Semantic Data Management

Graph Embeddings - Final Project

Emmanuel F. Werr
emmanuel.werr@estudiantat.upc.edu

Agustina Sofia Martinez
agustina.martinez@estudiantat.upc.edu

Facultat d'Informàtica de Barcelona
Master of Data Science

June 15, 2023

Research Aspect - Graph Embedding

Graph Embedding also known as Network Embedding deals with representing complex objects such as texts, images or graphs into a low-dimensional space in which the aim is preserve the most significant information about the objects while reducing the dimensionality of the non-relational data compared to the original dataset, where a similarity graph is constructed using the pairwise feature similarity, connected nodes closer to each other.

It addresses the problems of graph analytics and representation learning by providing techniques to transform graph data into vector representations that facilitate analysis, modeling, and capturing the essential features of the graph elements. On the one hand, graph analytics aims to mine useful information from graph data. On the other hand, representation learning obtains data representations that make it easier to extract useful information when building classifiers or other predictors. Is it important to mention that it does not require the learned representations to be low dimensional.

There are different types of graphs:

- **Homogeneous Graph:** $G_{homo} = (V, E)$ is a graph in which $|Tv| = |Te| = 1$. All nodes in G belong to a single type and all edges belong to one single type.
- **Heterogeneous Graph:** $G_{hete} = (V, E)$ is a graph in which $|Tv| > 1 and/or |Te| > 1$
- **Knowledge Graph:** $G_{known} = (V, E)$ is a directed graph whose nodes are entities and edges are subject-property-object triple facts. Each edge of the form (head entity, relation, tail entity) (denoted as $< h, r, t >$) indicates a relationship of r from entity h to entity t.

In terms of proximity measures commonly adopted to quantify the graph property preserved in the embedded space depending on: First-order proximity focuses on capturing the local pairwise similarity between nodes that are directly connected by edges. It measures the strength of the direct connection between pairs of nodes. And also the second-order proximity refers to the measure of similarity between nodes that are indirectly connected in a graph. It considers the relationships that involve traversing multiple edges and nodes to establish a connection, capturing these indirect relationships. The more similar two nodes neighbourhoods are, the larger the second order proximity value between them.

Problems Settings of Graph Embedding

The problem mainly consists of setting the embedding input which is a graph and the output which is a set of low dimensional vector(s) representing part of a graph.

Next, the four types of **Input** will be introduced.

- **Homogeneous Graph:** Nodes and Edges belong to a single type respectively. It can be categorized as "directed" where Nodes connected by higher-weighted edges are embedded closer to each other (e.g, social network graph) or it can be "undirected" here all nodes and edges are treat equally, as only the basic structural information of the input graph is available.

- **Heterogeneous Graph:** which mainly exist in three scenarios: Community-based Question Answering, Multimedia Networks and Knowledge graphs.

- **Graph with Auxiliary Information:** Contains auxiliary information of a node/ edge/whole-graph in addition to the structural relations of nodes such as Label, Attribute, Node Feature, Information Propagation and Knowledge Base.

- **Graph Constructed from Non- relational Data:** Usually happens when the input data is assumed to lie in a low dimensional manifold, it has the challenge to determine how to compute the relations between non-relational data and the construct such a graph, and how to preserve the node proximity of the constructed graph.

Categorized according to the granularity of the **output**, graph embedding yields four distinct categories including:

- **Node Embedding:** Represents each node as a vector in a low dimensional space. Nodes that are "close" in the graph are embedded to have similar vector representations. The challenge mainly come from defining the node proximity in the input graph.

- **Edge Embedding:**The aim is to represent an edge as a low-dimensional vector. Particularly it is useful in two scenarios: Knowledge Graph Embedding to predict a missing entity/relation given two other components, and embeds a node pair as a vector feature to either make the node pair comparable to other nodes or predict the existence of a link between two nodes.

- **Hybrid Embedding:** Combination of different types of graph components. The main challenge is generate the kind of embedding target structure that is not given.

- **Whole-Graph Embedding:** Commonly used for small graphs. Is represented as one vector and two similar graphs are embedded to be closer. It benefits the graph classification task by providing a straightforward and efficient solution for calculating graph similarities.

Graph Embedding Techniques

Graph embedding techniques generally aim to represent a graph in a lower-dimensional space while preserves crucial graph property information. The differences between different graph embedding algorithms lie in how they define the similarities of different graph components and how to preserve them in the embedded space.

Next, will be introduced the insight of each graph embedding techniques such us: Matrix Factorization (Includes: Graph Laplacian Eigenmaps and Node Proximity Matrix Factorization), Deep Learning (Includes: DL with Random Walk, Without Random Walk), Edge Reconstruction based Optimization (Includes: Maximizing Edge Reconstruction Probability, Minimizing Distance-based, Minimizing Margin-based Ranking Loss), Graph Kernel (Includes: Graphlet, Subtree Patterns, Random Walks) and Generative Model (Includes: Embed Graph into the Latent Semantic Space, Incorporate Latent Semantics for Graph Embedding).

We have decided to delve deeper into some of the commonly used graph embedding techniques that have shown promising results.

DeepWalk

DeepWalk uses random walks to generate embeddings for nodes in a graph. The method basically consists of three steps:

- **Sampling:** A graph is sampled using random walks, where a few random walks are performed from each node. It is sufficient to perform between 32 and 64 random walks from each node. However, good random walks typically have a length of around 40 steps.
- **Training skip-gram:** The skip-gram network takes a node from the random walk as a one-hot vector input and maximizes the probability of predicting neighboring nodes. Typically, it is trained to predict around 20 neighbor nodes, with 10 nodes on the left and 10 nodes on the right.
- **Computing embeddings:** Embedding is the output of a hidden layer of the network. DeepWalk computes embedding for each node in the graph.

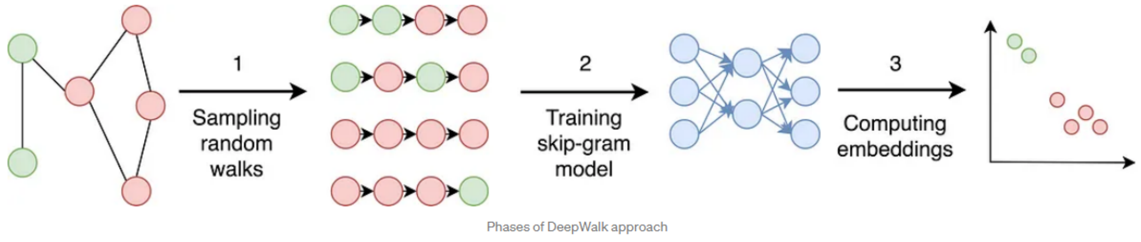


Figure 1: DeepWalk Diagram

The DeepWalk method performs random walks randomly, which means that the embeddings may not effectively preserve the local neighborhood of each node.

Generative Model

Can be defined by specifying the joint distribution of the input features and the class labels, conditioned on a set of parameters. These models learn to generate new graph instances that resemble the original graph distribution. By leveraging the generative process, the learned representations capture the underlying graph structure and can be used for various downstream tasks.

Matrix Factorization

Matrix factorization methods, such as Singular Value Decomposition (SVD) or Non-

Negative Matrix Factorization (NMF), decompose the graph's adjacency matrix into lower-dimensional matrices. The resulting matrix representations capture the latent features and relationships among nodes, allowing for efficient analysis and computation.

Edge Reconstruction based Optimization
This technique focuses on reconstructing the graph's edges in the embedding space. It formulates the embedding task as an optimization problem, aiming to minimize the difference between the original graph's adjacency matrix and the reconstructed adjacency matrix derived from the embeddings. By optimizing the embeddings to preserve the graph's edge structure, this approach generates informative representations.

Graph2vec
It uses the skip-gram network and gets an ID of the document on the input. It is trained to maximize the probability of predicting random words from the document.

Graph2vec approaches consist of three steps:

- **Sampling and relabeling all subgraphs from the graph:** A sub graph is a set of nodes that appear around the selected node, with nodes in the sub graph not being further than the specified number of edges away.
- **Training the skip-gram model:** Graphs are similar to documents, as documents are sets of words, while graphs are sets of subgraphs. During this phase, the skip-gram model is trained to maximize the probability of predicting subgraphs that exist in the input graph. The input graph is provided as a one-hot vector
- **Computing embeddings:** with providing a graph ID as a one-hot vector at the input. Embedding is the result of the hidden layer.

Since the task involves predicting subgraphs, graphs with similar subgraphs and comparable structures will have similar embeddings.

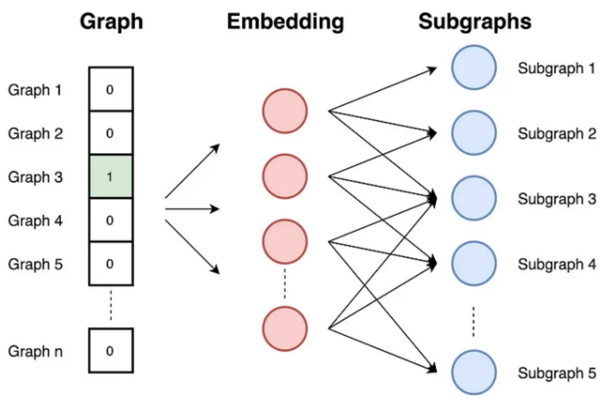


Figure 2: Graph2Vec Diagram

Applications

Graph embedding benefits a wide variety of graph analytics applications as the vector representations can be processed efficiently in both time and space.

- **Node Related Applications:**
  - **Node Classification:** To assign a class label to each node in a graph based on the rules learnt from the labelled nodes. Intuitively similar nodes have the same labels and it is conducted by appying a classifier on the set of labelled node embedding for training.
  - **Node Clustering:** Aims to group similar nodes together, so that nodes in the same group are more similar to each other than those in other groups.
  - **Node Recommendation/ Retrieval/Ranking:** Recommend top K nodes of interest to a given node based on certain criteria such as similarity.
- **Edge Related Applications:**
  - **Link Prediction:** Its output vectors can also help infer the graph structure.
  - **Triple Classification:** It aims to classify whether an unseen triplet $< h, r, t >$ is correct or not.
- **Graph Related Applications:**
  - **Graph Classification:** It assigns a class label to a whole graph. This is important when the graph is the unit of data.
  - **Visualization:** It generates visualizations of a graph on a low dimensional space. Usually all nodes are embedded as 2D vectors and then plotted ina 2D space with different colours indicating nodes categories.

Practical Section

For this section the aim was to keep it simple while using one of the Neo4j provided graphs. We decided to go with the popular Movies Recommendation graph because it is simple enough to play around with these concepts, but still large and rich enough to produce meaningful results. The graph contains information about movies, genres, actors, directors, users (reviewers), and movie reviews by users. More info can be found at the official Neo4j-Graph-Examples GitHub Repo.

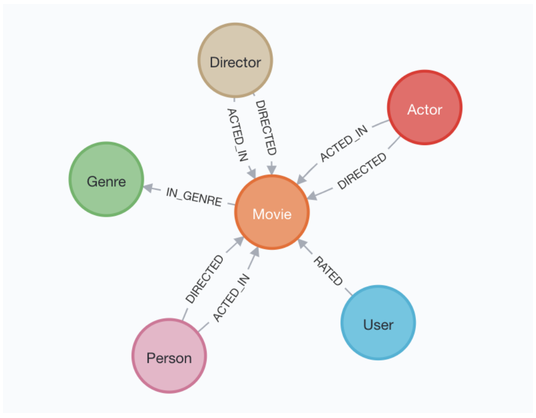
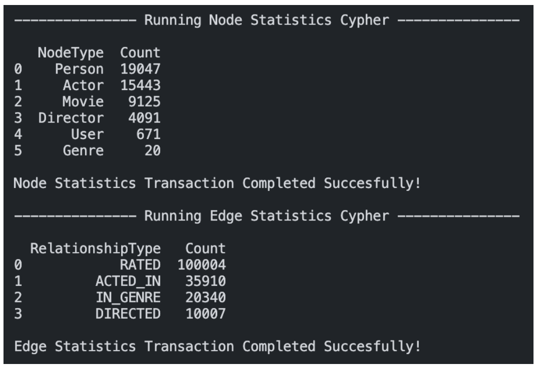


Figure 3: Movie Recommendation Graph Database Schema

Below are some node and edge statistics computed using Cypher and the Neo4j python client. As a reminder, all of the code used for our implementation of this recommender can be found either as screenshots in this report or inside the 'project.py' script attached with our submission.



```
-------------- Running Node Statistics Cypher --------------

   NodeType  Count
0    Person  19047
1     Actor  15443
2     Movie   9125
3  Director   4091
4      User    671
5     Genre     20

Node Statistics Transaction Completed Succesfully!

-------------- Running Edge Statistics Cypher --------------

  RelationshipType   Count
0            RATED  100004
1         ACTED_IN   35910
2         IN_GENRE   20340
3         DIRECTED   10007

Edge Statistics Transaction Completed Succesfully!
```

Figure 4: Node & Edge Distribution Statistics

The goal is to recommend to a user, movies to watch that other similar users have seen but they have not. The main steps involved in enabling this functionality are the following.

- Inspect the original graph to determine the nodes and relationships that will provide the most meaningful context for our embeddings.
- Project a new sub-graph containing the nodes and relationships identified in the previous step.
- Compute the embeddings and tune the hyperparameters.

- Compute similarities between nodes. A sensible option would be to use cosine similarity. But since Neo4j does not provide a built-in method for this, we will use KNN within Graph Data Science and achieve similar results.
- Recommend movies to a user that their most similar other users have seen but they have not.

The embeddings will represent a myriad of features (related to nodes and their connections) in high-dimensional vector-space and allow us to provide high-quality movie recommendations. We will implement all of the above steps inside Neo4j using Cypher and the Graph Data Science plugin.

Node2Vec vs. FastRP

After selecting the most meaningful nodes and relationships to create a new graph projection, we experimented with both Node2Vec and FastRP embeddings from the Neo4j Graph Data Science library. Both techniques transform the complex topological information from our graph into high-dimensional vector representations for each node. Node2Vec generates embeddings by performing random walks on the graph, while FastRP uses a much simpler and faster method based on random projections. However, the two techniques present different trade-offs.

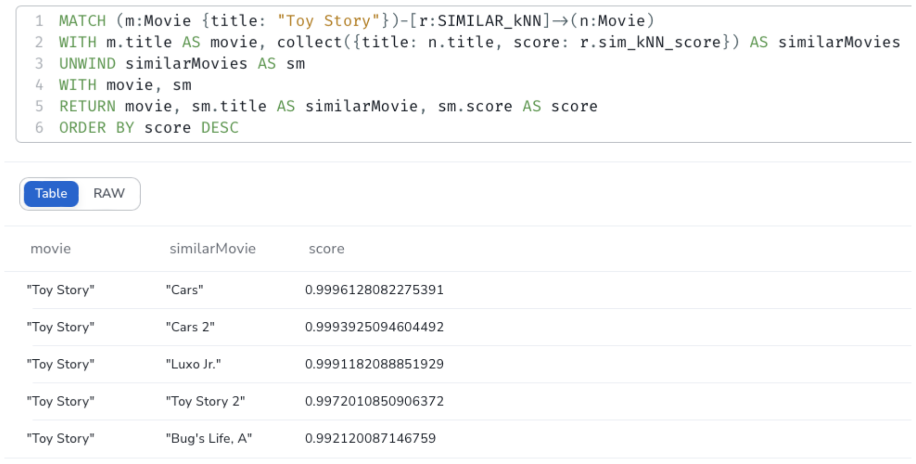
Node2Vec tends to produce more informative embeddings, as it captures local (specific neighborhoods) and global (broader structure) graph properties. This results in a robust understanding of the graph's topological context. However, Node2Vec is computationally intensive due to the performing of many random walks, which makes it less scalable for larger graphs. FastRP, on the other hand, stands out for its computational efficiency and scalability. It computes embeddings in linear time, making it a more suitable choice for very large graphs. However, FastRP provides less refined embeddings compared to Node2Vec, as it does not capture the neighborhood context in the same granular manner.

After testing both techniques, we opted to implement Node2Vec for our movie recommendation system. This decision was primarily influenced by the superior quality of the embeddings generated by Node2Vec. Even though Node2Vec is more resource-intensive than FastRP, our use case does not involve an exceedingly large graph, making the computational costs manageable. Most importantly, we believe Node2Vec's ability to capture both local and global properties of the graph will prove advantageous for the recommendation task. Moving forward, we believe the best approach would be to fine-tune our Node2Vec implementation to ensure more efficient and personalized movie recommendations to users.

Embedding Similarity Results

Since we are calculating node embeddings for all our nodes, we can not only identify similar users, but also similar movies. After our embeddings have been calculated and we have computed similarity using KNN with Grad Data Science, we can use some simple

Cypher within Neo4j Aura to test if indeed they are working properly. Below are some test queries with the goal to find similar movies.

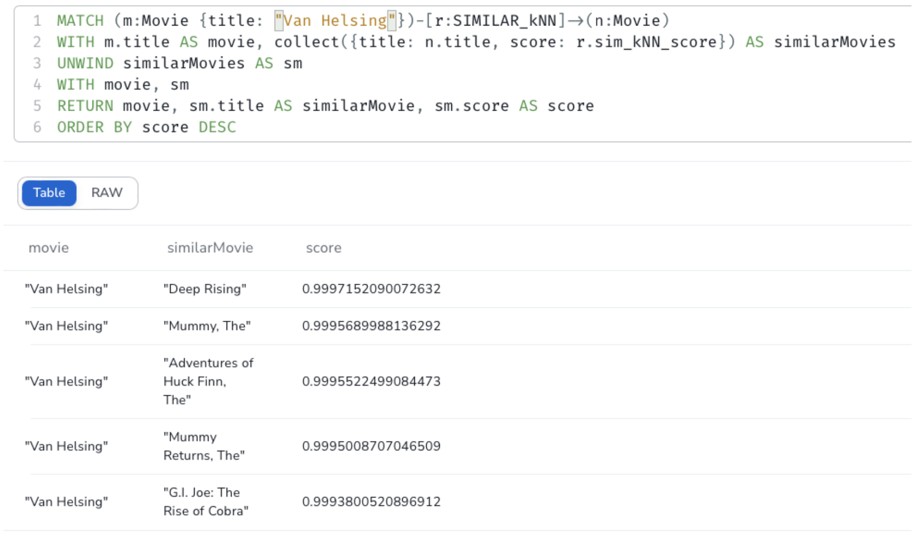


```
MATCH (m:Movie {title: "Toy Story"})-[r:SIMILAR_kNN]→(n:Movie)
WITH m.title AS movie, collect({title: n.title, score: r.sim_kNN_score}) AS similarMovies
UNWIND similarMovies AS sm
WITH movie, sm
RETURN movie, sm.title AS similarMovie, sm.score AS score
ORDER BY score DESC
```

movie	similarMovie	score
"Toy Story"	"Cars"	0.9996128082275391
"Toy Story"	"Cars 2"	0.9993925094604492
"Toy Story"	"Luxo Jr."	0.9991182088851929
"Toy Story"	"Toy Story 2"	0.9972010850906372
"Toy Story"	"Bug's Life, A"	0.992120087146759

Figure 5: Cypher Query Output of Most Similar Movies to Toy Story

We can infer from the results above that the embeddings are doing a good job at capturing node similarity. According to our query, the most similar movies to ”Toy Story” are all children's animations by Pixar. By contrast, the query below features a different kind of movie entirely in ”Van Helsing”, and the most similar movies to that one appear to be Sci-fi action-adventure movies with some gore/horror involved.



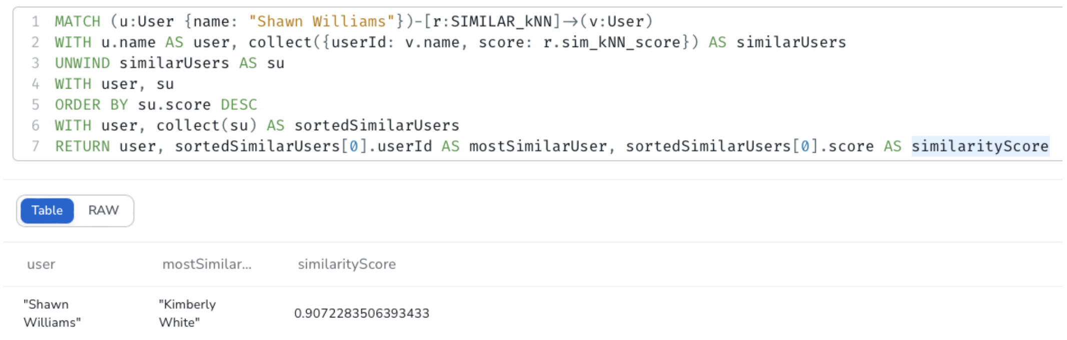
```
MATCH (m:Movie {title: "Van Helsing"})-[r:SIMILAR_kNN]→(n:Movie)
WITH m.title AS movie, collect({title: n.title, score: r.sim_kNN_score}) AS similarMovies
UNWIND similarMovies AS sm
WITH movie, sm
RETURN movie, sm.title AS similarMovie, sm.score AS score
ORDER BY score DESC
```

movie	similarMovie	score
"Van Helsing"	"Deep Rising"	0.9997152090072632
"Van Helsing"	"Mummy, The"	0.9995689988136292
"Van Helsing"	"Adventures of Huck Finn, The"	0.9995522499084473
"Van Helsing"	"Mummy Returns, The"	0.9995008707046509
"Van Helsing"	"G.I. Joe: The Rise of Cobra"	0.9993800520896912

Figure 6: Cypher Query Output of Most Similar Movies to Toy Story

Movie Recommendations

Now we can use Cypher within Neo4j to implement the functionality of identifying similar users to each other and recommend them relevant unseen movies. We will walkthrough an example of this by selecting a random user from our graph, finding their most similar user, and recommending the most relevant movies to watch. The Cypher query and output below show how we can identify user 'Shawn Williams' and their most similar user 'Kimberly White'.

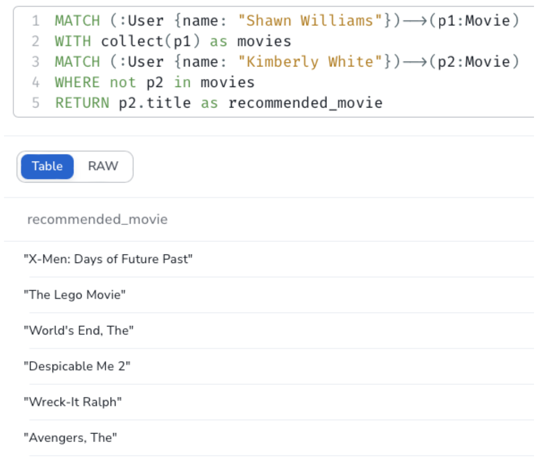


```
MATCH (u:User {name: "Shawn Williams"})-[r:SIMILAR_kNN]→(v:User)
WITH u.name AS user, collect({userId: v.name, score: r.sim_kNN_score}) AS similarUsers
UNWIND similarUsers AS su
WITH user, su
ORDER BY su.score DESC
WITH user, collect(su) AS sortedSimilarUsers
RETURN user, sortedSimilarUsers[0].userId AS mostSimilarUser, sortedSimilarUsers[0].score AS similarityScore
```

user	mostSimilar...	similarityScore
"Shawn Williams"	"Kimberly White"	0.9072283506393433

Figure 7: Identified Most Similar User to "Shawn Williams"

Then we can execute a new Cypher query to determine the movies that "Kimberly White" has seen but "Shawn White" has not. Our assumption is that these movies will likely be of interest to Shawn given his shared similarities with Kimberly. These all seem like wonderful movie choices for Shawn to watch.



```
MATCH (:User {name: "Shawn Williams"})⟶(p1:Movie)
WITH collect(p1) as movies
MATCH (:User {name: "Kimberly White"})⟶(p2:Movie)
WHERE not p2 in movies
RETURN p2.title as recommended_movie
```

recommended_movie
"X-Men: Days of Future Past"
"The Lego Movie"
"World's End, The"
"Despicable Me 2"
"Wreck-It Ralph"
"Avengers, The"

Figure 8: Top Recommended Movies for "Shawn Williams"

Future Work

There are several areas of development that would further improve this recommender. First, a comparative study could be conducted with different embedding techniques such as GraphSAGE, GNN, and Graph Convolutional Networks. Additionally, personalizing these algorithms by incorporating more user-specific information like age, location, or previously liked genres, would certainly yield more tailored recommendations.

Secondly, exploring hybrid approaches using both collaborative filtering as well as content-based recommendations would likely also yield better results. This was merely a simple implementation that explored what was possible with Graph Embeddings within Neo4j, and we are excited to continue learning about these methods for years to come.