CS768-2025 Assignment Report

Utkarsh Pant (22B0914)
Anilesh Bansal (22B0928)
Chaitanya Garg (22B0979)
Harsh Kumar (22B0973)
Saatwik Patnaik (22B1284)

Github : Link

Task 1: Building a Citation Graph

Dataset Overview

We worked with a manually curated dataset of 6,545 research papers from NeurIPS and ICML. Each paper folder contained the title (`title.txt`), abstract (`abstract.txt`), and bibliographic information (`.bbl` and `.bib` (optional)).

Preprocessing

The script in `preprocess.ipynb` was responsible for:

- Iterating through the directories.
- Parsing bibliographies from `.bbl` and `.bib` files using regular expressions and BibTex parsers. Separate functions were created for parsing bib and bbl files becuase of the difference in their regex
- Extracting citation keys and matching them to paper titles within the dataset to establish edges between papers.

Only successfully resolved citations between papers in the dataset were considered for the citation graph.

The papers were matched using their 'cleaned_titles' which is basically the titles with all non alpha-numeric characters and multiple spaces removed.

Citation Graph Construction

A directed graph G is created using networkx.DiGraph(), where each node represents a paper, initialized with attributes like title and abstract. This choice of a directed graph reflects the directional nature of citations (one paper citing another).

Paper identifiers are derived from filenames, and titles are normalized (e.g., removing special characters, converting to uppercase) to create a mapping (title_to_paper) for matching cited papers.

Citation data is sourced from a JSON file (parsed_citations.json), suggesting preprocessing has already parsed the bibliography files (.bbl/.bib) to identify citation relationships. The add_edges

function processes this data, using a title_match function to link citing and cited papers, adding directed edges accordingly.

The structure of the json file is list of dicts like the one given below:

```
"paper_id": "0709.0928v1",
"citations": [ list of citations with the following structure
  {
    "key": "cited_paper_id",
    "author": "Archambeau et~al.\\/",
    "year": "2006",
    "title": "ROBUST PROBABILISTIC PROJECTIONS.",
    "source": "bbl"
  }, ...
```

This approach handles the challenge of noisy entries by relying on pre-parsed data.

Results

- **Number of Nodes (Papers)**: ∼6545
- **Number of Edges (Citations)**: *29398*
- **Number of Isolated Nodes**: *438* (nodes with degree 0)
- **Average In-Degree**: *4.49*
- **Average Out-Degree**: *4.49*
- **Graph Diameter**: *13* (computed using largest connected component)

Degree Distribution

A histogram of node degrees was plotted, revealing a **long-tailed distribution**, which is typical in citation networks.

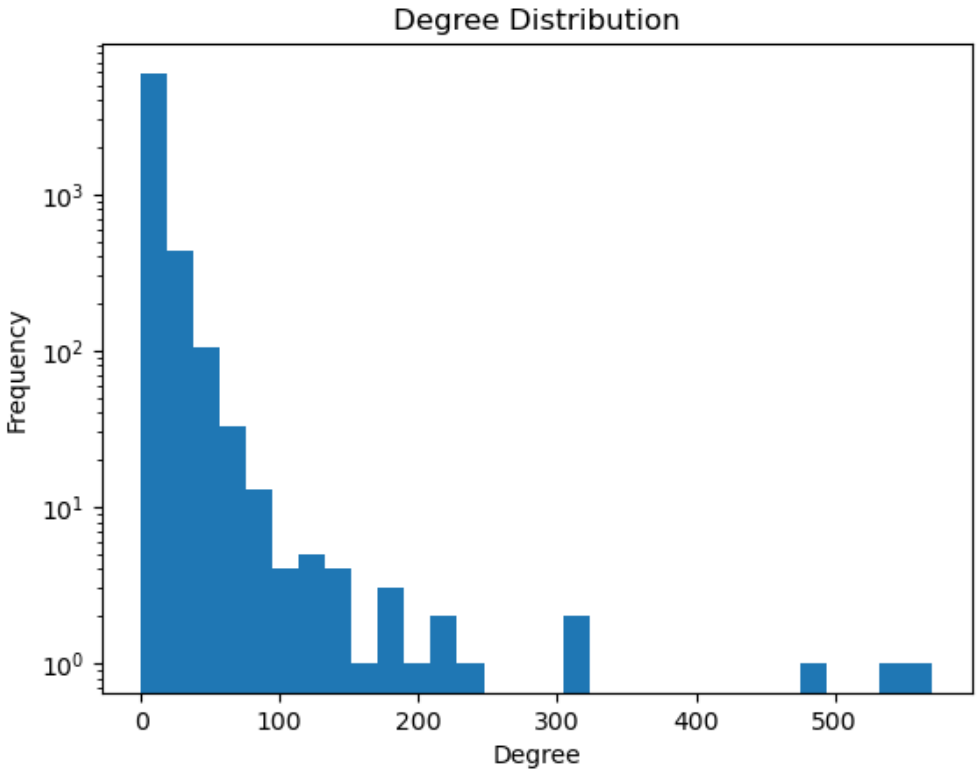

Figure 1: Degree distribution histogram of the citation network

Task 2: Link Prediction (Machine Learning)

The code's machine learning approach for link prediction is sophisticated, leveraging Graph Neural Networks (GNNs) and semantic embeddings, detailed as follows:

Feature Extraction

- The code employs the `sentence-transformers` library, specifically the `all-mpnet-base-v2` model, to generate 384-dimensional embeddings. These are created by encoding the concatenation of each paper's title and abstract, capturing semantic content for similarity analysis.

Data Splitting

- A random split is implemented by labelling nodes as test or train nodes. Then the adjacency matrix are re-computed relative to the testing and training graph by using the node0labels

Graph Neural Network (GNN) Model

- The chosen model is GraphSAGE, implemented using `torch_geometric.nn.SAGEConv`deal for predicting citations for new papers. Further therees a `nn.Sequential` model that's used for link prediction which takes the node embedding u and v of nodes, takes $u||v$ as input and outputs a single logit.

- The model learns node representations by aggregating features from neighbors, and for link prediction, it processes pairs of nodes. The link prediction module concatenates node embeddings and uses linear layers with ReLU activation to output a probability score for an edge.

Training the Model

- Training is conducted on a CPU device using the Adam optimizer with a learning rate of 0.01. Positive examples are the actual edges in the training graph, while negative examples are generated via the `negative_sampling` function, randomly selecting node pairs without edges.

- The main idea is to run the link predictor NN on all pairs of nodes and then take the sigmoid of the output. If the value is ¿ 0.5, it suggests that there exists a edge. Otherwise there doesnt exist an edge.

- The training loop shuffles positive edges, batches them, generates negative samples, and optimizes using binary cross-entropy loss with the predicted sigmoid, iterating over epochs to improve prediction accuracy.

- The negative sampling is done to train NN on both - when it should know an edge exists and when an edge doesnt exist.

Evaluation

- The evaluation focuses on *recall@K*, computed for K values like 5, 10, 20, 30, and 50. The `recall_at_k` function ranks predicted links for a query node and checks how many actual citations appear in the top K, using a dictionary (`citations`) to track actual citations.

- Performance is assessed on both training and test sets, plotting recall metrics to compare effectiveness. This aligns with the assignment's requirement to update `evaluation.py` for outputting top-K predictions, evaluated by `run_evaluation.py`.

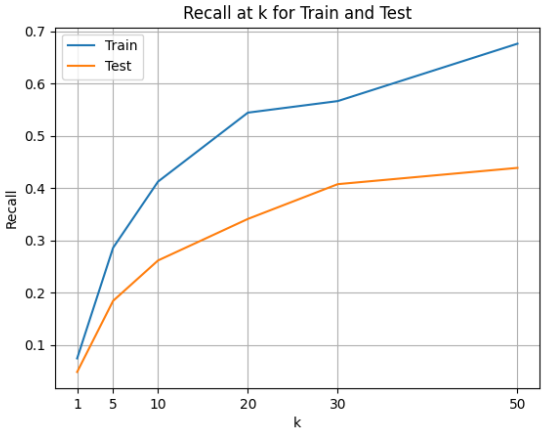

Figure 2: Recall Vs K for testing and training