MODEL RE-TRAINING FOR DYNAMIC GRAPHS

By

VARUN TEJA PURAM

Bachelor of Technology

Mahatma Gandhi Institute of Technology

Hyderabad, Telangana

2019

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 2022

MODEL RE-TRAINING FOR DYNAMIC GRAPHS

Thesis Approved:

ACKNOWLEDGEMENTS

The Master's degree from the computer science department, Oklahoma State University has greatly increased my interest in research and has given me great knowledge. I will continue to do more research in the upcoming years.

Firstly, I am very grateful and thankful to Dr. Johnson P Thomas, who has given me constant support and without him, I would not be able to do this.

I am also very thankful to Dr. K.M George who has encouraged, supported, and given important ideas to me.

Thank you, Dr. Mayfield, for being my committee member.

I would also like to thank my parents, Mr. Ashok Rao Puram and Mrs. Shoba Rani for their continued support throughout my studies.

Name: VARUN TEJA PURAM

Date of Degree: JULY 2022

Title of Study: MODEL RE-TRAINING FOR DYNAMIC GRAPHS

Major Field: Computer Science

Abstract: In Machine Learning, the most critical assumption is that training and testing datasets should have similar distributions. The model will be effective if the new test data is similar to the past data on which the model was trained. If there are substantial differences between the training data and the testing data, the machine learning algorithm will generate results that are not very accurate. In many applications, the data has *dynamic periodicity*, that is, the data changes with time. As the distribution of the data keeps changing, at some point, the model will therefore have to be retrained.

In this research I look at the dynamic behavior of graph data. As data changes, there will be addition/deletions of nodes/edges of the graph. As we are dealing with large sets of graph data, we use embedding vector spaces (for graph data) for training and testing. Embedding vector spaces in each timestamp are different and training the model each time when data changes is expensive. To address these challenges, we use the *dfs_dynode2vec* algorithm where the current timestamp graph embedding vectors initializes from the previous embedding vectors. For each timestamp, data might change significantly or insignificantly. We propose a statistical model '*Significant testing*' which determines whether the model should be retrained or not. If the change is insignificant, the model need not to be trained again and embedded vectors for that timestamp are not generated. We have considered several aspects in determining the statistical significance of the change. These include edge centrality, betweenness centrality and norm calculations.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE SURVEY 4

III. INTRODUCTION TO MACHINE LEARNING ON GRAPHS 8

Formal Definition of Graphs...8
Node Degree 9
Node Centrality...9
Node Embedding 14
Random Walks on Graphs 15
Random walk-based node embedding Technique 17
Matrix Norms...20

IV. STATISTICAL APPROACH TO RETRAINING GRAPH DATA 22

Problem Specification...22
Proposed Methodology 23

V. RESULTS 31

Dataset...31
Experiments 32

VI. CONCLUSION...42

REFERENCES. ……………………………………………………………… …..44

LIST OF TABLES

Table Page

5.1 Results when 10% high centrality edges were removed....................................35

5.2 Results when 20% leaf nodes were removed...35

5.3 Results when 10% leaf nodes and 2% high nodes were removed 36

5.4 Results for values for norms for different accuracy values (experiment -4) 38

5.5 Results for values for norms for different accuracy values (experiment -5) 40

LIST OF FIGURES

Figure Page

2.1 Continues traning-continuos testing 5

3.1 Node centrality example 11

3.2. Node centrality example continued 12

3.3 Node centrality example results...12

3.4 Edge centrality example...14

3.5 Node embedding example..15

3.6 Random walk example...16

3.7 Skip gram architecture 19

5.1 Graph as embedding vector space -1 33

5.2 Graph as embedding vector space -2 34

5.3 High edge centralities values 38

5.4 Accuracy vs Norm values 41

CHAPTER I

INTRODUCTION

Data equals Knowledge. Data is everywhere and plays a very important part in day-to-day life. The term ‘small data’, for example, phone numbers of family, refers to small datasets which humans can remember and analyze. ‘Big data’ refers to large datasets that may be structured or unstructured and requires a machine to store and analyze. Data can be in any form like image, video, text, numeric or bits.

Data may be represented as graphs to capture relationships between data and attributes of data. Graphs are ubiquitous data structures composed of nodes and edges, that is a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Nodes represent entities and edges establish the relationship between them. Many real-world applications are best modelled with graphs. As an example, airlines graphs consist of airports as nodes and edges as the distance between airports. Both nodes and edges can represent attributes. For example, a node may specify the name of the airport and whether it is an international airport or not. There are many algorithms on graphs like traversal, shortest path, cycle detection and so on. Lately machine learning algorithms can run on graph data.

Machine Learning algorithms on graphs include node prediction, link prediction, anomaly prediction and so on. A machine learning model takes graph data in terms of vectors for training and testing. When big data is represented in the form of graphs, *graph embeddings* are required. Graph embedding is a node embedding technique that embeds the nodes by preserving the graph structure and its properties into low dimensional vectors that can be viewed as projections in the latent space. This reduces the dimensionality of the graph which improves the time required for the machine learning algorithms to process the data.

There are several node embeddings techniques like matrix factorization approaches which use Laplacian eigenmaps to reduce the dimensionality and random walk-based approaches like Node2vec [14] which uses Gradient descent to optimize the random walks and Deep Walk [15] which uses skip gram to preserve the neighbor structure of the graph.

Data has dynamic periodicity as data changes with time. Incoming data may have associated with it a timestamp. The critical assumption of machine learning is that training and testing data should have similar distributions to get good results. With the dynamic behavior of data, over time, the training data may not have a similar distribution to the testing data resulting in decreasing accuracy. Hence, this means the model has to be retrained. Model retraining for each timestamp is costly in terms of time and computation. For big data, retraining the machine learning model may takes weeks. Hence, retraining should be done only when the accuracy is below a threshold. This will remove the need to retrain with every change in data. When data changes significantly the existing model may be retained.

In my research I used the random walk-based approach [14] for embedding the nodes. Node embedding for each timestamp will be different as we will get a different corpse of walks from random walks. To address this problem, Dynode2vec [15] is an algorithm that uses previous timestamps vectors for current timestamp vectors (which may produce similar vectors to the previous timestamp). I modify this algorithm by taking advantage of DFS (Depth First Search) traversal for the evolving nodes. In this thesis I also identify the key attributes of a graph using a statistical approach to determine if the model must be retrained. I will also define the bounds within which changes to the attributes will determine if the model must be retrained.

Previous work reported in the literature is presented in chapter 2. There is a brief introduction on Machine Learning on Graphs in chapter 3. Problem Statement and proposed methodology is presented in chapter 4, results obtained are described in chapter 5 and conclusions in chapter 6.

CHAPTER II

LITERATURE SURVEY

Machine learning algorithms require a lot of data, so-called ‘big data’. However as more and more data are appended to the dataset (the so-called ‘volume’, ‘velocity’ and ‘variety’ properties of big data), the accuracy of the machine learning algorithms may change as the distribution of the data may change. To maintain the required level of accuracy, the machine learning algorithm must be retrained with the updated dataset.

The key research question is: ‘as data changes, at what point should the model be retrained?’ Very little work has been reported in the literature on how the accuracy of machine learning algorithms change as data changes.

The Automated Retraining of Machine Learning Models [1] looked at this problem. They proposed the methodology shown below:

```
Algorithm1(Data)

   1) Gather data and create data set
   2) Train Machine Learning Model
   3) Evaluations [ Predictions]
            if (Prediction becomes inaccurate):
   4)          Model Retraining
   5)          Go to step 3
           else:
```

```
The Model is retained.
```

In this work, they run the model every time they need to predict to measure the accuracy each time. If the accuracy falls below a threshold the model is retrained.

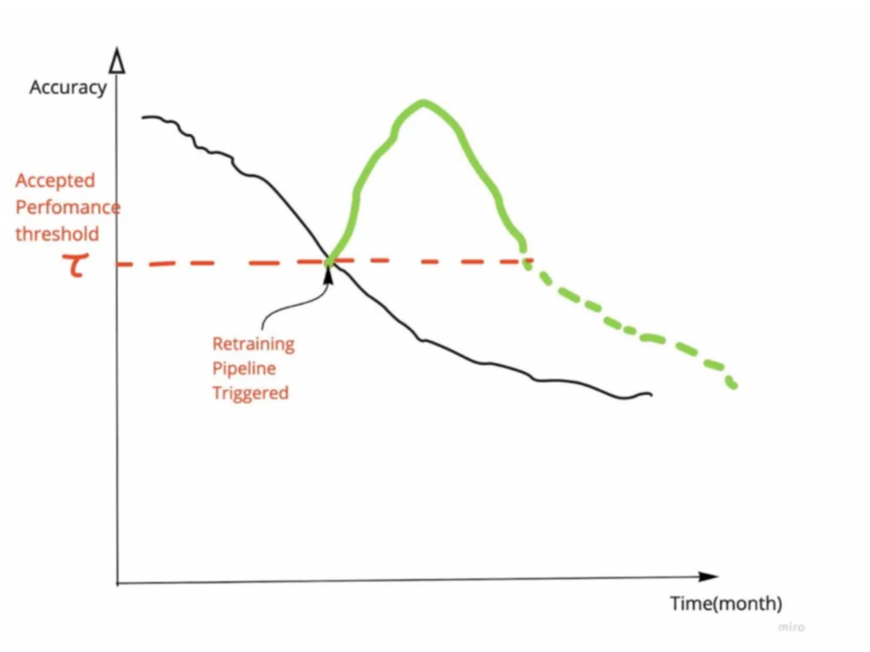


Figure 2.1: Continues training- continues testing

In [3], the authors use Model Drifting to measure the degradation of a model's prediction power due to changes in the environment. They take advantage of the Jenson-Shannon divergence [4] to identify prediction drift (a change in the distribution of the predicted label – $p(\hat{y}|X)$) [5] in real-time model output and compare it with the accuracy using training data. They set a threshold value as shown in figure 2.1 [2]. Once the threshold value is reached, they will retrain the model. If the data has significantly changed the machine learning model will give inaccurate results. However, this can only be determined after running the model. In other words, they do not train the model until it is run, and the accuracy obtained which may lead to inaccurate predictions. In this case also they are running the model each time to measure the accuracy.

Web services such as Azure [6], AWS etc. face the same problem as data stored in these clouds change. Because of the enormous resources available in such clouds, they store all the previous data in the pipeline and retrain the model each time a prediction is required. This requires a lot of resources, including processing power to retrain and a lot of main memory to store all this data.

Our goal is to determine when to retrain the model without making any predictions on the updated data and measuring the accuracy. None of the above approaches do this. They all make predictions every time and then decide whether to retrain the model. Other approaches retrain the model each time. In my research, rather than making a prediction each time and measuring its accuracy, I identify the key points in graphical data, such as node centrality and edge centrality. Significant changes to these critical points determine whether the model must be retrained. Hence, in my approach the model is not run each time to make a prediction.

Dynode2vec

Embedding vector spaces for each timestamp are different. To preserve embedding vectors, *dynode2vec* [15] *is* a dynamic graph embedding algorithm proposed by Aijun et. al. in 2019. This algorithm generates whole random walks for every time stamp and the embedding vector spaces in each timestamp are different. *dynode2vec*() employs the dynamic skip gram model, where the previous time stamp learned embedding vectors are transferred as the initial weights for current time stamp. They will train random walks on the set of evolving nodes, as shown in the equation given below.

$$\Delta V_t = V_{add} \cup \{v_i \in V_t | \exists e_i = (v_i, v_j) \in (E_{add} \cup E_{del})\}$$

--- Evolving Nodes

```
Algorithm: - Dynnode2vec

Input: Graphs G = G1, G2,...,GT
Output: Embedding vectors Z1, Z2 , . . . , ZT
Run static node2vec for the Graph G1
For t=2 to N do
    Find a set of evolving nodes, $\Delta V_t$
    Sample new random walks ($Walk_n$) for $\Delta V_t$
    Train Skip-Gram $Skip_t$ with $Walk_n$ and obtain $Z_t$
end for
```

This above algorithm will preserve the embedding vector spaces for previous and current timestamps. But the evolving nodes in this algorithm are limited to the set of new nodes or nodes which are affected by edges. In my research, I follow a similar algorithm to embed the dynamic graph but my algorithm finds the set of evolving nodes by taking advantage of depth-first search (dfs) traversal.

CHAPTER III

INTRODUCTION TO MACHINE LEARNING ON GRAPHS

3.1 Formal definition of Graphs

A graph is a collection of nodes and edges where nodes represent entities and edges represent relationships or interactions between the nodes. Nodes and edges may have attributes or features that contain more information about that instance.

Formally, a graph can be represented as $G = (V, E)$ where V is the set of nodes and E is the set of edges. If an edge exists between nodes u and v where $u \in V$, $v \in V$ then $(u, v) \in E$. If it is an undirected graph then $(u, v) \in E \leftrightarrow (v, u) \in E$.

In an Adjacency matrix $A \in R^{|V| * |V|}$, every node indexes a particular row or column in the adjacency matrix. $A[u, v] = 1$ if $(u, v) \in E$, else $A[u, v] = 0$. If the graph is undirected, the adjacency matrix is symmetric in nature. If the graph is directed the matrix need not to be symmetric. In the weighted graph $A[u, v]$ will not be in domain $\{0,1\}$. It will depend on the weight of the edge. For example, states in the USA are nodes and distance between each state are represented with a weighted edge.

3.2 Node Degree

Node degree is the number of edges connected to node. Node degree is denoted as d_u, where $u \in V$. The node degree d_u, is:

$$d_u = \sum_v A(u, v)$$

The above equation is for an undirected graph. For directed graphs there will be in-degree and out-degree associated with each node.

In-degree is:

$$d_u^{in} = \sum_v A(u, v)$$

Out-degree: -

$$d_u^{out} = \sum_v A(v, u)$$

Total Degree: -

$$d_u^{total} = d_u^{in} + d_u^{out}$$

For weighted graphs, there is a weighted degree associated to each node.

$$W\_D_u = \sum_v A(u, v)$$

3.3 Node Centrality

The degree of a node is the number of neighbors of the node. One of the important measures of centrality is Eigenvector Centrality.

a) Eigenvector Centrality

Eigenvector centrality not only considers the neighbor nodes but also their importance. In general, we define the node eigenvector centrality x_u where $u \in V$ by a recurrence relation where node centrality is directly proportional to the average node centrality of neighbors. Relative score is assigned to each node after iterations become saturated, that is, after some number of iterations the relative score remains unchanged. We can say a node will have a high value if it is connected to high score nodes rather than it having more node degree. Node degree depends upon only the numbers of nodes connected to it, but here it also depends on how important the connected nodes are.

$$x_u = \frac{1}{\lambda} \sum_{v} A[u,v] x_{v'} \quad \forall u \in V$$

Here λ is a constant. With a small re-arrangement we can re-write the equation as the eigenvector notation [7] below:

$$Ax = \boldsymbol{\lambda} x$$

If we start with $(1,1,1,\ldots v)^T$ as vector x^0 in the first iteration, then x^1 contains the degree of all the nodes. We will iterate the process until the principle eigen value is obtained.

In the example [8] below, we illustrate the simplification caused by Eigenvalue centrality

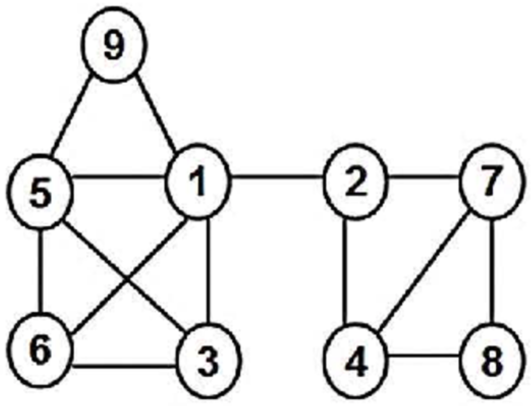


Figure 3.1: Node centrality example

$$\underset{\text{Adjacency Matrix}}{\left[\begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 5 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 6 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 7 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 8 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 9 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right]} \underset{\text{EVC Column Vector (beg. of Iteration 2)}}{\begin{bmatrix} 0.516 \\ 0.309 \\ 0.309 \\ 0.309 \\ 0.413 \\ 0.309 \\ 0.309 \\ 0.206 \\ 0.206 \end{bmatrix}} = \overset{\text{Product Vector}}{\begin{bmatrix} 1.546 \\ 1.134 \\ 1.238 \\ 0.824 \\ 1.340 \\ 1.238 \\ 0.824 \\ 0.618 \\ 0.929 \end{bmatrix}} \equiv \overset{\text{Normalized Value = 3.34}}{\underset{\text{EVC Column Vector (end of Iteration 2)}}{\begin{bmatrix} 0.463 \\ 0.339 \\ 0.371 \\ 0.247 \\ 0.401 \\ 0.371 \\ 0.247 \\ 0.185 \\ 0.278 \end{bmatrix}}}$$

$$\underbrace{\begin{array}{c|ccccccccc|} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 5 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 6 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 7 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 8 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 9 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}}_{\text{Adjacency Matrix}} \underbrace{\begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix}}_{\text{EVC Column Vector (beg. of Iteration 1)}} = \overset{\text{Product Vector}}{\begin{bmatrix}5\\3\\3\\3\\4\\3\\3\\2\\2\end{bmatrix}} \equiv \overset{\text{Normalized Value = 9.69}}{\underset{\text{EVC Column Vector (end of Iteration 1)}}{\begin{bmatrix}0.516\\0.309\\0.309\\0.309\\0.413\\0.309\\0.309\\0.206\\0.206\end{bmatrix}}}$$

Figure 3.2: Node Centrality example continued.

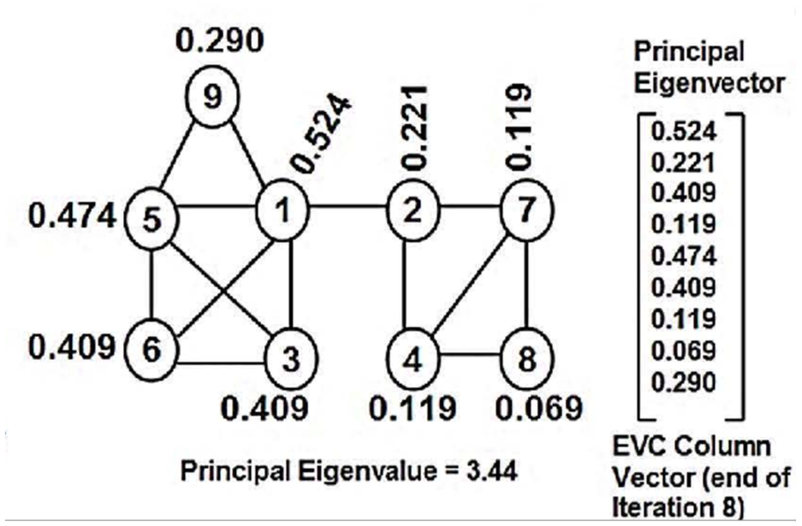


Figure 3.3: Result for Node centrality example

From the principal eigen vector, Node 1 is having centrality = 0.524 and so on. Node centrality plays a crucial role in my research.

b) Betweenness Centrality

Betweenness centrality [9] is one of the popular node centrality measures which depends on the shortest paths of two nodes. This measure depends upon the flow of information of the graph. For every two nodes in the connected graph there exists a shortest path between them. The number of edges the path passes through is minimized and for weighted graphs, the sum of the weights of those edges in the paths are minimized. Node $u \in V$ has the highest betweenness centrality if $(x, y) \in V$ which has the shortest path between them, consists of node u frequently compared to other nodes. Betweenness centrality is calculated as:

$$b(u) = \sum_{x \neq y \neq u} \frac{\sigma_{(xy|u)}}{\sigma_{xy}}$$

σ_{xy} = Number of shortest paths from node x to node y

σ_{xy}(u) = Number of shortest paths that pass-through u.

c) Edge Centrality

Edge Betweenness centrality [10] is the centrality measure for edges like betweenness centrality for nodes; this measure calculates how powerful the edge is in the graph. Every edge in the graph will have an edge betweenness centrality value. If the edge has high centrality score and this particular edge serves as the connection between two or more networks, removing the edge leads to drastic change in graph behavior.

$$b(e) = \sum_{x \neq y \neq u} \frac{\sigma_{(xy|e)}}{\sigma_{xy}}$$

σ_{xy} = Number of shortest paths from node x to node y

$\sigma_{xy|e}$ = Number of shortest paths that pass-through edge e.

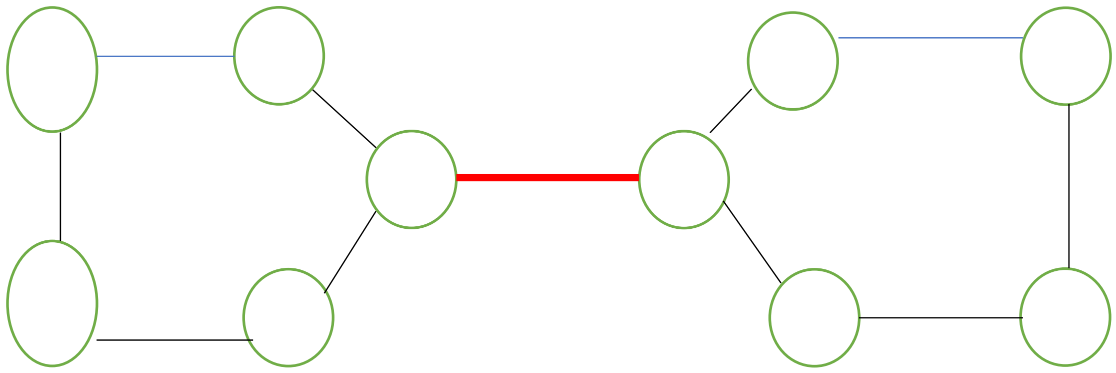


Figure 3.4: Edge centrality example

In Figure 3.4, the red edge has the most edge betweenness centrality as it is in the shortest paths of many pairs of vertices. If the edge is removed, the behavior of the graph may change. I have considered how when data changes, the model will be affected.

3.4 Node Embedding

Embedding is a popular technique in the machine learning world. Embedding allows representing complex objects like text, graph, images in a vector format while preserving all the important information about the data. Node embedding is the most well-known technique for graph embedding

Node Embedding is the process of embedding each node in the graph to a low dimensional vector space. Node embedding must preserve the graph structure such that nodes which are close or neighbors in graph, should be close to each other in the embedding space. These low dimensional embeddings can be viewed as projections into a latent space as shown in Figure 3.5.

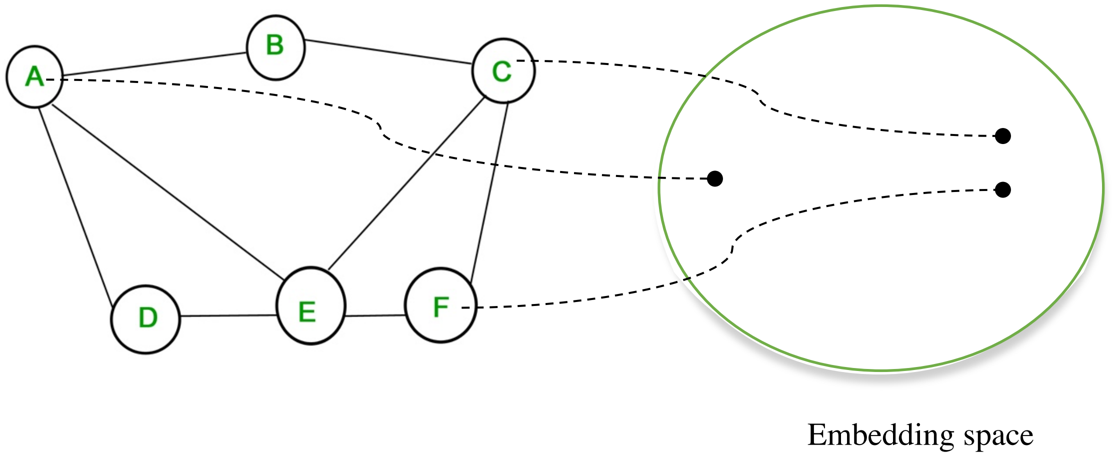


Figure 3.5: Node Embedding Example

There are several node embeddings techniques like Matrix Factorization-based and Random walks. Matrix Factorization approaches are directly inspired by classical techniques for dimensionality reduction called Laplacian Eigenmaps [12]. For data with dynamic periodicity, we can use random walks to learn the previous timestamps vectors. Hence in this research I use the Random-walk based node embedding methods.

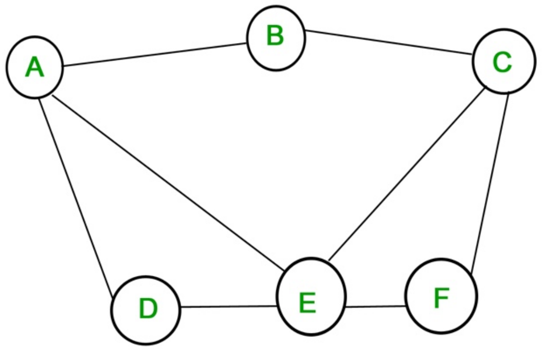
3.5 Random Walks on Graphs [11]

For a given graph $G = (V, E)$ and starting node $u \in V$, we select a neighbor of u, $v \in V$ $[(u,v) \in E]$ at random and move towards node v. Random walks then select the neighbor nodes of the node v and selects any one randomly and moves towards it. We do this *walk_length* ($\in N$), number of times. Walk length specifies the length of the sequence of random walks. These sequences of nodes are called random walks for a node starting at u in a graph G. Each node can be associated to a specific random walk/walks.

Let us consider $G = (V, E)$ be a graph or digraph (directional). If G is a digraph, then node $u \in V$, $d_u^{out} > 0$ for every vertex u. If starting node is u then the probability that it goes to node $v \in V$, where $(u,v) \in E$ is $\frac{1}{d_u}$ or $\frac{1}{d_u^{out}}$ if the graph is digraph.

$$p_{vu} = \begin{cases} \frac{1}{d_u}\ if\ (u,v) \in E \text{ in the Graph } G \\ \frac{1}{d_u^{out}}\ if\ (u,v) \in \text{ in the digraph } G \\ 0 \quad \text{Otherwise} \end{cases}$$

Random Walks is one of the most important and popular node embedding methods. The walks we get after running random walks for any graph is called a *corpse of walks*. It is an initial step to embed the nodes in the form of vectors on which we train the machine learning model. In the corpse we can observe random walks starting from each vertex that runs till *walk_length*. In the example below consider a undirected graph $G = (\{A,B,C,D,E,F\}, edges)$. Let take *walk_length* = 4. Random walks might be different for each instance. Random walks for the graph in Figure 3.6 are below.



A-> D -> E->F	A->B->C->F
D-> A -> B->C	D->E->F->C
E-> C-> F -> C	E->F->C->B
F -> C-> B->A	F->E->D->A
C -> B ->A->D	C->F->E->D
B-> A -> D->E	B->C->F->E
(1st Run)	(2nd Run)

(CORPSE)

Figure 3.6: Random walks example

Random walks are not the same for all time instances. The walks will vary with respective to neighboring nodes and their incoming probabilities and outgoing probalities. Incoming and outgoing probabilities are values we assign to the random walks. This means from each node has p1 probability of being an incoming node and p2 probability of being an outgoing node where p1+p2 =1. This means the random walk will select an outgoing node which is a node that is different node than the previous node in the walk with probability p2 and select an incoming node which is a walk that returns to the previous node with probability p1.

3.6 Random walk-based node embedding Techniques

Many real-world examples use random walk-based node embedding techniques to optimize the node embeddings so that every node which are neighbors have similar embeddings if they tend to reoccur on short random walks.

a) Node2Vec

Node2Vec [13] is an algorithmic framework for learning continuous feature representations for nodes in networks. In node2vec, we map nodes to a low-dimensional space of features that maximizes the likelihood of preserving network neighborhoods of nodes. We define a flexible notion of a node's network neighborhood and design a biased random walk procedure, which efficiently explores diverse neighborhoods. Node2Vec generalizes prior work which is based on rigid notions of network neighborhoods. The added flexibility in exploring neighborhoods is the key to learning richer representations. The main drawback of this algorithm is that it does not preserve the embedding vector spaces for the next time stamp data.

b) Deep Walk:

Deep Walk [14] can be represented as:

Deep Walk = Random Walks + Skip Gram

Skip Gram [16][17] is an algorithm that is used to create word embeddings i.e., high-dimensional vector representation of words. These embeddings are meant to encode the semantic meaning of words such that words that are semantically similar will lie close to each other in that vector's space. It assumes words that are semantically similar are used in similar contexts.

An example

→ Hi, Good Evening;   Hello, Good Evening;

Here hi, hello have the same meaning. Vector representation of "Hi, Hello" will be close to each other in the embedding space as they contain the same words in the sentences and mean the same.

Vocabulary set: {Hi, Hello, Good, Evening}. (All unique words in the corpse)

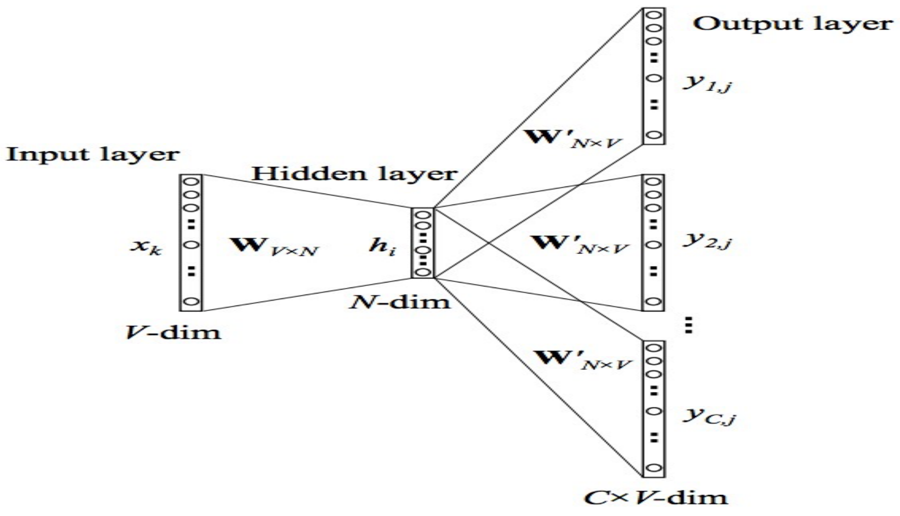


Figure 3.7: Skip gram Architecture.

If we observe the above architecture, initially x_k is the one-hot vector which is the vector to represent each word in the vocabulary. We will place 1 in place of that word and the remaining will all be 0's as shown below.

$X_{hi} = [1\ 0\ 0\ 0]^T$

$X_{hello} = [0\ 1\ 0\ 0]^T$

$X_{good} = [0\ 0\ 1\ 0]^T$

$X_{Morning} = [0\ 0\ 0\ 1]^T$

The respective 1-hot vector will multiply with initial weight matrix (which in dynnode2vec(). We will take advantage of the dynamic skipgram model for obtaining current time stamp vectors with previous skipgram as initial weights). Calculations on hidden layer are performed to give the embedding vectors.

Coming to Deep Walk, till now we are considering only graphs and data represented as graphs, but in the above subsection, I said skipgram is used to create word embeddings from words and words that are converted into vectors. Below we show how skipgram and random walks capture graph structure.

We can use SkipGram to capture the community structure of a graph by viewing the graph as a kind of language where:

- Each node is a unique word in the language
- Random walks of finite length on the graph constitute a sentence with the respective node where it starts. For each node there will be at least one random walk which acts as sentence of the node.

The final output will be the vectors for each node preserving the graph structure.

3.7 Matrix Norms

The norm of Matrix A ∈ R^{m*n} is similar to the absolute value of the Matrix, denoted by ‖ A ‖.

It is calculated by:

$$\| A \| = \sum_{i=0}^{m} \sum_{j=0}^{n} \sqrt{a_{ij}^2} \ .$$

This essentially calculates the square root to the sum of the squares of the all the numbers in the matrix.

Example:

$$\text{If } A^{3*3} \ = \begin{matrix} 2 & -1 & 2 \\ 1.23 & 0 & 3 \\ 0.01 & -1.34 & 1 \end{matrix}$$

$$\| A \| = \sqrt{2^2 + (-1)^2 + 2^2 + 1.23^2 + 0^2 + \ 3^2 + \ 0.01^2 + \ (-1.34)^2 + 1}$$

$= \sqrt{4 + 1 + 4 + 1.51 + 0 + 9 + 0.00 + 1.79 + 1}$

$= \sqrt{22.3}$

$\|A\| = 4.72$

Norm of the matrix tells us the difference between the two matrices. To find out how similar the two matrices are, we subtract the two matrices A – B and find the norm of the resultant matrix, $\|A - B\|$. If the value is low or close to 0 then we can say that A and B are almost identical.

CHAPTER IV

STATISTICAL APPROACH TO RETRAINING GRAPH DATA

4.1 Problem Specification:

Machine learning graphs models learn from the graph data in order to solve tasks such as node classification, edge classification, node prediction and so on. If the data is stable or static, the graph structure and properties do not change. But in the real world most of the data have dynamic periodicity (data changes with time). This means that there will additions/deletions of nodes/edges in the graph.

As the distribution of the data changes with time, to get good accuracy, we need to retrain the model. But the data may or may not have changed sufficiently to significantly affect the accuracy of the machine learning task. The research problem is then to determine when to retrain the model. For this problem I propose a solution based on statistical approaches called significant testing considering several aspects of graph measures like node centrality, edge centrality, betweenness centrality and norm.

Embedding vector spaces will be different in each timestamp. Many of the existing graph embedding methods are for static graphs. The only work that looks at embedding for dynamic graphs is the dynamic graph embedding algorithm called '*dfs_dynode2vec*' [16]. This algorithm takes advantage of depth first search (*dfs*) traversal on the existing algorithm *dynode2vec* [16].

4.2 Proposed Methodology

The goal is to determine when the model should be retrained before it is used for testing.

```
Algorithm_A (Data):
Step 1: - Node Embedding: (dfs_dynnode2vec())
Step 2: - $\text{Existing}_{\text{Model}}$ = Train the model
Return $\text{Existing}_{\text{Model}}$


Algorithm_B ($\text{Data}_{\text{new}}$):
 If (Data changes significantly):
         Algorithm_A($\text{Data}_{\text{new}}$)
 Else:
         Node Embedding : (dfs_dynnode2vec ())
         Evaluation with $\text{Existing}_{\text{Model}}$
 Return Accuracy
```

Algorithm_A should be implemented first for the original graph data and store the *$\text{Existing}_{\text{Model}}$*. Whenever the data changes, *Algorithm_B* is executed to determine if the change is significant enough to affect the results of the machine learning task. If the change in data is significant, then the model is retrained.

Embedding vector spaces are different each time the data changes. [16] proposed a solution to preserving vector spaces between timestamps as mentioned in the Literature Review section. The proposed approach for preserving embedding vector spaces is shown below:

```
dfs_dynnode2vec (graph$_i$ ,walklength):
        1)  Run  Node2Vec(Deep  Walk)for  the  graph$_0$(Initial
            graph)
        2) for graph$_i$ (i = 1: N):
              X ← New nodes
              Z ← Nodes effect by addition/removal of edges.
              for (all X and Z nodes)
                 Y  ← dfs_walklength(node, walklength)
              end
              $\delta$V$_i$  =  X ∪ Y ∪ Z
              corpse$_i$ =  Run random walks for $\delta$V$_i$
              Vectors$_i$ = Train Skipgram skip$_i$ with corpse$_i$
          end
  end
dfs_walklength (node, walklength):
          for i in range (X ∪ Z):
          // get all nodes
          end
                  dfs (walklength, node)
   end
```

The above algorithm `dfs_dynode2vec()` is a graph embedding technique. `dynnode2vec()` [16] in the algorithm takes advantage of depth first search to find the nodes with the distance of the walklength which is done by `dfs_walklength()`

The above algorithm takes as input *graph$_i$* and *walklength*. *walklength* defines the number of walks on each node when random walk is implemented. Step 1 is done only for the initial graph, that is, run the static node2vec (Deep walk) [14]. When data changes at time stamp i, the graph is represented by *graph$_i$* . For each new node or node affected due to removal of an edge, apply `dfs_walklength` algorithm to get all nodes within the walklength. For the evolving nodes represented by δV_i, apply random walks and store in *corpse$_i$*. Next train the skip gram model for *corpse$_i$* .

The proposed methodology for determining whether or not to retrain the model is outlined below:

Constructing data in matrix formats:

Graph data can be represented in many ways. Matrix form is one of them. The Adjacency matrix shows if two nodes are connected or not and the weighted matrix will show the weight between two nodes. In this project, we considered a matrix with edge centralities. In this work, we measure the difference between the original matrix and the new matrix to decide if the model has to be retrained

```
**Algorithm_original_matrix** (Graph_original):
    Step 1: - Find Edge centralities for Graph_original in
    dictionary format
    Step 2: - initialize a Nd array M which is in Matrix format
    Step 3: - append. Edge centralities (M)
**Return** M
```

```
**Algorithm_new_matrix** (M, Graph_new):
    Step 1: - Get all nodes in new graph
    Step 2: - Append new nodes to M
    Step 3: - With the same structure of M take a new Nd array
              $M^1$
    Step 4: - Find Edge centralities of Graph_new
    Step 5: - append. edge_centralities($M^1$)
    Step 6: - Fill remaining values with 0's in M and $M^1$.
**Return** M, $M^1$
```

The above algorithm "`algorithm_original_matrix()`" takes as input the original graph data. This algorithm finds edge centralities of each edge in a graph which is output in the format of a dictionary `{[(node`$_i$`, node`$_j$`), edge_centrality`$_{ij}$`]}`. That means for every pair of nodes in the graph we have an edge centrality value. The matrix datatype is not available in any of the major programming languages (C, java, python). I have therefore taken a Nd array (Nd array is an N dimensional array which is a primitive data structure that stores same kind of variables). When we considered an Nd array, it will be like ‘n’ rows arranged in the linearly format which is in a matrix-like format. Initialize all the values of edge centralities in M with their respective indices (i, j).

`Algorithm_new_matrix()` constructs a new matrix when the data changes. It takes as inputs graph$_{new}$ and Matrix M and return an updated matrix M and new Matrix M^1. M is the matrix which contains edge centralities of the original matrix and M^1 is the matrix which contains edge centralities values when the data changes that is for new data. In the first step we will get all the nodes from the new data and check the indices of M. If nodes have been deleted, some of the nodes will not be present in M. So, we add new indices to M which means we are extending the matrix with indices because M and M^1 should have similar structure to perform the Norm (matrices extending is for preserving the structure of the two matrices). We have the structure of M (original data matrix), To preserve the structure of the matrix with the same dimensions take another Nd array M^1, find the edge centralities of the new graph and initialize them in M^1 with respective to their indices (i,j). There might be chance of some of them will have null values, initialize them to 0’s and return M and M^1.

Calculating Norm:

We have two matrices M and M^1 $\in R^{n*n}$, where

if a = {nodes in Graph$_{original}$ }

b = {nodes in Graph$_{new}$}

n = #$_{nodes}$ in {a ∪ b}

M is the original data matrix and M^1 is the new data matrix.

Let,

$A = M - M^1$

$|| A || = \sum_{i=0}^{n} \sum_{j=0}^{n} \sqrt{a_{ij}^2}$

Constructing data in vector formats

In the above algorithms, we represented the data in Matrix format. Next, we represent it in vector format. Vector format is a 1-dimensional array in vector format. Here the main idea is to verify the data change in the shorter format.

```
Algorithm_original_vector (Graph_original):
   Step 1: - Find Edge centralities for Graph_original in
   dictionary format
   Step 2: - initialize a 1d array V which is in vector format
   Step 3: - append. Edge centralities (V)
Return M
Algorithm_new_vector (M, Graph_new):
    Step 1: - Get all nodes in new graph
    Step 2: - Append new nodes to V
    Step 3: - With the same structure of M take a new 1d array
              V^1
    Step 4: - Find Edge centralities of Graph_new
    Step 5: - append. edge_centralities(V^1)
    Step 6: - Fill remaining values with 0's in V and V^1.
Return V, V^1
```

The above algorithm "`algorithm_original_vector()`" takes as input the original graph data. This algorithm finds edge centralities of each edge in a graph which is output in the format of a dictionary `{[(node`$_i$`, node`$_j$`), edge_centrality`$_{ij}$`]}`. That means for every pair of nodes in the graph we have an edge centrality value. I have therefore taken a 1d array, V which is in a vector-like format. Initialize all the values of edge centralities in V with their respective indices i {(i = (node$_i$, node$_j$)}.

`Algorithm_new_vector ()` constructs a new vector when the data changes. It takes inputs as *graph$_{new}$* and Vector V and returns an updated vector V and new vector V^1. V is the vector which contains edge centralities of the original vector and V^1 is the vector which contains edge centralities values when the data changes, that is, for new data. In the first step we will get all the nodes from the new data and check the indices of V. As the data may have changed, some of the nodes may not be present in V. So, we add new indices to V which means we are extending the Vector with indices because V and V^1 should have similar structure to perform the Norm (vectors extending is for preserving the structure of the two vectors V and V^1). We have the structure of V (original data vector), To preserve the structure of the vector with the same dimensions take another 1d array V^1, find the edge centralities of the new graph and initialize them in V^1 with respective to their indices i. Because of changes in the data, some of them may have null values. Initialize them to 0's and return V and V^1.

Calculating Norm:

We have two matrices V and $V^1 \in R^n$, where

if a = {nodes in $\text{Graph}_{\text{original}}$ }

b = {nodes in $\text{Graph}_{\text{new}}$}

n = $\#_{\text{nodes}}$ in {a ∪ b}

V is the original data vector and V^1 is the new data vector.

Let,

$A = V - V^1$

$|| A || = \sum_{i=0}^{n-1} \sqrt{a}_i$

CHAPTER V

RESULTS

5.1 Dataset

The Cora data is used for all experiments [18]. There are two columns in the data set

<ID of cited paper>	<ID of citing paper>
1) u_1	v_1
2) u_2	v_2
….	
5429) u_{5429}	v_{5429}

There are 2485 unique papers and 5429 interactions in the dataset. An interaction between two papers occurs when a paper *a* is cited by paper *b*. Each paper is represented as a node and each interaction as an edge. Papers u1 and v1 are linked so there will an edge connecting these two nodes in the graph.

Each node has an attribute, 'node subject'. This defines the subject of the paper. A paper has only one subject. For example, if node u_1 has node subject 'Probabilistic methods', then that node will be classified with label 'Probabilistic Methods'. There are 7 unique labels representing different subjects:

```
{'Case_Based','Genetic_Algorithm','Neural_Networks',
'Probabilistic_Methods','Reinforcement_Learning','Rule_Lear
ning','Theory'}.
```

The Cora dataset is a static dataset, that is, the data does not change. But for my research I need data which behaves with dynamic periodicity, I therefore selected randomly 80% of the original data and constructed a graph out of this data. This is the training data or graph. The model was trained on this data. Some or all of the remaining 20% of the data (the testing data or graph) was added and deleted from the 80% of the data to emulate the dynamic behavior of the data. This results in a new graph that behaves like a dynamic graph. In the next step, the new nodes are classified, and the accuracy measured.

5.2 Experiments

Experiments were conducted to preserve the embedding vector spaces for two different random walks

```
colors = {'Case_Based': 'black',
            'Genetic Algorithms': 'red',
            'Neural_Networks': 'blue',
            'Probabilistic_Methods': 'green',
            'Reinforcement_Learning': 'aqua',
            'Rule_Learning': 'purple',
            'Theory': 'yellow'}
```

1) Experiment 1:

This experiment visualizes the results when the static Node2vec algorithm is run for similar data at two different times. Fig. 5.1 and Fig. 5.2 below are low dimensional plots of the embedded graph where each color denotes one of the node subjects listed above. Both the figures preserve the graph structure as similar nodes are plotted together, but they do not have the same embedding vector spaces. This leads to low accuracy

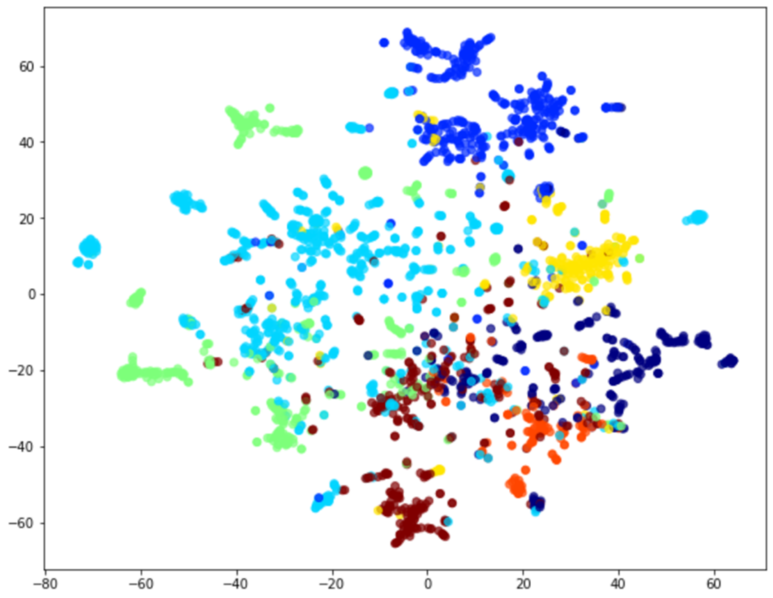


Figure 5.1: Graph as embedding vector space 1

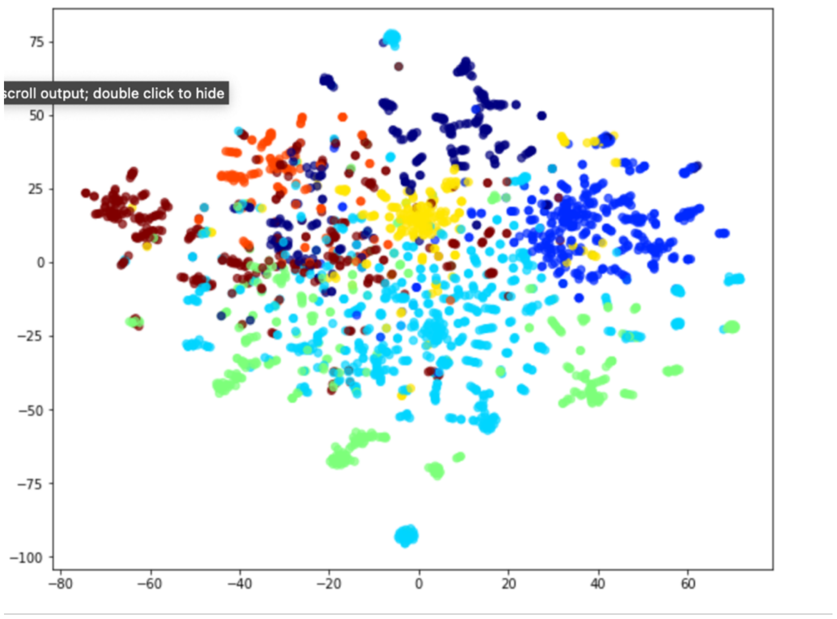


Figure 5.2: Graph as embedding vector space 2

Experiment 2

The second experiment classified the 20% Test data which was removed from the original Graph. The classification accuracy was 0.87.

Experiment 3

The `dynnode2vec()` and `dfs_dynnode2vec()` algorithms were run with change in data. The training was done on 80% of the original graph (training graph). The remaining 20% of the graph was used for testing.

<u>3.1 Changes in high centrality edges</u>

a) The training graph is 80% of the original graph. The top 10% of high centrality edges in the training graph were deleted

b) 20% of new nodes (from the test data which is the remaining 20% data) were added to the graph.

c) The trained model (80% of the graph) was used to classify the new 20% nodes.

Results:

Method	Accuracy
Dynnode2vec	0.287
Dfs_dynode2vec	0.298

Table 5.1: Results when 10% high centrality edges were removed

The results show that deleting high centrality edges results in low accuracy. The model therefore needs to be retrained.

3.1 Changes in leaf nodes

a) The training graph is 80% of the original graph. 20 % of leaf nodes were deleted from the training graph
b) 20% of new nodes (from the test data which is the remaining 20% data) were added to the graph.
c) The trained model (80% of the graph) was used to classify the new 20% nodes.

Results:

Method	Accuracy
Dynnode2vec	0.712
Dfs_dynode2vec	0.759

Table 5.2 – Results when 20% leaf nodes were removed

The results show that when leaf nodes are deleted, the accuracy is not significantly impacted. There is therefore no need to retrain the model. The same number of nodes may be deleted, but because they have low centrality value, the effect is low.

3.3 Changes in high centrality edges and leaf nodes

a) The training graph is 80% of the original graph. 10 % of leaf nodes were and the top 2% of high centrality edges were deleted from the training graph

b) 20% of new nodes (from the test data which is the remaining 20% data) were added to the graph.

c) The trained model (80% of the graph) was used to classify the new 20% nodes.

Method	Accuracy (20%)
Dynnode2vec	0.632
Dfs_dynode2vec	0.681

Table 5.3 – Results when 10% leaf nodes and Top 2% high centrality edges

These results show that high centrality nodes and edges change graph features more than the low centrality edges and nodes. High centrality edges have more impact than low centrality in terms of the accuracy of the machine learning tasks.

Experiment 4

Experiment 3 shows that high centrality nodes and edges change graph features more than low centrality edges and nodes. This means removing low centrality nodes doesn't impact the trained model but when high centrality nodes are removed it impacts the trained model and leads to low accuracy.

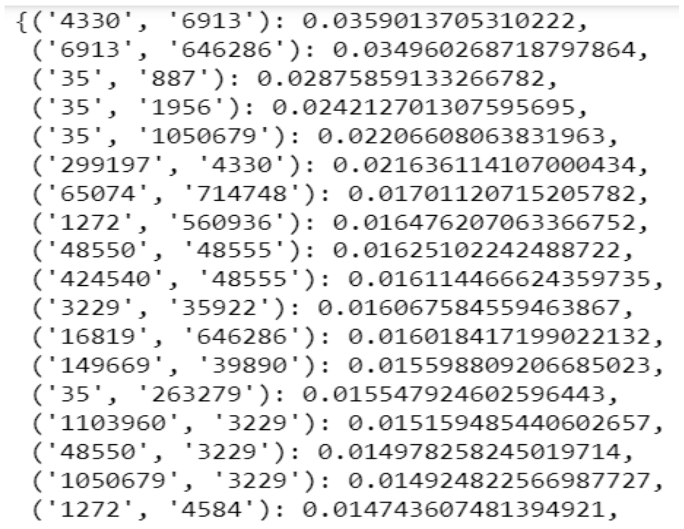
To determine if the model needs to be retrained we use Norm calculations. Algorithms `algorithm_original_matrix()` and `algorithm_new_matrix` proposed in the methodology section are used to calculate the norms of the matrix M - M^1 to determine the difference between the two matrices, in other words, are the two matrices identical or very different . The results are shown in the table below

Accuracy for original data with original trained model.	Accuracy for new data when tested with original trained model	Norm of M-M^1
87.4	19.4	0.158
87.4	29.8	0.154
87.4	48.3	0.135
87.4	53.2	0.127
87.4	55.8	0.109
87.4	56.3	0.088
87.4	61.5	0.075
87.4	65.9	0.074
87.4	68.1	0.065
87.4	68.8	0.049
87.4	69.5	0.0061
87.4	69.9	0.0001
87.4	71.2	0.0001
87.4	72.6	0.0001
87.4	72.9	0.0001

87.4	73.5	0.0001
87.4	75.9	0.0001
87.4	76.5	0.0001
87.4	77.8	0.0001
87.4	78.2	0.0001
87.4	79.5	0.0001
87.4	80	0.0001
87.4	81.5	0.0001
87.4	83	0.0001
87.4	84.5	0.0001

Table 5.4: Results for values of norms for different accuracy values (experiment 4)

Table 5.4 shows that entries with norms = 0.0001 has little change in accuracy between the original and the new data. When the norms are above > 0.061, the accuracy falls due to significant changes in data. When we removed high centrality edges which have high values in the matrix, then the norm value will be of higher value. When we deleted low centrality edges which have low values in the matrix, the norm will be nearly equal to zero.



```
{('4330', '6913'): 0.0359013705310222,
 ('6913', '646286'): 0.034960268718797864,
 ('35', '887'): 0.02875859133266782,
 ('35', '1956'): 0.024212701307595695,
 ('35', '1050679'): 0.02206608063831963,
 ('299197', '4330'): 0.021636114107000434,
 ('65074', '714748'): 0.01701120715205782,
 ('1272', '560936'): 0.016476207063366752,
 ('48550', '48555'): 0.01625102242488722,
 ('424540', '48555'): 0.016114466624359735,
 ('3229', '35922'): 0.016067584559463867,
 ('16819', '646286'): 0.016018417199022132,
 ('149669', '39890'): 0.015598809206685023,
 ('35', '263279'): 0.015547924602596443,
 ('1103960', '3229'): 0.015159485440602657,
 ('48550', '3229'): 0.014978258245019714,
 ('1050679', '3229'): 0.014924822566987727,
 ('1272', '4584'): 0.014743607481394921,
```

Figure: 5.3: High Edge centralities values.

Analysis of results

Figure 5.3 shows us the highest edge centrality values for nodes. For example {('4330', '6913'): 0.035} means nodes 4330 and 6913 are connected with an edge centrality of 0.035. Whenever we remove the high centrality edges from the graph, in the new matrix, the edge between 4330 and 6913 is deleted and the position of 4330 and 6913 will become 0. The norm now value becomes more than 0.035. Hence, whenever we remove high centrality edges, the norm value increases and accuracy decreases, Table 5.4 shows that the values of the norms keep increasing when the accuracy goes down.

Norm value equaled almost 0 until the accuracy reached 69%. Once high centrality edges were removed, the norm value increased more than the highest edge centrality value of the graph which is 0.036. All the norm values greater than 0.036 have low accuracy which is shown in Table 5.4. Here the Threshold value was set at the highest edge centrality value. These results show that when the norm of the difference is greater than the highest edge centrality value then the model must be retrained, otherwise retraining is not needed.

Experiment 5

I also checked the norm with the data in vector format. In vector format the data is in a 1-dimensional array. In the above experiment the graph data is in matrix format whereas in this experiment the data is in vector format and the results are compared.

Accuracy for original data with original trained model.	Accuracy for new data when tested with original trained model	Norm of $V-V^1$
87.4	19.4	0.111
87.4	29.8	0.108
87.4	48.3	0.0954

87.4	53.2	0.0898
87.4	55.8	0.077
87.4	56.3	0.0632
87.4	61.5	0.05302
87.4	65.9	0.05233
87.4	68.1	0.05304
87.4	68.8	0.03606
87.4	69.5	0.0009
87.4	69.9	0.0001
87.4	71.2	0.0001
87.4	72.6	0.0001
87.4	72.9	0.0001
87.4	73.5	0.0001
87.4	75.9	0.0001
87.4	76.5	0.0001
87.4	77.8	0.0001
87.4	78.2	0.0001
87.4	79.5	0.0001
87.4	80	0.0001
87.4	81.5	0.0001
87.4	83	0.0001
87.4	84.5	0.0001

Table 5.5: Results for values for norms for different accuracy values (experiment 5)

<u>Analysis of results</u>

Table 5.5 shows that in experiment 5's the values of norms are lower than values of the norms for Matrix data representation by about $\frac{1}{\sqrt{2}}$ times. Here with the above threshold value set at 0.036 (high edge centrality value of the graph), the accuracy is acceptable and does not decrease much (because of the low change in data) until the norm reached 0.03. When the norm increases accuracy decreases. This experiment shows very similar results as the previous above experiment.

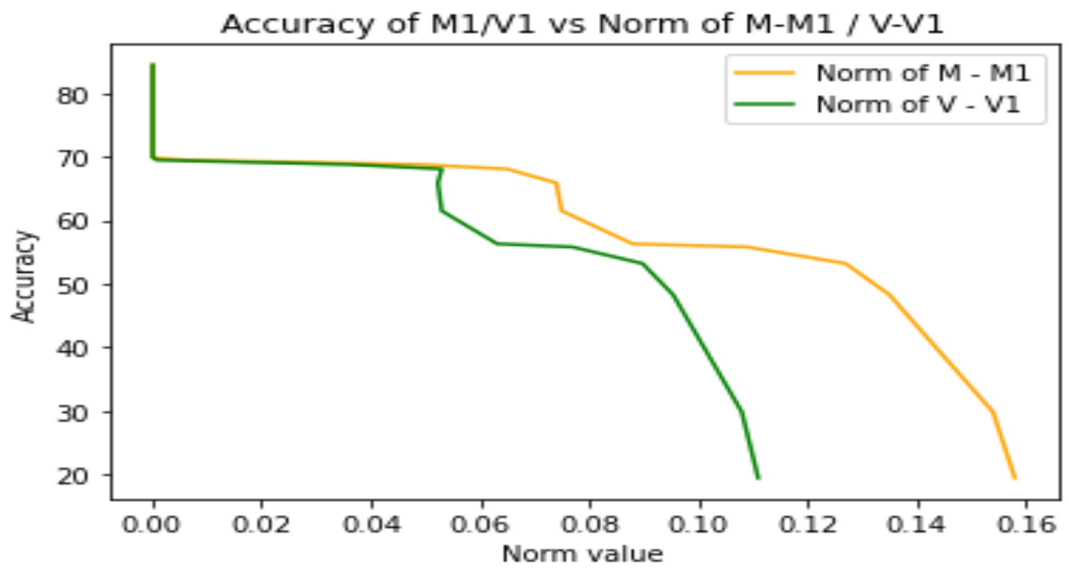


Figure 5.4 Accuracy vs Norm value of M – M^1 and V – V^1

In Fig 5.4, the graph is constructed between accuracy and norm values of different methods ($M-M^1$ and $V–V^1$) . If we observe both the lines, green for $V-V^1$ and yellow for M- M^1 both are almost identical till a norm value of approximately 0.05. After this they follow a similar pattern although the norm values diverge.

CHAPTER VI

CONCLUSIONS

This research proposes a method on when to retrain a machine learning model for dynamic graphs as data keeps changing. Initially for graphs to be trained, each node in the graph is converted to a vector. We encountered a problem in the sense that embedding vector spaces are different for each time stamp. To preserve the embedding vector spaces, we proposed the algorithm "`dfs_dynode2vec()`".

Each graph is essentially represented by a matrix like structure such as an adjacency matrix and weighted matrix. Here we introduce Matrix norms which evaluates the difference between any two matrices. I considered the edge centrality values as the weights for each edge. The higher the centrality, the higher the importance of the node. When high edge centrality values are removed there will large changes in the next timestamp matrix when compared to the initial matrix. The value of the norm will be high. From this work I conclude that when the Norm exceeds a threshold value then the model must be retrained. Otherwise, there is no need to retrain the model.

I used the Cora Dataset which is a small static dataset to validate the approach. The dataset was manipulated to display dynamic behavior. The results show that model retraining will depend on whether the edges removed are low centrality edges or high centrality edges. Future work will include calculating the complexity of the algorithms which are proposed and deriving the change in complexity from the old graph to the new graph when nodes are added/deleted. Other work will include testing this algorithm with real-world dynamic datasets.

REFERENCES

1) Kavikondala, Akanksha & Muppalla, Vivek & Prakasha, Dr. Krishna & Acharya, Vasundhara. "Automated Retraining of Machine Learning Models". 8. 445-452. 10.35940/ ijitee. L 3322.1081219, (2019). International Journal of Innovative Technology and Exploring Engineering.
2) Retraining Model During Deployment: Continues traning-continuos testing, https://neptune.ai/blog/retraining-model-during-deployment-continuous-training-continuous-testing (Date of last access: July 2nd , 2022)
3) Data Tron Blog, https://datatron.com/what-is-model-drift/ (Date of last access: July 2nd , 2022)
4) Jensen-Shannon Divergence https://notesonai.com/Jensen%E2%80%93Shannon+Divergence (Date of last access: July 2nd , 2022)
5) Concept_drift_in_machine https://www.aporia.com/blog/concept_drift_in_machine_learning_101/ (Date of last access: July 2nd , 2022)
6) Retraining and updating azure machine learning models https://azure.microsoft.com/es-es/blog/retraining-and-updating-azure-machine-learning-models-with-azure-data-factory/ (Date of last access: July 2nd , 2022)

7) Eigenvector centrality https://en.wikipedia.org/wiki/Eigenvector_centrality (Date of last access: July 2nd , 2022)

8) Natarajan Meghanathan , "Evaluation of Correlation Measures for Computationally Light vs. Computationally Heavy Centrality Metrics on Real-World Graphs" CIT Journal of Computing and Information technology, Vol 25, Issue 2, June 2017.

9) Betweenness centrality https://en.wikipedia.org/wiki/Betweenness_centrality (Date of last access: July 2nd , 2022)

10) Lu L., Zhang M , "Edge Betweenness Centrality". In: Dubitzky W., Wolkenhauer O., Cho KH., Yokota H. (eds) Encyclopedia of Systems Biology. Springer, New York, NY. https://doi.org/10.1007/978-1-4419-9863-7_874

11) Random walks on graph https://people.math.osu.edu/husen.1/teaching/571/random_walks.pdf (Date of last access: July 2nd , 2022)

12) M. Belkin and P. Niyogi, "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation," Technical Report, University of Chicago, Chicago, 2001.

13) A Grover, J.Leskovec, "Node2vec: Scalable Feature Learning for Networks", KDD. Aug 2016: 855–864. doi: 10.1145/2939672.2939754 2016 https://arxiv.org/abs/1607.00653

14) Bryan Perozzi, Rami Al- Rofu, Steven Skiena, "Deep walk: Online Learning of Social Representations" arXiv:1403.6652 [cs.SI] (2014) https://arxiv.org/abs/1403.6652v2

15) Sedigheh Mahdevi, Shima Khoshraftar and Aijun An, "Dynode2vec Scalable Dynamic Network Embedding" arXiv:1812.02356 [cs.LG] (2018) https://arxiv.org/abs/1812.02356v2

16) An illustrated explanation of using skip gram to encode structure of a graph, https://medium.com/@_init_/an-illustrated-explanation-of-using-skipgram-to-encode-the-structure-of-a-graph-deepwalk-6220e304d71b (Date of last access: July 2nd, 2022)

17) Word2vec tutorial skip gram model http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/ (Date of last access: July 2nd, 2022)

18) CORA dataset https://relational.fit.cvut.cz/dataset/CORA (Date of last access: July 2nd, 2022)

VITA

VARUN TEJA PURAM

COMPUTER SCIENCE

Master of Science

Thesis: MODEL RE-TRAINING FOR DYNAMIC GRAPHS

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in your Computer Science at Oklahoma State University, Stillwater, Oklahoma in July 2022

Completed the requirements for the Bachelor of Science in Computer Science at Mahatma Gandhi Institute of Technology, Hyderabad, Telangana , India in 2019.