2025

Link Prediction

Γεώργιος Βλέτσας 4924 **Στέφανος Γκερς-Κουτσογιάννης 5046**

Team Name : out of cite, out of mind

Names in Kaggle: Stefanos Gersch-Koutsogiannis, Giorgos Vletsas

Περιεχόμενα

Εισαγωγή 2
Κατασκευή Train αρχείου 2
Εξαγωγή Χαρακτηριστικών από τον Γράφο 5
Εξαγωγή Χαρακτηριστικών από τα abstructs 8
Εξαγωγή Χαρακτηριστικών από τους authors 13
Τελική Εκπαίδευση και Πρόβλεψη 15

Εισαγωγή

Στην παρούσα εργασία θα παρουσιάσουμε τον πηγαίο κώδικα που υλοποιήσαμε για την επίλυση του προβλήματος του **Link Prediction**, με βάση τα δεδομένα του διαγωνισμού που διεξήχθη στην πλατφόρμα **Kaggle**. Η καλύτερη επίδοσή μας στον διαγωνισμό ήταν **0.10336**. Αρχικά, θα επεξηγήσουμε αναλυτικά τον κώδικα που οδήγησε στο παραπάνω αποτέλεσμα, εξηγώντας τη λογική και τις τεχνικές που αξιοποιήθηκαν.

Στη συνέχεια, θα αναλύσουμε και άλλες τεχνικές που δοκίμασαμε κατά τη διάρκεια του project, οι οποίες είτε δεν απέδωσαν ικανοποιητικά αποτελέσματα είτε αποδείχθηκαν ακατάλληλες για το συγκεκριμένο πρόβλημα.

Κατασκευή Train αρχείου

Για την εκπαίδευση του μοντέλου μας, δημιουργήθηκε ένα αρχείο **train.txt**, το οποίο περιέχει ζεύγη κόμβων που είτε συνδέονται μεταξύ τους (θετικά δείγματα label =1) είτε όχι (αρνητικά δείγματα label=0). Τα δεδομένα του training προέκυψαν από το πλήρες αρχείο ακμών **edgelist.txt**, αφού πρώτα **φιλτραρίστηκαν και αφαιρέθηκαν οι ακμές του test.txt,** έτσι ώστε να αποφευχθεί **data leakage**. Με αυτόν τον τρόπο διασφαλίζεται ότι ο γράφος εκπαίδευσης δεν περιέχει πληροφορία από το test.txt. Η υλοποίηση αυτή έγινε στο αρχείο **CreateTrainFile.py**

Στη συνέχεια δίνονται τα βήματα του κώδηκα της δημιουργίας του train.txt, με αντίστοιχη επεξήγηση.

```
# Διαβαζω τον πλήρη γράφο
edge_df = pd.read_csv("edgelist.txt", header=None, sep=",",
names=["source", "target"])
G_full = nx.from_pandas_edgelist(edge_df, source="source",
target="target")

# Διαβαζω τις ακμές του test set
test = pd.read_csv("test.txt", header=None, names=["source", "target"])
test_edges = set([tuple(x) for x in test.values])

#κρατάω όλους τους κόμβους, αφαιρώ μόνο τις ακμές του test set
G_train = nx.DiGraph()
G_train.add_nodes_from(G_full.nodes())
train_edges = [e for e in G_full.edges() if e not in test_edges]
G_train.add_edges_from(train_edges)
```

- Αρχικά φορτώνουμε το αρχείο edgelist.txt και κατασκευάζουμε τον πλήρη γράφο G_full.
- Στη συνέχεια, διαβάζουμε τις ακμές του test set από το αρχείο test.txt και τις αποθηκεύουμε ως σύνολο για γρήγορο έλεγχο.
- Για να αποφύγουμε data leakage, δημιουργούμε έναν νέο γράφο G_train που περιέχει όλους τους κόμβους του πλήρους γράφου, αλλά **μόνο τις ακμές που δεν βρίσκονται στο test.txt**. Έτσι εξασφαλίζουμε ότι το μοντέλο δεν θα εκπαιδευτεί πάνω σε ακμές που θα πρέπει να προβλέψει.

```
# Δημιουργία θετικών δειγμάτων
positive_samples = [(u, v, 1) for u, v in G_train.edges()]
```

- Όλες οι ακμές που υπάρχουν στον training γράφο θεωρούνται κόμβοι που συνδέονται.

```
# Δημιουργία αρνητικών δειγμάτων
nodes = list(G_train.nodes())
existing_edges = set(G_train.edges()) | set((v, u) for u, v in
G_train.edges())
num_neg = len(positive_samples)
negative_samples = set()

while len(negative_samples) < num_neg:
    u, v = random.sample(nodes, 2)
    if (u, v) not in existing_edges and (v, u) not in existing_edges:
        negative_samples.add((u, v))

negative_samples = [(u, v, 0) for u, v in negative_samples]
```

- Δημιουργούμε ισάριθμα θετικά και αρνητικά δείγματα, δηλαδή ένα τυχαίο ζεύγος κόμβων που **δεν συνδέονται** μεταξύ τους στον train γράφο. Με αυτό τον τρόπο το dataset γίνεται **ισορροπημένο**. Αυτά προκύπτουν τυχαία, με την εντολή random.sample(nodes, 2) που επιλέγει δύο διαφορετικούς κόμβους. Για κάθε τέτοιο ζεύγος, ελέγχουμε αν δεν υπάρχει ακμή μεταξύ τους και τότε το κρατάμε σαν αρνητικό.

```
# Συνδυασμός, ανακάτεμα και αποθήκευση
train_data = positive_samples + negative_samples
random.shuffle(train_data)

train = pd.DataFrame(train_data, columns=["source", "target", "label"])
train.to_csv("train.txt", index=False, header=False)

print("train.txt created")
```

- Τέλος, τα θετικά και αρνητικά δείγματα ανακατεύονται και αποθηκεύονται σε ένα αρχείο train.txt με μορφή "source", "target", "label" όπου label είναι 1 για τα θετικά και 0 για τα αρνητικά.

Εξαγωγή Χαρακτηριστικών από τον Γράφο

Σε αυτό το στάδιο, αξιοποιήσαμε τη δομή του γράφου για να εξάγουμε χρήσιμα features που μπορούν να χρησιμοποιηθούν για την πρόβλεψη. Αρχικά εφαρμόστηκε η τεχνική **Node2Vec** για την απόκτηση πυκνών embeddings των κόμβων έτσι ώστε να τα περάσουμε στο **GCN** στη συνέχεια για να εκπαιδευτεί και να ανακατασκευάσει τα Node2Vec embeddings, πράγμα που λειτουργεί ως ένα self-supervised objective. Τέλος, δημιουργήσαμε και παραδοσιακά γραφικά χαρακτηριστικά όπως **Jaccard Similarity**, **Common Neighbors**, **Adamic-Adar**, και **Preferential Attachment**.

```
#A: αξιοποιηση γράφου
train = pd.read_csv("train.txt", names=["source", "target", "label"])
train_edges = train[train["label"] == 1]
G_train = nx.from_pandas_edgelist(train_edges, source="source", target="target")

# Δημιουργία Node2Vec μοντέλου
if os.path.exists('node2vec_embeddings.npy') and os.path.exists('node2vec_node_id_map.npy'):
    print("Φόρτωση αποθηκευμένων Node2Vec embeddings")
    embedding_matrix = np.load('node2vec_embeddings.npy')
    node_id_map = np.load('node2vec_node_id_map.npy', allow_pickle=True).item()
else:
    print("Υπολογισμός Node2Vec embeddings")
    node2vec = Node2Vec(G_train, dimensions=64, walk_length=20, num_walks=100, workers=4)
    n2v_model = node2vec.fit(window=10, min_count=1)

    #Δημιουργεία λεξικου node_id_map που αντιστοιχίζει
    #κάθε κόμβο του γράφου σε έναν ακέραιο δείκτη.
    node_id_map = {node: idx for idx, node in enumerate(G_train.nodes())}
    # Δημιουργία πίνακα embeddings 64 διαστάσεων απο 0
    embedding_matrix = np.zeros((len(node_id_map), 64))
    # Γέμισμα του πίνακα με τα embeddings του Node2Vec
    for node, idx in node_id_map.items():
        embedding_matrix[idx] = n2v_model.wv[str(node)]

    np.save('node2vec_embeddings.npy', embedding_matrix)
    np.save('node2vec_node_id_map.npy', node_id_map)
```

- Σε αυτό το κομμάτι γίνεται η εξαγωγή embeddings από τον γραφο με την χρήση της Node2vec και λογο του ότι αργεί πάρα πολύ να τρέξει όλη η υλοποίηση αποθηκέυουμε τα embenddings την πρώτη φορά που θα υπολογιστούν και στην συνέχεια απλά τα κάνουμε load για να γλιτώσουμε χρόνο.

```
# Δημιουργία PyTorch Geometric Data object
features = torch.tensor(embedding_matrix, dtype=torch.float)
edge_index = torch.tensor(list(G_train.edges())).t().contiguous()

edge_index = edge_index.type(torch.long)
index_map = {node: i for i, node in enumerate(G_train.nodes())}
edge_index = edge_index.apply_(lambda x: index_map[x])

pyg_data = Data(x=features, edge_index=edge_index)

class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return x
# Εκπαίδευση GCN για να μάθει τα embeddings των κόμβων
model = GCN(in_channels=64, hidden_channels=32, out_channels=64)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
model.train()

prev_loss = float('inf')
for epoch in range(100000):
    optimizer.zero_grad()
    out = model(pyg_data.x, pyg_data.edge_index)
    loss = F.mse_loss(out, pyg_data.x)
    loss.backward()
    optimizer.step()

    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
    if epoch > 0 and loss.item() > prev_loss:
        print("stopping: loss increased.")
        break
    if (prev_loss - loss.item()) < 0.0001:
        print("stopping: loss below threshold.")
        break
    prev_loss = loss.item()
final_embeddings = out.detach().cpu().numpy()

def gcn_sim(u, v):
    i, j = node_id_map[u], node_id_map[v]
    return cosine_similarity([final_embeddings[i]], [final_embeddings[j]])[0][0]
```

- Στο παραπάνω κομμάτι κώδικα αρχικά μετατρέπουμε τα Node2Vec embeddings σε torch.tensor, ώστε να χρησιμοποιηθούν στο GCN. Το edge_index περιέχει τις ακμές του γράφου ως δείκτες. Μετατρέπουμε τις ακμές σε αριθμητικά. Στην συνέχεια ορίζουμε το GCN μοντέλο **class GCN(torch.nn.Module)** και το εκπαιδεύουμε με στόχο να να προσαρμόζει τα embeddings, με βάση τη δομή του γράφου δηλαδή ποιος κόμβος συνδέεται με ποιον. Το τελικό αποτέλεσμα είναι βελτιωμένα embeddings που περιέχουν τόσο τη γνώση από το Node2Vec όσο και πληροφορία από τη συνδεσιμότητα του γράφου, η εκπαίδευση σταματά είτε όταν το loss αρχίζει να αυξάνεται είτε όταν η βελτίωση είναι πάρα πολύ μικρή. Τέλος εξάγουμε τα τελικά embeddings από το GCN και αυτά είναι τα embeddings που θα χρησιμοποιηθούν για υπολογισμό στην συνάρτηση **gcn_sim(u, v)** η οποία εξάγει το cosine similarity μεταξύ 2 κόμβων.

```
def jaccard_sim(u, v):
    return next(jaccard_coefficient(G_train, [(u, v)]))[2]

def common_neighbors(u, v):
    return len(list(nx.common_neighbors(G_train, u, v)))

def adamic_adar(u, v):
    return list(nx.adamic_adar_index(G_train, [(u, v)]))[0][2]

def preferential_attachment(u, v):
    return list(nx.preferential_attachment(G_train, [(u, v)]))[0][2]
```

- Τέλος για να ενισχύσουμε το μοντέλο και να πάρουμε όσο ποιο πολύ πληροφορία μπορούμε από τον γράφο υπολογίσαμε και μερικά κλασικά similarities κόμβων, τα οποία βασίζονται στη δομή του γράφου. Αυτά είναι το **jaccard_sim** μετράει πόσο κοινή γειτονιά έχουν δύο κόμβοι, όσο περισσότεροι κοινοί γείτονες, τόσο μεγαλύτερο similarity, **common_neighbors** απλά μετράει πόσους κοινούς γείτονες έχουν οι κόμβοι, **adamic_adar** ίδιο με το common_neighbors απλά δίνει σημασία σε κόμβους που έχουν λίγες συνδέσεις διότι αν δύο κόμβοι έχουν κοινό γείτονα που δεν συνδέεται με πολλούς άλλους, αυτή η σύνδεση θεωρείται πιο σημαντική γενικότερα και τέλος **preferential_attachment** το οποίο μετράει αν δύο κόμβοι έχουν πολλούς γείτονες, και αν έχουν είναι πιο πιθανό να συνδεθούν.

Κατά τη διάρκεια της εργασίας επιχειρήθηκε επίσης η εφαρμογή των μοντέλων Node2Vec και GraphSAGE. Το Node2Vec, παρότι κατάφερε να παραγάγει embeddings, δεν βελτίωσε καθόλου την απόδοση του μοντέλου.

Αντίστοιχα, με το GraphSAGE, δεν καταφέραμε να το κάνουμε να τρέξει. Παρουσιάστηκαν πολλά τεχνικά ζητήματα κατά την εκτέλεση (σφάλματα σε, memory errors κατά βάση)

Εξαγωγή Χαρακτηριστικών από τα abstructs

Για το project μας, στοχεύσαμε στην εξαγωγή χαρακτηριστικών (**feature**) από τα επιστημονικά abstracts του dataset, χρησιμοποιώντας διάφορες τεχνικές μετατροπής κειμένου σε **embeddings**. Σκοπός μας ήταν να μετατρέψουμε κάθε abstract σε ένα διάνυσμα**.** Έπειτα από την εξαγωγή των embeddings, υπολογίσαμε το **cosine similarity** μεταξύ των vectors, δηλαδή τη γωνία που σχηματίζουν μεταξύ τους. Αυτό μας δείχνει πόσο κοντά νοηματικά είναι δύο abstracts μεταξύ τους. Όσο μεγαλύτερο το cosine similarity, τόσο πιο παρόμοια τα abstracts.

Για την εξαγωγή των embeddings, χρησιμοποιήσαμε και ποιο απλές κλασικές μεθόδους όπως (TF-IDF, Word2Vec), όσο και σύγχρονες μεθόδους deep learning, βασισμένες σε pretraining μοντέλα τα γνωστά **transformers models**.

1. TF-IDF

- Πιο συγκεκριμένα, κάθε λέξη παίρνει ένα βάρος που σχετίζεται με το πόσο συχνά εμφανίζεται στο κείμενο (TF), αλλά και με το πόσο μοναδική είναι σε σχέση με όλα τα κείμενα του dataset (IDF).
- Το αποτέλεσμα είναι ένα **αραιό διάνυσμα μεγάλων διαστάσεων** για κάθε abstract.

Χρησιμοποιήσαμε TfidfVectorizer από το Scikit-learn με max_features=5000 για να κρατήσουμε μόνο τις 5000 πιο σημαντικές λέξεις.

```
#tfidf
tfidf = TfidfVectorizer(max_features=5000)
tfidf_emb = tfidf.fit_transform(abstracts)
```

2. Word2Vec

- Εκπαιδεύσαμε το δικό μας Word2Vec μοντέλο πάνω στα abstracts, κατά την εκπαίδευση, το κάθε abstract καθαρίζεται και στη συνέχεια γίνεται σπάσιμο σε λέξεις.
- Για κάθε abstract βρίσκουμε τα embeddings όλων των λέξεών και υπολογίζουμε το μέσο όρο τους. Έτσι δημιουργούμε ένα ενιαίο διάνυσμα για κάθε abstract

```
# Word2Vec
if os.path.exists("word2vec.model"):
    print("Loading cached Word2Vec model")
    w2v_model = Word2Vec.load("word2vec.model")
else:
    print("Training Word2Vec model")
    tokenized_abstracts = [preprocess(doc) for doc in abstracts]
    w2v_model = Word2Vec(sentences=tokenized_abstracts, vector_size=300, window=5,
min_count=2, workers=4, epochs=30)
    w2v_model.save("word2vec.model")
    print("Saved Word2Vec model")

def get_w2v_vector(words, model):
    vectors = [model.wv[w] for w in words if w in model.wv]
    if len(vectors) == 0:
        return np.zeros(model.vector_size)
    return np.mean(vectors, axis=0)

if os.path.exists("word2vec_emb.npy"):
    print("Loading cached Word2Vec embeddings...")
    word2vec_emb = np.load("word2vec_emb.npy")
else:
    print("Computing Word2Vec abstract embeddings")
    word2vec_emb = np.array([get_w2v_vector(preprocess(doc), w2v_model) for doc in
abstracts])
    np.save("word2vec_emb.npy", word2vec_emb)
    print("Saved word2vec_emb.npy")
```

3. SBERT (all-MiniLM-L6-v2)

- Το SBERT είναι ένα μοντέλο βασισμένο σε transformers. Στο project μας, χρησιμοποιήσαμε την παραλλαγή all-MiniLM-L6-v2, η οποία είναι γρήγορη και ελαφριά.
- Το SBERT μετατρέπει κάθε abstract σε ένα διάνυσμα που συνοψίζει το νόημα του κειμένου. Τα αποτελέσματα αποθηκεύτηκαν τοπικά για να μη χρειάζεται επαναυπολογισμός η οποία είναι αρκετά χρονοβόρα διαδικασία.
- Το SBERT έχει ήδη προ εκπαιδευτεί σε μεγάλο όγκο δεδομένων για semantic tasks, κάτι που το κάνει πολύ αποδοτικό για σύγκριση κειμένων.

```
# SBERT
if os.path.exists("sbert_emb.npy"):
    print("Loading cached SBERT embeddings...")
    sbert_emb = np.load("sbert_emb.npy")
else:
    print("Computing SBERT embeddings...")
    sbert_model = SentenceTransformer('all-MiniLM-L6-v2')
    sbert_emb = sbert_model.encode(abstracts, show_progress_bar=True)
    np.save("sbert_emb.npy", sbert_emb)
    print("Saved SBERT embeddings to sbert_emb.npy")
```

4. SciBERT

- Το SciBERT είναι μια ειδική παραλλαγή του BERT, η οποία έχει προ εκπαιδευτεί πάνω σε επιστημονικά άρθρα. Αυτό το καθιστά ιδανικό για το dataset μας, αφού όλα τα abstracts είναι επιστημονικά.
- Για κάθε abstract, το SciBERT επιστρέφει αναπαραστάσεις λέξεων. Για να φτιάξουμε ένα συνολικό διάνυσμα ανά abstract υπολογίσαμε τον μέσο όρο όλων των token embeddings, αγνοώντας όμως τα padding tokens μέσω του attention mask.

```
# SciBERT
sci_tokenizer = AutoTokenizer.from_pretrained("allenai/scibert_scivocab_uncased")
sci_model =
AutoModel.from_pretrained("allenai/scibert_scivocab_uncased").to(device)

if os.path.exists("scibert_emb.npy"):
    print("Loading cached SciBERT embeddings")
    scibert_emb = np.load("scibert_emb.npy")
else:
    print("Computing SciBERT embeddings")
    scibert_emb = []
    batch_size = 16
    sci_model.eval()
    with torch.no_grad():
        for i in range(0, len(abstracts), batch_size):
            batch = abstracts[i:i+batch_size]
            inputs = sci_tokenizer(batch, padding=True, truncation=True,
return_tensors="pt", max_length=512).to(device)
            outputs = sci_model(**inputs)

            token_embeddings = outputs.last_hidden_state
            attention_mask = inputs['attention_mask'].unsqueeze(-
1).expand(token_embeddings.size()).float()

            sum_embeddings = torch.sum(token_embeddings * attention_mask, dim=1)
            sum_mask = torch.clamp(attention_mask.sum(dim=1), min=1e-9)
            mean_embeddings = sum_embeddings / sum_mask

            scibert_emb.append(mean_embeddings.cpu().numpy())

    scibert_emb = np.vstack(scibert_emb)
    np.save("scibert_emb.npy", scibert_emb)
    print("Saved SciBERT embeddings to scibert_emb.npy")
```

5. MPNet

- Το MPNet είναι ένα μοντέλο από τη σειρά των sentence-transformers. Είναι προ εκπαιδευμένο ώστε να προβλέπει λέξεις με βάση μετατεταγμένες εισόδους και είναι εξαιρετικό σε semantic similarity tasks.
- Στο project μας, χρησιμοποιήσαμε το προ εκπαιδευμένο μοντέλο all-mpnet-base-v2, το οποίο δέχεται ως είσοδο ένα abstract και επιστρέφει ένα embedding. Το μοντέλο είναι από τα πιο δυνατά που προσφέρει η πλατφόρμα sentence-transformers.

```
#MpNet
mpnet_model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')
mpnet_model.to(device)

if os.path.exists("mpnet_embeddings.npy"):
    print("Loading cached MPNet embeddings")
    mpnet_emb = np.load("mpnet_embeddings.npy")
else:
    print("Computing MPNet embeddings")
    mpnet_emb = mpnet_model.encode(abstracts, show_progress_bar=True)
    np.save("mpnet_embeddings.npy", mpnet_emb)
    print("Saved mpnet_embeddings.npy")
```

Τέλος είναι όλα τα similarities.

```
def sbert_sim(u, v):
    return cosine_similarity([sbert_emb[u]], [sbert_emb[v]])[0][0]

def tfidf_sim(u, v):
    return cosine_similarity(tfidf_emb[u], tfidf_emb[v])[0][0]

def scibert_sim(u, v):
    return cosine_similarity([scibert_emb[u]], [scibert_emb[v]])[0][0]

def mpnet_sim(u, v):
    return cosine_similarity([mpnet_emb[u]], [mpnet_emb[v]])[0][0]

def word2vec_sim(u, v):
    return cosine_similarity([word2vec_emb[u]], [word2vec_emb[v]])[0][0]
```

Κατά την εξέληξη του project, πειραματιστήκαμε και με άλλα μοντέλα, με στόχο τη βελτίωση της απόδοσης. Συγκεκριμένα, Doc2Vec, SPECTER και απλό BERT. Ωστόσο τα αποτελέσματα έδειξαν ότι δεν είναι χρήσιμα για το μοντέλο μας και πολλές φορές κάποιο από τα παραπάνω πρόσθετε θόρυβο στο τελικό score, με αποτέλεσμα να μην ενταχθούν στο τελικό pipeline.

Εξαγωγή Χαρακτηριστικών από τους authors

Για την αξιοποίηση των πληροφοριών συγγραφικής συνάφειας, κατασκευάστηκε ένα **δίκτυο (co-authorship graph)**, όπου κάθε κόμβος αντιπροσωπεύει έναν author και κάθε ακμή δηλώνει τη συνεργασία δύο authors στο ίδιο άρθρο, με τους συγγραφείς κάθε paper να συνδέονται μεταξύ τους πλήρως

Πάνω σε αυτόν τον γράφο εφαρμόστηκε ο αλγόριθμος **Node2Vec**, ώστε να εξαχθούν embeddings για κάθε συγγραφέα. Τα embeddings επιχειρούν να ενσωματώσουν πληροφορία δομής και ένωσης μεταξύ των συγγραφέων στο γράφο.

Στη συνέχεια, για κάθε paper, δημιουργήθηκε ένα **paper-level author embedding**, υπολογίζοντας τον μέσο όρο των embeddings των συγγραφέων του. Με αυτόν τον τρόπο, κάθε paper αποκτά ένα χαρακτηριστικό διάνυσμα που ενσωματώνει τη ταυτότητά του.

Τέλος, για κάθε ζεύγος papers, ορίστηκε ως χαρακτηριστικό το author cosine similarity μεταξύ των paper-level author embeddings.

```
with open("authors.txt", "r", encoding="utf-8") as f:
    author_lines = f.readlines()

paper_authors = []
for line in author_lines:
    authors = line.strip().split("|--|")[1].split(",")
    paper_authors.append([a.strip() for a in authors if a.strip()])

G_authors = nx.Graph()
for authors in paper_authors:
    for i in range(len(authors)):
        for j in range(i+1, len(authors)):
            G_authors.add_edge(authors[i], authors[j])

if os.path.exists("author_embeddings.pkl"):
    print("Loading cached author embeddings...")
    with open("author_embeddings.pkl", "rb") as f:
        author_embeddings = pickle.load(f)
else:
    print("Computing author embeddings")
    node2vec_auth = Node2Vec(G_authors, dimensions=64, walk_length=10,
num_walks=100, workers=4)
    model_auth = node2vec_auth.fit(window=10, min_count=1)

    author_embeddings = {a: model_auth.wv[a] for a in G_authors.nodes()}
    with open("author_embeddings.pkl", "wb") as f:
        pickle.dump(author_embeddings, f)
    print("Saved author embeddings")
```

```
#C1 Compute paper-level author embeddings
if os.path.exists("paper_author_emb.npy"):
    paper_author_emb = np.load("paper_author_emb.npy")
else:
    print("Computing paper-level author embeddings")
    paper_authors = []
    for line in author_lines:
        authors = line.strip().split("|--|")[1].split(",")
        authors = [a.strip() for a in authors if a.strip()]
        paper_authors.append(authors)

    paper_author_emb = []
    for authors in paper_authors:
        vecs = [author_embeddings[a] for a in authors if a in author_embeddings]
        if vecs:
            paper_author_emb.append(np.mean(vecs, axis=0))
        else:
            paper_author_emb.append(np.zeros(model_auth.vector_size))
    paper_author_emb = np.vstack(paper_author_emb)
    np.save("paper_author_emb.npy", paper_author_emb)
    print("Saved paper_author_emb.npy")

def author_sim(u, v):
    return cosine_similarity([paper_author_emb[u]], [paper_author_emb[v]])[0][0]
```

Τελική Εκπαίδευση και Πρόβλεψη

Στο τελικό στάδιο, για κάθε ζεύγος κόμβων, κατασκευάστηκε ένα διάνυσμα με fetures με βάση όλη την παραπάνω προεργασία που έγινε και ποιο συγκεκριμένα.

- **Author-based similarity**, βασισμένη στα embeddings του author γράφου.
- **Jaccard similarity**, **common neighbors**, **Adamic-Adar και Preferential Attachment.**
- **abstruct embeddings**, με cosine similarity από μοντέλα **SBERT**, **TF-IDF**, **SciBERT**, **MPNet**, **Word2Vec**.
- **GCN-based similarity**, μέσω embeddings που εξήχθησαν από Graph Convolutional Network.

Όλα τα παραπάνω χαρακτηριστικά κανονικοποιήθηκαν με **StandardScaler** και δόθηκαν ως είσοδος σε έναν τελικό classifier **XGBoost**, ο οποίος εκπαιδεύτηκε με στόχο τη βελτιστοποίηση της log-loss. Η διαδικασία πραγματοποιήθηκε με κατάλληλο tuning των υπερπαραμέτρων.

Μετά την εκπαίδευση στο σύνολο των train δεδομένον, το μοντέλο εφαρμόστηκε στο **test set** για την παραγωγή πιθανοτήτων ύπαρξης ακμής σε κάθε ζεύγος. Το τελικό αρχείο πρόβλεψης υποβλήθηκε στη μορφή submission.csv.

Για την επιλογή του τελικού classifier αξιολογήθηκαν διάφορα μοντέλα μηχανικής μάθησης, όπως:

- **MLP**
- **Random Forest**
- **Logistic Regression**
- **XGBoost**

Αρχικά εφαρμόστηκε **Cross-Validation** για να συγκριθεί η απόδοσή τους ως προς την **log loss**. Από αυτή την αρχική αξιολόγηση, τα δύο μοντέλα με την καλύτερη απόδοση ήταν:

- **XGBoost**
- **MLP** με log loss κοντά στην απόδοση του XGBoost

Κατόπιν, έγινε **Grid Search** με Cross-Validation **στα δύο μοντέλα (MLP και XGBoost)**, ώστε να εντοπιστούν οι βέλτιστες υπερπαράμετροι. Η διαδικασία ανέδειξε ως βέλτιστο ταξινομητή το **XGBoost**, με τις παρακάτω παραμέτρους:

- learning_rate = 0.05
- max_depth = 5
- n_estimators = 500

Το μοντέλο αυτό πέτυχε την **χαμηλότερη log loss** στο validation set και χρησιμοποιήθηκε στο τελικό pipeline για την παραγωγή των προβλέψεων του test set.

```
pairs = train[["source","target"]].values
labels = train["label"].values
filtered_pairs = [(u, v) for u, v in pairs if u in G_train and v in G_train]
filtered_labels = [labels[i] for i, (u, v) in enumerate(pairs) if u in G_train and
v in G_train]

#D training
def compute_features(filtered_pairs):
    X = []
    for u, v in filtered_pairs:
        feats = [
            author_sim(u, v),
            jaccard_sim(u, v),
            common_neighbors(u, v),
            adamic_adar(u, v),
            sbert_sim(u, v),
            tfidf_sim(u, v),
            scibert_sim(u, v),
            mpnet_sim(u, v),
            word2vec_sim(u, v),
            gcn_sim(u, v),
            preferential_attachment(u, v),

        ]
        X.append(feats)
    return np.array(X)

#Train
print("Computing train features...")
X_train = compute_features(filtered_pairs)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

final_model = XGBClassifier(
    learning_rate=0.05,
    max_depth=5,
    n_estimators=500,
    eval_metric='logloss',
    random_state=42
)

final_model.fit(X_train_scaled, filtered_labels)
train_probs = final_model.predict_proba(X_train_scaled)[:, 1]
```

```
train_logloss = log_loss(filtered_labels, train_probs)
print(f"Log Loss στο train set : {train_logloss:.5f}")

#Test prediction
print("Loading test data")
test_df = pd.read_csv("test.txt", header=None, names=["source", "target"])
test_pairs = test_df.values

print("Computing test features")
X_test = compute_features(test_pairs)
X_test_scaled = scaler.transform(X_test)
probs = final_model.predict_proba(X_test_scaled)[:, 1]

#Αποθήκευση submission
submission = test_df.copy()
submission["Label"] = probs
submission.reset_index(inplace=True)
submission = submission[["index", "Label"]]
submission.columns = ["ID", "Label"]
submission.to_csv("submission.csv", index=False)
print("saved submission.csv")
```

Παρακάτω παρουσιάζεται ενδεικτικά ο κώδικας που χρησιμοποιήθηκε για την επιλογή του βέλτιστου μοντέλου, ο οποίος δεν περιλαμβάνεται στο τελικό αρχείο υλοποίησης, καθώς εκεί χρησιμοποιείται απευθείας το μοντέλο με τις βέλτιστες υπερπαραμέτρους.

```
#param_grids = {
    #"MLP": {
    #    'hidden_layer_sizes': [(128, 32),(128,64,32), (128, 128,32), (256,)],
    #    'alpha': [0.001, 0.01],
    #    'max_iter': [400, 500],
    #},

    #"XGBoost": {
        #'n_estimators': [100, 200, 300, 500, 600, 700],
        #'max_depth': [3, 5, 7],
       # 'learning_rate': [0.1, 0.05, 0.01],
   # }
#}
```

```
# Ορισμός μοντέλων
#models = {
    #"MLP": MLPClassifier(random_state=42),
    # "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    # "Logistic Regression": LogisticRegression(max_iter=1000, random_state=42),
    #"XGBoost": XGBClassifier(eval_metric='logloss', random_state=42)
#}

# Cross-validation
"""
best_model = None
best_loss = float("inf")

print("Grid Search με Cross-Validation (Log Loss):")
for name in models:
    print(f"\n{name}:")
    grid = GridSearchCV(
        estimator=models[name],
        param_grid=param_grids[name],
        scoring='neg_log_loss',
        cv=5,
        n_jobs=-1,
        verbose=1
    )
    grid.fit(X_train_scaled, filtered_labels)

    loss = -grid.best_score_
    print(f"Best Log Loss = {loss:.5f}")
    print("Best Params:", grid.best_params_)

    if loss < best_loss:
        best_loss = loss
        best_model = (name, grid.best_estimator_)

# Τελικό μοντέλο
print(f"\nΚαλυτερο μοντελο: {best_model[0]} με Log Loss = {best_loss:.5f}")
"""
```