Τεχνολογίες Γραφημάτων
Εργασία
Πρόβλεψη Συνδέσμων(Link Prediction)

Ομάδα 13

Καλδάνης Θωμάς
Χατζηφωτιάδης Φώτης

Εισαγωγή

Σκοπός της εργασίας είναι η κατανόηση του προβλήματος της πρόβλεψης συνδέσμων (link prediction) σε γραφήματα μέσα από τρεις διαφορετικές προσεγγίσεις.

1. Κλασικές ευριστικές μέθοδοι.
2. Μια μέθοδος ρηχών προεκπαιδευμένων ενσωματώσεων κόμβων, Node2Vec στην περίπτωσή μας.
3. Ένα end-to-end GNN μοντέλο που προπονείται απευθείας για το έργο της πρόβλεψης συνδέσμων.

Πλαίσιο και Αξιολόγηση:

Στην εργασία μας χρησιμοποιούμε το **Cora Dataset**. Αφού φορτώσουμε το dataset, με την χρήση του **NetworkX** αφαιρούμε τα self-loops και τις πολλαπλές ακμές από το γράφημα και ελέγχουμε εάν είναι συνεκτικό ή όχι(διαπιστώνουμε πως δεν είναι).

```
Nodes: 2708
Edges: 5278
Connected:  False
```

Για τον λόγο αυτό, διατηρούμε μόνο το μεγαλύτερο συνεκτικό υποσύνολο του γραφήματος.

```
Nodes:  2485
Edges:  5069
Connected:  True
```

Για τον διαχωρισμό σε train/test, επιλέγουμε τυχαία το 10% των ακμών για το test set και το γράφημα που απομένει είναι το training graph. Η αφαίρεση των ακμών ελέγχεται επαναληπτικά ώστε το training graph να παραμείνει συνεκτικό. Κάθε φορά επιλέγουμε μια ακμή και την αφαιρούμε προσωρινά από το training graph. Εάν δεν χαλάει η συνεκτικότητα, η ακμή προστίθεται στο test set, διαφορετικά την ξαναβάζουμε στο training graph. Τέλος εμφανίζουμε τα αποτελέσματα της δειγματοληψίας και ελέγχουμε και πάλι αν είναι συνεκτικός ο γράφος μας.

```
Training edges:  4563
Positive test edges:  506
Training graph connected:  True
```

Για την αρνητική δειγματοληψία, επιλέγουμε τυχαία ζεύγη κόμβων και αν δεν συνδέονται στο αρχικό γράφημα τα κρατάμε, διαφορετικά επιλέγουμε νέα. Επιβεβαιώνουμε ότι το πλήθος των θετικών ακμών είναι ίσος με το πλήθος των αρνητικών.

```
Negative test edges:  506
```

ΜΕΡΟΣ Α:

Σαν μετρική αξιολόγησης, χρησιμοποιούμε την **AUC**, η οποία μετράει την ικανότητα ενός μοντέλου να ξεχωρίζει σωστά τις θετικές απο τις αρνητικές ακμές. Η συνάρτηση **compute_auc**, είναι αναγκαία μιας και ο αλγόριθμος δεν καταλαβαίνει από μόνος του τι είναι οι θετικές και οι αρνητικές ακμές, οπότε πρέπει εμείς να πούμε ότι οι θετικές ακμές αντιστοιχούν στο label 1 και οι αρνητικές στο label 0, και πάνω σε αυτό το σύνολο από 0 και 1 να υπολογίσουμε το **ROC-AUC**. Στη συνέχεια, υλοποιούμε τις συναρτήσεις για όλες τις ευρεστικές μεθόδους. Κάθε μέθοδος υπολογίζει με τον τρόπο της το τελικό σκορ των θετικών και αρνητικών ακμών.

- **Common Neighbors:** Χρησιμοποιεί τον αριθμό των κοινών γειτόνων των δύο κόμβων ώστε να υπολογίσει την πιθανότητα να υπάρχει ακμή μεταξύ τους.
- **Jaccard Coefficient:** Υπολογίζει τον λόγο των κοινών γειτόνων προς το σύνολο όλων των γειτόνων των δύο κόμβων.
- **Adamic-Adar Index:** Δίνει μεγαλύτερο βάρος στους κοινούς γείτονες που έχουν μικρότερο βαθμό θεωρώντας τους ως πιο σημαντικούς για την πρόβλεψη.

Τέλος αξιοποιούμε την συνάρτηση **compute_auc**, με τον τρόπο πού ήδη αναφέραμε ωστε να υπολογίσουμε την auc κάθε μεθόδου.

```
Common Neighbours AUC: 0.7508592541673828
Jaccard AUC: 0.747959271352466 1
Adamic Adar AUC: 0.7530894093018168
```

Βλέπουμε πως τα αποτελέσματα κυμαίνονται περίπου στο 0.75, κάτι που σημαίνει πως και τα τρία μοντέλα μπορούν να ταξινομήσουν σωστά περίπου το 75% των περιπτώσεων.

ΜΕΡΟΣ Β:

Για το δεύτερο μέρος της άσκησης χρησιμοποιήθηκε ένα shallow μοντέλο ενσωματώσεων κόμβων, το **Node2Vec**. Το **Node2Vec** εκπαιδεύτηκε πάνω στο training graph και παρήγαγε για κάθε κόμβο ένα **embedding 64 διαστάσεων.**

Οι παράμετροι που χρησιμοποιήθηκαν ήταν:

```
node2vec = Node2Vec(training_graph, dimensions=128, walk_length=80, num_walks=10, workers=multiprocessing.cpu_count())
```

- **dimensions**=128
- **walk_length**=80
- **num_walks**=10

Κατέληξα σε αυτές τις παραμέτρους καθώς και νωρίτερα είχα δοκιμάσει μοντέλα που έτρεχαν και για περισσότερη ώρα(υπερδιπλάσια), ωστόσο παρουσίαζαν overfitting και όχι μόνο δεν παρουσίαζαν συνεπή καλύτερα αποτελέσματα, αλλά μερικές φορές και χειρότερα.

Τα embeddings κόμβων χρησιμοποιήθηκαν για τη δημιουργία edge embeddings μέσω του **Hadamard** product:

```python
def hadamard(u, v, embeddings):
    return embeddings[u] * embeddings[v]
```

```python
def make_edge_dataset(pos_edges, neg_edges, embeddings):
    X = []
    y = []

    for u, v in pos_edges:
        X.append(hadamard(u, v, embeddings))
        y.append(1)

    for u, v in neg_edges:
        X.append(hadamard(u, v, embeddings))
        y.append(0)

    return np.array(X), np.array(y)
```

Το training και το test αποτελούνται από θετικές και αρνητικές ακμές με **Hadamard** embeddings.

Στη συνέχεια κατασκευάστηκε σύνολο εκπαίδευσης αποτελούμενο από **θετικά και αρνητικά edges**, το οποίο χρησιμοποιήθηκε για την εκπαίδευση ενός **MLP ταξινομητή** με:

- 1 κρυφό επίπεδο 64 νευρώνων
- max_iter = 300

Για την αξιολόγηση του μοντέλου χρησιμοποιήθηκε το **AUC** στο test set

```
Test AUC: 0.9225636238653938
```

Αποτέλεσμα που μας κάνει να συμπεράνουμε πως το **Node2Vec** σε συνδυασμό με ένα **MLP** ταξινομητή έχει πολύ καλύτερα αποτελέσματα από απλές heuristic μεθόδους.

ΜΕΡΟΣ Γ:

Η 4η άσκηση έγινε σε ξεχωριστό notebook για να είναι πιο καθαρή και ευδιάκριτη η χρήση των μεταβλητών και των συναρτήσεων.

Τα θετικά παραδείγματα αντιστοιχούν στις πραγματικές ακμές του γράφου, οι οποίες λαμβάνονται μέσω της συνάρτησης **g.edges()**. Τα αρνητικά παραδείγματα παράγονται δυναμικά με την συνάρτηση

dgl.sampling.global_uniform_negative_sampling η οποία επιστρέφει τυχαία ζεύγη κόμβων τα οποία δεν συνδέονται με ακμή.

Τα δεδομένα διαχωρίζονται σε 90/10, δηλαδή 90% training edges και 10% testing edges.

Ως encoder χρησιμοποιήθηκε **Graph Convolutional Network(GCN)** δύο επιπέδων, το οποίο χαρτογραφεί κάθε κόμβο σε ένα embedding 16 διαστάσεων:

```
class GCN(torch.nn.Module):
    def __init__(self, in_feats, h_feats):
        super(GCN, self).__init__()
        self.conv1 = GraphConv(in_feats, h_feats)
        self.conv2 = GraphConv(h_feats, h_feats)

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = torch.relu(h)
        h = self.conv2(g, h)
        return h
```

```
in_feats = train_g_dgl.ndata["feat"].shape[1]
h_feats = 16
model = GCN(in_feats, h_feats)
```

Για **Predictor** χρησιμοποιήσαμε τον **Dot Product** ο οποίος υλοποιήθηκε από την συνάρτηση **DotPredictor**:

```
class DotPredictor(torch.nn.Module):
    def forward(self, graph, h):

        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))

            return graph.edata['score']
```

```
pred = DotPredictor()
```

Στην προπόνηση χρησιμοποιήθηκε το **Binary Cross-Entropy loss,** ωστόσο χρησιμοποίησα την **BCEWithLogitsLoss()** γιατί εμφανιζόταν error:
RuntimeError: all elements of input should be between 0 and 1
το οποίο δημιουργείται από τις γραμμές

```
pos_score = score_edges(h, (pos_u, pos_v))
neg_score = score_edges(h, (train_neg_u, train_neg_v))

scores = torch.cat([pos_score, neg_score])
```

λόγω του ότι δεν επιστρέφουν δυαδικές τιμές. Επίσης το training έγινε:

- **epochs=100**
- **learning rate = 0.05**, γιατί είχε τα καλύτερα αποτελέσματα μετά από testing
- **Adam Optimizer**, για σταθερότερη σύγκλιση

Για την αξιολόγηση του μοντέλου χρησιμοποιήθηκε **AUC** στο test set:

```
pos_score = pred(test_pos_g, h)
neg_score = pred(test_neg_g, h)

auc = compute_auc(pos_score, neg_score)
print("GCN Link Prediction Test AUC:", auc)
```

```
from sklearn.metrics import roc_auc_score


def compute_auc(pos_scores, neg_scores):
    scores = torch.cat([pos_scores, neg_scores]).numpy()
    labels = np.concatenate([
        np.ones(len(pos_scores)), np.zeros(len(neg_scores))
    ])
    #print(labels)
    return roc_auc_score(labels, scores)
```

Τα αποτελέσματα κυμαίνονται γύρω ≈86%, γεγονός που σημαίνει ότι το μοντέλο εκπαιδεύεται αρκετά καλά και τα αποτελέσματα του είναι καλύτερα απο heuristics και αρκετά κοντά σε αυτά του **Node2vec** με **MLP Classifier.**

```
Epoch 0 | Loss: 0.6931 | TRAIN AUC: 0.7579
Epoch 5 | Loss: 0.6839 | TRAIN AUC: 0.7508
Epoch 10 | Loss: 0.6768 | TRAIN AUC: 0.7521
Epoch 15 | Loss: 0.6548 | TRAIN AUC: 0.7633
Epoch 20 | Loss: 0.6241 | TRAIN AUC: 0.7968
Epoch 25 | Loss: 0.5710 | TRAIN AUC: 0.8647
Epoch 30 | Loss: 0.5446 | TRAIN AUC: 0.8588
Epoch 35 | Loss: 0.5347 | TRAIN AUC: 0.8715
Epoch 40 | Loss: 0.5251 | TRAIN AUC: 0.8844
Epoch 45 | Loss: 0.5246 | TRAIN AUC: 0.8879
Epoch 50 | Loss: 0.5123 | TRAIN AUC: 0.8998
Epoch 55 | Loss: 0.5184 | TRAIN AUC: 0.8979
Epoch 60 | Loss: 0.5112 | TRAIN AUC: 0.9014
Epoch 65 | Loss: 0.5124 | TRAIN AUC: 0.9089
Epoch 70 | Loss: 0.5135 | TRAIN AUC: 0.9044
Epoch 75 | Loss: 0.5055 | TRAIN AUC: 0.9108
Epoch 80 | Loss: 0.4994 | TRAIN AUC: 0.9146
Epoch 85 | Loss: 0.4980 | TRAIN AUC: 0.9162
Epoch 90 | Loss: 0.5052 | TRAIN AUC: 0.9130
Epoch 95 | Loss: 0.5054 | TRAIN AUC: 0.9143
GCN Link Prediction Test AUC: 0.8596408337729711
```

Πίνακας σύγκρισης AUC

Common Neighbors	0.7508
Jaccard Coefficient	0.7479
Adamic-Adar Index	0.7530
Node2Vec with MLP CLASSIFIER	≈0.92
GCN with Dot Predictor & BCELoss	≈0.86

Παρατηρούμε ότι οι απλές ευριστικές μέθοδοι έχουν αρκετά καλά αποτελέσματα, δεν πλησιάζουν όμως την ποιότητα των επόμενων μεθόδων. Αυτό συμβαίνει γιατί οι ευριστικές μεθόδοι περιορίζονται στις τοπικές σχέσεις των κόμβων και δεν μπορούν να ανακαλύψουν σχέσεις μακρύτερες μεταξύ κόμβων.

Η μέθοδος Node2Vec σε συνδυασμό με MLP Classifier έχει την καλύτερη απόδοση καθώς έχει μεγαλύτερο βάθος στις αναζητήσεις του και λόγο του walk_length = 80 εξερευνά μεγαλύτερο μέρος του γράφου και καλύπτει πιο μακρινές σχέσεις μεταξύ των κόμβων, κάτι που εξηγεί και τις μικρές διακυμάνσεις στα αποτελέσματά του.

Τέλος το GCN παρουσιάζει ελαφρώς χειρότερα αποτελέσματα σε σχέση με το Node2Vec, κάτι που μπορεί να οφείλεται στην απλή αρχιτεκτονική του νευρωνικού δικτύου, εμφανίζει ευκολότερα διακυμάνσεις στην ακρίβειά του (απο 78% έως 94%) και με παραπάνω fine-tuning(πχ παραπάνω layers), πιθανότατα να ξεπερνούσε και το Node2Vec.