Software Engineering Department
Braude College

Capstone Project Phase B – 61999

Citation networks evolution using Dynamic Network Embeddings

Maintenance Guide

25-1-R-16

Students:

Avishay Bar 316165984

Shay Bistrizky 206622086

Supervisors:

Prof. Zeev (Vladimir) Volkovich

Dr. Renata Avros

GIT Repository Link

Contents

1 Project Overview

1.1 Objective

The objective of this project is to develop a time-aware embedding framework for analysing the evolving structure and influence patterns within citation networks. Unlike traditional static graph analysis methods that treat the network as fixed, this project recognizes that the academic landscape is inherently dynamic: new papers are published each year, citation relationships change over time, and the influence of a given work may increase, decrease, or stabilize in response to ongoing research trends.

To address this, we implement the Dynamic Bernoulli Embedding (DBE) model, which learns a separate low-dimensional embedding for each paper at each time step (typically yearly). These embeddings are optimized to reflect both the local citation structure and the temporal continuity of the network. The result is a trajectory for each node in embedding space, which can be analysed to reveal how the paper's role and importance evolve over time.

The main goal is to use these trajectories to classify papers into dynamic behaviour archetypes, such as:

- Rising Stars: consistently increasing influence.
- Emerging Nodes: newly published papers with a sudden spike in attention.
- Steady Nodes: foundational works with stable long-term influence.
- Falling Stars: formerly influential papers that are losing relevance.

This classification enables deeper insights into the lifecycle of academic contributions, the dissemination of knowledge, and the temporal structure of scientific progress. It also supports applications such as trend analysis, research evaluation, and early detection of breakthrough works.

1.2 Technologies Used

To implement a scalable framework for modeling temporal citation networks, this project leverages a combination of widely adopted machine learning libraries, graph processing tools, and scientific computing frameworks. Each technology plays a specific role in supporting the full pipeline.

1.2.1 Execution Environment

The project is developed and executed on Google Colab, utilizing an NVIDIA A100 GPU for accelerated training of the embedding model. This environment ensures sufficient computational resources for handling large graphs and running multi-year dynamic simulations efficiently.

1.2.2 Core Libraries

- PyTorch: Used to implement and train the Dynamic Bernoulli Embedding (DBE) model. It provides efficient GPU-accelerated tensor operations.
- NumPy: Provides fast operations on multidimensional arrays, essential for computing distances and manipulating embeddings.
- Pandas: Used for managing tabular data, especially for storing and exporting node characterizations and training metrics.
- NetworkX: Handles graph construction and analysis. It is used to build and manipulate directed citation graphs from yearly data.

2 Data Preparation

This project operates on temporal citation data extracted from a collection of JSON files, where each file contains metadata for a batch of academic papers. Proper preparation of this input data is critical for building accurate and consistent dynamic citation graphs used throughout the embedding and classification pipeline.

The data is provided in .jsonl format, where each line is a valid JSON object representing a single academic paper. Each paper entry is expected to contain at least the following fields:

- `"id"` - a unique identifier for the paper (string or integer)
- `"year"` - the publication year of the paper (int)
- `"references"` - a list of paper IDs this paper cites (list of strings or integers)

Example (simplified):

```
{
  "id": "P12345",
  "year": 2009,
  "references": ["P111", "P222", "P333"]
}
```

The citation data used by this project is stored in a structured directory layout that organizes papers by their year of publication. Each file in the directory corresponds to a specific year and contains all the papers published in that year in a .jsonl format.

Here's what the directory might look like on disk:

```
/data
├── papers_2000.jsonl
├── papers_2001.jsonl
├── papers_2002.jsonl
├── ...
└── papers_2024.jsonl
```

3 Parameters and Folder Setup

This section defines the configuration interface and directory structure required to run the model. It enables flexibility and reproducibility by allowing users to specify all necessary parameters through command-line arguments or notebook cells (in the case of Google Colab).

3.1 Argument Parser

The project uses Python's argparse module to define all runtime parameters.

These include data paths, model and training hyperparameters, time span for analysis, and device selection. Importantly, the script uses `parse_known_args()` instead of `parse_args()` to avoid conflicts with Jupyter/Colab's internal flags such as `-f`.

This ensures smooth execution in notebook environments, where Colab automatically passes hidden arguments.

3.2 Supported CLI Arguments

The main parameters are as follows:

Parameter	Description	Default Value
`--data_dir`	Directory containing input JSON files	./data
`--output_dir`	Directory for saving all results	./outputs
`--start_year`	First year in the temporal sequence	1995
`--end_year`	Last year in the temporal sequence	2020
`--embedding_dim`	Dimensionality of node embeddings	64
`--walks_per_node`	Number of random walks per node	4
`--walk_length`	Length of each random walk	40
`--context_size`	Size of the context window	4
`--neg_samples`	Number of negative samples per positive pair	10
`--epochs`	Training epochs per year	5
`--batch_size`	Batch size for training	256
`--learning_rate`	Learning rate for optimizer	1e-3
`--validation_ratio`	Ratio of data used for validation	0.25
`--device`	"cuda" or "cpu" based on availability	auto-detect
`--resume`	Resume from last completed year	False (default)

These parameters offer full control over the modeling process and are typically adjusted when experimenting with different embedding strategies or dataset sizes.

3.3 Resume Flag

The `--resume` flag allows the pipeline to continue training from the last completed year. This is particularly useful in long-running jobs or environments like Colab where runtime can expire.

Internally, the script checks for the existence of output files for each year and skips reprocessing those already completed. This mechanism provides robustness and efficiency during retraining or re-runs.

4 Yearly Snapshots Construction

This component is responsible for constructing yearly snapshots, from the raw paper data stored in JSON Lines format. Each snapshot corresponds to a specific publication year and captures the citation relationships between papers as a directed graph.

These graphs serve as the foundation for all subsequent embedding and analysis stages.

The **`SnapshotBuilder`** class parses the input dataset year by year and incrementally builds a cumulative citation network. After processing each year:

- A subgraph snapshot is saved to disk (**`snapshots/G_<year>.pkl`**)
- Global ID mappings (**`id2idx`** and **`idx2id`**) are stored to maintain consistent indexing across all years.

This process ensures that node indices remain stable over time, which is critical for computing dynamic embeddings.

4.1 Internal Workflow

1. Initialize Mapping

   Each paper is assigned a unique integer index using two dictionaries:

   ```
   self.id2idx: Dict[str, int] = {}  # Maps paper ID → index
   self.idx2id: List[str] = []       # List of paper IDs by index
   ```

   These mappings enable consistent reference across years and are serialized to a JSON file (**`id_mappings.json`**).

2. Load Yearly Papers

   Each **`{year}.jsonl`** file is parsed line-by-line. For each paper a node is added for the paper and directed edged are created from the paper to all its referenced papers.

3. Save Graph Snapshot

   After processing all papers for the year, the current state of the graph is saved using Python's pickle module:

   ```
   with open(out_dir / f"G_{year}.pkl", "wb") as fh:
       pickle.dump(cumulative.copy(), fh)
   ```

4. Save ID Mappings

   At the end of the process, the final ID mappings are saved to:

   `outputs/snapshots/id_mappings.json` .This file ensures that node indices can be reused across steps like walks, embeddings, and evaluation.

5 Random-Walk Corpus Generation

To enable learning of temporal embeddings from graph structure, this stage generates or loads random walks over the citation network. These walks serve as input in a skip-gram style training setup, enabling the model to learn the local and global structure of each yearly graph snapshot.

This process includes walk generation, walk storage/loading, and negative sampling preparation for model training.

The purpose of the random-walk corpus generation stage is to simulate co-occurrence patterns between nodes in each yearly graph, enabling the learning of meaningful node embeddings. By performing random walks over the citation network, the model captures local and global structures, like word-context relationships in natural language.

At the heart of this stage is the `RandomWalkGenerator` class, which performs a specified number of random walks (`r`) of a given length (`L`) from every node in the graph. Each walk follows the outbound edges (via `G.successors()`), effectively simulating paths through citation chains. These walks capture both local and global structural properties of the citation network, and they are stored as NumPy arrays (`np.ndarray`) for downstream use in training. This approach mirrors techniques like DeepWalk, where co-occurrence patterns are learned from sequential paths in the graph.

To improve runtime efficiency and ensure reproducibility, the implementation supports resume-aware logic. For each year, before initiating a new walk generation, the script checks whether a corresponding file (`walks_<year>.npy`) already exists in the `walks/` directory. If the file exists, it is loaded directly using `np.load`. If not, the graph snapshot (`G_<year>.pkl`) is loaded from disk, walks are generated via the `RandomWalkGenerator`, and the resulting list is saved back to disk for future use. This logic avoids unnecessary recomputation and ensures consistent walk data across runs. Finally, all yearly walk lists are collected into a master list called `year_walks`, which serves as the training corpus for dynamic embedding learning in subsequent stages of the pipeline.

The `WalkDataset` class prepares the training pairs by converting random walks into positive and negative examples.

For each node in a walk:

- Positive pairs are formed with nodes within `context_size` on either side.
- Negative pairs are randomly sampled from the rest of the graph.

This avoids the need for full pairwise computations and makes training efficient.

6 Training

This stage is the heart of the Dynamic Bernoulli Embedding (DBE) system. It defines how the model is constructed, how it is trained year by year, and how previously trained states are managed during resume scenarios.

6.1 DBE Model

The **`DynamicBernoulliEmbedding`** class implements the core neural architecture of the system. It is designed to learn time-aware node embeddings across a sequence of yearly citation networks. The model maintains a tensor of shape **`[T, N, D]`**, where **`T`** is the number of years, **`N`** is the number of nodes (papers), and **`D`** is the embedding dimensionality. Each slice **`embeddings[t]`** captures the vector representation of all nodes in year **`t`**.

Additionally, the model learns a shared transformation matrix $\alpha \in \mathbb{R}^{DxD}$ that encodes how embeddings interact for link prediction. The model is trained to distinguish between positive edges (real context pairs from walks) and negative edges (randomly sampled nodes) using a binary cross-entropy loss on the dot product between embeddings.

To ensure embeddings evolve smoothly over time, a temporal regularization term penalizes large deviations between a node's embedding at year `t` and its value in year **`t-1`**. Another regularization term applies L^2 penalty on the α matrix to avoid overfitting. This combination of objectives ensures that the model not only predicts edges accurately but also learns stable and interpretable trajectories of node evolution.

The model's **`forward()`** method takes a batch of **`(target, context, label)`** pairs and returns either the loss or the raw logits, depending on the mode. It supports training per year while sharing the α matrix across all years.

6.2 Training Manager

The **`DBETrainer`** class encapsulates the training loop, handling optimization, early stopping, validation, and logging. It is initialized with the DBE model and training parameters, including learning rate, batch size, number of epochs, early stopping patience, and the computation device (**`cpu`** or **`cuda`**).

Training is done year-by-year. For each year, the trainer receives random-walk data, splits it into training and validation subsets, and constructs a **`WalkDataset`** to generate training pairs. It uses PyTorch's **`DataLoader`** to create minibatches and performs standard forward-backward optimization with the Adam optimizer.

Validation is performed after every epoch by evaluating the model on the held-out walks. The trainer tracks both training and validation loss and accuracy. Early stopping is triggered if the validation loss fails to improve after a given number of epochs (**`patience`**). The best model weights for that year are restored before continuing.

In addition to optimization, the trainer periodically logs node-level statistics—such as the L2 norm of each node's embedding—by appending them to a CSV file. These logs provide insight into how node influence changes during training.

After training each year, the trainer saves the yearly embedding slice **`M_<year>.pt`** and the updated **`alpha.pt`** matrix. This design supports the **`--resume`** flag, which allows training to continue from the last completed year without redoing past work. The resume logic automatically detects completed years by scanning for existing `.pt` files.

When all years are trained, the trainer consolidates all metric logs into a DataFrame and saves it as a **`.parquet`** file, ensuring reproducibility and enabling further offline analysis. Overall, the **`DBETrainer`** abstracts the complexity of dynamic model training while offering robustness and transparency.

7 Node Classification

After training is complete and the embeddings for all years have been generated, the `NodeCharacterizer` class performs post-training analysis to classify the dynamic behavior of nodes (i.e., academic papers) over time. This classification aims to understand how the influence of each paper evolves throughout the temporal citation network.

The goal is to label every node with a semantic category reflecting its temporal role in the network. This allows the identification of influential patterns in citation dynamics, which can support downstream analytics, visualization, or research insights.

The four semantic labels used are:

- Rising Star: A node whose influence is growing steadily and consistently over time.
- Emerging: A node that has recently experienced a sharp spike in influence.
- Falling Star: A node that was once influential but is now declining in prominence.
- Steady: A node that remains stable in influence and connectivity over time.

Two core metrics are computed for each node:

1. Velocity (vel):
   Year-to-year movement of a node in embedding space. For node i at time t, it is:

   $$\text{velocity}_i(t) = \| v_i(t+1) - v_i(t) \|$$

   This metric captures how much the embedding changes from year to year — often corresponding to changes in the paper's influence or connections.

2. Drift (drift):
   Distance of each node's embedding from the centroid (mean embedding vector) at each year.

   $$\text{drift}_i(t) = \| v_i(t) - \mu(t) \|$$

   where μ(t) is the centroid of all node embeddings at year t. Drift captures how "central" or "peripheral" a paper is in the influence landscape.

These metrics are standardized across all nodes to obtain global mean (μ) and standard deviation (σ) for velocity and drift, which are used as thresholds.

Then, for each node n, the following rules are applied in order:

- Rising Star:
  If the velocity of node n exceeds (μ + 2σ) for two or more consecutive years, the node is considered a Rising Star. This indicates consistent growth in connectivity and influence.
- Emerging:
  If the node has a single spike of velocity above (μ + 2σ), it is labeled as Emerging. This marks the sudden rise of new research.
- Falling Star:
  If the node's drift from the centroid increases by at least 0.5σ over time, and this drift

generally increases (i.e., does not decline much), it is labeled as Falling Star. These are papers that are becoming less central and gradually lose citations.

- Steady:
  Nodes not meeting any of the above conditions are considered Steady. These maintain a stable position and act as core reference points in the network.

Note:

The characterization step only runs if the final embedding slice for the last year (`M_<end_year>.pt`) is found in the embeddings folder. If training hasn't completed, this step will automatically skip, preventing premature classification.