

In this paper, we propose a new GCN model, *Adaptive Graph Convolutional Neural Networks* (AdaGCN), to address this problem. Distinct from existing GCNs, in our model, edge weights are set as trainable variables, which allows the model to adaptively learn the weights between connected users. Note that though trainable edge weights increase the capacity of our model, the increasing number of parameters may make the model harder to train and prone to overfitting. To solve this issue, we design an additional regularization on edge weights based on *the label smoothness*, which assumes that adjacent users are more likely to have the same label (i.e., they tend to be both hateful or both normal). We further employ the *Label Propagation Algorithm* (LPA) and a leave-one-out loss function as the implementation of label smoothness regularization, which can be naturally combined with the loss function of GCN part of our model.

Empirically, we apply AdaGCN on a dataset of Twitter social networks that contains $100k+$ Twitter users and $2m+$ social edges, of which approximately $5k$ users were annotated as hateful or not. The experimental results show that our model achieves AUC score of 0.80 and F1 score of 0.47, outperforming all the baselines including the original GCN (Kipf and Welling, 2016), Graph Attention Networks (Veličković et al., 2017), LPA (Zhu et al., 2003), SVM, etc. Besides, results also show that the performance of AdaGCN has the lowest standard error, indicating the strong stability of our model.

Online social platforms can utilize our proposed method to evaluate and detect hateful users. After detection, they can take necessary measures to warn or punish those users, preventing them from hurting others and spreading hate speech. Technology is meant to bring convenience and happiness to humanity, but nothing comes purely beneficial. Hate speech and hateful users are exactly an issue unintentionally brought along by Informatics technology. We hope that our proposed method will alleviate this problem and provide a clean and friendly Internet world to all users.

Keywords: hateful user detection; graph convolutional neural networks; label propagation.

Contents

1 Introduction **3**

2 Related Work **5**

2.1 Hateful User Detection 5

2.2 Graph Convolutional Neural Networks 5

3 Problem Formulation **6**

4 Our Approach **6**

4.1 Label Propagation Algorithm 7

4.2 Graph Convolutional Neural Networks 8

4.3 Adaptive GCN 8

5 Experiments **9**

5.1 Dataset 10

5.2 Baselines 10

5.3 Experiment Setup 10

5.4 Empirical Study 11

5.5 Results 11

5.5.1 Learning Curves 11

5.5.2 Comparison with Baselines 12

5.5.3 Impact of Unlabeled Users 12

5.5.4 Hyperparameters Sensitivity 13

6 Conclusion **13**

Reference **14**

Appendix **15**

layers.py 15

models.py 17

train.py 19

Acknowledgements **22**

1 Introduction

The world has seen a proliferation of online social networks in the recent decade. As we step into the Internet Era, the number of Internet users has grown exponentially and cyberspace has become increasingly important in our lives. In 2018, the number of active Internet users reached 3.8 billion (51% of the world's population, from statista.com). The idea of the instant messenger was first brought to the spotlight in 1996 when Mirabilis Company released software called ICQ. This technology allows people to chat with others and build circles on the Internet regardless of time, location, and their real identity. After twenty years of development, online social networks have become a vital part of our lives. Well-known platforms such as Twitter[1] and Facebook[2] have more than 500 million active users[3], taking up 10% of total Internet users. In China, similar platforms like Weibo[4] and Wechat[5] dominate, accounting for 80.9% (from sohu.com) of total Chinese Internet users.

Hateful Users

Online social networks bring us friends, ideas, and even a chance to build up another personality. However, along with these benefits, it also causes a serious problem: *hateful users*. Hateful users are defined as people who spread negative speech, promote violence, and attack or threaten other users on the Internet based on race, ethnicity, national origin, sexual orientation, gender, religious affiliation, age, disability, or disease[6]. The damage of hateful users and their hate speech is huge. Since online social networks play such an important role, victims are badly hurt by abusive words or even becomes hateful users themselves. Moreover, more than 30% of the online social platform users are teenagers (under 24 years old) [7], who are more vulnerable to aggressive words. For example, schools often have forums where students can express their opinions. When a student poses some hateful content on the forum, other students, even including the ones who don't stir up troubles, would fight back and cast abusive words on each other. In this regard, hate speech can be seen as an infectious plague, starting with one and spreading exponentially. What's even worse is that hate speech and violence don't purely stay online but could also cause fighting and bullying in real life. In spite of the serious damage brought along by hateful users, however, the anonymity and mobility afforded by the Internet have made hate speech and poisonous opinions effortless in a landscape beyond the realms of traditional law enforcement (Wulczyn et al., 2017).

Hateful User Detection

Since hateful users and hate speech are seriously hazardous to the online public, detecting hateful users becomes increasingly urgent. However, manually labeling every piece of online speech is impractical due to the large amount of content and the expensive cost of human labor. A simple idea is building lexicons of hate speech as filters to perform content-based search and matching user-generated content to abusive words, but this method fails to find hate speech that doesn't contain commonly used abusive words.

The requirements of scalability and accuracy on hateful user detection models lead us to seek solutions from the perspective of machine learning, which can automatically obtain statistical patterns from given data and make predictions on new data points. In this paper, we formulate hateful user detection as a binary classification problem. Yet, this classification problem is nontrivial, because whether a user is hateful or not largely depends on the following two complex factors: (1) The attributes of the user himself/herself, including his/her demographic characteristics (e.g., age, gender, location) and content from his/her previous posts; (2) The social circle that the user is in, i.e., how his/her neighbors in social networks behave and how they influence this user. A well-designed hateful user detection model should consider information from both *user profiles* and *social network structure*, then combine them delicately to achieve high performance in real-time detecting scenarios.

[1] `www.twitter.com`

[2] `www.facebook.com`

[3] `www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/`

[4] `www.weibo.com`

[5] `www.wechat.com`

[6] `help.twitter.com/en/rules-and-policies/hateful-conduct-policy`

[7] `www.statista.com/statistics/274829/age-distribution-of-active-social-media-users-worldwide-by-platform/`

A Promising Method: Graph Convolutional Neural Networks (GCNs) Because of the important graphical features carried by online social networks, we consider using Graph Convolutional Neural Networks (GCNs) as the base model for hateful user detection. GCNs (Kipf and Welling, 2016; Defferrard et al., 2016; Hamilton et al., 2017; Veličković et al., 2017) are a type of neural network models that address the problem of semi-supervised node classification in graphs. Generally speaking, for each node in the graph, GCNs aggregate features of its neighbors and itself (e.g., by averaging them together), then use the aggregated feature as the new representation for this node. This procedure can be performed iteratively, for example, for K times, so that the final representation of each node is the mixture of its neighbors up to K steps away. In this way, GCNs are able to capture both raw features of nodes and structural information of the graph when learning node representations.

Though GCNs are naturally applicable to hateful user detection, original GCNs fail to consider an important factor: *the weights of edges* in the social network. Specifically, for a given user, his/her neighbors may have different impacts on his mind and behavior when considering the propagation of public opinions. A user is more likely to be influenced by the neighbors that he/she is familiar or share high similarity with, so the edges between them should have larger weights in general. However, existing models simply treat neighbors of a user equally and do not distinguish their importance to the current user, i.e., treating the social network as an unweighted graph when aggregating neighborhood information. Such simplification fails to characterize real social networks and may lead to sub-optimal performance in hateful user detection.

Our Proposed Method: AdaGCN

To address this problem, we propose a new GCN model, *Adaptive Graph Convolutional Neural Networks* (AdaGCN), for hateful user detection. A significant difference between AdaGCN and existing GCNs is that edge weights in the social network are not fixed but set as trainable variables, which allows our model to adaptively learn the impact factor between each pair of users. However, this added flexibility of edge weights makes the training process prone to overfitting due to the increase of model parameters. Therefore, additional regularization on edge weights is needed to guide the training process and achieve better generalization. We propose taking the *label smoothness* (Zhu et al., 2003) as the additional regularization, which assumes that adjacent users in the social network tend to have similar labels (here the label means whether the user is hateful or not). Under this assumption, an optimal set of edge weights that satisfies label smoothness should minimize the total pairwise label discrepancy between users. It can be shown that label smoothness regularization is equivalent to *Label Propagation Algorithm* (LPA). To this end, we design a leave-one-out loss function for LPA to assist learning the edge weights, provideing an extra supervised signal for the training process.

We apply our model on a real-world dataset containing $100k+$ Twitter users and $2m+$ social edges, of which approximately $5k$ users were annotated as hateful or not. We first conduct an empirical study to examine the relationship between labels and edges and confirms that the label smoothness assumption applies to the real scenario. Then we conduct extensive experiments on the dataset. According to the experimental results, our model achieves AUC score of 0.80 and F1 score of 0.47 in the detection, which surpasses all of the baselines, including original GCN (Kipf and Welling, 2016), GAT (Veličković et al., 2017), LPA (Zhu et al., 2003), SVM, etc.. Moreover, the standard error of Adaptive GCN is the lowest among all methods, indicating the strong stability of our model. The results demonstrate that our proposed model solves the hateful user detection problem effectively and outperforms existing state-of-the-art methods by a large margin.

Novelty

Our proposed model, AdaGCN, outstands among all GCN models in the following aspects: (1) We set edge weights in the social network as trainable to simulate the different impact of neighbors on a given user; (2) We introduce LPA as well as the leave-one-out loss to impose extra supervision on the edge weights, which effectively mitigates the issue of overfitting; (3) Our model combines GCNs and LPA together, and achieves the highest performance among all methods.

Practical Significance

Online social platforms can utilize our proposed method to evaluate and detect hateful users. After detection, they can take necessary measures to warn or punish those users, preventing them from hurting others and spreading hate speech. In the meantime, our model can also be used to assess the social impact of different types of hate speech. Technology is meant to bring convenience and happiness to humanity, but nothing comes purely beneficial. Hate speech and hateful users are exactly an issue unintentionally brought along by Informatics technology. We hope that our proposed method will alleviate this problem and provide a clean and friendly Internet world to all users.

2 Related Work

In this section, we discuss two lines of related work: hateful user detection and graph convolutional neural networks.

2.1 Hateful User Detection

Hateful user detection aims to distinguish hateful users on the Internet. Existing work in this field can be divided into two categories: content-based methods and network-based methods.

Content-based methods aim to train a supervised language model given content that users posted on the Internet as input, then classify unlabeled content as hateful or not. Burnap and Williams (2016) used lexical features to develop a supervised binary classifier for hateful contents on Twitter. Their model tends to have high recall but leads to high rates of false positives, since the presence of offensive words can lead to the misclassification of unhateful tweets. To address this problem, Davidson et al. (2017) built a multi-classifier to distinguish contents between "hate speech," "offensive language only", and those with neither. Syntactic features have also been leveraged to identify hate speech better. For example, research suggests that sentences where a relevant noun and verb occur (e.g., *kill* and *Jews*) have a high possibility to be a hate speech (Gitari et al., 2015).

However, these content-based methods have two drawbacks in general. First, due to the subjectiveness of hate speech, there is hardly a universal feature of hate speech in lexical or syntactic level. A sentence utilizing irony or sarcasm without any offensive word could also be hateful. For example, *"May I introduce you to the toilet and offer you a one-way visit down the drain?"* is an abusive sentence without any words typically considered as hateful. Second, those methods require concrete texts as input, which implies that they can only be used to judge given content rather than predict the behavior of users who have not published any hate speech yet. This feature greatly limits the number of possible hateful users that can be found.

Network-based methods make use of the relationships among users on social media, i.e., social networks. They aim to detect hateful users, which are nodes in the network, and this can be formulated as a semi-supervised node classification task in a graph. An obvious benefit of network-based methods is that they can predict whether a user is hateful or not without requiring any prior speech content from him. For example, Ribeiro et al. (2018) employed a node embedding algorithm that creates a low-dimensional representation for every user. It shows high accuracy and precision on the task but fails to consider the edge weights of the users' network, which play a crucial role in graph-based node classification. Wang et al. (2018) proposed a model called Signed Heterogeneous Information Network Embedding to extract users' latent representations from the network and predict the sign (positive or negative) of unobserved sentiment links between users. But this model mainly focuses on classifying edges in a graph, which is different from our task of finding hateful users (i.e., classifying nodes in a graph).

2.2 Graph Convolutional Neural Networks

Graph Convolutional Neural Networks (GCNs) aim to generalize Convolutional Neural Networks (CNNs) to the field of graphs. In general, GCNs fall into two categories: spectral-based methods and spatial-based methods.

Spectral-based GCNs define the graph convolution operation based on spectral graph theory. The first prominent research on GCNs was presented by Bruna et al. (2013), in which the convolution operator is defined in the Fourier domain by computing the eigendecomposition of the graph Laplacian matrix. Henaff et al. (2015) introduced a parameterization of the spectral filters with smooth coefficient, spatially localized the filters. Defferrard et al. (2016) proposed a new model with fast localized spectral filtering, avoiding an explicit use of the graph Fourier basis and extra computations. Kipf and Welling (2016) restricted the filters within the one-step neighborhood around each node, which further promoted the efficiency and accuracy of GCN models.

In contrast, spatial-based GCNs define graph convolution operation via aggregating information from neighbors and don't need to perform eigendecomposition. Therefore, they are more suitable for large graphs. A significant challenge for spatial-based GCNs is to define a convolution operator that works on nodes with different degrees while maintaining the weight-sharing property of CNNs. To address this problem, Atwood and Towsley (2016) introduced a method to train a specific weight matrix for each node degree, Niepert et al. (2016) extracted and normalized neighborhoods containing a fixed number of nodes, and GraphSAGE (Hamilton et al., 2017) sampled a fixed-size neighborhood of each node and then aggregated the information to the central node. Though the above mentioned spatial-based GCNs outperform spectral-based GCNs in time complexity and are capable of handling large graphs, neither of them consideres an important factor of the graph: the edge weights. Specifically, the strength of impact a node receives from its neighbors largely depends on their similarity, but existing models simply treat neighbors equally and do not distinguish their importance to the central node.

It is worth noticing that Graph Attention Networks (Thekumparampil et al., 2018; Veličković et al., 2017) use an attention mechanism to learn the similarity between nodes. A significant difference between these attention mechanisms and our work is that attention weights are learned based merely on feature similarity, while we propose that edge weights should be consistent with the distribution of labels on the graph, which requires less handcrafting of the attention function and is more task-oriented.

3 Problem Formulation

We formulate the hateful user detection problem in this paper as follows. Let $\mathcal{G} = (\mathcal{V}, A)$ be an online social network, where $\mathcal{V} = \{v_1, \cdots, v_n\}$ is the set of nodes representing users and $A \in \{0,1\}^{n\times n}$ is the adjacency matrix (including self-loops). Meanwhile, we have a raw user feature matrix $X \in \mathbb{R}^{n\times k}$ available (k is the number of features), where the i-th row $X_i \in \mathbb{R}^k$ represents the feature vector of user i. In addition, we are aware of the labels of a subset of nodes $\mathcal{L}$, such that $\mathcal{L} \subset \mathcal{V}$. Label $Y_i \in \{1,0\}$ for node $i \in \mathcal{L}$ indicates that whether user i is hateful or not. Given the input above, our task is to learn a mapping $\mathcal{M} : v \to \hat{y}$, where $\hat{y} \in [0,1]$ represents the predicted probability that user v is hateful.

The key notation in this paper is listed in Table 1.

Notation	Description
$\mathcal{G} = (\mathcal{V}, A)$	The social network
$\mathcal{V}$	The set of all users
$\mathcal{L} \subset \mathcal{V}$	The set of labeled users
v_i	The i-th user in $\mathcal{V}$
$A \in \{0,1\}^{n\times n}$	The adjacency matrix of $\mathcal{G}$
$X \in \mathbb{R}^{n\times k}$	Raw user feature matrix
$Y \in \{0,1\}^n$	Labels of users
$X^{(k)}$	User representation matrix in the k-th layer of GCN
$Y^{(k)}$	Predicted labels in the k-th iteration of LPA
$\mathcal{N}_i$	The set of neighbors of user i
D	Diagonal degree matrix of A
$W^{(k)}$	Transformation matrix in the k-th layer of GCN

Table 1: Key notation in this paper.

4 Our Approach

In this section, we first briefly introduce the mechanism of Label Propagation Algorithm and Graph Convolutional Neural Networks, respectively. Then we propose our new model, Adaptive Graph Convolutional Neural Networks (AdaGCN), which combines GCN and LPA by learning weights of edges adaptively in the graph.

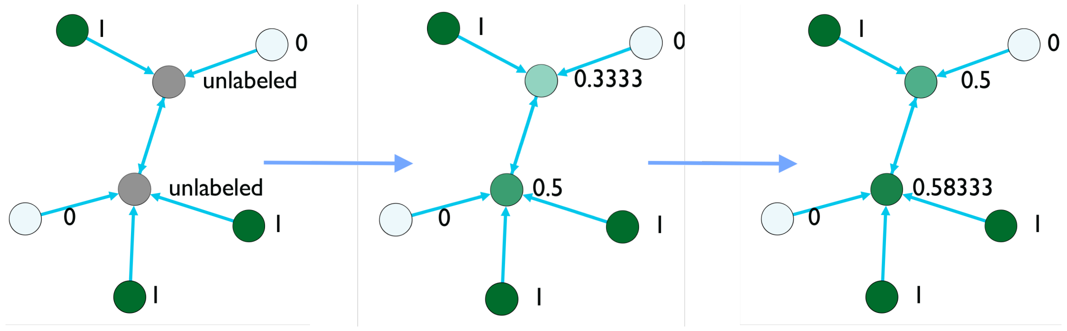

Figure 1: The iteration process of LPA. Labeled nodes are marked as 1 (green) or 0 (white), indicating they are hateful users or not. The initial label of unlabeled users (grey nodes) are set as 0. Then, the label of every node is updated by averaging the current labels of its neighbors, and the labels of labeled nodes are reset to their initial values. This process is repeated until the labels converge.

4.1 Label Propagation Algorithm

Label propagation (Zhu et al., 2003) is an lightweight algorithm for semi-supervised node classification problem. LPA assumes that adjacent nodes in the graph tend to have the same labels, which is known as *label smoothness assumption*. This assumption motivates us to minimize the following energy function:

$$E(Y) = \frac{1}{2} \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} a_{ij} \left(y_i - y_j\right)^2, \tag{1}$$

where a_{ij} is ij-th entry of A, i.e., the weight of edge connecting v_i and v_j. We show by the following theorem that, in the optimal solution of Y, the predicted label of each unlabeled node should equal the average label of its neighbors:

Theorem 1. *The optimal solution of* Y, *i.e.,* $\arg\min_Y E\left(Y\right)$, *satisfies that*

$$y_i = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{V}} a_{ij} y_j, \ \forall\, i \in \mathcal{V} \backslash \mathcal{L}. \tag{2}$$

Proof. The derivative of function $E\left(Y\right)$ with respect to y_i where $i \in \mathcal{V} \backslash \mathcal{L}$ is:

$$\frac{\partial E\left(Y, A\right)}{\partial y_i} = \sum_j a_{ij} \left(y_i - y_j\right).$$

The minimum-energy label function Y^* satisfies that

$$\frac{\partial E\left(Y, A\right)}{\partial y_i} |_{Y=Y^*} = 0.$$

Thus,

$$y_i^* = \frac{1}{\sum_j a_{ij}} \sum_j a_{ij} y_j^* = \frac{1}{A_{ii}} \sum_j a_{ij} y_j^*, \forall i \in \mathcal{V} \backslash \mathcal{L}.$$

□

This theorem indicates that for unlabeled nodes, their labels can be obtained by averaging the labels of their neighbors. This leads us to the following iterative label propagation method:

Theorem 2. *If we set the initial labels of all nodes as*

$$Y^{(0)} = \begin{cases} y_i, & \forall i \in \mathcal{L} \\ 0, & \forall\, i \in \mathcal{V} \backslash \mathcal{L} \end{cases} \tag{3}$$

and then perform the following two steps iteratively:

1. *Do $Y^{(k+1)} = D^{-1}AY^{(k)}$, i.e., propagating the labels of every node to its neighbors. Here D is the diagonal degree matrix of A, where $d_{ii} = \sum_j a_{ij}$ and $d_{ij} = 0$ for $i \neq j$;*

2. *Reset $y_i^{(k+1)} = y_i$ for all $i \in \mathcal{L}$, i.e., recovering the labels of labeled nodes to their initial values;*

until Y converges, we will reach the optimal solution as mentioned in Theorem 1.

The proof of Theorem 2 can be found in Zhu (2002). The procedure stated in Theorem 2 is the label propagation algorithm. We also illustrate LPA in Figure 1.

4.2 Graph Convolutional Neural Networks

GCNs (Kipf and Welling, 2016) are multi-layer feedforward neural networks that propagate node features on graphs. In each GCN layer, nodes aggregate features of their neighbors to obtain their new feature vectors. Specifically, the feature propagation scheme in layer k is:

$$X^{(k)} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}X^{(k-1)}W^{(k)}\right), \tag{4}$$

where $W^{(k)}$ is the trainable weight matrix in the k-th layer, $\sigma(\cdot)$ is an activation function such as ReLU or tanh, $X^{(k)}$ is the nodes representation at layer k, and $X^{(0)} = X$. The term $D^{-\frac{1}{2}}$ is used to normalize the adjacency matrix and keep the node representation matrix $X^{(k)}$ stable. In order to keep models consistent, in this paper, we choose to normalize the adjacency matrix in the way similar to LPA:

$$X^{(k)} = \sigma\left(D^{-1}AX^{(k-1)}W^{(k)}\right). \tag{5}$$

Finally, the supervised signal for training GCN is defined as follows:

$$L_{GCN}(W) = \sum_{i \in \mathcal{L}} J(\hat{y}_i, y_i) + \lambda \|W\|_2^2, \tag{6}$$

where $\hat{y}_i$ is the predicted labels of user i, which is exactly the i-th row of $X^{(K)}$ where K is the number of GCN layers. J is the cross entropy loss between the predicted labels and ground truth, $\|W\|_2^2$ is the L2 regularizer, and λ is the training weight of $\|W\|_2^2$.

4.3 Adaptive GCN

When propagating node features, GCNs treat all the edge weights equally. However, edge weights play an important role in hateful user detection since different neighbors are supposed to have different impacts on a given user. To address this problem, the key is to make the edge weights *trainable* during the optimization process. This means that the loss function in Eq. (6) is revised as follows:

$$L_{GCN}(W, A) = \sum_{i \in \mathcal{L}} J(\hat{y}_i, y_i) + \lambda \|W\|_2^2, \tag{7}$$

Note that the difference between Eq. (6) and (7) is that A is trainable and becomes model parameters. For example, Graph Attention Networks (GAT) (Thekumparampil et al., 2018; Veličković et al., 2017) use an attention mechanism to learn the edge weights:

$$\mathbf{h}_i^k = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha\left(\mathbf{h}_i^{k-1}, \mathbf{h}_j^{k-1}\right) \mathbf{W}^{k-1}\mathbf{h}_j^{k-1}\right), \tag{8}$$

where $\mathbf{h}_i^k$ is the representation vector of user i in the k-th layer of GAT, $\mathcal{N}_i$ is the set of neighbors of user i, and $\alpha(\cdot)$ is an attention function controlling the contribution of neighbor j to user i. However, the weights learned by the attention function α are only based on the similarity of node features, which is not necessarily correlated with the similarity of node labels. Since our goal is to predict node labels, we want to assign weights to edges based on the distribution of hateful users, which is more task-oriented.

Intuitively, the adjacent nodes with the same labels are likely to have larger edge weights. Therefore, a naive way to assign edge weights is to simply strengthen edges that connect two nodes with the

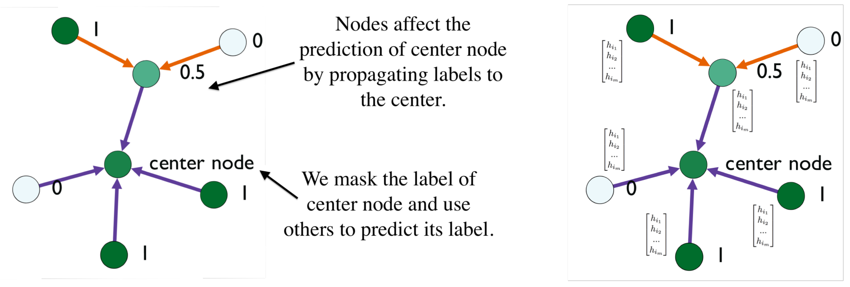
1
0
0.5
Nodes affect the prediction of center node by propagating labels to the center.
center node
We mask the label of center node and use others to predict its label.
0
1
1
center node

Figure 2: Left: LPA combined with leave-one-out loss. Right: The propagation scheme of AdaGCN, in which feature vectors and labels are propagated simultaneously.

same label. However, it is worth noticing that most nodes ($\sim 95\%$ in our dataset) in the graph are unlabeled, which greatly limits the number of edge weights that we can assign.

To solve the issue above, we turn to the label propagation algorithm. Note that the results of label propagation largely depends on the edge weights. Therefpre, LPA may provide us a principled way to guide the training process of edge weights. But in traditional LPA, the final predicted labels $Y^{(K)}$ does not provide any signal to update the edge weights (adjacency) matrix A, since the labeled part of $Y^{(K)}$ equals their ground truth, and we do not know the ground truth of the unlabeled part in $Y^{(K)}$. Therefore, we propose utilizing the *leave-one-out loss*.

We describe how leave-one-out loss works as follows. For each labeled node v, we mask its label out, which means that we treat this labeled node unlabeled. Then we use the labels of remaining nodes to predict v. The predicting process is similar to LPA, while the only difference is that the label of v is hidden. Then, we calculate the difference between the predicted label and the original label of v, which can serve as a supervised signal for regularizing the edge weights:

$$L_{LPA}(A) = \sum_{i \in \mathcal{L}} J\left(y_i, l_i\right), \tag{9}$$

where J is the cross-entropy loss function, and l_i is the predicted label of the i-th user by LPA after masking its label out. Given the above regularization, an ideal edge weight matrix A should reproduce the true label of each masked-out user while satisfying the label smoothness assumption.

We can combine the loss function of GCN and LPA, obtaining the complete loss function for AdaGCN:

$$\begin{aligned} L_{AdaGCN}(W, A) &= L_{GCN}(W, A) + \lambda_0 L_{LPA}(A) + \lambda_1 \|W\|_2^2 \\ &= \sum_{i \in \mathcal{L}} J(\hat{y}_i, y_i) + \lambda_0 \sum_{i \in \mathcal{L}} J\left(y_i, l_i\right) + \lambda_1 \|W\|_2^2, \end{aligned} \tag{10}$$

where λ_0, λ_1 are balancing hyperparameters. In the above loss function, the first term $L_{GCN}(W, A)$ corresponds to the part of GCN that learns the transformation matrix W and edge weights A simultaneously, while the second term $L_{LPA}(A)$ corresponds to the part of LPA and can be seen as an added constraint on edge weights A. Therefore, $L_{LPA}(A)$ serves as regularization on A to assist GCN in training edge weights.

The proposed model AdaGCN is illustrated in Figure 2. The implementation of essential parts of AdaGCN can be found in the Appendix. We divide the main function of our model into three programs that run together: layers.py, models.py, and train.py. The first section defines the structure of GCN layers and LPA layers, the second section builds up the structure of our network, and the third section feeds the input into our model, calculates the results, and trains model parameters.

5 Experiments

In this section, we first introduce the dataset and baselines. Then, we evaluate our proposed model on the task of hateful user detection and present the results.

5.1 Dataset

Our dataset (Ribeiro, 2018) comes from Kaggle[8]. The dataset was collected by Manoel Horta Ribeiro et al. in 2018 containing 100,386 Twitter users and 2,286,592 edges, among which 4,970 users were annotated as hateful or not by crowdsourcing (544 users are detected as hateful). The dataset also includes a feature vector of length 1,042 for each user, of which most entries are quantitative indicators such as the number of the user's followers, followees, and posts. These features are highly related to a user's behavior and personality, thus helpful to hatefulness detection. The statistics of the dataset is shown in Table 2.

Number of users	100,386
Number of labeled users	4,971
Number of detected hateful users	544
Ratio of hateful users	10.9%
Number of edges	2,286,592
Dimension of user features	1,042
Average number of neighbors for a user	45.6

Table 2: Statistics of the dataset.

5.2 Baselines

We use the following methods as baselines in our experiment. The first three baselines only use the features of each user without the graph structure, whereas the fourth baseline only uses the graph structure without user features. The rest of our baselines are GCN-based methods.

- **Logistic Regression** is a standard method for binary classification problems. In our experiment, we use the Logistic Regression function in Python sklearn package and choose the "lbfgs" solver.
- **Support Vector Machine** is another supervised learning model for binary classification. Similarly, we used the package provided by Python sklearn as the implementation of SVM.
- **Multi-layer Perceptron** is a feedforward artificial neural network model. We use the package from Python sklearn, and choose the adam solver, relu activation function, and use one hidden layer with 100 units.
- **Label Propagation Algorithm** (Zhu et al., 2003) is a semi-supervised node classification method that we discussed in Section 4.1.
- **Graph Convolutional Neural Network** (Kipf and Welling, 2016) is another semi-supervised representation learning method that we discussed in Section 4.2. We use two hidden layers, and train for 200 epochs.
- **Graph Attention Network** (Veličković et al., 2017) is a kind of models using attention mechanism to learn edge weights. By deciding the strength of edges on the graph, this model effectively utilizes the similarity between nodes' features. We take the code from Thekumparampil et al. (2018) and leave the hyperparameters unchanged.

5.3 Experiment Setup

To improve the time and space efficiency of our model, we only select a portion of unlabeled users from the original dataset to construct the social network. Specifically, we sample s unlabeled nodes from the set of all unlabeled nodes, and keep all labeled nodes. Edges are then extracted according to the selected nodes. We set $s = 5,000$, which is close to the number of labeled nodes (4,971). Since unlabeled nodes only contribute to the result by their structural information, these s unlabeled nodes are chosen according to their degrees in the social network (high degree is preferred). We divide the labeled nodes into three parts: training set, validation set, and test set, to train the model, determine optimal parameter settings, and to test the final results, respectively. The ratio of training/validation/test sets is set as $8 : 1 : 1$.

[8] `www.kaggle.com`

We evaluate the performance of models using two metrics: the Area Under Curve (AUC) and the F1 score. Compared with the accuracy metric, AUC and F1 can accurately reflect the performance of classification models even when the sizes of different classes are unbalanced. We train our model for 2,000 epochs using Adam Optimizer and record the performance of the test set when the AUC score of the validation set is maximized. We normalize the feature matrix, transform the input network into the adjacency matrix, and initialize weights in networks according to Glorot and Bengio (2010). During the training process, we employ the L2 regularization and dropout technique. Other hyperparameters, such as the number of GCN layers and LP layers, are listed in the result section.

5.4 Empirical Study

To examine label smoothness assumption, we conduct an empirical study to investigate the relationship between labels and edges in the Twitter dataset. For every pair of labeled nodes, we examine them from two aspects: whether they are connected and whether they are both hateful users. The result is shown in Table 3.

	Connected	Disconnected	Overall	Connected rate
Both hateful	6,104	289,288	295,392	2.07%
At least one normal	13,136	24,397,342	24,410,478	0.05%
Overall	19,240	24,686,630	24,705,870	0.08%
Hateful rate	31.70%	1.17%	1.20%	

Table 3: Statistics of connectivity and labels for labeled nodes.

Table 3 presents the connectivity and the hatefulness status of every labeled pair. The hateful rate is defined as the number of "both hateful" pairs divided by the corresponding overall number. The connected rate is defined as the number of connected pairs divided by the corresponding overall number.

From Table 3, we can see the connected rate for pairs that are both hateful is 2.07%, which is far higher than the connected rate for all pairs (0.08%). On the other hand, when two users are connected, they are more likely to be hateful users (31.70% $\gg$ 1.20%). Therefore, the statistics show that hateful users tend to be connected, and vise versa.

5.5 Results

5.5.1 Learning Curves

We present the learning curves of loss, AUC, and F1 on training set and validation set in Figures 3 and 4, respectively. Our model starts with randomly initialized variables for which AUC is about 0.5. During the training process, the loss of both training and validation decline steadily. The AUC score and F1 score increase fast at first as our model learns to distinguish hateful users.

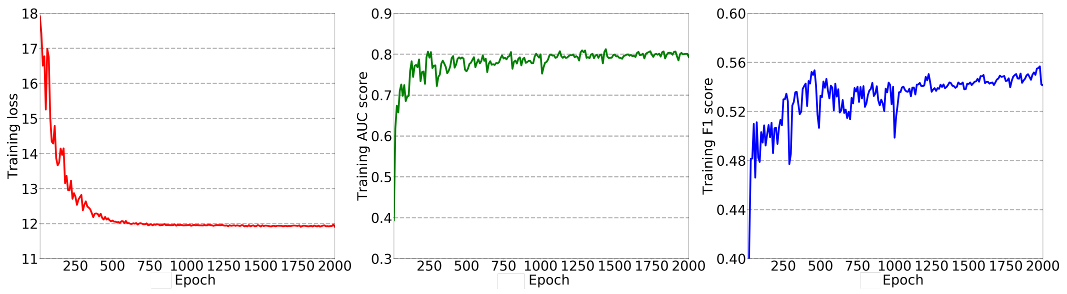

Figure 3: Loss, AUC, and F1 score on the training set.

Due to the dropout and L2-loss mechanism, the AUC and F1 score of training and validation data stay close throughout the process, indicating that our model doesn't overfit.

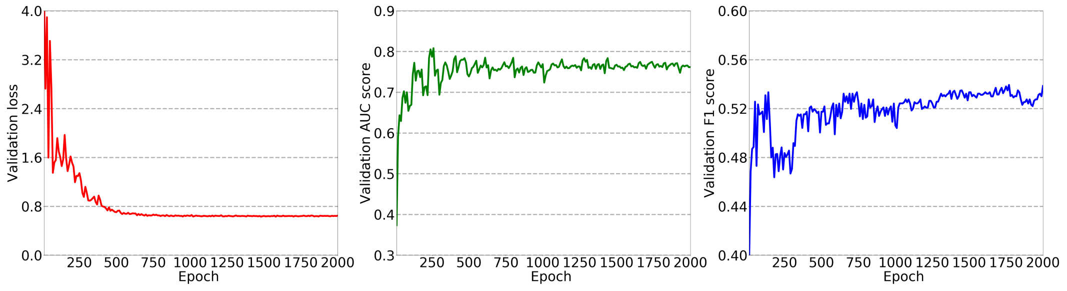

Figure 4: Loss, AUC, and F1 score on the validation set.

5.5.2 Comparison with Baselines

The experimental results of the six baselines and our model are shown in Table 4.

Method	AUC	F1
Logistic Regression	64.9±3.4	-
Support Vector Machine	50.4±4.5	-
MLP Classifier	65.7±3.5	-
Label Propagation	78.0±1.0	36.6±6.1
Graph Convolutional Neural Network	67.8±3.3	20.9±4.3
Graph Attention Network	70.5±2.1	27.2±3.3
AdaGCN	**79.5**±0.6	**47.4**±1.8

Table 4: Mean and standard deviation of AUC and F1 score for all methods. The highest score in each column is marked in bold.

From Table 4, we can see that models using only node features (LR, SVM, MLP) or graph structure (LPA) will lead to information loss, and cannot fully extract the feature of hateful users. The result shows that our proposed AdaGCN model outperforms all six baselines by at least 1.5% in AUC score and at least 11.2% in F1 score.

5.5.3 Impact of Unlabeled Users

Unlabeled users are used mainly to construct the social network. To investigate their impact on the performance of our model, we increase the number of unlabeled users from 0 to 10,000. For each case, we construct the corresponding social network and run our model on it. The results are shown in Figure 5.

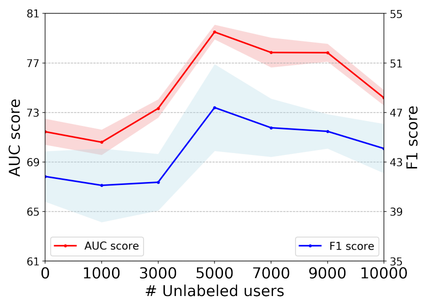

Figure 5: Impact of the number of unlabeled users.

From Figure 5, we can find that sampling approximately 5,000 unlabeled users is optimal. If the number of unlabeled users is too small, the nodes in the social network would be too few to provide meaningful information about the social network structure. On the other hand, if the number of unlabeled users is too large, the learning process will take an extremely long time and the performance will decrease due to the possible noisy nodes.

5.5.4 Hyperparameters Sensitivity

We vary the weight of LPA loss (λ_0), the dimension of GCN hidden layers, learning rate, the weight of l2 loss (λ_1), dropout rate, and the number of LPA layers, respectively, and the results are shown as in Figure 6.

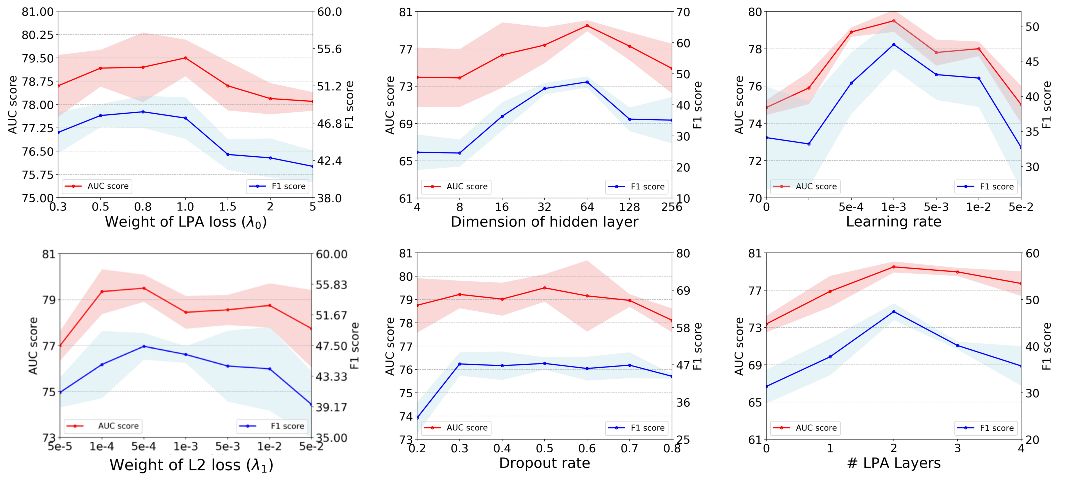

Figure 6: Hyperparameters sensitivity.

From Figure 6, we find that all hyperparameters have impact on the performance of our model. If the dimension of hidden layers is larger than 64, the neural network model would become too complicated and difficult to train; if it's smaller, the model does not have sufficient capability to learn structural information from the graph. As our contribution to the original GCN models, the weight of LPA loss serves to provide a signal from labels, and we observe that the performance reaches its peak when $\lambda_0 = 1.0$. If λ_0 is too small, the impact of LPA loss would not be large enough. On the contrary, if λ_0 is too large, the LPA loss would dominate the loss function, which overwhelms the model and leads to performance decrease.

By comparing the experimental results, we find that when λ_0, dimension of hidden layer, learning rate, λ_1, dropout rate, and the number of LPA layers are set as 1.0, 64, 10^{-3}, 5×10^{-4}, 0.5, and 2, respectively, the performance of our model reaches the optimum.

6 Conclusion

In this paper, we propose a new model, AdaGCN, to detect hateful users in social networks. Different from existing GCN models, our proposed AdaGCN treats weights of edges in the social network trainable and learns transformation matrices and edge weights simultaneously. In order to prevent overfitting, we design a leave-one-out loss function using label propagation to provide an extra supervised signal for the training process of our model. By combining LPA and GCN, we achieve the end-to-end model AdaGCN.

On a dataset of 100,386 Twitter users, our model demonstrates a robust performance of 0.795 ± 0.006 AUC score and 0.474 ± 0.018 F1 score, surpassing all the baselines. Moreover, the standard error of AdaGCN is the lowest among all methods, showing the strong stability of our model.

The limitation of our model mainly lies in the time complexity. The original dataset contains $100k+$ users, which is too large for our model to be trained within a reasonable time limit. Currently, we deal with this problem by removing unlabeled users with small degrees from the graph. In the future, we plan to investigate a more efficient algorithm to sample unlabeled users and maintain the structural information of the social network as much as possible.

The model proposed in this paper will allow online social platforms to detect hateful users and prevent their hateful actions in the future. We believe our work will alleviate the hate speech problem and contribute to a clean and friendly Internet environment.

References

Atwood, J. and Towsley, D. (2016). Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1993–2001.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*.

Burnap, P. and Williams, M. L. (2016). Us and them: identifying cyber hate on twitter across multiple protected characteristics. *EPJ Data Science*, 5(1):11.

Davidson, T., Warmsley, D., Macy, M., and Weber, I. (2017). Automated hate speech detection and the problem of offensive language. In *Eleventh international aaai conference on web and social media*.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852.

Gitari, N. D., Zuping, Z., Damien, H., and Long, J. (2015). A lexicon-based approach for hate speech detection. *International Journal of Multimedia and Ubiquitous Engineering*, 10(4):215–230.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034.

Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023.

Ribeiro, M. (2018). Hateful users on twitter. `https://www.kaggle.com/manoelribeiro/hateful-users-on-twitter#users_neighborhood_anon.csv`.

Ribeiro, M. H., Calais, P. H., Santos, Y. A., Almeida, V. A., and Meira Jr, W. (2018). Characterizing and detecting hateful users on twitter. In *Twelfth International AAAI Conference on Web and Social Media*.

Thekumparampil, K. K., Wang, C., Oh, S., and Li, L.-J. (2018). Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735*.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2017). Graph attention networks. *arXiv preprint arXiv:1710.10903*.

Wang, H., Zhang, F., Hou, M., Xie, X., Guo, M., and Liu, Q. (2018). Shine: Signed heterogeneous information network embedding for sentiment link prediction. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 592–600. ACM.

Wulczyn, E., Thain, N., and Dixon, L. (2017). Ex machina: Personal attacks seen at scale. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1391–1399. International World Wide Web Conferences Steering Committee.

Zhu, X., Ghahramani, Z., and Lafferty, J. D. (2003). Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919.

Zhu, Xiaojin, Z. (2002). Learning from labeled and unlabeled data with label propagation.

Appendix

layers.py

This program serves at the bottom level, constructing the structure of GCN layers and LPA layers. Plus, it also defines how layers operate data and pass them to the next layer.

```
import numpy as np
import tensorflow as tf

flags = tf.app.flags
FLAGS = flags.FLAGS

def uniform(shape, scale=0.05, name=None):
    """Uniform init."""
    initial = tf.random_uniform(shape, minval=-scale, maxval=scale, dtype
        =tf.float32)
    return tf.Variable(initial, name=name)


def glorot(shape, name=None):
    """Glorot & Bengio (AISTATS 2010) init."""
    init_range = np.sqrt(6.0 / (shape[0] + shape[1]))
    initial = tf.random_uniform(shape, minval=-init_range, maxval=
        init_range, dtype=tf.float32)
    return tf.Variable(initial, name=name)


def zeros(shape, name=None):
    """All zeros."""
    initial = tf.zeros(shape, dtype=tf.float32)
    return tf.Variable(initial, name=name)


def ones(shape, name=None):
    """All ones."""
    initial = tf.ones(shape, dtype=tf.float32)
    return tf.Variable(initial, name=name)

def sparse_dropout(x, keep_prob, noise_shape):
    """Dropout for sparse tensors."""
    random_tensor = keep_prob
    random_tensor += tf.random_uniform(noise_shape)
    dropout_mask = tf.cast(tf.floor(random_tensor), dtype=tf.bool)
    pre_out = tf.sparse_retain(x, dropout_mask)
    return pre_out * (1./keep_prob)


def dot(x, y, sparse=False):
    """Wrapper for tf.matmul (sparse vs dense)."""
    if sparse:
        res = tf.sparse_tensor_dense_matmul(x, y)
    else:
        res = tf.matmul(x, y)
    return res


class Layer(object):
    def __init__(self, **kwargs):
        self.vars = {}
        self.sparse_inputs = False

    def _call(self, inputs):
        return inputs
    def _call(self, inputs, init, mask):
        return inputs

    def __call__(self, inputs, init=-1, mask=-1):
        if init == -1:
            outputs = self._call(inputs)
        else:
            outputs = self._call(inputs, init, mask)
        return outputs

class GraphConvolution(Layer):
    def __init__(self, ADJ, input_dim, output_dim, placeholders, dropout
        =0.,
                 sparse_inputs=False, act=tf.nn.relu, bias=False,
                 featureless=False, **kwargs):
        super(GraphConvolution, self).__init__(**kwargs)

        if dropout:
            self.dropout = placeholders['dropout']
        else:
            self.dropout = 0.

        self.act = act
        self.adj = ADJ
        self.sparse_inputs = sparse_inputs
        self.featureless = featureless
        self.bias = bias

        # helper variable for sparse dropout
        self.num_features_nonzero = placeholders['num_features_nonzero']

        with tf.variable_scope('_vars'):
            self.vars['weights_1'] = glorot([input_dim, output_dim], name
                ='weights_1')
            if self.bias:
                self.vars['bias'] = zeros([output_dim], name='bias')

    def _call(self, inputs):
        x = inputs
        # dropout
        if self.sparse_inputs:
            x = sparse_dropout(x, 1-self.dropout, self.
                num_features_nonzero)
        else:
            x = tf.nn.dropout(x, 1-self.dropout)

        # convolve
        if not self.featureless:
            pre_sup = dot(x, self.vars['weights_1'], sparse=self.
                sparse_inputs)
        else:
            pre_sup = self.vars['weights_1']
        support = dot(self.adj, pre_sup, sparse=True)
        output = support
        return self.act(output)

class LabelProp(Layer):
    # Label Propagation Layer.
    def __init__(self, ADJ, **kwargs):
        super(LabelProp, self).__init__(**kwargs)
        self.adj = ADJ

    def _call(self, inputs, init, mask):
        x = inputs * mask
        x = x + init
        new_lab = tf.sparse_tensor_dense_matmul(self.adj, x)
        return new_lab
```

models.py

This program constructs the structure of AdaGCN, which contains GCN layers and LPA layers defined in layers.py. This program also defines the loss function, the accuracy, and the output of our model.

```
from layers import *
from metrics import *

flags = tf.app.flags
FLAGS = flags.FLAGS


class Model(object):
    def __init__(self, **kwargs):
        allowed_kwargs = {'name', 'logging'}
        for kwarg in kwargs.keys():
            assert kwarg in allowed_kwargs, 'Invalid␣keyword␣argument:␣'
                + kwarg
        name = kwargs.get('name')
        if not name:
            name = self.__class__.__name__.lower()
        self.name = name

        logging = kwargs.get('logging', False)
        self.logging = logging

        self.vars = {}
        self.placeholders = {}

        self.gcn_layers = []
        self.gcn_activations = []
        self.gcn_inputs = None
        self.gcn_outputs = None

        self.lp_layers = []
        self.lp_activations = []
        self.lp_inputs = None
        self.lp_outputs = None

        self.loss = 0
        self.accuracy = 0
        self.optimizer = None
        self.opt_op = None

    def _build(self):
        raise NotImplementedError

    def build(self):
        """ Wrapper for _build() """
        with tf.variable_scope(self.name):
            self._build()

        # Build sequential layer model: GCN
        self.gcn_activations.append(self.gcn_inputs)
        for layer in self.gcn_layers:
            hidden = layer(self.gcn_activations[-1])
            self.gcn_activations.append(hidden)
        self.gcn_outputs = self.gcn_activations[-1]

        # Build sequential layer model: LP
        self.lp_activations.append(self.lp_inputs)
        for layer in self.lp_layers:
            hidden = layer(self.lp_activations[-1], self.lp_inputs, self.
                placeholders['labels_mask2'])
            self.lp_activations.append(hidden)
        self.lp_outputs = self.lp_activations[-1]

        # Store model variables for easy access
        variables = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
            scope=self.name)
        self.vars = {var.name: var for var in variables}

        self._loss()
        self.opt_op = self.optimizer.minimize(self.loss)

        self._accuracy()

    def predict(self):
        pass

    def _loss(self):
        raise NotImplementedError

    def _accuracy(self):
        raise NotImplementedError

class AdaGCN(Model):
    def __init__(self, ADJ, placeholders, input_dim, **kwargs):
        super(AdaGCN, self).__init__(**kwargs)

        self.gcn_inputs = placeholders['features']
        self.lp_inputs = placeholders['labels']
        self.input_dim = input_dim
        self.adj = ADJ
        self.output_dim = placeholders['labels'].get_shape().as_list()[1]
        self.placeholders = placeholders
        self.optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.
            learning_rate)
        self.build()

    def _loss(self):
        # Weight decay loss
        for var in self.gcn_layers[0].vars.values():
            self.loss += FLAGS.weight_decay * tf.nn.l2_loss(var)

        # Cross entropy error
        self.loss += masked_softmax_cross_entropy(self.gcn_outputs, self.
            placeholders['labels'],
                                                  self.placeholders['
                                                      labels_mask'])
        self.loss += masked_softmax_cross_entropy(self.lp_outputs, self.
            placeholders['labels'],
                                                  self.placeholders['
                                                      labels_mask']) *
                                                      FLAGS.lp_weight

    def _accuracy(self):
        self.accuracy = masked_accuracy(self.gcn_outputs, self.
            placeholders['labels'], self.placeholders['labels_mask'])

    def _build(self):
        self.gcn_layers.append(GraphConvolution(
                                              ADJ=self.adj,
                                              input_dim=self.input_dim,
                                              output_dim=FLAGS.hidden1,
                                            placeholders=self.
                                                placeholders,
                                            act=tf.nn.relu,
                                            dropout=True,
                                            sparse_inputs=True))

        self.gcn_layers.append(GraphConvolution(
                                            ADJ=self.adj,
                                            input_dim=FLAGS.hidden1,
                                            output_dim=self.output_dim,
                                            placeholders=self.
                                                placeholders,
                                            act=lambda x: x,
                                            dropout=True))

        self.lp_layers.append(LabelProp(ADJ=self.adj))
        self.lp_layers.append(LabelProp(ADJ=self.adj))

    def predict(self):
        return tf.nn.softmax( self.gcn_outputs)
```

train.py

This program serves to feed the input into our model, train model parameters, and calculate the results.

```
import tensorflow as tf
import numpy as np
import scipy as sp
import pandas as pd
import csv
import networkx as nx
import time
import sys
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from metrics import *

from utils import *
from models import AdaGCN

# Load data
adj = graph_partition('data/users_clean_modified.graphml')
adj = tuple_to_sparsetensor(adj)
y, features, train_mask, val_mask, test_mask = features_partition('data/
    users_neighborhood_anon_modified.csv')
y_train = (np.eye(train_mask.shape[0]) * train_mask).dot(y)
y_val = (np.eye(val_mask.shape[0]) * val_mask).dot(y)
y_test = (np.eye(test_mask.shape[0]) * test_mask).dot(y)

val_mask = val_mask.reshape(1, -1)[0]
test_mask = test_mask.reshape(1, -1)[0]
train_mask = train_mask.reshape(1, -1)[0]

# Settings
flags = tf.app.flags
FLAGS = flags.FLAGS
flags.DEFINE_string('dataset', 'cora', 'Dataset␣string.')
flags.DEFINE_string('model', 'gcn', 'Model␣string.')
flags.DEFINE_float('learning_rate', None, 'Initial␣learning␣rate.')
flags.DEFINE_integer('epochs', None, 'Number␣of␣epochs␣to␣train.')
flags.DEFINE_integer('hidden1', None, 'Number␣of␣units␣in␣hidden␣layer␣1.
    ')
```

```
flags.DEFINE_float('dropout', None, 'Dropout␣rate␣(1␣−␣keep␣probability).
    ')
flags.DEFINE_float('weight_decay', None, 'Weight␣for␣L2␣loss␣on␣embedding
    ␣matrix.')
flags.DEFINE_float('lp_weight', None, 'Weight␣for␣lp␣loss')


def oneRun(learning_rate, epochs, hidden1, dropout, weight_decay,
    lp_weight, Seed, cnt):
    # Set random seed
    seed = Seed
    np.random.seed(seed)
    tf.set_random_seed(seed)

    FLAGS.learning_rate = learning_rate
    FLAGS.epochs = epochs
    FLAGS.hidden1 = hidden1
    FLAGS.dropout = dropout
    FLAGS.weight_decay = weight_decay
    FLAGS.lp_weight = lp_weight

    model_func = AdaGCN

    # Define placeholders
    placeholders = {
        'features': tf.sparse_placeholder(tf.float32, shape=tf.constant(
            features[2], dtype=tf.int64)),
        'labels': tf.placeholder(tf.float32, shape=(None, y_train.shape
            [1])),
        'labels_mask': tf.placeholder(tf.int32),
        'labels_mask2': tf.placeholder(tf.float32, shape=(None, y_train.
            shape[1])),
        'dropout': tf.placeholder_with_default(0., shape=()),
        'num_features_nonzero': tf.placeholder(tf.int32)  # helper
            variable for sparse dropout
    }

    # Create model
    model = model_func(ADJ=adj, placeholders=placeholders, input_dim=
        features[2][1], logging=True)

    # Initialize session
    sess = tf.Session()

    # Define model evaluation function
    def evaluate(features, labels, mask, placeholders):
        t_test = time.time()
        feed_dict_val = construct_feed_dict(features, labels, mask,
            placeholders)
        outs_val = sess.run([model.loss, model.accuracy, model.predict()
            ], feed_dict=feed_dict_val)
        return outs_val[0], outs_val[1], outs_val[2], (time.time() −
            t_test)

    # Init variables
    sess.run(tf.global_variables_initializer())

    result_max_AUC = 0.6
    test_AUC = 0, test_f1 = 0, train_loss = []
    train_auc = [], train_f1 = []
    val_auc = [], val_f1 = [], val_loss = []

    # Train model
    for epoch in range(FLAGS.epochs):
        t = time.time()
        # Construct feed dictionary
        feed_dict = construct_feed_dict(features, y_train, train_mask,
            placeholders)
        feed_dict.update({placeholders['dropout']: FLAGS.dropout})

        # Training step
        outs = sess.run([model.opt_op, model.loss, model.accuracy],
            feed_dict=feed_dict)

        # Validation
        cost, acc, pred, duration = evaluate(features, y_val, val_mask,
            placeholders)
        val_AUC, val_Acc, val_F1 = calc_AUC(y_val, pred, val_mask)
        if epoch > 30:
            if val_AUC > result_max_AUC:
                result_max_AUC = val_AUC
                test_cost, test_acc, test_pred, duration = evaluate(
                    features, y_test, test_mask, placeholders)
                test_AUC, test_acc, test_f1 = calc_AUC(y_test, test_pred,
                     test_mask)

        #Recording
        if (epoch+1) % 10 == 0:
            print("[turn]", '%02d' % (cnt),
                      "Epoch:", '%04d' % (epoch + 1),
                      "train_loss=", "{:.5f}".format(outs[1]),
                      "val_loss=", "{:.5f}".format(cost),
                      "val_AUC=", "{:.5f}".format(val_AUC),
                      "val_f1=", "{:.5f}".format(val_F1),
                      "time=", "{:.5f}".format(time.time() - t))
            val_auc.append(val_AUC)
            val_f1.append(val_F1)
            val_loss.append(cost)
            cost, acc, pred, duration = evaluate(features, y_train,
                train_mask, placeholders)
            train_AUC, train_Acc, train_F1 = calc_AUC(y_train, pred,
                train_mask)
            train_auc.append(train_AUC)
            train_f1.append(train_F1)
            train_loss.append(outs[1])

        if (epoch+1) % 100 == 0:
            std_outputs(y_val, pred, val_mask)
            print("pre_answer:␣", "{:.5f}".format(test_AUC), "{:.5f}".
                format(test_f1), "{:.5f}".format(result_max_AUC))

    print("Optimization␣Finished!")
    print("answer:␣", "{:.5f}".format(test_AUC), "{:.5f}".format(test_f1)
        )
    print("train_auc␣=", train_auc)
    print("train_f1␣=", train_f1)
    print("train_loss␣=", train_loss)
    print("val_auc␣=", val_auc)
    print("val_f1␣=", val_f1)
    print("val_loss␣=", val_loss)
    print('END!')
```

Acknowledgements

On February 10, 2018, a 22-year-old boy named Ted Senior hanged himself on the woodland after his chatting content with a girl was maliciously judged on social media[9]. In my life, I've seen my friend expressing opinions on the forum, but received anonymous insults and threatening comments, which made him painful and scared. I was deeply affected by these online hateful actions and the consequences. Information technology brings us into the Internet Era, and online social networks truly make our lives more efficient and interesting. Yet nothing comes purely beneficial. Hate speech and hateful users are exactly an issue unintentionally brought by information technology, which I am eager to find an effective way to detect and control.

Here, I want to express my sincere gratitude to my mentor Hongwei Wang, who guides me through the research and gives me gratis help. Mr. Wang is a postdoctoral fellow at Stanford University and has done intensive studies in the Graph Neural Network field. In 10th grade, I've taken courses on Artificial Intelligence and learned how to build simple Artificial Neural Networks in my school. With the guidance of Mr. Wang, I spent two months studying the background knowledge of Graph Convolutional Neural Networks and Label Propagation Algorithm, and then used another two months to implement the programs and conduct experiments, gradually building the structure and improving the performance of my new model. Finally, I spent one month writing the paper. Because of the time difference between Beijing and San Francisco, Mr. Wang often discusses with me at midnight. He patiently teaches me to conduct every experiment and complete the paper in accordance with strict academic requirements. His strict academic attitude deeply affected me, and the academic training I received will benefit me whenever I conduct researches in the future.

At last, I would like to thank the S.-T. Yau High School Science Award for allowing me to start this exciting exploration in my favorite field. My favorite subjects are Computer Science and Mathematics. C++, Python, Java... As I learned more and more programming languages, I gradually realized that algorithms are the essence of all programs, connecting abstraction to reality and reducing problems into abstract forms to which computing power can be applied. Mathematics is the foundation of algorithms. In competitive algorithm-designing courses, I was introduced to the Pólya enumeration theorem, the application of Simpson's rule in computing geometry, and the Mobius inversion in number theory. Fourier Transformation convolute functions into elegant forms, which shocked me as much as magicians producing roses out of thin air. The experience of attending the S.-T. Yau High School Science Award makes me think further on how to use mathematical tools to optimize algorithms and solve real-life problems.

Thanks again to my teachers, family members, and the ones who have helped me in the research!

[9] `telegraph.co.uk/news/2018/03/07/medical-student-boasted-online-fling-attractive-girl-kills-rugby/`

Introduction of Participant

Xingjian Bai is currently attending 12th grade in the Experimental High School Attached to Beijing Normal University. He has a wide range of interests, thinks actively, and loves mathematics and computer science. He is the chairman of both the Computer Club and the Humanities Institute of his school. Having studied algorithms and programming for 5 years.

- Invited to attend the Canadian Computing Olympiad (CCO), the highest level programing competition in Canada, in May 2018; Achieved the highest score among all the competitors, including all four national team members in Canada.

- Won the silver medal of the National Olympiads of Informatics (NOI) 2018.

- Won the 3rd place, first prize in the National Olympiads of Informatics Provincial (NOIP).

- Participated in the USA Computing Olympiads (USACO) Open and achieved a full score in March 2019;
- Invited as the only foreign participant to attend the USA Computing Olympiad Training Camp in May 2019 (only top 25 US high school students can participate, to select the USA national team).

- Won the Finalist Award (7%) in the High School Mathematical Contest in Modeling (HiMCM) 2018.

- Won first place in the Math League 2018 in China Region.

- Won the Distinction Honor Roll (1%) in American Mathematics Competition (AMC) 2019, and scored 8 points in AIME.

- Won the Golden Sail Award (top 2%) for two consecutive years. This scholarship is offered by the Experimental High School Attached to BNU to encourage students who are well-rounded and have distinguished achievements in extracurricular competitions.

致谢信

2018年2月10日，一个名叫Ted Senior的22岁男孩在林地上吊自杀，原因是一些人在社交媒体上恶意地分享和评判他与一名女孩的聊天内容。在我身边，我的同学好友在学校论坛发表观点，但是遭受匿名的辱骂和攻击，这种羞辱让他感到非常痛苦。我深深地被这些可恨的行为和可怕的后果所触动。计算机科学带来了信息时代，社交网络改变了我们的生活，我们期望技术会让世界更美好。但没有什么是尽善尽美的。网络暴力是信息技术无意中带来的一个问题，我渴望找到一种方法来发现和控制它们。

感谢王鸿伟老师引导我探索这个领域，无偿的给予我帮助。王鸿老师目前在斯坦福做博士后，在这个领域有深厚的积累。高一时我学习过人工智能的课程，学会了构建人工神经网络。在王老师带领下，我用两个月时间研究图神经网络和标签传播算法相关知识，然后用两个月时间编程实验，逐步构建和完善新的模型，最后用一个月时间撰写论文。因为时差的缘故，王老师经常在深夜给我指导。他耐心指导我按照严格的学术要求进行实验和完成论文。王老师的博学和严谨，给我树立了榜样，他给予我的学术训练，将使我终身受益。

最后，我要感谢丘成桐中学科学奖，让我在喜爱的领域展开了一次激动人心的探索。中学时代，我最大的乐趣是编程算法和数学。C++、Html、CSS、Javascript 、Go、Python，一门又一门的编程语言成为我思考的延伸和解决问题的工具。当我学习了越来越多的编程语言，我体会到算法是一切程序的精髓，是抽象和现实之间的桥梁。我们要用算法模型将现实问题归纳、抽象，变成计算力可以应用的形式。数学是算法的底层规律。在算法设计课上，我接触了群论中的Polya引理，计算几何中的辛普森积分，数论中的莫比乌斯反演。傅理叶变换把函数卷积成优雅简洁的形式，给我的震撼不亚于魔术师凭空变出一朵玫瑰。这一次课题研究，让我进一步思考如何运用数学规律完善编程算法，然后去解决现实问题，收获很大。

再一次感谢在课题研究中给予我帮助的老师和家人！

参赛队员介绍

白行健，男，北京师范大学附属实验中学国际部高三学生，兴趣广泛，思维活跃，热爱数学和计算机科学。担任学校计算机社社长和人文社社长。

•2018年5月受邀参加加拿大信息学奥林匹克竞赛（CCO）决赛及国家队集训营，获得决赛金奖第一名。

•2018年7月获得中国信息学奥林匹克竞赛决赛（NOI）银牌。

•2018年11月获得中国信息学联赛（NOIP）北京市提高组一等奖第3名。

•2019年1月参加美国信息学奥林匹克竞赛(USACO)公开赛获得满分，5月作为唯一的外国学生受邀参加美国国家队集训营（USACO前25名美国中学生参加,选拔国家队成员）。

•2018年获得美国高中生数学建模竞赛（HiMCM）**Finalist**奖项（一等奖7%）。

•2018年获得美国数学大联盟杯（Math League）中国赛区第一名。

•2018年获得美国数学竞赛(AMC) Distinction Honor Roll（荣誉奖1%），AIME获得8分。

•2017、2018连续两年获得北京师范大学附属实验中学金帆奖（top 2%）。

学术诚信声明

本参赛团队声明所提交的论文是在指导老师指导下进行的研究工作和取得的研究成果。尽本团队所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果。若有不实之处，本人愿意承担一切相关责任。

参赛队员：白行健　　指导老师：王鸿伟

2019 年 9 月 14 日