Graduate Group 01 - Aidan Lambrecht, Alex Norris, Kyle Lesinger
COMP 6130
November 24, 2020

Analysis On "Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective"

SECTION 1: INTRODUCTION

Graph Neural Networks are currently one of the most sought-after topics in machine learning. In the last three years, papers on Graph Neural Networks or GNNs have held a large space in major machine learning and AI conferences around the world. This report analyzes and adds onto the findings from a "*Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective*" by Xu, Chen, Liu, Chen, Weng, Hong, and Lin [Kaidi et al. 2019]. Through study of the above mentioned paper, we explore the empirical results from our own testing on a variety of machines and also build on the existing work hoping to add further capability for more advanced testing.

The major problem that this paper discusses is the lack of adversarial robustness. To start, one can define adversarial robustness as the overall capability to withstand adversarial attacks and continue to operate on a high level. Adversarial attacks work to trick the machine or network with noisy or bad data in order to produce a false result or low classification score. Many of the modern GNNs today are highly susceptible to these adversarial attacks resulting in major damage to the network's classification accuracy. The authors proposed a new topology attack and defense method that can increase the adversarial robustness of GNNs in order to decrease misclassification rates of graphed data against adversarial attacks. Their results indicate that this new framework can increase the robustness against greedy and gradient based attacks while still achieving high accuracy on the original graph.

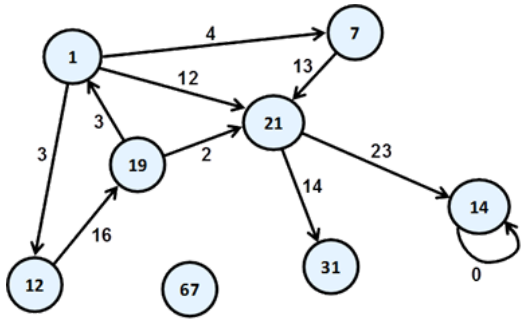

Figure 1. Directed Weighted Graph

Moving onto the assembly of a contemporary GNN, we start with the basic building blocks. Graph Neural Networks are constructed with the common computer science abstract data type, the graph. These graphs are made up of a collection of vertices (nodes or points), and

edges. The nodes and their edges may be directed (one-way) or undirected (two-way). The final component of these graphs is the weight associated with each edge and potential value that can be assigned to each node.

Now that the graph and its attributes have been presented, the next major component in a GNN is the neural network. Neural networks have resurged in the world of computer science, becoming one of the major staples in the field once again after becoming dormant in the research scene many years ago. Within the GNN, the neural network performs as a feed forward network that uses backpropagation to train itself.

The network starts off with an input layer consisting of adjacency matrices that then passes the input layer through a pre-defined number of hidden layers that translate the input to reach a desired output. Typically these neural networks are trained to reach an optimal model that can accurately classify items at a high accuracy. During training, these networks utilize gradient descent to obtain a local minimum that results in a better performing model. Gradient descent is applied in order to minimize a loss function. The loss function is the ultimate metric when determining how the model will perform given that set of input parameters. In conclusion, the gradient descent function selects the optimal parameters that will result in the best model output in classification accuracy.

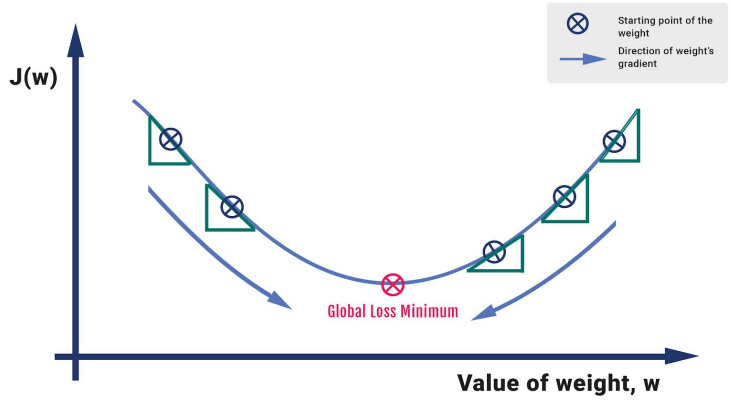

Figure 2. Gradient Descent

SECTION 2: EXPLANATION OF ALGORITHMS

To address the problem of adversarial robustness [Xu *et al.,* 2019] proposes two new attack methods and an adversarial training method. The two attack methods are called Projected Gradient Descent (PGD) Topology Attack and Min-max Topology Attack, and the adversarial training method makes use of these attacks to increase overall robustness through training.

To understand the first attack, one must first understand PGD. As described above, gradient descent is a first-order optimization method used to find the local minimum of a graph. By taking the negative gradient at the current point in the graph, one can iteratively move toward a local minimum in the graph. This helps to optimize the loss function of the network, which results in a more accurate final prediction.

However, sometimes the step that the gradient asks the algorithm to take is not inside the search space that defines the problem to be solved. This is where “projection” comes into play.

In order to move the step onto the desired search space, PGD projects the step onto the closest point in the search space. An illustration depicting this process is shown in Figure 3 [Andersen, 2020].

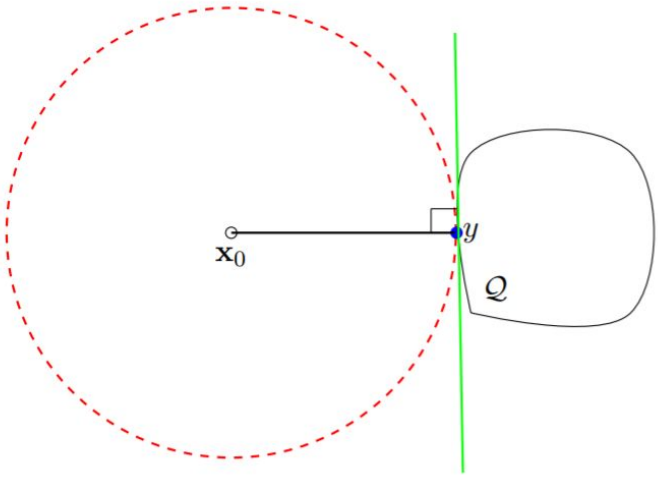

Figure 3. Example of Step X_0 being projected onto point y in search space Q

Equation 1 shows the PGD attack formula used to minimize the performance of the GNN where f(s) describes the confidence with which the GNN classifies the node in the given graph, S is the set of all edges in the graph, and W is the weights matrix of the GNN.

$$\begin{aligned} \underset{\mathbf{s}}{\text{minimize}} \quad & f(\mathbf{s}) := \textstyle\sum_{i \in \mathcal{V}} f_i(\mathbf{s}; \mathbf{W}, \mathbf{A}, \{\mathbf{x}_i\}, y_i) \\ \text{subject to} \quad & \mathbf{s} \in \mathcal{S}, \end{aligned}$$

Equation 1. PGD attack formula

Equation 2 shows the loss function that is used in conjunction with the attack formula to measure the success of the attack and to steer the attack in the most beneficial direction.

$$f_i(\mathbf{S}, \mathbf{W}; \mathbf{A}, \{\mathbf{x}_i\}, y_i) = \max \left\{ Z_{i,y_i} - \max_{c \neq y_i} Z_{i,c}, -\kappa \right\}$$

Equation 2. Loss function for PGD attack

In the case of this graph attack, the step is projected onto the closest edge in the graph S. This is how edges are selected for perturbation, and these perturbed edges will result in lower classification accuracy for the GNN being attacked.

The other attack method, Min-max Topology Attack, builds on top of the PDG attack. This attack adds an efficiency component to the general attack described above. By minimizing the number of perturbations that occur in the graph while maximizing the classification accuracy drop, this attack serves to prevent detection while still doing maximum damage. Equation 3

shows the Min-max Attack formula used to guide the attack. By minimizing performance of the GNN based on the selected edge s and maximizing performance based on the weights matrix W, this attack perturbs edges in the given graph.

$$\underset{\mathbf{s}\in\mathcal{S}}{\text{minimize}}\ \underset{\mathbf{W}}{\text{maximize}}\ f(\mathbf{s},\mathbf{W}) = \sum_{i\in\mathcal{V}} f_i(\mathbf{s},\mathbf{W};\mathbf{A},\{\mathbf{x}_i\},y_i),$$

Equation 3. Min-max Attack formula

The last focus of the paper was on its new adversarial training framework based on the attacks described above. By essentially doing the opposite of the Min-max attack, GNNs can be trained to classify attacked edges and nodes accurately. The formula for this method is shown in Equation 4.

$$\underset{\mathbf{W}}{\text{minimize}}\ \underset{\mathbf{s}\in\mathcal{S}}{\text{maximize}}\ -f(\mathbf{s},\mathbf{W}),$$

Equation 4. Adversarial training formula

SECTION 3: DATASETS AND ENVIRONMENT SETUP

The primary datasets utilized in the paper are the commonly used Citeseer and Cora citation networks. A citation network type dataset represents a collection of documents that act as the nodes, and citation links that represent the edges. Each document was also classified into one of several categories. Coupled with this graph is a matrix representing all individual words that occur within each document ten or more times. Each document has its own vocabulary.

A third dataset was created and tested. Built from the directed graph dataset available on Canvas, we call this dataset, unsurprisingly, ‘Canvas.’ Because the only features of this dataset are nodes and edges, it was necessary to create synthetic features to train a similar network to the original GNN described in the paper. This was done in several steps.

The first step was to assign categories to each node. This was done via the script ‘add_cats.py’ found in Appendix C. This attached an extra value (category) to each line and ensured that each ‘document’ was always classified in the same category, regardless of the which ‘citation’ it was paired with on that specific line.

The second step was creating a synthetic vocabulary for the documents. To do so, an arbitrary vocabulary size of 2500 was chosen (falling between the Cora and Citeseer vocabulary sizes), and for each document, between 200 and 300 indices were randomly chosen to be value ‘1’ (meaning there were ten or more instances of that ‘word’ in the current document).

To simulate consistent vocabularies between similarly classified documents, all documents of the same category had their vocabularies filled by a random number generator with the same seed as the others of that category. The difference was in how many indices were set high. This ensures similar, but not identical, vocabularies between documents in the same category. This process was accomplished in ‘create_dataset.py,’ shown in Appendix D.

With this, the Canvas dataset was completed, and the three datasets, Cora, Citeseer, and Canvas, were used in the training and evaluation of six GNN models. The sizes of each part of the datasets are shown in Table 1.

Table 1. Dataset sizes and statistics

	Cora	**Citeseer**	**Canvas**
Documents	2708	3327	5242
Vocabulary	1433	3703	2500
Categories	7	6	7

To prepare the python environment and for training and evaluations on these datasets, there are a few required packages that must be installed through conda and can run on windows subsystem for linux. The required packages include scipy, numpy, nomkl, python 3.6, and tensorflow 1.13.1. After setting up the environment and installing the necessary packages, cloning the GitHub directory is needed and then processing of graph data can occur.

All model training and attacks can be completed in the command line. To train models, you must edit the train.py scripts to include the names of the graph datasets and then run the command line code. To train robust models, the adv_train_pgd.py script must be edited as well by changing the names of the datasets. However, this can also be done through command line arguments. Attacking natural models requires a hidden layer of 16 while attacking robust models requires a hidden layer of 64.

SECTION 4: EXPERIMENTATION AND RESULTS

For the evaluation of the project, several experiments were run. The code base comes equipped with two types of attack: the Min-max Topology Attack described earlier and a Carlili-Wagner Attack (CW) discussed in [Carlini and Wagner, 2017]. All of our models were attacked with both methods with a 5% edge perturbation rate via the attack.py script in the code base. Five percent was chosen because the authors of the paper also used this value, making our comparison to their results all the more relevant.

After each model was attacked, evaluations were run with several different metrics. This was accomplished by modifying the ‘attack.py’ script (found in Appendix D) in the code base and adding an evaluation function. Among those metrics are Hamming Loss, F1 Micro Score, F1 Macro Score, and Tensorflow’s own Accuracy metric. Hamming Loss is a measure of the amount of labels incorrectly predicted, represented as a fraction of incorrect predictions to total predictions. The lower the score, the better performance [Scikit Learn Hamming].

F1 score is a measure of the precision and recall of a model. In this case, the higher the score, the better a model has performed. Equation 5 describes the F1 formula.

$$F1 = 2 * (precision * recall) / (precision + recall)$$

Equation 5. F1 formula

More specifically, F1 Macro Score, according to [Scikit Learn F1], "Calculate[s] metrics for each label, and find[s] their unweighted mean," while F1 Micro Score "Calculate[s] metrics globally by counting the total true positives, false negatives and false positives."

Tensorflow's Accuracy metric is the exact opposite of Hamming Loss: instead of giving a fraction representing the number of incorrect predictions, it counts the number of correct predictions. Since Accuracy and Hamming Loss are redundant, only Hamming Loss is displayed in the results. Results for each type of attack are shown in Tables 1 and 2 in Appendix A.

In addition to these results, efficiency tests were also performed on the models. Both training times and attack times for each model were recorded. Tests were run on a Dell XPS 15 (2016), the Auburn Gate Machines, and the Alabama Supercomputer Authority GPU cluster. The specifications of each machine are shown in Table 3 in Appendix B.

However, due to time constraints, several models were unable to be trained on all available machines. Results are presented as they are available. In Appendix B, Table 4 displays model training times, and Table 5 shows model attack times for the PGD attack, while Table 6 shows model attack times for the CW attack.

SECTION 5: ANALYSIS

Accuracy:

An easy comparison to make is between the results of the implemented natural Cora (nat_cora) model and the results of the one in the original paper. Looking back at their results, one can see that their nat_cora model scored ~72% accuracy on their classification graph under a CE attack. However, our implemented model only ~68% accuracy, which is a significant drop.

This inconsistency has thrown some confusion onto their work. Our models routinely performed 4-10% worse on classification tasks than theirs did. One explanation for this is that their attack scripts could be stronger than they thought, doing more damage to GNN classification ability than expected. However, that implies that something else was different between their training and testing methods and ours, though we took care to implement their project in the same manner as they did.

The other striking thing about our accuracy results is that the adversarial (robust) models show no improvement in classification accuracy over the naturally trained models. In two cases, the robust models even performed worse than their natural counterparts. This is particularly interesting because their robust models performed on average ~5% better than the natural ones.

A possible reason for this discrepancy is that their adversarial training method was merely ineffective. This appears to not be the case, or at least not entirely, because the Canvas models show the effects of this training. Under the PGD attack, the robust Canvas model shows almost 20% accuracy improvement over the natural model, and under the CE attack, there is a smaller but still astounding 11% accuracy improvement.

This indicates that there is some merit to the proposed adversarial training network, but it is strange that we could not observe this improvement over all six of our models.

Efficiency:

Unsurprisingly, one can observe that in both the training times and attack times, runtime decreases as the machine hardware improves. From the Dell XPS 15, there is a substantial decrease in runtime to the Auburn Gate Machines and another decrease when running on the Alabama Supercomputer Authority GPU Cluster.

Training times for the robust models were prohibitively long, particularly for the Citeseer and Canvas models, taking upwards of thirty five hours for Citeseer and seventy hours for Canvas.

Strengths and Weaknesses:

It can be easy to highlight our implementation of the GCN and the attacks as ineffective and inconclusive, but the network still had some positive areas throughout analysis. Specifically, there exist several strengths to point out within the project. To start, the network scaled upward relatively well, as a couple years ago it would have been difficult to implement a fully functioning GCN or GNN on outdated WSL systems with limited access to processing hardware. With the introduction of WSL 2 and better virtualization technology, the algorithm was able to run on limited personal computers. The adversarial attacks resulted in increased robustness on our custom Canvas dataset from both the PGD and CW attacks. The algorithm is clearly working well here and should hold as a positive outcome.

Throughout testing and evaluation there was high amounts of variance and instability within the results. Specifically from testing the Cora and Citeseer models, both natural and robust. While the authors' work claims a major improvement, our implementation failed to produce similar results, detracting from the major selling points the authors advertised.

Another valid weakness within the program is the lack of hardware and resource monitoring. We had made plans to include this functionality but they were fully developed. Without proper hardware management, training on large datasets may produce errors claiming memory issues. It would be worth adding support for resource monitoring and estimation so that it would be easier to draw the line for what is possible on hardware X and what is not. The original implementation lacks a clear way to handle and process new graph datasets. We ended up implementing this functionality, but it detracts from the authors' original usability.

SECTION 6: CONCLUSION

In this work, we investigated and verified the results of Xu et al, 2019 who proposed two new graph attack methods and one new adversarial training method for GNNs. After training networks on both of the datasets that they used (Cora and Citeseer) as well as one of our own making (Canvas), we ran evaluations with Hamming Loss, F1 Micro Score and F1 Macro Score as metrics.

Results show several discrepancies with their work, despite our best efforts at replicating their methods, perhaps meaning that their proposed attacks are more effective than they thought and that their adversarial training framework is not effective.

Further research can be done into replicating their results, though it would also be useful to perform more thorough experimentation. We could test more graph attack methods or change the percentage of edges perturbed in each graph. We could also test more datasets, such as the PubMed dataset, which is an even larger graph dataset than either Cora or Citeseer.

Real-world application of this technology involves normal GNN activities, including GPS route optimization, drug side-effect predictions, image classification, and more. However, the adversarial training framework could be applied to GNNs deployed for these tasks, making them simultaneously less susceptible to attacks and more reliable, while still maintaining performance.

Works Cited

[Andersen, 2020] Ang, Andersen. *Projected Gradient Algorithm*. UMONS, Belgium. <https://angms.science/doc/CVX/CVX_PGD.pdf> Accessed Nov. 24, 2020.

[Carlini and Wagner, 2017] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. *In Security and Privacy (SP), 2017 IEEE Symposium* on, pages 39–57. IEEE, 2017.

[Kaidi Xu, et al 2019] Kaidi Xu, Honggee Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, Xue Lin. Topology Attack and Defense for Graph Neural Networks: An Optimization Perspective. *arXiv:1906.04214*, 2019.

[Scikit Learn F1] Scikit Learn. *sklearn.metrics.f1_score*. scikit learn developers, 2020. <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html> Accessed Nov. 24, 2020.

[Scikit Learn Hamming] Scikit Learn. *sklearn.metrics.hamming_loss*. scikit learn developers, 2020. <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.hamming_loss.html> Accessed Nov. 24, 2020.

Appendix A: Evaluation Results

Table 1. Attack results for Min-max Topology Attack

MIN-MAX ATTACK	**Nat Cora**	**Rob Cora**	**Nat Citeseer**	**Rob Citeseer**	**Nat Canvas**	**Rob Canvas**
Hamming Loss	0.316	0.31	0.398	0.404	0.603	0.4
F1 Micro	0.684	0.69	0.602	0.596	0.397	0.6
F1 Macro	0.6465	0.687	0.55137	0.57499	0.39219	0.59649

Table 2. Attack results for CW Attack

CW ATTACK	**Nat Cora**	**Rob Cora**	**Nat Citeseer**	**Rob Citeseer**	**Nat Canvas**	**Rob Canvas**
Hamming Loss	0.317	0.349	0.406	0.403	0.614	0.505
F1 Micro	0.683	0.651	0.594	0.597	0.386	0.495
F1 Macro	0.6678	0.6441	0.56301	0.57947	0.3882	0.48667

Appendix B: Efficiency Results

Table 3. Machine specifications

MACHINE	CORE/THREAD COUNT	CLOCK SPEED	MEMORY
Dell XPS 15	4 / 8	2.60 GHz	16 GBs
AUB Gate	8 / 16	2.80 GHz	24 GBs
ASA GPU*	32 / 64	4.3 GHz	32 GBs

***denotes use of tensorflow-GPU and accelerated performance**

Table 4. Efficiency results for model training

TRAINING TIMES (mins)	**Nat Cora**	**Rob Cora**	**Nat Citeseer**	**Rob Citeseer**	**Nat Canvas**	**Rob Canvas**
Dell XPS 15	0.116	425	0.70	3100	1.1	7500
AUB Gate	0.30	725	0.85	2100	1.67	4250
ASA GPU	0.09	270	0.25	600	0.92	2874

Table 5. Efficiency results for PGD attack

PGD ATTACK TIMES (mins)	**Nat Cora**	**Rob Cora**	**Nat Citeseer**	**Rob Citeseer**	**Nat Canvas**	**Rob Canvas**
Dell XPS 15	2.50	1.49	10.16	10.83	24.00	24.50
AUB Gate	2.55	2.57	6.51	6.58	14.73	16.67
ASA GPU	1.30	1.15	4.89	5.12	11.26	12.79

Table 6. Efficiency results for CW Attack

CW ATTACK TIMES (mins)	**Nat Cora**	**Rob Cora**	**Nat Citeseer**	**Rob Citeseer**	**Nat Canvas**	**Rob Canvas**
Dell XPS 15	1.46	1.48	10.79	10.66	26.16	27.32
AUB Gate	2.48	2.53	6.65	6.55	15.33	16
ASA GPU	1.3	1.15	3.78	3.59	8.72	8.80

Appendix C: add_cats.py

```
# add categories to the directed graph dataset from canvas

import numpy as np
import pickle as pkl
import networkx as nx
import scipy.sparse as sp
from scipy.sparse.linalg.eigen.arpack import eigsh
import sys

values = np.array([[0,0]])
xvalues = np.array([], dtype=int)
yvalues = np.array([], dtype=int)
with open("data/CA-GrQc.txt", 'rb') as f:
  for line in f:
    x,y = line.split()
    x = int(x)
    y = int(y)
    values = np.append(values,[[x,y]], axis=0)
    xvalues = np.append(xvalues,x)
    yvalues = np.append(yvalues,y)

with open("data/DGAddedCats.txt", 'w') as f:
  cat = 0
  x_init = xvalues[0]
  for x, y in zip(xvalues, yvalues):
    if x != x_init:
      if cat >= 6:
        cat = 0
      else:
        cat = cat + 1
      x_init = x
    f.write(str(x) + ' ' + str(y) + ' ' + str(cat) + '\n')
```

Appendix D: create_dataset.py

```
# create_dataset.py
# create dataset from canvas set for additional testing
# gonna pickle some things

import numpy as np
import random
import pickle as pkl
import networkx as nx
import scipy.sparse as sp
import collections
from scipy.sparse.linalg.eigen.arpack import eigsh
import sys

# read in dataset.txt
# pickle the dataset into subsets including .x, .tx, .allx as
scipy.sparse.csr.csr_matrix
# pickle the dataset into subsets including .y, .ty, .ally as numpy.ndarray
# also:
#  ind.dataset_str.graph => a dict in the format {index:
[index_of_neighbor_nodes]} as collections.defaultdict object;
#  ind.dataset_str.test.index => the indices of test instances in graph, for the
inductive setting as list object.
values = np.array([[0,0]])
xvalues = np.array([], dtype=int)
yvalues = np.array([], dtype=int)
zvalues = np.array([], dtype=int)
with open("data/DGAddedCats.txt", 'rb') as f:
  for line in f:
    x,y,z = line.split()
    x = int(x)
    y = int(y)
    z = int(z)
    values = np.append(values,[[x,y]], axis=0)
    xvalues = np.append(xvalues,x)
    yvalues = np.append(yvalues,y)
    zvalues = np.append(zvalues,z)
```

```
#delete first empty element
# index = [0]
# values = np.delete(values, 0,0)
# ones = np.ones([2500], dtype=int)

# choose between 150 and 300 indices per paper to make hot using the same random
seed
# to allow the GNN to make correlations between papers with similar vocabularies
#  -Arbitrarily choose the vocabulary to be 2500 words
xtot = np.empty((5242,2500))
numHot = random.randint(150, 300)
x_cur = xvalues[10]
i = 0
for x, z in zip(xvalues, zvalues):
  hot = np.zeros((2500))
  random.seed(z)
  if x != x_cur:
    x_cur = x
    for j in range(0, numHot):
      index = random.randint(0, 2499)
      hot[[index]] = 1
    xtot[[i]] = hot
    i = i + 1

x_train = sp.csr_matrix(xtot[0:140])
x_test = sp.csr_matrix(xtot[4242:5242])
x_all = sp.csr_matrix(xtot[0:4242])

# now get the y values in a numpy array
# first 100 entries are used as labeled training examples
# 4k to 5k are used as labeled testing examples
y_all = np.zeros(shape=(5242,7), dtype=int)
x_cur = xvalues[10]
i = 0
for x, z in zip(xvalues, zvalues):
  if x != x_cur:
    x_cur = x
    y_all[i,z] = 1
    i = i + 1
```

```
y_train = y_all[0:140]
y_test = y_all[4242:5242]
y_all = y_all[0:4242]

# now make .graph
# contains an entry for every unique 'document' and each of the 'papers' it cites

# lets us reference each document in the order of appearance in xvalues via
indexing
conversion = collections.defaultdict(list)
x_init = xvalues[10]
i = 0
for x in xvalues:
  if x_init != x:
    x_init = x
    conversion[x].append(i)
    i = i + 1

# now create graph with 'conversion'
d = collections.defaultdict(list)
d1 = collections.defaultdict(list)
x_init = xvalues[0]
i = 0
for x, y in zip(xvalues, yvalues):
  if x_init != x:
    x_init = x
    i = i + 1
  d[i].append(conversion[y][0])
  d1[i].append(y)

print(d[0])
print(d[1])

print(d1[0])
print(d1[1])
print('\nDG:')
print('\tlen graphdg: ',len(d))
print('\tx_test size: ', x_test.toarray().shape)
print('\tx_train size: ', x_train.toarray().shape)
```

```
print('\ty_test size: ', y_test.shape)
print('\ty_train size: ', y_train.shape)

# now make .index
index = random.sample(range(4242, 5242), 1000)

# pickle some things
with open("data/ind.canvas.x", 'wb') as f:
  pkl.dump(x_train, f)
  f.close()

with open("data/ind.canvas.y", 'wb') as f:
  pkl.dump(y_train, f)
  f.close()

with open("data/ind.canvas.tx", 'wb') as f:
  pkl.dump(x_test, f)
  f.close()

with open("data/ind.canvas.ty", 'wb') as f:
  pkl.dump(y_test, f)
  f.close()

with open("data/ind.canvas.allx", 'wb') as f:
  pkl.dump(x_all, f)
  f.close()

with open("data/ind.canvas.ally", 'wb') as f:
  pkl.dump(y_all, f)
  f.close()

with open("data/ind.canvas.graph", 'wb') as f:
  pkl.dump(d, f)
  f.close()

with open("data/ind.canvas.test.index", 'w') as f:
  for num in index:
    f.write(str(num) + '\n')
  f.close()
```

```
# load/unpickle cora/citeseer to see what it's like
names = ['x', 'y', 'tx', 'ty', 'allx', 'ally', 'graph']
objects = []
for i in range(len(names)):
  with open("data/ind.{}.{}".format("citeseer", names[i]), 'rb') as f:
    if sys.version_info > (3, 0):
      objects.append(pkl.load(f, encoding='latin1'))
    else:
      objects.append(pkl.load(f))

x, y, tx, ty, allx, ally, graph = tuple(objects)

print('\nCiteseer:')
print('\tsize graph: ', len(graph))
# print(graph)
print('\tsize tx: ', tx.toarray().shape)
print('\tsize x: ', x.toarray().shape)

# print('tx:\n', tx)
#print('ty:\n', ty)

print('\tsize ty: ', ty.shape)
print('\tsize y: ', y.shape)
# print(tx.toarray())

names = ['x', 'y', 'tx', 'ty', 'allx', 'ally', 'graph']
objects = []
for i in range(len(names)):
  with open("data/ind.{}.{}".format("cora", names[i]), 'rb') as f:
    if sys.version_info > (3, 0):
      objects.append(pkl.load(f, encoding='latin1'))
    else:
      objects.append(pkl.load(f))

x, y, tx, ty, allx, ally, graph = tuple(objects)

print('\nCora:')
# print(graph)
```

```
print('\tsize graph: ', len(graph))
print('\tsize tx: ', tx.toarray().shape)
print('\tsize x: ', x.toarray().shape)
print('\tsize ty: ', ty.shape)
print('\tsize y: ', y.shape)

names = ['x', 'y', 'tx', 'ty', 'allx', 'ally', 'graph']
objects = []
for i in range(len(names)):
  with open("data/ind.{}.{}".format("pubmed", names[i]), 'rb') as f:
    if sys.version_info > (3, 0):
      objects.append(pkl.load(f, encoding='latin1'))
    else:
      objects.append(pkl.load(f))

x, y, tx, ty, allx, ally, graph = tuple(objects)

print('\nPubMed:')
print('\tsize graph: ', len(graph))
print('\tsize tx: ', tx.toarray().shape)
print('\tsize x: ', x.toarray().shape)
print('\tsize ty: ', ty.shape)
print('\tsize y: ', y.shape)
```

Appendix E: modified attack.py

```
from __future__ import division
from __future__ import print_function

import time
import tensorflow as tf
import scipy.sparse as sp
import copy
import numpy as np
import matplotlib.pyplot as plt
import os
from sklearn.metrics import hamming_loss, f1_score
from utils import *
from models import GCN, MLP

# Set random seed
seed = 123
np.random.seed(seed)
tf.set_random_seed(seed)

# Settings
flags = tf.app.flags
FLAGS = flags.FLAGS
flags.DEFINE_string('model_dir', 'rob_canvas', 'saved model directory')
flags.DEFINE_string('dataset', 'canvas', 'Dataset string.') # 'cora', 'citeseer',
'pubmed'
flags.DEFINE_integer('steps', 100, 'Number of steps to attack')
flags.DEFINE_float('learning_rate', 0.001, 'Initial learning rate.')
flags.DEFINE_integer('hidden1', 64, 'Number of units in hidden layer 1.')
flags.DEFINE_float('dropout', 0., 'Dropout rate (1 - keep probability).')
flags.DEFINE_integer('early_stopping', 10, 'Tolerance for early stopping (# of
steps).')
flags.DEFINE_string('method', 'CW', 'attack method, PGD or CW')
flags.DEFINE_float('perturb_ratio', 0.05, 'perturb ratio of total edges.')
flags.DEFINE_float('weight_decay', 5e-4, 'Weight for L2 loss on embedding
matrix.')

# Load data
```

```
adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask =
load_data(FLAGS.dataset)
total_edges = adj.data.shape[0]
n_node = adj.shape[0]

# Some preprocessing
features = preprocess_features(features)
# for non sparse
features = sp.coo_matrix((features[1], (features[0][:, 0], features[0][:, 1])),
shape=features[2]).toarray()

support = preprocess_adj(adj)
# for non sparse
support = [sp.coo_matrix((support[1], (support[0][:, 0], support[0][:, 1])),
shape=support[2]).toarray()]
num_supports = 1
model_func = GCN

# Define placeholders
placeholders = {
  'lmd': tf.placeholder(tf.float32),
  'mu': tf.placeholder(tf.float32),
  's': [tf.placeholder(tf.float32, shape=(n_node, n_node)) for _ in
range(num_supports)],
  'adj': [tf.placeholder(tf.float32, shape=(n_node, n_node)) for _ in
range(num_supports)],
  'support': [tf.placeholder(tf.float32) for _ in range(num_supports)],
  'features': tf.placeholder(tf.float32, shape=features.shape),
  'labels': tf.placeholder(tf.float32, shape=(None, y_train.shape[1])),
  'labels_mask': tf.placeholder(tf.int32),
  'label_mask_expand': tf.placeholder(tf.float32),
  'dropout': tf.placeholder_with_default(0., shape=()),
  'num_features_nonzero': tf.placeholder(tf.int32) # helper variable for sparse
dropout
}

# Create model
# for non sparse
```

```
model = model_func(placeholders, input_dim=features.shape[1],
attack=FLAGS.method, logging=False)

# Initialize session
sess = tf.Session()


def hamming(features, support, labels, mask, placeholders):
  t_test = time.time()
  feed_dict_val = construct_feed_dict(features, support, labels, mask,
placeholders)
  feed_dict_val.update({placeholders['support'][i]: support[i] for i in
range(len(support))})
  outputs = []
  predictions = []
  pred = tf.arg_max(model.outputs, 1)
  pred = sess.run(pred, feed_dict=feed_dict_val)
  final = []
  for truth, x in zip(mask, pred):
    if truth == True:
      final.append(x)
  actual = []
  for row in labels:
    for i,x in enumerate(row):
      if x == 1:
        actual.append(i)
        break
  hloss = hamming_loss(actual, final)
  f1m = f1_score(actual, final, average='micro')
  f1ma = f1_score(actual, final, average='macro')
  return hloss, f1m, f1ma

# Define model evaluation function
def evaluate(features, support, labels, mask, placeholders):
  t_test = time.time()
  feed_dict_val = construct_feed_dict(features, support, labels, mask,
placeholders)
  feed_dict_val.update({placeholders['support'][i]: support[i] for i in
range(len(support))})
```

```
  outs_val = sess.run([model.attack_loss, model.accuracy],
feed_dict=feed_dict_val)
  return outs_val[0], outs_val[1], (time.time() - t_test)


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

model.load(FLAGS.model_dir, sess)
adj = adj.toarray()
lmd = 1
eps = total_edges * FLAGS.perturb_ratio
xi = 1e-5

label = y_train
label_mask = train_mask + test_mask

original_support = copy.deepcopy(support)
feed_dict = construct_feed_dict(features, support, label, label_mask,
placeholders)
feed_dict.update({placeholders['lmd']: lmd})
feed_dict.update({placeholders['dropout']: FLAGS.dropout})
feed_dict.update({placeholders['adj'][i]: adj for i in range(num_supports)})
# feed_dict.update({placeholders['s'][i]: np.random.uniform(size=(n_node,n_node))
for i in range(num_supports)})
feed_dict.update({placeholders['s'][i]: np.zeros([n_node, n_node]) for i in
range(num_supports)})


if FLAGS.method == 'CW':
  label_mask_expand = np.tile(label_mask, [label.shape[1],1]).transpose()
  feed_dict.update({placeholders['label_mask_expand']: label_mask_expand})
  C = 0.1
else:
  C = 200 # initial learning rate

if os.path.exists('label_' + FLAGS.dataset + '.npy'):
  label = np.load('label_' + FLAGS.dataset + '.npy')
else:
```

```
  ret = sess.run(model.outputs, feed_dict=feed_dict)
  ret = np.argmax(ret, 1)
  label = np.zeros_like(label)
  label[np.arange(label.shape[0]), ret] = 1
  np.save('label_' + FLAGS.dataset + '.npy', label)
feed_dict.update({placeholders['labels']: label})

print('{} attack begin:'.format(FLAGS.method))
for epoch in range(FLAGS.steps):

  t = time.time()
  # mu = C/np.sqrt(np.sqrt(epoch+1))
  mu = C / np.sqrt(epoch + 1)
  feed_dict.update({placeholders['mu']: mu})

  # s \in [0,1]
  if FLAGS.method == 'CW':
    a, support, l, g = sess.run([model.a, model.placeholders['support'],
model.loss, model.Sgrad],
                  feed_dict=feed_dict)
    # print('loss:', l)
  elif FLAGS.method == 'PGD':
    a, support, S, g = sess.run([model.a, model.placeholders['support'],
model.upper_S_real, model.Sgrad],
                  feed_dict=feed_dict)
  else:
    raise ValueError('invalid attack method: {}'.format(FLAGS.method))
  upper_S_update = bisection(a, eps, xi)

  feed_dict.update({placeholders['s'][i]: upper_S_update[i] for i in
range(num_supports)})

  upper_S_update_tmp = upper_S_update[:]
  if epoch == FLAGS.steps - 1:
    acc_record, support_record, p_ratio_record = [], [], []
    for i in range(20):
      print('random start!')
      randm = np.random.uniform(size=(n_node, n_node))
      upper_S_update = np.where(upper_S_update_tmp > randm, 1, 0)
```

```
      feed_dict.update({placeholders['s'][i]: upper_S_update[i] for i in
range(num_supports)})
      support = sess.run(model.placeholders['support'], feed_dict=feed_dict)
      cost, acc, duration = evaluate(features, support, y_test, test_mask,
placeholders)
      pr = np.count_nonzero(upper_S_update[0]) / total_edges
      if pr <= FLAGS.perturb_ratio:
        acc_record.append(acc)
        support_record.append(support)
        p_ratio_record.append(pr)
      print("Epoch:", '%04d' % (epoch + 1), "val_loss=", "{:.5f}".format(cost),
         "val_acc=", "{:.5f}".format(acc), "time=", "{:.5f}".format(time.time() -
t))
      print("perturb ratio", pr)
      print('random end!')
    # Validation
    support = support_record[np.argmin(np.array(acc_record))]
  cost, acc, duration = evaluate(features, support, y_test, test_mask,
placeholders)
  # Print results
  print("Epoch:", '%04d' % (epoch + 1), "val_loss=", "{:.5f}".format(cost),
     "val_acc=", "{:.5f}".format(acc), "time=", "{:.5f}".format(time.time() - t))

print("attack Finished!")
print("perturb ratio", np.count_nonzero(upper_S_update[0]) / total_edges)

# Testing after attack

test_cost, test_acc, test_duration = evaluate(features, support, y_train,
train_mask, placeholders)
hloss, f1, f1m = hamming(features, support, y_train, train_mask, placeholders)
print("Train set results:", "cost=", "{:.5f}".format(test_cost),
   "accuracy=", "{:.5f}".format(test_acc), "time=",
"{:.5f}".format(test_duration))
print("\tHamming Loss: ", "{:.5f}".format(hloss))
print("\tF1 Micro Score: ", "{:.5f}".format(f1))
print("\tF1 Macro Score: ", "{:.5f}".format(f1m))
```

```
test_cost, test_acc, test_duration = evaluate(features, support, y_val, val_mask,
placeholders)
hloss, f1, f1m = hamming(features, support, y_val, val_mask, placeholders)
print("Validation set results:", "cost=", "{:.5f}".format(test_cost),
   "accuracy=", "{:.5f}".format(test_acc), "time=",
"{:.5f}".format(test_duration))
print("\tHamming Loss: ", "{:.5f}".format(hloss))
print("\tF1 Micro Score: ", "{:.5f}".format(f1))
print("\tF1 Macro Score: ", "{:.5f}".format(f1m))

test_cost, test_acc, test_duration = evaluate(features, support, y_test,
test_mask, placeholders)
hloss, f1, f1m = hamming(features, support, y_test, test_mask, placeholders)
print("Test set results:", "cost=", "{:.5f}".format(test_cost),
   "accuracy=", "{:.5f}".format(test_acc), "time=",
"{:.5f}".format(test_duration))
print("\tHamming Loss: ", "{:.5f}".format(hloss))
print("\tF1 Micro Score: ", "{:.5f}".format(f1))
print("\tF1 Macro Score: ", "{:.5f}".format(f1m))

del sess
```