Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Using Graph Embedding with Machine Learning on Simulation Models

Rajpal Singh

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Goetz Botterweck

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

[illegible]

Rajpal Singh

August 19, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Rajpal

Rajpal Singh

August 19, 2022

Using Graph Embedding with Machine Learning on Simulation Models

Rajpal Singh, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Goetz Botterweck

Graphs are the general language for describing and analyzing entities with relations and interactions. Graphs provide the optimal structure for the representation and manipulation of non-euclidean data. In this research, simulation models are structured in graphs and are used for cost prediction through machine learning models. In the comparison of simulation, graph neural networks with machine learning help to work seamlessly and takes much fewer resources and time for structuring and training the models.

This dissertation aims to apply tools for converting JSON data to graphs. With the help of graph embedding techniques and methods, graphs are converted into vectors which machine learning models can train the data with the fan cost value provided as the target value for prediction.

We carried out an experiment with the dataset that consist of 10000 graphs, of which 1200 were taken with the target value (fan cost) associated with each graph for prediction. NetworkX, a python framework, was used to structure graphs consisting of rooms as nodes and links between them as edges. Random Walks methods such as DeepWalk and Node2Vec were used to transform graphs focusing on node-level structuring. Moreover, Weisfeiler-Lehman methods such as Graph2Vec and GL2Vec, along with Laplacian matrix with eigenvalues methods such as NetLSD, were used as whole graph level structuring for graphs into vectors. Vectors were then trained with machine learning regression models like Linear Regression, Lasso/Ridge Regression, AdaBoost, Random Forest and Multi-Layer Perceptron (MLP). MLP with K Fold cross-validation has proven to be the best fitting model with an accuracy score of 21%, an error rate of 0.7 among whole-graph level methods and an accuracy score of 16% error rate of 0.14 for node-level methods.

Acknowledgments

My sincere thank you to my supervisor, Goetz Botterweck, and for the valuable guidance, assistance, and providing the resources that I needed. I would like to extend my sincere thanks to my PhD Colleague, Michael Mittermaier for guiding me through every step working on this research. I would also like to thank my family and friends for their support.

Rajpal Singh

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Over the years, graphs [4] have proven to be the language to relate and describe complex and non-linear relationships. Graphs with Deep learning [5], [6] have shown promising progress with applications like recommendation systems on social networking sites [7], predicting drug side effects [8], [9], protein-protein interactions [10], knowledge graphs [11], and many more. They have proven to be the next-level skills developed by the machine using deep learning, machine learning and graphs. Machine learning can predict outcomes with graphs by converting non-euclidean input (such as pictures, sounds, and texts) into vectors. With non-euclidean data, there is a chance that one may lose information like the quality of edges, information associated with nodes, type of connection between two nodes, and there is not any apparent specified way to map these correlations. Specious correlations can be introduced to map these correlations, such as considering the closeness between two unconnected nodes in the embedding rather than the relationship between two connected nodes. Processing this non-euclidean structured data as an input for Graph Neural Networks (GNNs) or graph methods with embedding techniques can convert the non-alignment space data into vector form without losing the additional information related to nodes and edges.

Deep Learning with machine learning has drawn much attention with its technological advancement and features. Neural Networks components such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and autoencoders are models of deep learning which are now widely used for pattern recognition, data mining, and image processing. They have proven very advanced techniques and are used in various applications. Graph Neural Networks (GNN) is a relatively recent branch which deals with graphs used to identify complex structuring and relationships between objects. A graph consists of nodes as objects and edges connected to different nodes as connections. The centrality of the nodes can also determine connections concerning the number of neighbours or distance to the node from each node. Social Networks, Economic Networks, Citation Networks, Communication Networks, Internet, and Neuron Networks, can all be described as graphs having real-world applications like Social Networking sites (Facebook, Instagram, LinkedIn), route navigation through maps [12], drug targeting with the protein, protein-protein interactions, and much more.

Graph embedding [13], [14] has become a sensitive area for research for machine learning models to work with graphs. It represents the projection of nodes into a low-dimensional vector in a network by encoding the node properties and network structures. Embedding tasks can be classified into node-level, link-prediction, subgraph (community) and whole graph level. Random walks/ biased random walks models such as DeepWalk [15], Node2Vec [16], and skip-gram models like Word2Vec are standard embedding methods involved in node-level tasks. LINE [17] is used for edge-level tasks for arbitrary graph types (directed, undirected and weighted). Graph Convolutional Network (GCN) [18] is a semi-supervised learning approach which uses convolutional operations on the graph relating nodes to their neighbours. Structural Deep Network Embedding (SDNE) [19] is for preserving the first-order (pairwise similarity between nodes linked by edges) and second-order (similarity of the nodes' neighbourhood structures) proximities. Graph2Vec [20] and GL2Vec [21] use Weisfeiler-Lehman kernels [22] in the graph for whole graph embeddings considering its iterations. An implementation of Spectral Features (SF) [23] and Family of Graph Spectral Distance (FGSD) [24] calculates the k lowest eigenvalues of the normalized Laplacian. If the graph has fewer eigenvalues than k, the representation is padded with zeros. This paper uses node and graph embedding techniques to evaluate the graphs using machine learning models.

1.2 Motivation

Over the years, Deep Learning (DL) has proven promising results and usually performs better than the traditional machine learning methods. Machine learning models using gradient boosting work better on tabular data and outperform neural network models. Also, transforming non-linear data, one of the critical features of deep learning, can be done by ML models such as Decision trees and Support Vector Machines (SVMs). Deep Learning (DL) uses the neuron layer, which allows parameter-sharing. In the case of Graph Neural Networks (GNNs) and Graph Convolutional Networks (GCN), it embeds an object into low dimensional space (convolutional layer), which is very complex for ML models. DL is very efficient in extracting features and replacing the complex method for feature selection, making it the right way to deal with non-tabular datasets.

Simulation models can be created using MATLAB and python using MATLAB Engine API and Simulink. Models created are then tested concerning some parameters to which it returns the output by analysing the model. The time and resources required to create and analyse the model were not optimal. Therefore, ideas were proposed if deep learning and machine learning could be applied to create the Simulink model into a low-dimensional vector space so that the machine can understand the correlations between the data and can analyse and predict the results requiring less time and resources as compared to analysing it in higher dimensional space. Graphs were introduced and best suited to create a simulation model and convert it into a low-dimensional space. Also, graph embedding techniques were used to maintain the integrity of the data such that no information is lost when converting it.

The objective of this dissertation was to create a graph (network) out of any simulation model and apply graph embedding techniques having all the properties and features of the object intact and then train the machine learning model to predict the target values. This approach can deal with large datasets in no time and with much less computing power. It can be used to analyse the simulation and other data models and will also help users to have better insights on graphs and further help to explore and innovate new techniques and algorithms in this field. Time comparison has been mentioned in the evaluation and results section 5.3.

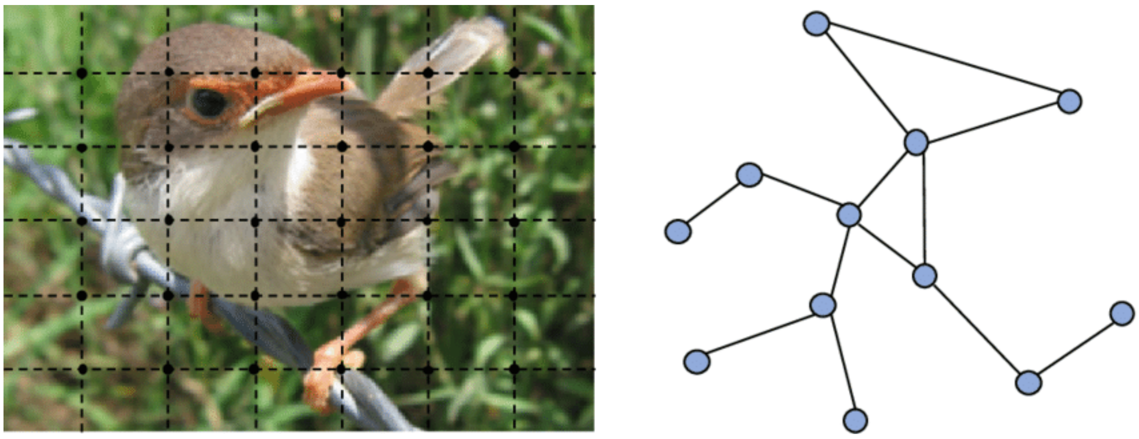


Figure 1.1: Image and graph in non-euclidean space
Note: Taken from [1]

1.3 Research Objectives

The main objective of this research was to organise the simulation models built using Simulink into a low-dimensional space vector that machines can understand and analyse, refer to figure 1.1. Graphs were used as a data structure to establish connections using nodes and edges satisfying different entities to the Simulink models. Inspired by convolutional neural networks (CNNs) and recurrent neural networks (RNNs), graph embedding techniques use a relationship model to handle the graphical representation of information and also use a method called "message passing". Graph embedding requires features to represent it in vector by using adjacency matrix/sparse adjacency matrix using weights, rankings, types, signs or properties associated with edges.

A graph is ubiquitous in real-world applications and is analysed as a deep learning method in ongoing research. This dissertation focuses on the different methods applied to different simulation models and investigates their variation based on different models. Some research questions are addressed while performing this investigation, such as:

- **RQ1 - Feasibility Study or Exploratory Research**: Is it possible to predict simulation models' outcomes with the machine learning approach?

  Within this, we will explore technical questions such as:

  - How to generate a graph, and what information is required? (answered in 2.1, 2.2)
  - How should data structures that have a graph structure rather than temporal or geographical structures be handled? (answered in 2.4, 2.5)
  - What libraries and frameworks are available to build a graph which can be further used by Graph Neural Networks (GNNs), and how are they implemented? (answered in 2.2)

- **RQ2 - Evaluation of a Particular Instance**: How good are the results of predictions, and what factors are responsible?

  Within this, we will explore technical questions such as:

  - What factors can affect graphs when training with machine learning models? (answered in 3.4, 6)
  - How well does the approach work in terms of accuracy? (answered in 5.2.2)

1.4 Challenges

While working on this research, many things were unfamiliar, and some challenges were faced.

1.4.1 Dataset Creation

- Dataset in euclidean form should be represented in graphical structure for applying graph embedding. Since graph embedding techniques can be applied to graphs, searching for a way to convert euclidean or tabular data to a graph was the first challenge.
- Finding a python framework, library or tool is one way of structuring graphs, and there are many to choose from, which are explained in section 2.2.

1.4.2 Graph Embedding

Procuring vector representation of graphs is a tedious and complex task and possesses challenges that are:

- The first challenge is to choose the feature and property of the graph that the embedding should preserve. Node properties and connection of edges should be preserved when converting into vectors. This decision can be challenging given the wide range of distance measures and features provided for graphs, and performance may vary depending on the application.
- Most networks (graphs) are extensive, containing millions of nodes and edges; therefore, embedding methods should be scalable enough to possess large graphs without losing the properties of the graphs.
- Dimension of the vectors can be hard to select. Finding optimal dimensions for graph embedding methods can considerably impact training an ML model. It can affect the model accuracy and prediction score.

1.4.3 Machine Learning

Challenges in machine learning can be:

- Identify the Data pre-processing tools.
- Finding correlation between data points to calculate variance and bias.

- Option to add cross-validation and hyper-parameter tuning for correlating data.
- Choosing optimal ML model for prediction.

1.5 Dissertation Structure

The current state of the art is outlined in Chapter 2, along with several tools and methodologies and the components that go into creating citation networks. In Chapter 3, the application's design is described along with the justifications for the choices taken. Chapter 4 provides a breakdown of the steps used to create the application. Chapter 5 provides evaluations and results of the application. Chapter 6 summarises what points can be discussed, limitations and threats of validity to the application. The seventh and last chapter summarises this fundamental research and its findings. It also discusses the future work that can be done to improve this research.

Chapter 2

State of the Art

This chapter gives an overview of the current state of the art and background on the obstacles and possibilities of applying graph embedding. The review begins with the dataset used to build the network, which is an essential component in any graph network. It mentions tools or libraries for structuring the graph and provides insights into graph models and their overall structure. Moreover, different tasks are mentioned that the graph can perform. At last, machine learning models are explained that will be used for training graph vectors.

2.1 Dataset

For creating a graph from simulation models to predict the results, firstly, datasets of the simulation models are required, which should correlate with them. Building a simulation model requires some tabular data stored in key-value pairs or rows and columns. Similarly, for creating a graph, these tabular data can be helpful and should be appropriately analysed to have some features and properties that can be used to build a network. Nowadays, organisations provide various APIs that use graph data to carry out their research and implementation. Dataset can also be created manually and downloaded from the internet to conduct this research.

2.1.1 Building a Dataset

Various tools and frameworks of computer languages can be required to build a graph dataset from tabular data that can be applied to Graph Neural Networks (GNNs) and other graph embedding techniques. Also, one should be creative enough to identify nodes, edges and features on which its structure will be based. Pre-built datasets can be found on the internet from sites such as Stanford, Kaggle, and many other organization websites such as meta, apple, amazon, who perform research on graphs with neural networks, and machine learning. Datasets can also be created manually with a vision to create a graph that requires all its parameters and properties.

2.2 Tools and Libraries

In python, creating a graph dataset from tabular data will requires libraries and tools such as Pytorch Geometric, NetworkX, Deep Graph Library (DGL), Graph Nets, and Spektral.

Library Name	License	Programming Language	Main Contributor(s)
Pytorch Geometric	MIT 6	Python, Pytorch	Matthias Fey
Deep Graph Library	Apache 2.0	Python, Pytorch, TF, MxNet	Distributed MLC
Graph Nets	Apache 2.0	Python, Pytorch	DeepMind
Spektral	MIT	Python, TF2/Keras	Daniele Grattarola
Networkx	BSD	Python	Aric Hagberg

Table 2.1: Python Libraries for graphs in Neural Networks

2.2.1 Pytorch Geometric

Pytorch [25] is an open source machine learning framework based on torch library and is used for applications like Natural Language Processing and Computer vision. PyG (Pytorch Geometric) [26] is a library built in Pytorch which is used for structuring and training graphs for graph embedding and graph neural networks (GNNs) for a wide range of applications on structured data.

It includes several recently published approaches from relational learning and 3D data processing in addition to general graph data structures and processing techniques. PyTorch Geometric utilizes sparse GPU acceleration, offers specialized CUDA kernels, and implements effective mini-batch management for input samples of various sizes to achieve excellent data throughput. It is made up of many techniques for deep geometric learning, also known as deep learning on graphs and other irregular structures, that are drawn from some academic publications. It also includes numerous standard benchmark datasets (based on accessible interfaces to create your own), the GraphGym experiment manager, and functional transforms for learning on arbitrary graphs and on 3D meshes or point clouds. It also supports multiple GPUs, DataPipe, distributed graph learning via Quiver [27], and many other features.

2.2.2 Deep Graph Library

Deep learning on graphs is made simple, high-performing, and scalable with the help of the Deep Graph Library (DGL) [28]. Because DGL is framework-independent, the remaining functionality in an end-to-end application that includes a deep graph model can be written using any famous framework, such as PyTorch, Apache MXNet [29], or TensorFlow [30].

DGL offers a robust graph object installed on the CPU or the GPU. For improved control, it combines structural data with characteristics. It offers many functions for working with graph objects, such as powerful and adaptable primitives for message passing in Graph Neural Networks.

2.2.3 Graph Nets

Graph-Nets [5] is a DeepMind's lower-level Graph Neural Network model and library. It is so flexible that it can be modified to work with Temporal Graphs and nearly any other existing GNN. It uses TensorFlow and Sonnet to build graph networks, refer to figure 2.1. Both the CPU and GPU versions of TensorFlow are compatible with the library. It enables the freedom to add Temporal Graphs and build nearly any current GNN using just six core functions. Even though it is only approximately three years old, it feels outdated because Graph Nets require TensorFlow 1.

A graph network receives a graph as its input and output. Edge (E), node (V), and global-level (u) properties are present in the input graph. The output graph's structure is the same, but its properties have been modified. The more prominent family of "graph neural networks" includes graph networks [31].

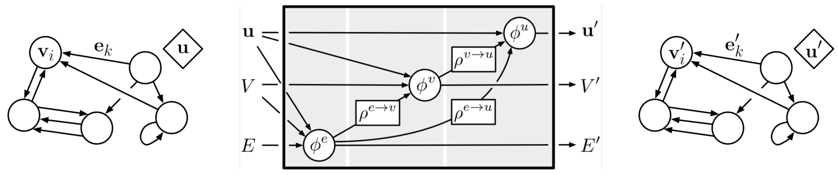


Figure 2.1: graph-nets implementation
Note: Taken from [2]

2.2.4 Spektral

Based on TensorFlow 2 and the Keras API, Spektral [32] is an open-source graph deep learning toolkit for Python. The primary objective of this library is to offer a straightforward, adaptable foundation for developing GNNs. To categorize users in a social network, predict chemical features, create new graphs using GANs, cluster nodes, forecast linkages, and any other activity where graphs represent data, utilize Spektral. It uses some of the most well-liked layer architectures for deep learning on graphs. According to Keras' founding ideas, this library is designed to be user-friendly for novices while retaining versatility for professionals. The poor training speeds for the majority of jobs as compared to competing libraries like DGL[28] and PyG [26] are the unfortunate trade-off for Spektral's ease of use.

2.2.5 NetworkX

NetworkX [33] is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. This package is used in this project to carry out the research, refer to subsection 3.2.1.

2.3 Definitions and Preliminaries

Graphs are general language for describing and analyzing entities with relations/interactions.

Definition 1: A graph G(V,E) is a collections of nodes $V = \{v_1, v_2, v_3, ..., v_n\}$ and edges $E = \{e_1, e_2, e_3, ..., e_n\}$. The adjacency matrix S of graph G contains non-negative weights associated with each edge: $s_{ij} \geq 0$. If v_i and v_j are not connected to each other, then $s_{ij} = 0$. For undirected weighted graphs, $s_{ij} = s_{ji} \forall i, j \in [n]$.

The edge weights s_{ij} defines the connection and similarity measures between the nodes v_i and v_j. The greater the weight of the edge, the strong connection and more similarity is expected to be between them.

Definition 2: (First-order proximity) It refers to local pairwise proximity between the vertices in the network. Probability between vertices v_i and v_j with undirected edge (i,j) is defined as:

$$p_1(v_i, v_j) = \frac{1}{1 + exp(-u_i^{-T}.u_j^{-})}$$

Whereas empirical probability is defined as:

$$p^(i, j) = \frac{w_{ij}}{W}$$

Definition 3: (Second Order Proximity) It is applicable for both directed and undirected graph, (for detailed definitions, paper [34]). Here, each vertex plays two roles: first is of vertex itself and second is specific context of other vertices with respect to it. For vertex v_i, two vectors are introduced u_i and u_j, where u_i is a representation of v_i when it is treated as a vertex itself and u_j when it is treated as specific context. For directed edge (i.j) probability of context v_j generated by v_i if defined by:

$$p_2(v_j|v_i) = \frac{exp(u_j^{-T}.u_i^{-})}{\sum_{k=1}^{|V|} exp(u_k^{-T}.u_i)}$$

where empirical formula is defined by:

$$p_1(v_i, v_j) = \frac{1}{1 + exp(-u_i^{-T}.u_j^{-})}$$

Definition 4: (Negative Sampling) It is the approach for sampling multiple edges according to some noisy distribution for each edge (i,j). Objective function for each edge is defined as:

$$p_2(v_j|v_i) = log\sigma(u_j^{-T}.u_i) + \sum_{i=1}^{K} E_{v_n \sim p_n(v)}[log\sigma(-u_n^{-T}.u_i)]$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. The first term models the observed edges, the second term models the negative edges drawn from the noise distribution and K is the number of the negative edges.

Definition 5: (Graph embedding) Given a graph G = (V, E), a graph embedding is a mapping $f : v_i \to y_i \in R^d, \forall i \in [n]$ such that $d \ll |V|$ and the function f preserves some proximity measure defined on graph G.

Conversion of higher dimensional data such as 3D model, images, audio, words to lower dimensional vector embedding is used. First order proximity in a embedding model can be defined by minimizing $\sum_{i,j} s_{i.j}||yi - yj||_2^2$. Suppose there are nodes having pairs (v_i, v_j) and (v_i, v_k), and connection strength is compared and found to be $s_{i,j} > s_{i,k}$. Therefore in the embedding space v_i and v_j will be mapped close to each other than v_i and v_k.

2.4 Algorithmic Approach

Over the years, considerable research has been done using the graph embedding approach to design embedding algorithms using various graph features. The focus was to stimulate the features reasoned to activities and connections in the networks (graphs), as sometimes algorithms can vary according to the type of connection to be built. In today's world, social media platform is a success in connecting to the outer world and in this platform type of connections can vary related to the distance between two users, everyday things between users, places users have visited, likes and dislikes. These things can help recommend a user to an individual.

Graphs have an arbitrary size and complex topological structure, i.e., no spatially locality like grids, no reference point and are dynamic and have multimodal features, paper [35] divides the body of research on graph attention models into three categories depending on the types of tasks (node/edge or graph level), attention type (similarity or weight-based), and issue settings (the kind of input and output). There are different levels of tasks done by the graph that can be applied to different applications-

- **Node Classification-** This task is used where we are trying the properties of the nodes. *Example-* Categorize online user or any item
- **Link Prediction-** We try to predict whether there are missing links between a pair of node. *Example-* Knowledge graph completion.
- **Graph Classification-** We categorise different graph and predict its properties. *Example-* Represent molecules as graph and predict properties of different molecules or drugs.
- **Clustering-** Here Goal is to identify closely linked sub-parts of graphs where nodes are densely connected with each other. *Example-* Social circle detection.
- **Other Tasks-**
  - Graph generation- drug discovery
  - Graph evolution- physical simulation

2.5 Types of Graphs

There are numerous types of graphs that represent unique forms and require particular GNN designs. Each type of graph possesses different challenges in graph embedding. Particular graph neural networks and graph embedding methods modelled for specific graph types are highlighted here:

2.5.1 Homogeneous Graph

A homogeneous graph can be defined as the graph in which nodes and edges are of a single type only, i.e., every node has the same properties, and every edge defines the same type of connection. These graph types can be undirected or directed graphs. Graphs with unidirectional unweighted edges are the most common and basic graphs. More research and studies can be found on this type of graph, e.g., [36], [37]. All nodes and edges are treated equally as these graphs present common relationships, e.g., college representing courses and passenger transport facilities. These type of graphs does not present a relationship of any magnitude. Unidirectional graphs also presents bidirectional relationship between the nodes as unidirectional edge $e_{i.j}$ represents relation of v_i to v_j and also v_j to v_i.

Weights added to the edges can provide more information on how strong the connection is between two nodes and can help represent it more accurately in low-dimensional vector space. Directional graphs meanwhile represents unidirectional relationship between two nodes as directional edge $e_{i.j}$ represents relation of v_i to v_j and not v_j to v_i. Directional graphs can be used in various applications, e.g., social media platforms defining followers and following.

Challenge: The difficulty is capturing the variety of connection patterns seen in graphs. Since homogeneous graphs only contain structural information, the difficulty of homogeneous graph embedding is to figure out how to maintain the connection patterns seen in the input graphs while embedding.

2.5.2 Heterogeneous Graph

A heterogeneous graph can be defined as a graph in which nodes and edges can be of multiple types, i.e., nodes can be of multiple types. They can have different properties, and edges can have more than one relationship with different nodes.

Knowledge graphs An example can be a knowledge graph based on a movie constructed from freebase [38]. There can be different types of entities (nodes) such as *actor, producer, director* and relations (edges) can be *acted in, produced by, directed by*

Another example can be **Multimedia graphs** having different types of entities (nodes) such as images and texts. In both [39] and [40], graphs with two types of nodes (image and text) and three types of links are embedded (the co-occurrence of image-image, text-text and image text). [41] does a social curation, including user and image nodes. It takes advantage of user-image linkages to place persons and images together so they can be directly contrasted for image suggestion.

Challenge: Heterogeneous graph embedding involves embedding various object types (such as nodes and edges) into the same space. The issue is how to investigate the global consistency between them. Additionally, there can be an imbalance between objects of various sorts. When embedding, this data skewness should be taken into account.

2.5.3 Auxiliary Informative Graphs

This type of graph contains auxiliary information of node/edge/whole-graph and connectivity between the nodes. Additional information can be of different types explained below:

- **Label**: When embedding a graph in a low dimensional plane, nodes with different labels are embedded far from each other [42]. Moreover, [43] explains the classifier and embedded objective functions. In [44], the entity categories are organized hierarchically. For example, the category ”book” has two subcategories: ”author” and ”written work,” which creates a more complex knowledge graph.

- **Attribute**: In this type of graph, attribute values can be discrete, continuous, or both. A graph containing a discrete node attribute value (such as the atomic number of a molecule) is embedded in [45]. Discrete and continuous values for nodes and edges can be seen in [46].

- **Node Features**: In this type of graph, mostly node features are provided in texts. The features are then further processed to extract feature vectors using strategies like bag-of-words, topic modelling, or considering ”word” as a particular sort of node described in [47], [48], [49]. Images as node features are also possible [50]. Node features provide unstructured information, which is very informative in the vectors generated by graph embedding.

- **Information Propagation**: One example of this type of graph can be ”retweet” on Twitter. From the given network G = (V, E), another network $G^c = (V^c, E^c)$ is constructed for every other network c [51], where V^c is the node that has been adopted from V and E^c is the edge that corresponds to nodes V and V^c. Topo-LSTM [65] views a cascade as a dynamic directed acyclic graph for embedding rather than just a list of nodes.

- **Knowledge Base**: Wikipedia [52], Freebase [38], YAGO [53], DBpedia [54], etc, are most commonly known example of knowledge graphs. Wikipedia [52] uses text articles as links between entities as suggested by users.

Other Auxiliary information can be user check-in data (user-location) [55] and user item preference ranking list [56]. Node contents and labels are used by [57] to support the graph embedding procedure.

Challenge: In addition to structural graph information, the auxiliary information aids in the definition of node similarity. Using these two information sources to specify the node similarity to be kept is the difficult part of embedding a graph with auxiliary information.

2.5.4 Graph Constructed from Non-Relational Data

Here, the graph will be constructed from the data in a low-dimensional form. Input is in the form of feature matrix $X \in R^{|V|xN}$, with each row, X_i, representing an N-dimensional feature vector for the i^{th} training instance. Then the similarity between X_i and X_j is constructed using similarity matrix S. One way is to build an adjacency matrix A based on euclidean distance and not centrality or any property of neighbouring nodes. Another way is to consider neighbouring nodes using the K Nearest Neighbours (KNN) graph. In [58], the graph calculates the geodesic distance in adjacency matrix A. KNN graph is constructed using similarity matrix, and the shortest distance is calculated.

Challenge: How to compute the relationships between the non-relational data and generate such a graph is the first difficulty confronted by embedding graphs formed from non-relational data. The difficulty now is maintaining the node closeness of the built graph in the embedded space, which is the same as other input graphs after the graph has been constructed.

2.6 Machine Learning Approach

In order to understand how this project exploits the power of machine learning and graph embeddings to understand the correlation between various dimension of rooms present in the house and their interconnections between them to accurately predict the cost of air conditioning in an house it is pertinent to have basic knowledge of various machine learning algorithms including Linear Regression, Decision trees (Random forests, Adaboost), and Neural Networks (Multi-Layer Perceptron (MLP)).

2.6.1 Linear Regression

Linear Regression [59] is one of the most basic machine learning algorithms, which is governed by two parameters, Weights and bias, through the equation target Variables

= Weights*(Input variables) + bias. It aims to estimate accurate values of weights bias based on which good predictions can be made with such weights in an iterative process. In Linear Regression, the first weights are initialized a random manner. These randomly initialized weights are then used to make a prediction. The predicted values are then compared to the actual target variable using a loss function. Depending on the loss, function values of weights are modified following optimization algorithms such as gradient descent.

Linear Regression is usually the first algorithm which is applied. It is an algorithm with low computational overhead; however, due to its simplicity, it sometimes fails to grasp the complex trends in data.

2.6.2 Lasso/Ridge Regression

Sometimes regression models overfit the training data, i.e., the model trained performs exceptionally well on training data but fails on unseen training data. Overfitting usually occurs when the model is provided with many features. To solve the problem of overfitting, there are various regularization techniques. The two most popular regularization techniques are lasso and ridge regularization. In Lasso regression in the loss function, a penalty factor is added. The penalty factor consists of a penalty parameter lambda multiplied by the summation of the absolute value of weights. Due to this penalty factor, the algorithm actively tries to keep the weights of features low. In ridge regression, the penalty factor is composed of a penalty parameter lambda multiplied by the summation square of weights. The main difference between lasso and ridge regression is that lasso imposes a more strict penalty and can make the weights of some features zero.

2.6.3 Decision Trees

Decision trees [60], [61] are a type of machine learning algorithm which splits the given dataset on the given feature until all sets are homogeneous; that is, they belong to the same class. The choice of which features to split on is taken using various methods, including splitting on cross-entropy. Entropy is a measure of randomness [62]. The goal of a decision tree is to reduce randomness present in data that increases homogeneity. Thus at each node of a tree, we split the data points left at that node on a feature that leads to a maximum drop in entropy. Another metric on which a split can be done is Information gain which is the basic subtraction of weighted entropy of the condition of data points after the split from before the split. We thus want to maximize the gain in information at each split. Gini Index is another metric on which split decisions can be made. Gini Index is the addition of the square of probabilities of each class subtracted from one. Gini Index is favoured as a split method for more significant distributions of data, while information gain is preferred when there are multiple features to split on. Decision trees can also be applied for regression problems. When decision trees are used as regressors, the only thing different in regression trees is how the feature on which split is done is selected. The Sum of squared errors is the criteria used in the regression of decision trees. Decision trees are easily prone to overfitting. A common strategy used to prevent overfitting is pruning. The pruning strategy stops data splitting when a certain threshold is reached.

2.6.4 Random Forest

Random Forest [63] is an ensemble machine learning method that leverages decision trees' bagging to produce better results. Bagging is based on the idea that weak meta learners working in tandem produce robust predictions. In random forest, the same is implemented. A critical hyperparameter for a random forest is the number of decision trees that a random forest will train. Random Forest works by training several decision trees on different versions of the same dataset. Each tree is trained on a different version of the dataset. Such different versions are created by randomly sampling data points from the dataset without replacement. It solves the critical flaw of overfitting in decision trees prone to high variance. However, Random forest also leads to a loss of interpretability decision trees offer.

2.6.5 AdaBoost

AdaBoost [64] is boosting mechanism. Boosting is a mechanism that concentrates on predicting solid models by ensembling various week predictors. AdaBoost achieves this by first assigning equal weights to all data points. Then various strategies are employed to give increased weight to misclassified data points. On this new unevenly weighted dataset, a new model is trained, which learns better to classify the data points misclassified earlier. This way, a new model is built sequentially to get more robust predictions. In a simplistic sense, AdaBoost decreases bias and variance in the learners. Variance is reduced by using shallow trees, and bias is reduced by using multiple trees.

2.6.6 MLP Regressor

Multi-layer perceptron (MLP) [65] is a fully connected neural network with three layers. Each layer in a neural network is made of nodes. A connection between two nodes has a particular weight associated with it. Inputs are fed into the first layer. The number of features in the input vector equals the number of nodes. The nodes then transfer the inputs to all the nodes of the next layer by multiplying them by the weights of the connection. This process repeats until the output layer, where the predictions are made. The predictions made are then compared to an actual value, and the loss value is calculated according to this loss value. The algorithm of backpropagation corrects the weights.

2.7 Conclusion

The above sections in this chapter focused on the similar work done by authors on finding different approaches to graph embedding techniques which different graph embedding methods can implement. The literature review gave critical insights that helped in completing the research objectives1.3 and also stated the future works 7.2 which can be done.

Chapter 3

Design

This chapter gives an overview of the approach taken to do this research. It highlights the structure, flow and design, starting with collecting and retrieving data and pre-processing it. Then it explains the tools and methods for graphs to work with machine learning models.

3.1 Dataset

This application/project can work on any data represented in networks with nodes and edges and features attached to them. Dataset used here is manually designed, which was used for building simulation models using MATLAB and python. This dataset is household data with information represented in key-value pairs stored in JSON (JavaScript Object Notation) files. Information used in household data is explained in the table 3.1.

There were around 10000 graphs made for simulation, which were also used for struc-

Variable name	Description	type
name	type of room	string
length	length of the room	float
width	width of the room	float
specific port	connectivity of the room	string
building length	length of the building	float
building width	width of the building	float
building height	height of the building	float
links (source)	connectivity from the type of room	string
links (target)	connectivity to the type of room	string
is ac	one way connectivity	boolean

Table 3.1: Dataset variables, descriptions and types

turing the graph for neural networks. Networkx [33], which is a python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks, is used here for converting the JSON files into networks. This python package uses simple python syntaxes and can be used to convert any form of tabular data into networks.

3.2 Data Operations

Python is easy to use and has built-in libraries and frameworks for developing complex applications or writing code. Handling data, Machine Learning, and Deep Learning all this methodology are performed and widely used with python.

A python script was used to create a data frame from JSON files created as key-value pairs. Then, for converting this tabular data into graphs, various python libraries and tools use simple functions and methods for converting. NetworkX [33] is used here for the conversion of data into graphs. Various graph embedding techniques are developed and categorised based on the node, edge, cluster(subgraph) and whole graph for converting graphs into a low dimensional vector the machine understands.

There were other libraries/packages for structuring and analysing graphs, such as Pytorch Geometric [25], Deep Learning Graphs [28] and Spektral [32], but using them was more strenuous than using NetworkX. Also, different pre-requisite packages were required to install on the machine for every other library to get installed. Also, even after that, it did not provide simplicity and a less obstructive way of implementing them.

3.2.1 NetworkX

NetworkX [33] is a python based framework used for network analysis. NetworkX graph nodes may be any Python object, and edges can carry arbitrary data; this flexibility makes NetworkX perfect for describing networks seen in a variety of scientific subjects.

It uses simple python syntaxes for constructing a graph and has all the functions for defining its properties related to nodes, edges, and graphs, allowing both brief and unambiguous network algorithm formulations as well as straightforward and adaptable network representations. NetworkX employs the dynamic and expanding ecosystem of Python packages to offer additional capabilities like sketching and numerical linear algebra explained in table 3.2.

Networkx Class	Type	Self-loops allowed	Parallel edges allowed
Graph	undirected	Yes	No
DiGraph	directed	Yes	No
MultiGraph	undirected	Yes	Yes
MultiDiGraph	directed	Yes	Yes

Table 3.2: Compatibility of graphs in NetworkX

3.3 Karate Club

Karate Club [66] is an unsupervised machine learning library for NetworkX built upon existing open source libraries for linear algebra, machine learning, and graph signal processing, including Numpy, Scipy, Gensim, PyGSP, and Scikit-Learn. It offers node and graph-level network embedding methods and incorporates a range of community detection techniques, both overlapping and non-overlapping. Implemented techniques are drawn from several conferences, seminars, and articles from prestigious publications in the fields of network science (NetSci, Complenet), data mining (ICDM, CIKM, KDD), artificial intelligence (AAAI, IJCAI), and machine learning (NeurIPS, ICML, ICLR).

3.3.1 Graph Level Tasks in Karate Club

Various tasks can be performed with the graph, and various embedding techniques are associated with every level of tasks provided by the karate club community. Graph level tasks are explained below with provided techniques:

- **Overlapping Community Detection**: Ego-Splitting Framework [67], Deep Autoencoder-like Nonnegative Matrix Factorization (DANMF), Non-negative Symmetric Encoder-Decoder (NNSED)

- **Non-Overlapping Community Detection**: Graph Embedding with Self Clustering (GEMSEC), Edge Motif Clustering (EdMot)

- **Neighbourhood-/Based Node Embedding**: Geometric Laplacian Eigenmap Embedding (GLEE), Diff2Vec, Network Embedding as Matrix Factorization (NetMF), DeepWalk, Node2Vec

- **Structural Node Embedding**: GraphWave, Role2Vec

- **Attributed Node Embedding**: Attributed Social Network Embedding (ASNE), Fusing Structure and Content via Non-negative Matrix Factorization for Embedding Information Networks (FSCNMF), Binarized Attributed Network Embedding Class (BANE)

- **Meta Node Embedding**: NEU- Fast Network Embedding Enhancement via High Order Proximity Approximation
- **Whole Graph Embedding**: Graph2Vec, GL2Vec, FGSD, NetLSD, IGE (Invariant Graph Embedding)

3.4 Methods for Machine Learning in Graphs

Graph embedding is used to convert graph data into vectors that machine learning models can use to train themselves. Prediction using ML models can be node level, link prediction, subgraph (clusters) or whole graph level. Since one specific output (fan cost) was generated for one graph model, whole graph level prediction was suitable for this problem. Also, node level prediction was used, and the idea was to divide the output (fan cost) among every node based on node features and the type of connections between them.

ML pipeline requires:

- Design features for nodes/links/graphs
- Obtain features for all training data

3.4.1 Node Level Prediction

Features for node-level prediction task used by DeepWalk (section 4.4.1) and Node2Vec(section 4.4.2) are:

Node Degree

- The degree k_v of node v is the number of edges (neighboring nodes) the node has.
- Treats all neighboring nodes equally. Refer to figure (3.1)

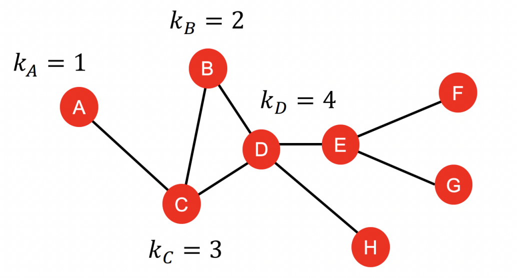


Figure 3.1: Example of Node Degree feature of node
Note: Taken from [3]

Node Centrality

- **Eigenvector Centrality**: A node v is important if surrounded by important neighboring nodes $u \in N(v)$. Centrality of node v is modelled as the sum of the centrality of neighboring nodes:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

  λ is normalization constant

- **Betweenness Centrality**: A node is important if it lies on many shortest paths between other nodes. Refer to figure (3.2)

$$c_v = \sum_{s \neq v \neq t} \frac{\textit{shortest path between } s \textit{ and } t \textit{ that contains } v}{\textit{shortest path between } s \textit{ and } t}$$

  .

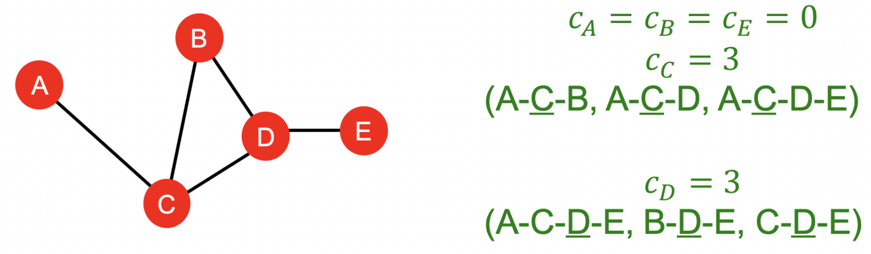

$c_A = c_B = c_E = 0$
$c_C = 3$
(A-$\underline{\text{C}}$-B, A-$\underline{\text{C}}$-D, A-$\underline{\text{C}}$-D-E)
$c_D = 3$
(A-C-$\underline{\text{D}}$-E, B-$\underline{\text{D}}$-E, C-$\underline{\text{D}}$-E)
A
B
C
D
E

Figure 3.2: Example of Betweenness Centrality feature of node
Note: Taken from [3]

- **Closeness Centrality**: A node is important if it has small shortest path lengths to all other nodes. Refer to figure (3.3)

$$c_v = \frac{1}{\sum_{u \neq v} \textit{shortest path between } u \textit{ and } v}$$

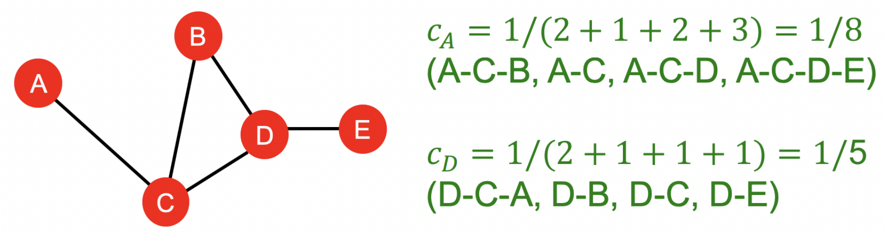

$c_A = 1/(2+1+2+3) = 1/8$
(A-C-B, A-C, A-C-D, A-C-D-E)
$c_D = 1/(2+1+1+1) = 1/5$
(D-C-A, D-B, D-C, D-E)
A
B
C
D
E

Figure 3.3: Example of Closeness Centrality feature of node
Note: Taken from [3]

- **Clustering Coefficient**: Measures how connected v's neighboring nodes are (Refer to figure 3.4):

$$e_v = \frac{edges\ among\ neighbouring\ nodes}{neighbouring\ nodes\ to\ node\ v\ (k_v)/2}$$

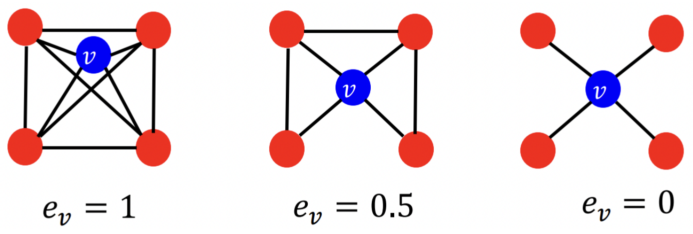


Figure 3.4: Example of Clustering Coefficient feature of node
Note: Taken from [3]

- **Graphlets**: Similar to clustering coefficient, Graphlets are small subgraphs that describe the structure of node u's network neighborhood. Refer to figure 3.5

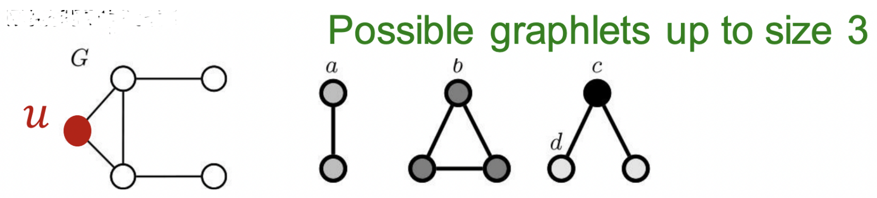


Figure 3.5: Example of graphlet feature of node
Note: Taken from [3]

3.4.2 Link Predictions and Features

The task is to predict new links based on the existing links. At test time, node pairs (with no existing links) are ranked, and top K node pairs are predicted. The key is to design features for a pair of nodes.

Link Level features are:

- **Distance-Based Feature**: Shortest distance between two nodes and it doesn't capture the degree of neighborhood overlap. Refer to figure 3.6

- **Local Neighbourhood Overlap**: Captures neighboring nodes shared between two nodes v_1 and v_2. Becomes zero when no neighbor nodes are shared. Refer to figure 3.7

  - **Common Neighbours**: $|N(v_1) \cap N(v_2))|$

    Example: $|N(A) \cap N(B))| =$ —C— $= 1$

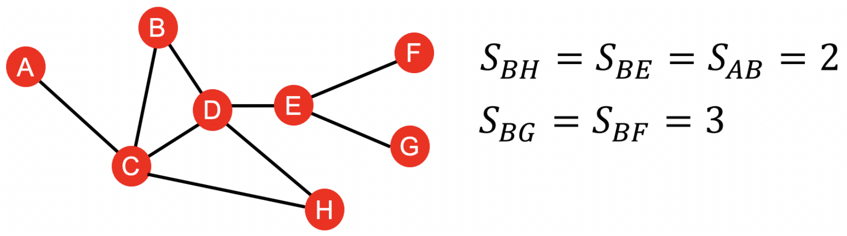


Figure 3.6: Example of shortest distance between two nodes
Note: Taken from [3]

- **Jaccard's Coefficient**:

$$\frac{|N(v_1) \cap N(v_2))|}{|N(v_1) \cup N(v_2))|}$$

Example: $\frac{|N(A) \cap N(B))|}{|N(A) \cup N(B))|} = \frac{|C|}{|C,D|} = \frac{1}{2}$

- **Adamic-Adar Index**:

$$\sum_{uinN(v_1) \cap N(v_2)} \frac{1}{logk_u}$$

Example: $\frac{1}{logk_c} = \frac{1}{log4}$

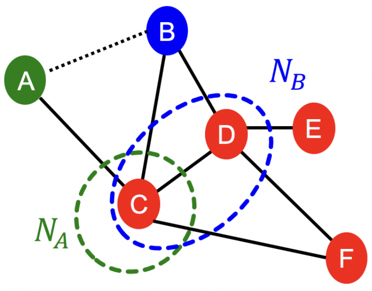


Figure 3.7: Example of Local Neighbourhood Overlap
Note: Taken from [3]

- **Global Neighbourhood Overlap**: Uses global graph structure to score two nodes. Katz index counts walks of all lengths between two nodes. Katz index is explained as :

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1 v_2}$$

3.4.3 Graph Level Features

Kernel methods are widely-used for traditional ML for graph-level prediction. They measure similarity between two graphs.

- **Graphlet Kernel**: Graph is represented as Bag-of-graphlets. It is computationally expensive. Given two graphs, G and $G^{'}$, graphlet kernel is computed as:

$$K(G, G') = f_G^T f_{G'}$$

- **Weisfeiler-Lehman Kernel**: It uses neighborhood structure to iteratively enrich node vocabulary and provide generalized version of Bag of node degrees since node degrees are one-hop neighborhood information. It is computationally efficient. Implementation of this can been seen in section 4.4.3.

3.5 Overview of the Approach

Below figure 3.8 explains the overview of the approach used to implement this research.

- **Data Preprocessing**: This block shows filtering out the JSON files with fan cost value (target value) calculated to train ML models. Data preprocessing is essential for Machine Learning, as input variables and target values are required for prediction. Implementation is explained in section 4.3
- **Conversion of JSON Data into Graphs**: Before applying any graph embedding techniques to the simulation model data, it should be represented in graphs. In this step, JSON data is converted into graphs with preserving important features and properties of the nodes and edges. NetworkX, a python framework, is used to do this conversion. Implementation is explained in section 4.3
- **Implementing Graph Embedding**: In the third block, using graph embedding methods, graphs are converted and represented in vectors. For training any graph with ML models, converting them in low-dimensional form is required. Implementation is explained in section 4.4
- **Applying Machine Learning**: In the last step, using graph machine learning models, vectors with the associated fan cost value (target values) are trained and is used for prediction. Calculating mean square error and r2 score for highlighting the best model used for prediction. Implementation is explained in section 4.5

3.6 Conclusion

This section highlights the concepts, tools and methods used in the implementation. It also explains a high-level overview of the entire flow of the system. In the next section,

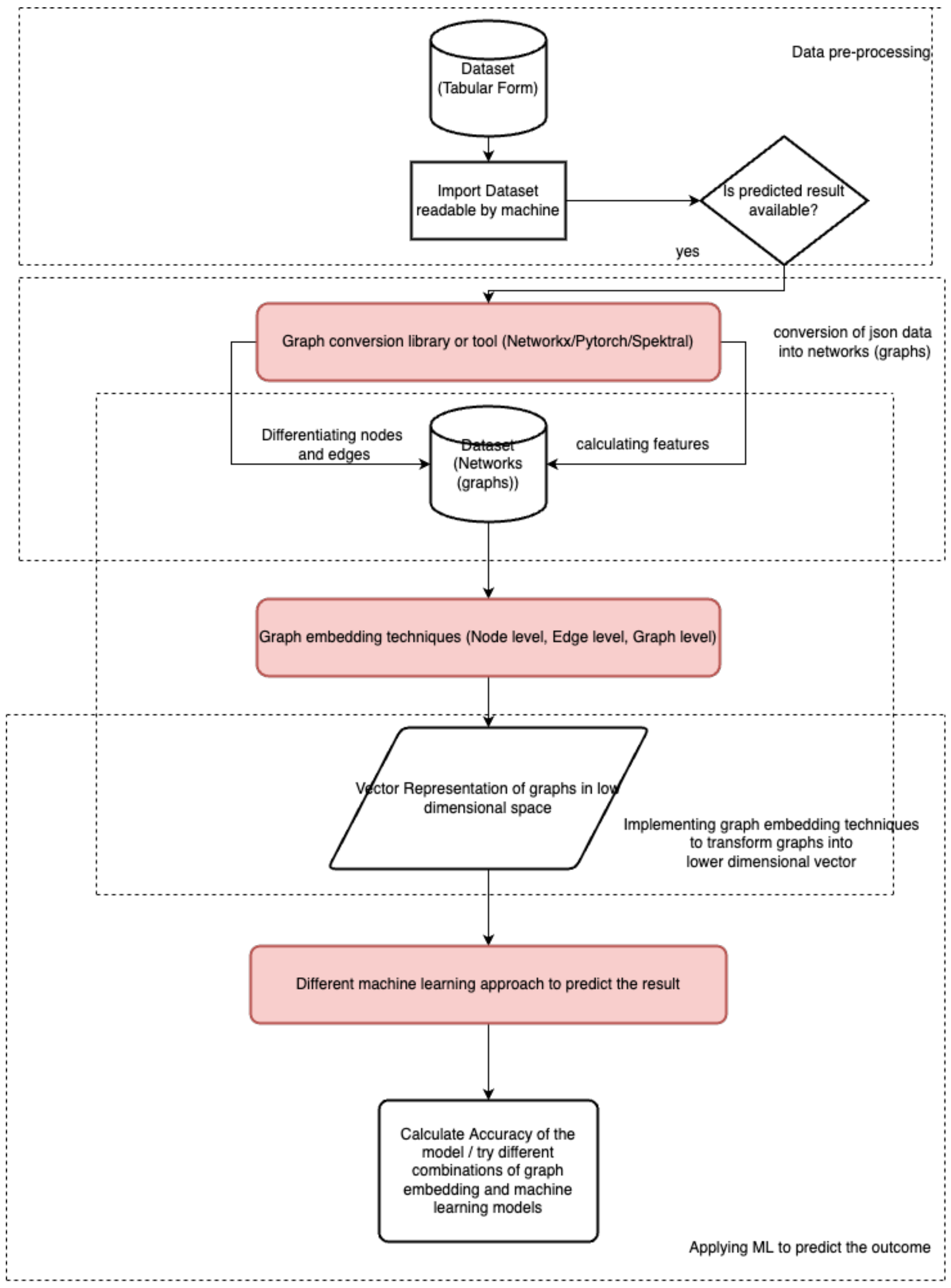

Data pre-processing
Dataset (Tabular Form)
Import Dataset readable by machine
Is predicted result available?
yes
Graph conversion library or tool (Networkx/Pytorch/Spektral)
conversion of json data into networks (graphs)
Differentiating nodes and edges
Dataset (Networks (graphs))
calculating features
Graph embedding techniques (Node level, Edge level, Graph level)
Vector Representation of graphs in low dimensional space
Implementing graph embedding techniques to transform graphs into lower dimensional vector
Different machine learning approach to predict the result
Calculate Accuracy of the model / try different combinations of graph embedding and machine learning models
Applying ML to predict the outcome

Figure 3.8: flow diagram of the approach used

we will see a detailed implementation.

Chapter 4

Implementation

4.1 Introduction

This section discusses how the dissertation implementation was carried out and what vital code snippets were used in this thesis to get the expected outcome. There were multiple python libraries used in the implementation of this thesis. The first package was the NetworkX package, about which the use is discussed in section 4.3. The other package we used was NumPy for matrix multiplications and other matrix operations in just a line of code. It also helps to do trigonometric calculations with its allowance of using different functions such as sin, cos, and the degree to radians. Further, the Pandas package was used during data collection and extraction. Pandas helped to read the CSV file into a data frame, and it also helped to clean the dataset by its default functions. The Scikit-Learn [68] package was also used, which is an open-source machine learning library that supports supervised and unsupervised learning and provides various tools for model fitting, data pre-processing, selection, evaluation, and many other utilities. To plot the estimated values of the kernel on a three-dimensional sphere. Packages like JSON were used to read and manipulate the JSON files and the Matplotlib library of python to create static, animated and interactive visualization. At last, machine learning models were used from the Scikit-Learn library, such as Linear Regression, Ridge, Lasso, Random Forest Regressor, AdaBoost, MLP Regressor and SVM, for training and prediction of the data.

4.2 Tool for Code Execution

Here, Google Colab is used for executing the code for this research as it is a free tool for executing python code and is suitable for executing Machine Learning and data analysis.

It is similar to a Jupyter notebook and allows for creating and sharing computational files without installing or downloading anything. It also provides runtime GPUs and TPUs for faster execution of code compared to machines as it uses the computing power of google servers. Python script execution frequently demands a lot of processing power and might take some time. While running Python programs, the performance of the local PC will not suffer.

Also, one can create a python environment on a personal computer and install the required package to run the code. Many available platforms have inbuilt packages and tools for python and R (data science) that aim to simplify package and environment management, such as **Anaconda, Miniconda, Miniforge and Mambaforge**.

4.3 Data Preprocessing

As discussed in section 3.1, tabular data in key-value pairs stored in JSON files should be converted to graphs with the property of nodes and edges intact. **Household data** is used here where different rooms and entrances define the nodes, and the property of the nodes are length, width and specific ports. A specific port defines whether the room has outflow, inflow or both (Mixed) concerning the fan's airflow.

Edges define the connection between the rooms as the rooms are isolated; therefore, **node centrality** features are not used for defining the connection. Features are defined by **node degree, Graphlet and Weisfeiler-Lehman Kernels and Local neighbourhood overlap**

4.3.1 Data Extraction and Conversion

Matching JSON and CSV Datasets

Here, in the code snippet 4.1, all the required packages are imported, and `path_to_json` is pointing to the JSON files which are stored in a given path folder. Because Google Colab is used here, all the files are stored in a drive in the *Dataset* folder.

Then there is another CSV file with all the values fan costs associated with every household, refer to figure 4.1. By comparing the JSON file names (e.g., household0001.json) with the CSV file value stored in the *name* column (household0001), JSON files are filtered out, and only the fan cost value associated with it is taken.

In the code snippet 4.2, the structure of the JSON file is mentioned, which has key-value pairs built into a graph structure.

```python

import os, json
import pandas as pd
import numpy as np
import glob
import math
import json

#path to json files
path_to_json = '/content/drive/MyDrive/Dataset/'

#reading from csv file that has fan cost values with respect to houses
dataset = pd.read_csv('/content/results.csv')
X = dataset.iloc[:,0].values
y = dataset.iloc[:,1].values

file_list = []
for x in X:
    #storing json files in a list that are available in csv file
    file_list+=glob.glob(path_to_json + x + '.json')

#reading from the files
for i in range(len(file_list)):
  data = pd.read_json(file_list[i], lines=True)
```

Listing 4.1: Data Conversion

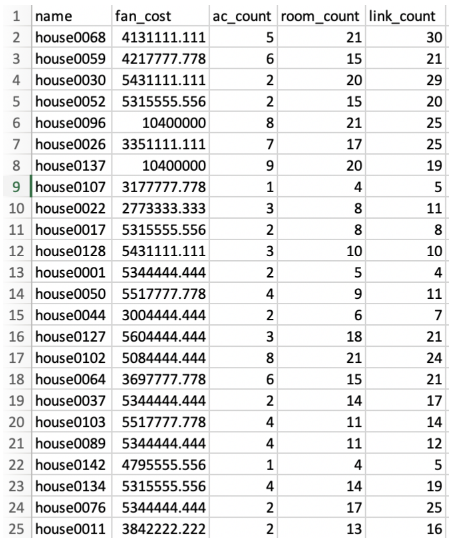
```json

{
  "building height": 2.5,
  "rooms": [
    {
      "name": "entrance",
      "length": 2,
      "width": 2,
      "x": 1,
      "y": 1,
      "specific port": "Door"
    },
    {
      "name": "hallway",
      "length": 2,
      "width": 38,
      "x": 3,
      "y": 1,
      "specific port": "Mixed"
    },
    {
      "name": "office1",
      "length": 18,
      "width": 40,
      "x": 1,
      "y": 3,
      "specific port": "Internal"
    }
  ],
  "building length": 20,
  "building width": 40,
  "name": "house0041",
  "links": [
    {
      "source": "hallway",
      "target": "entrance"
    },
    {
      "source": "office1",
      "target": "hallway"
    },
    {
      "is_ac": true,
      "source": "office1",
      "target": "hallway"
    }
  ]
}
```

Listing 4.2: JSON for a household

Converting JSON to Graph Dataset

Describing the features of the JSON file, rooms object are built into nodes having length, width, and specific port as node attributes or properties, and the edges between the rooms are built using the links object of the file having all the links between the rooms. **is_ac** node refers to the fan cost being installed, taking air from the source node and passing it on to the target node.

Referring to code snippet 4.3, NetworkX and torch packages are imported to draw the graph, and the Matplotlib library is there to plot it. `DiGraph()` class in the NetworkX package to draw the directed edges between the nodes. Since NetworkX does not support



1	name	fan_cost	ac_count	room_count	link_count
2	house0068	4131111.111	5	21	30
3	house0059	4217777.778	6	15	21
4	house0030	5431111.111	2	20	29
5	house0052	5315555.556	2	15	20
6	house0096	10400000	8	21	25
7	house0026	3351111.111	7	17	25
8	house0137	10400000	9	20	19
9	house0107	3177777.778	1	4	5
10	house0022	2773333.333	3	8	11
11	house0017	5315555.556	2	8	8
12	house0128	5431111.111	3	10	10
13	house0001	5344444.444	2	5	4
14	house0050	5517777.778	4	9	11
15	house0044	3004444.444	2	6	7
16	house0127	5604444.444	3	18	21
17	house0102	5084444.444	8	21	24
18	house0064	3697777.778	6	15	21
19	house0037	5344444.444	2	14	17
20	house0103	5517777.778	4	11	14
21	house0089	5344444.444	4	11	12
22	house0142	4795555.556	1	4	5
23	house0134	5315555.556	4	14	19
24	house0076	5344444.444	2	17	25
25	house0011	3842222.222	2	13	16

Figure 4.1: Sample of csv file used to train ”fan cost” as target value

graphs with mixed edges (directed and undirected), to acquire compatibility of both the edges directed graph with parallel edges was taken to implement this.

Also, the name attribute in the rooms object was added as a node using the `add_node` function with the required properties of length, width and a specific port. Also, the nodes were classified based on `specific port` to differentiate between the type of rooms the node specifies and labels are coloured accordingly.

```python

import networkx as nx
import torch
import matplotlib.pyplot as plt

#imported Directed graph from networkx
G = nx.DiGraph()
labels = []
#adding nodes to graph
for i in range(len(data.rooms[0])):
    G.add_node(data['rooms'][0][i]['name'], length=data['rooms'][0][i]['length'], width=data['rooms'][0][i]['width'])
    if 'specific port' in data['rooms'][0][i]:
        #adding labels as colors to identify nodes
      nx.set_node_attributes(G, {data['rooms'][0][i]['name']: data['rooms'][0][i]['specific port']}, name="specific_port")
      if data['rooms'][0][i]['specific port'] == "Mixed":
        labels.append('red')
      elif data['rooms'][0][i]['specific port'] == "Internal":
        labels.append('green')
      elif data['rooms'][0][i]['specific port'] == "Door":
        labels.append('orange')
    else:
      labels.append('blue')
    print(G.nodes[data['rooms'][0][i]['name']])

G.nodes(data=True)
```

Listing 4.3: JSON to graph conversion

In code snippet 4.4, edges are being added using `add_edge` function requiring parameters such as source node, target node.Weight can also be specified inside this function to give more information to the graph model. Connection between nodes are taken from the `links` object mentioned in the JSON file 4.2, having source and target entities as source node and target node respectively. Also, if attribute `is\_ac` is true for the connection, there is only a directed edge from source to target. Otherwise, a two-way connection between source and target is referred to as an undirected edge.

```
#adding edges to graph
for i in range(len(data['links'][0])):
    if('is_ac' in data['links'][0][i]):
      G.add_edge(data['links'][0][i]['source'], data['links'][0][i]['target'], weight=0.5, relation="ac")
    else:
      G.add_edge(data['links'][0][i]['source'], data['links'][0][i]['target'], weight=0.5, relation="notac")
      G.add_edge(data['links'][0][i]['target'], data['links'][0][i]['source'], weight=0.5, relation="notac")

G.edges(data=True)

# layout_pos = nx.spring_layout(G)
nx.draw(G, with_labels=True, node_color=labels)
plt.show()
```

Listing 4.4: JSON to graph conversion

While plotting graphs, Matplotlib is used, along with `draw()` function by NetworkX providing parameters such as `with_label` and `node_color` which helps to plot the graph with the label name and color specified above.

Converting String Labels into Integers

```
1 nx.convert_node_labels_to_integers(G, first_label=0, ordering='sorted',
     label_attribute="node_type")
2 mapping = dict(zip(G, range(len(G.nodes))))
3 G = nx.relabel_nodes(G, mapping)
4 sorted(G)[:3]
```

Listing 4.5: Converting node labels from strings to integers

String labels should be converted to integers before applying Graph Embedding as it produces an error - The node indexing is wrong. Referring to code snippet 4.5, there is a function called

`convert_node_labels_to_integers`

in NetworkX, which takes some parameters such as graph (G), first label starting sequence (first_label), order of the integers (ordering), and storing previous node label name (label_attribute). On line 2, there is a `mapping` defined which is used for mapping the integer labels to the nodes. `relabel nodes()` function takes the previous graph and mapping parameter and returns a new graph with nodes having integer labels.

4.4 Implementation of Graph Embedding Methods

Machine learning algorithms are designed for continuous data, and graph embedding is designed for continuous vector space. Before applying ML models to graphs, they should be converted to low dimensional vector space. Many graph embedding methods have been developed using graph embedding techniques. Some of the methods used in this research are explained below.

4.4.1 DeepWalk

DeepWalk is a node classification task used for graph embedding by the random walk technique. By simulating a stream of brief random walks, DeepWalk learns the embeddings of the vertices of a graph. DeepWalk does latent features of the vertices that capture neighbourhood similarity and community by modelling a stream of short random walks. It uses features of node-level task explained in section 3.4.1. The mapping function is used to learn a latent representation, and probability distribution of node co-occurrences is defined as:

$$\phi : v \in V \longrightarrow R^{|V| \times d}$$

. Estimation of likelihood between the vertices is defined by:

$$Pr(v_i | (\phi(v_1), \phi(v_2), ..., \phi(v_{i-1})))$$

```python
from karateclub import DeepWalk

deepwalk_model = DeepWalk(walk_number=10, walk_length=80, dimensions
    =124)
deepwalk_model.fit(G)
embedding = deepwalk_model.get_embedding()
print('Embedding array shape', embedding)
```

Listing 4.6: DeepWalk Embedding

Here, DeepWalk is imported from karate club and parameters provided in the models are:

- walk_number (int) (Number of random walks. Default is 10)
- walk_length (int) (Length of random walks. Default is 80)
- dimensions (int) (Dimensionality of embedding. Default is 128)

Other parameters can also be provided such as workers (int), window_size (int),epochs (int), learning_rate (float), min_count (int), seed (int). `fit()` method is used for fitting a graph into DeepWalk model and `get_embedding()` method is used for generate the vector for that graph. Refer to code snippet 4.6

4.4.2 Node2Vec

The Node2Vec is also node classification takes and it optimizes a neighborhood preservation goal to learn low-dimensional representations for nodes in a network. The method adapts to different definitions of network neighborhoods by simulating biased random walks and also uses features of node-level task explained in section 3.4.1. Probability of random walk method, jumping from current node j to node i and has common attribute α is:

$$T_{ij}^{\alpha} = \frac{\alpha A_{ij}}{\sum_k \alpha A_{kj}}$$

```python
from karateclub import Node2Vec

n2v_model = Node2Vec(walk_number=10, walk_length=80, p=0.6, q=0.4,
    dimensions=124)
n2v_model.fit(G)
embedding = n2v_model.get_embedding()
print('Embedding array shape', embedding)
```

Listing 4.7: Node2Vec Embedding

Same as DeepWalk, Node2Vec is also imported from karate club and parameters provided in the models are:

- walk_number (int) (Number of random walks. Default is 10)
- walk_length (int) (Length of random walks. Default is 80)
- p (float) – Return parameter (1/p transition probability) to move towards from previous node
- q (float) – In-out parameter (1/q transition probability) to move away from previous node
- dimensions (int) (Dimensionality of embedding. Default is 128)

Other parameters can also be provided and are same as DeepWalk. The main difference between DeepWalk and Node2Vec is that biased random walks are used in Node2Vec, hence, p and q defines the biased behavior of the method. Refer to code snippet 4.7

4.4.3 Graph2Vec and GL2Vec Methods with Weisfeiler-Lehman Kernels

For graph nodes, the approach generates Weisfeiler-Lehman tree characteristics derived from 3.4.3. To create representations for the graphs, a document-feature co-occurrence matrix is deconstructed using these characteristics. The process assumes that nodes don't have any string features, and the degree centrality of the WL-hashing is used by default. The feature extraction, however, takes place based on the values of this key if a node feature with the key "feature" is supported for the nodes.

```python
from karateclub import Graph2Vec, GL2Vec

#Graph2Vec
g2v_model = Graph2Vec(dimensions=128, wl_iterations=4, epochs=5000,
    learning_rate=0.5)
g2v_model.fit(graphs)
embedding = g2v_model.get_embedding()
print(embedding)

#GL2Vec
g2v_model = GL2Vec(dimensions=2, wl_iterations=4, epochs=1000,
    learning_rate=1.0)
g2v_model.fit(graphs)
embedding = g2v_model.get_embedding()
```

```python
print(embedding)
```

Listing 4.8: Graph2Vec Embedding

Imported from Karate Club, Graph2Vec model takes parameters such as:

- wl_iterations (int) – Number of Weisfeiler-Lehman iterations. Default is 2
- dimensions (int) – Dimensionality of embedding. Default is 128
- epochs (int) – Number of epochs. Default is 10
- learning_rate (float) – HogWild! learning rate. Default is 0.025

Other parameters which can also be used are attributed (bool) – Presence of graph attributes, workers (int) – Number of cores, down_sampling (float) – Down sampling frequency, min_count (int) – Minimal count of graph feature occurrences, seed (int) – Random seed for the model, erase_base_features (bool) – Erasing the base features. `fit()` method is used for fitting a list of graphs into both models and `get_embedding()` method is used for generate the vector for all the graphs. Refer to code snippet 4.8

4.4.4 Network Laplacian Spectral Descriptor (NetLSD)

It is an approach that is scale-adaptive, size-invariant, and efficiently allows computable graph representation and comparison of large graphs. It calculates heat-kernel traces of normalized Laplacian matrix and switches to the approximation of eigenvalues approach when dealing with large graphs over a vector of time scales. The heat equation associated with Laplacian:

$$\frac{\partial u_t}{\partial t} = Lu_t$$

```python
from karateclub import NetLSD

nl_model = NetLSD(scale_max=10.0, scale_steps=500)
nl_model.fit(graphs)
embedding = nl_model.get_embedding()
print(embedding)
```

Listing 4.9: NetLSD Embedding

This method only works on the graphs having undirected edges, doesn't work on the directed ones. Referring to code snippet 4.9, it takes parameters such as:

- scale_min (float) – Time scale interval minimum. Default is -2.0.

- scale_max (float) – Time scale interval maximum. Default is 2.0.
- scale_steps (int) – Number of steps in time scale. Default is 250.
- scale_approximations (int) – Number of eigenvalue approximations. Default is 200.
- seed (int) – Random seed value. Default is 42.

4.5 Applied Machine Learning Models

Deep learning networks such as graphs and images can provide overwhelming results when appropriately trained using machine learning models. Therefore in this section, vectors generated from graph embedding models are used as input features in machine learning models, and the target value is the fan cost. Since it is a regression problem, some regression models are used for training and predicting the results.

4.5.1 Data Preprocessing

Data preprocessing is an essential step in Machine Learning since the quality of data and the relevant information that can be extracted directly influence our model's capacity to learn. Hence, we must preprocess our data before feeding it into any model. For Data preprocessing, all the methods are used from **Scikit-Learn** library.

- Null or blank values have been checked and removed if there are any. In the CSV file, all the values that have fan cost in number format for the respective household, that household model is selected for training and prediction using ML models.
- Since fan cost values vary between large ranges, and values are scaled between 0 and 1 using **MinMaxScaler** (Refer code 4.10), the ML model can learn and understand the problem easily. MinMax Scaler is derived by:

$$X_{sc} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

  Standard Scaler can also be used.

```
    from sklearn.preprocessing import StandardScaler, MinMaxScaler

    dataset = pd.read_csv('/content/results.csv')
    X = dataset.iloc[:,0].values
    y = dataset.iloc[:,1].values


    scaler = MinMaxScaler()
    y = scaler.fit_transform(y.reshape(-1,1))
    y_new = np.round(y,3)
    print(y_new.shape)

```

Listing 4.10: Data Scaling

4.5.2 Linear Regression

This ML algorithm is used supervised learning, thus finding out the linear relationship between dependent (target) and independent (input) variable(s). For more detailed analysis, refer to subsection 2.6.1. Regression analysis is defined by-

$$Y_i = f(X_i, \beta) + e_i$$

Y_i = dependent variable, f = function, X_i = independent variable, β = unknown parameters, e_i = error terms

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error,
    mean_absolute_error

x_train, x_test, y_train, y_test = train_test_split(x_new, y_new,
    test_size=0.3)
ML_model = LinearRegression().fit(x_train, y_train)
y_predict = ML_model.predict(x_test)
print(y_predict - y_test)
ML_acc = r2_score(y_test, y_predict)
print('AUC:', ML_acc)
```

Listing 4.11: Training and Predicting using Linear Regression

In the above code snippet 4.11, `x_new` is the vector representation of the graph given as the input data (independent variables) and `y_new` is the fan cost value as a target variable. Data is divided into training and test data using train_test_split imported from `sklearn.model_selection`. Then the training data (x_train, y_train) is used to train the model using the `fit()` method in the linear regression model imported from `sklearn.linear_model`. Then the remaining test data (x_test) is predicted using `predict()` method and stored in `y_predict` variable. For knowing the score of the model `r2_score` is used for regression models taking predicted value (y_predict) and known value (y_test) of the test data.

4.5.3 Lasso and Ridge Regression

Referring to 2.6.2, Least Absolute Shrinkage and Selection Operator (Lasso) is a penalised regression method, which uses L1 regularisation technique for regularisation and model selection. By adding a penalization factor (lambda or alpha) values, the coefficients computed in the linear model are decreased towards the center point as the mean.

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{N}(y_i - \sum_{j} x_{ij}\hat{\beta}_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Similar to Lasso, Ridge also add constraints on the coefficients by adding penalization factor (lambda or alpha). While Lasso take magnitude of the coefficients, Ridge takes the square. It use L2 regularisation.

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{N}(y_i - \sum_{j} x_{ij}\hat{\beta}_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Where,
λ denotes the amount of shrinkage.
$\lambda = 0$ implies all features are considered and it is equivalent to the linear regression where only the residual sum of squares is considered to build a predictive model
$\lambda = \infty$ implies no feature is considered i.e, as λ closes to infinity it eliminates more and more features
The bias increases with increase in λ
variance increases with decrease in λ

For Lasso and Ridge, the model has been trained with K-fold cross-validation procedure. Training since the data is small, K-fold is applied. Also, the models are trained for different hyperparameters C, which is used for the penalty parameter. Refer to code 4.12

```python
from sklearn.metrics import r2_score, mean_squared_error,
    mean_absolute_error
from sklearn.model_selection import KFold

kf = KFold(n_splits=5)
mean_square_array = []; scores_array = []; std_error=[]; mae=[]
from sklearn.linear_model import Ridge, Lasso
```

```python
C = [0.0001, 0.01, 0.1, 1, 5, 10, 20, 50, 100]
for c in C:
  temp=[]; temp1=[]; temp2=[]
  print("C=", c, "=========================================")
  for train, test in kf.split(x_new):
    model = Lasso(alpha=1/(2*c), fit_intercept=False).fit(x_new[train],
   y_new[train])
    print("Lasso ==================================")
    print(model.intercept_, model.coef_)
    y_pred = model.predict(x_new[test])
    print("Model Score test", model.score(x_new[test], y[test]))
    print("R2", r2_score(y_new[test], y_pred))
    print("MSE", mean_squared_error(y_new[test],y_pred))
    temp.append(mean_squared_error(y_new[test],y_pred))
    temp1.append(r2_score(y_new[test], y_pred))
    temp2.append(mean_absolute_error(y_new[test], y_pred))
  mean_square_array.append(np.array(temp).mean())
  std_error.append(np.array(temp).std())
  scores_array.append(np.array(temp1).mean())
  mae.append(np.array(temp2).mean())
print("C=", c, "MSE=", np.array(mean_square_array).mean(), "STD=", np.
   array(std_error).mean(), "MAE=", np.array(mae).mean(), "R2=", np.
   array(scores_array).mean())
plt.errorbar(C,mean_square_array,yerr=std_error,linewidth=3)
plt.xlabel('C')
plt.ylabel('Mean square error')
plt.show()
```

Listing 4.12: Training and Predicting using Lasso/Ridge Regression

4.5.4 AdaBoost and Random Forest

Other regression models were also used for training the data such as AdaBoost and Random Forest regressor, which uses decision tree methods to train the model. Refer to 2.6.5, 2.6.4 and code snippets 4.13, 4.14

```python

from sklearn.ensemble import AdaBoostRegressor

regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=3),
   n_estimators=300, random_state=1)


regr_2.fit(x_train, y_train)
```

```
y_pred_adaboost = regr_2.predict(x_test)

print("y_2==========")
mse = mean_squared_error(y_test, y_pred_adaboost)
print('mse-', mse)
ML_acc = r2_score(y_test, y_pred_adaboost)
print('AUC:', ML_acc)
```

Listing 4.13: Training and Predicting using AdaBoost

```

from sklearn.ensemble import RandomForestRegressor

regr = RandomForestRegressor(max_depth=6, random_state=0)
regr.fit(x_train, y_train)
y_pred_rfreg = regr.predict(x_test)
mse = mean_squared_error(y_test, y_pred_rfreg)
rmse = np.sqrt(mse)
print(mse)
print(rmse)
print(r2_score(y_test, y_pred_rfreg))
```

Listing 4.14: Training and Predicting using Random Forest Regressor

4.5.5 Multi Layer Perceptron

Refer to 2.6.6 and code snippet 4.15

```

kf = KFold(n_splits=5)
mean_square_array = []; scores_array = []; std_error=[]; mae=[]
from sklearn.neural_network import MLPRegressor
C = [0.0001, 0.01, 0.1, 1, 5, 10, 20, 50, 100]
for c in C:
  temp=[]; temp1=[]; temp2=[]
  print("C=", c, "===========================================")
  for train, test in kf.split(x_new):
    model = MLPRegressor(random_state=1,alpha=1/(2*c), max_iter=5000).
   fit(x_new[train], y_new[train])
    print("Lasso ===================================")
    # print(model.intercept_, model.coef_)
    y_pred_mlp = model.predict(x_new[test])
    print("Model Score test", model.score(x_new[test], y[test]))
    print("R2", r2_score(y_new[test], y_pred_mlp))
    print("MSE", mean_squared_error(y_new[test],y_pred_mlp))
    temp.append(mean_squared_error(y_new[test],y_pred_mlp))
    temp1.append(r2_score(y_new[test], y_pred_mlp))
    temp2.append(mean_absolute_error(y_new[test], y_pred_mlp))
  mean_square_array.append(np.array(temp).mean())
  std_error.append(np.array(temp).std())
  scores_array.append(np.array(temp1).mean())
  mae.append(np.array(temp2).mean())
print("C=", c, "MSE=", np.array(mean_square_array).mean(), "STD=", np.
   array(std_error).mean(), "MAE=", np.array(mae).mean(), "R2=", np.
   array(scores_array).mean())
plt.errorbar(C,mean_square_array,yerr=std_error,linewidth=3)
plt.xlabel('C')
plt.ylabel('Mean square error')
plt.show()
```

Listing 4.15: Training and Predicting using MLP Regressor

Chapter 5

Evaluation and Results

This section describes the evaluation metrics which can be used to evaluate results and also results will be displayed and discussed.

5.1 Structuring Graphs

```
{'length': 2, 'width': 2, 'specific_port': 'Door'}
{'length': 2, 'width': 38, 'specific_port': 'Internal'}
{'length': 18, 'width': 10, 'specific_port': 'Mixed'}
{'length': 18, 'width': 3}
{'length': 18, 'width': 7}
{'length': 18, 'width': 3}
{'length': 18, 'width': 5, 'specific_port': 'Internal'}
{'length': 18, 'width': 3, 'specific_port': 'Mixed'}
{'length': 18, 'width': 3}
{'length': 18, 'width': 2}
{'length': 18, 'width': 2}
{'length': 18, 'width': 2}
```

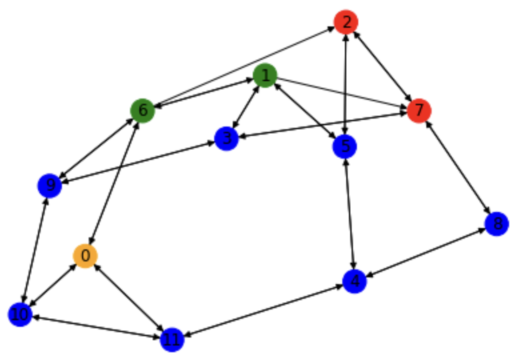


Figure 5.1: Graph generated from json file

Here, figure 5.1 represents graph build from json file. Also, nodes are colored according to the *specific port* mentioned which differentiates between the nodes. *Orange* color defines the node to be **Door** which is the outlet of the air from fan, *red* color defines the node to be **Mixed** through which the air flows both ways and *green* color defines the node to be

Internal, through which the airflow is into the fan. Directed edges are built from port Internal to Mixed to which the property **is_ac** is true i.e., air is flown from internal port to mixed. Edges both ways defines the connectivity to be mutual i.e., airflow is bidirectional.

5.2 Metrics for Evaluating Regression Models

Performance or competence of a regression model is explained in error, measuring the difference in predicting and testing values.

There are three or four major metrics which is used to measure the performance:

- Mean Square Error (MSE)
- Root Mean Square Error (RMSE)
- Mean Absolute Error (MAE)
- R2 Score

5.2.1 Mean Square Error (MSE)

MSE is an important measure of regression models, a loss function used to minimize errors between predicted and expected values. It is calculated as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the i^{th} expected value and $\hat{y}_i$ is the $i^t h$ predicted value. Squared of difference between both the values, results in removing the negative sign and squaring the error value. Larger the difference between expected and predicted value, larger the error will be.

MSE for Graph2Vec/GL2Vec Methods

Above graph 5.2 explains the square of the difference between the predicted and the expected value (fan cost value). Splitting the data into train and test, and without using K-fold, graph embedding vectors were trained with three models - Linear Regression, AdaBoost and Random Forest.

Here, it is observed that blue line has higher spikes mostly, followed by green and then orange. Mean square error should be close to zero (0), if model is performing good and has over-fitting and under-fitting reduced. Therefore, AdaBoost is seen to be performing well

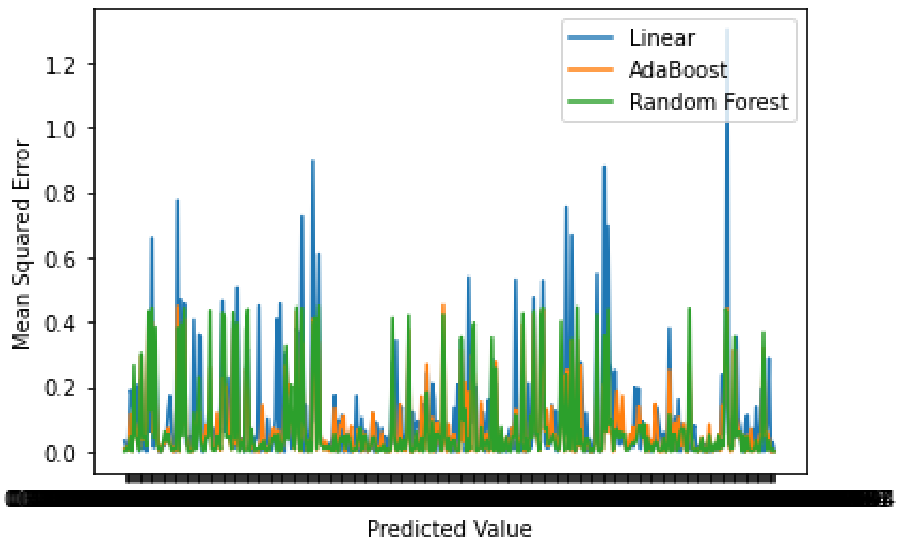


Figure 5.2: Mean square error without K-fold in Graph2Vec/GL2Vec

than Linear Regression and Random Forest. Producing less difference between predicted and expected value, the average mse of AdaBoost is seen to be around $\longrightarrow$ 0.18.

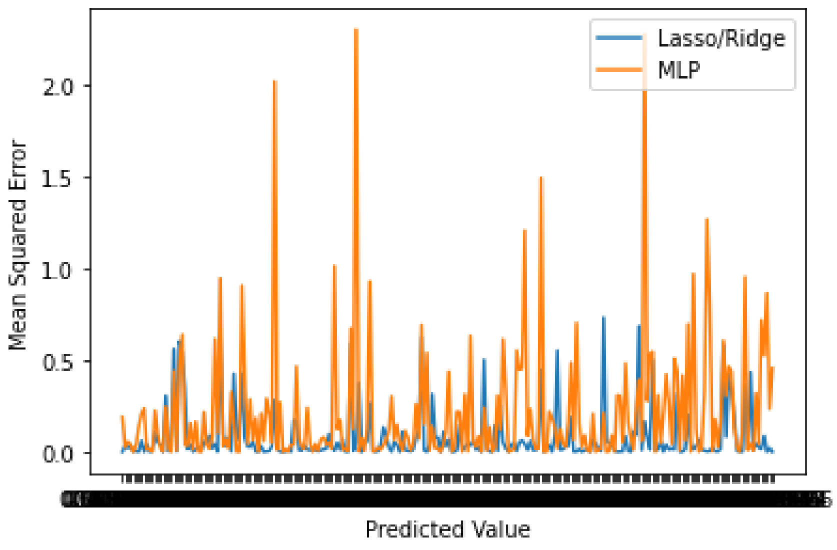


Figure 5.3: Mean square error with K-fold in Graph2Vec/GL2Vec

Graph 5.3 explains the error rate of the model by mean square error, to which models were trained by using K-fold cross validation splitting data into 5 chunks and 5 iterations with each time testing data chunk was different, graph embedding vectors were trained with ML models - Lasso/Ridge and MLP.

Here, it is observed that orange line has higher spikes mostly, followed by blue. Therefore, Lasso/Ridge is seen to be performing well than MLP Regressor. Producing less difference between predicted and expected value, the average mse of Lasso/Ridge is seen to be around $\longrightarrow$ 0.4.

MSE for DeepWalk/Node2Vec Methods

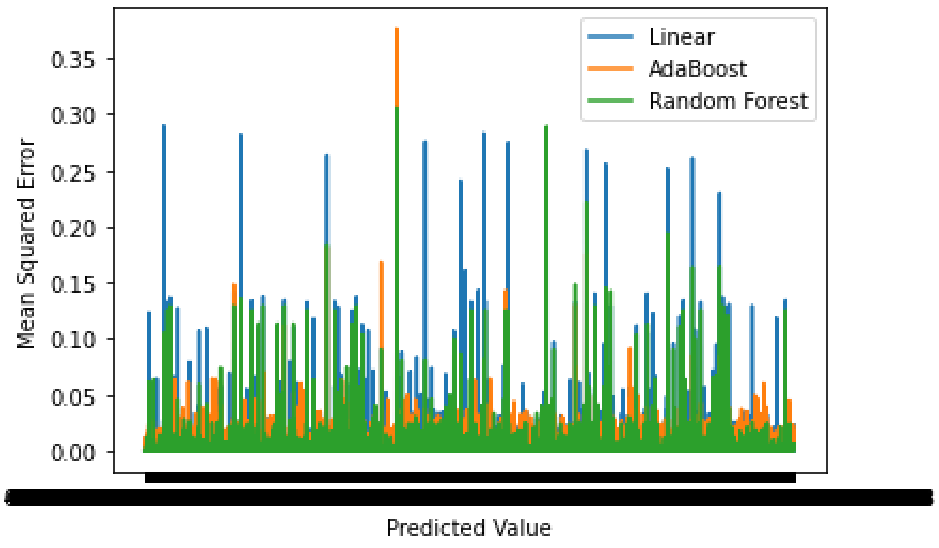


Figure 5.4: Mean square error with K-fold in DeepWalk/Node2Vec

Above graph 5.4 explains the MSE for random walks embedding methods (DeepWalk Node2Vec), ML models were applied without K-fold cross validation and the error between expected and predicted values were plotted. Three models Linear Regression, AdaBoost and Random Forest Regressor were evaluated for which Adaboost has been performing well than others and average mse score for adaboost model is $\longrightarrow$ 0.087

In figure 5.5, the cross validation (K-fold=5) is applied to data and ML models such as Lasso/Ridge and MLP Regressor were evaluated for random walk methods. Here, also Lasso/Ridge Model is performing better than MLP Regressor. Average MSE for Lasso/Ridge $\longrightarrow$ 0.11

ML Models	**Graph2Vec/GL2Vec**	**DeepWalk/Node2Vec**
Linear Regression	0.45	0.17
Lasso/Ridge	0.4	0.11
AdaBoost	0.18	0.087
Random Forest	0.37	0.15
MLP	0.7	0.14

Table 5.1: MSE for ML models used with different graph embedding methods

Table 5.1 explains the mean square error values for different machine learning models

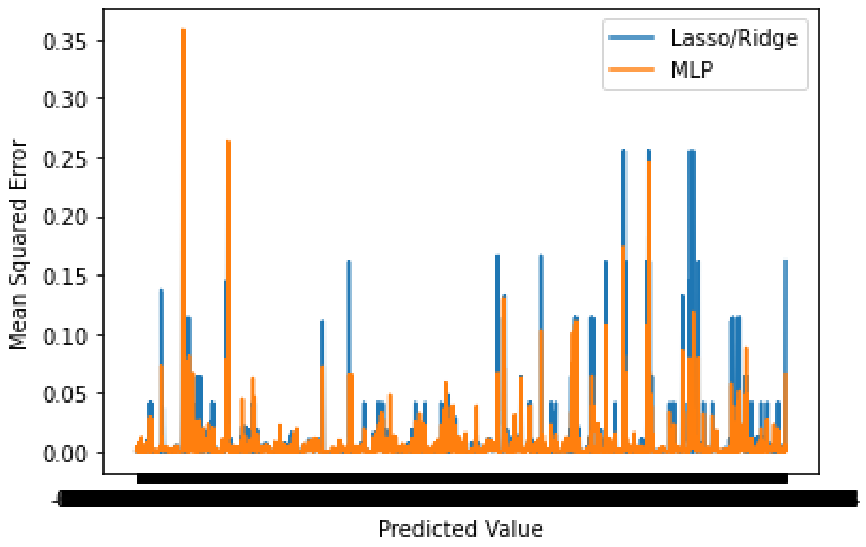


Figure 5.5: Mean square error with K-fold in DeepWalk/Node2Vec

used with different graph embedding techniques. It explains that AdaBoost method has been performing well on graph embedding methods i.e., the correlation between vector generated by graph embedding methods and target value has been well adjusted.

5.2.2 R2 Score

R-squared metric explains the variance of the dependent value explained by independent value in a regression model. If R2 score is 50%, it can be said that 50% of the changeability in the target value (dependent value) due to change in independent values can still be acknowledged by the model and the remaining 50%, it doesn't predict the change or variance in the values.

$$R^2 = 1 - \frac{\sum_{k=1}^{n}(y_{test\_k} - y_{pred_k})^2}{\sum_{k=1}^{n}(y_{test\_k} - (\sum_{i=1}^{n}\frac{1}{n}y_i))^2} = \frac{Variance\ explained\ by\ the\ model}{Total\ Variance}$$

where, y_{test} = expected value, y_{pred} = predicted value, $(\sum_{i=1}^{n}\frac{1}{n}y_i)$ = mean of expected value. k is the iteration of the values of the expected and predicted value.

Higher the R2 score, performance of the model is good. For worse models, it can give negative values also.

Note: For a model to give accurate predictions, it should have low variance and low bias.

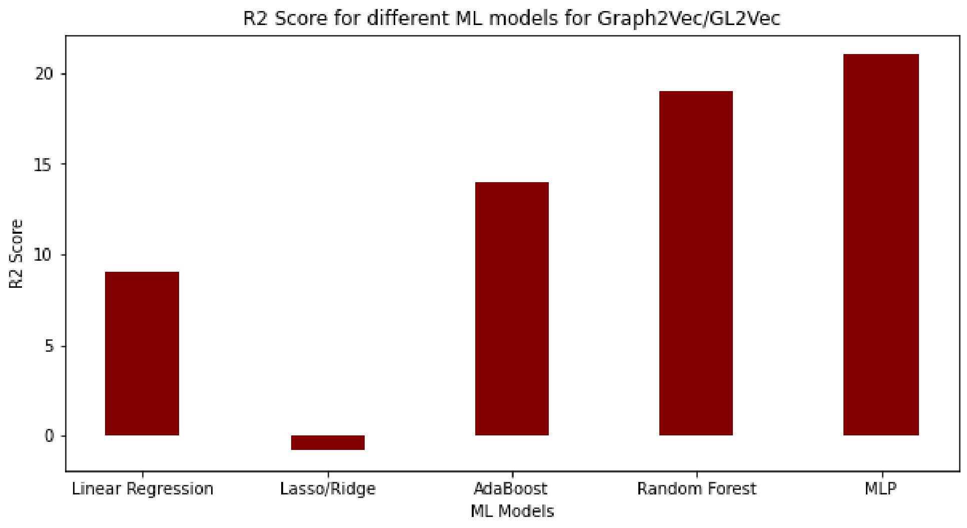


Figure 5.6: R2 scores for ML models for Graph2Vec/GL2Vec

R2 Score for Graph2Vec/GL2Vec Methods

Here in the figure 5.6, Multi-layer Perceptron (MLP) Regressor has been performing well with r2 score of 21% than the remaining other ML models, it means that 21% variance of dependent value has been is being handled by the model with respect to the vector generated. Here, the variance seems to be very high.

R2 Score for DeepWalk/Node2Vec Methods

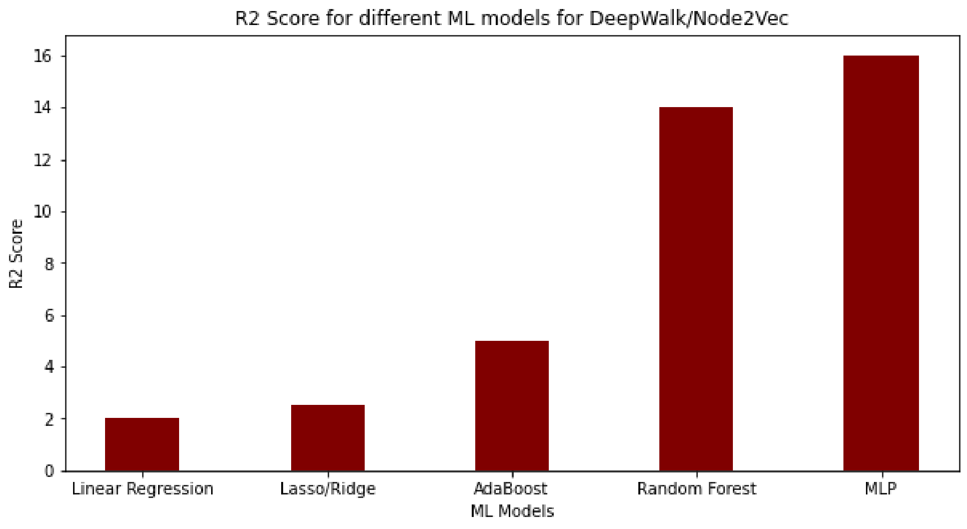


Figure 5.7: R2 scores for ML models for random walks

Here in the figure 5.7, Multi-layer Perceptron (MLP) Regressor again has been performing well with r2 score of 16% than the remaining other ML models, it means that

21% variance of dependent value has been is being handled by the model with respect to the vector generated. Here, the variance seems to be very high.

ML Models	**Graph2Vec/GL2Vec**	**DeepWalk/Node2Vec**
Linear Regression	9	0.17
Lasso/Ridge	-0.81	2.5
AdaBoost	14	5
Random Forest	19	14
MLP	21	16

Table 5.2: R2 Scores (in %) for ML models used with different graph embedding methods

Table 5.2, mentions all the scores achieved for both whole graph embedding methods and random walks methods for different ML models, which can help us in knowing the best suitable model for this type of dataset and problem. Also, can also explain the different the reason of the model to be performing good as well as lacking behind.

5.3 Keypoints for Evaluation

- Talking about the comparison between this and Simulink approach, time required for structuring 1200 graphs with the NetworkX took around 14 to 15 minutes; for applying graph embedding methods for conversion to vectors took 2 mins for running 2000 epochs and for predicting the outcome with different Machine Learning models took 2 mins. Compared to Simulink model which took 5 to 6 minutes modeling and prediction for 1 graph. This calculation has been tested and verified.

- Also, Simulink which is graphical programming environment based on MATLAB requires more computing power than running the python environment with some graph embedding and machine learning models.

- Mean Square Error value has been calculated by change in dimensions parameters of the graph embedding models. **Dimension=300** is set for all the graph embedding models, this means vector dimensions generated for each graph with be 1 x 300. For 1200 graphs which were used in this research, vector dimension of input variable (dependent variable) is 1200 x 300. The model explained above in code snippet 4.4.3, has dimension parameter which can have any value to model the output in the given dimension. Also in the graph2vec method, epochs used were 5000 which helps in reiterating the graphs and trains the methods well to give the best vector output

- R2 scores values has been calculated by change in dimensions parameters of the graph embedding models. **Dimension=2** is set for all the graph embedding models, this means vector dimensions generated for each graph with be 1 x 2. For 1200 graphs which were used in this research, vector dimension of input variable (dependent variable) is 1200 x 2. Epochs used were 5000 for the Graph2vec/GL2vec model.

- For decision tree models tree depth parameters was 3 (depth=3) for both AdaBoost and Random Forest Regressor.

- K-fold cross validations were used for Lasso/Ridge and MLP Regressor models.

- Weisfeiler-Lehman Iterations for Graph2vec and GL2vec models was set to 4 while performing graph embedding.

- Penalty parameter (C) was set for Lasso/Ridge and MLP because they support loss function (α) and were re-evaluted for different values of C.

Chapter 6

Discussion

During this research, there were many things which were not static. Many changes can be made in this research, other operations can be performed, and many other things can be tried. There are a few points I would like to mention which can change the scenario of this research performed above. Improvising things is what research should be, and different things should be tried and learned. Discussion points are:

- Dataset can be improved and should have more features so that there can be a correlation between the target value and input values.
- Parameters in graph embedding methods affect the results and predictions. The dimension of the vector has a massive effect on the accuracy score of the ML models; therefore, an optimal number of dimensions are required. Epochs and learning rates also affect the scores; increasing them can help in improvement. **Weisfeiler-Lehman (wl)** iterations in the whole graph embedding method do not affect the data of the vectors much; refer to code 4.8. **walk_number** and **walk_length** can affect the number and length of biased/unbiased walks from the source node and the vector output.
- Decision Tree models such as Adaboost and Random Forest perform better if an optimal number of **max_depth** of the decision tree is defined. Here, results were evaluated using different values for max_depth, and for max_depth = 6, Random Forest had an accuracy score of 22 more than MLP Regressor for Graph2Vec/GL2Vec.
- For MLP Regressor and Lasso/Ridge, penalty parameter $\alpha = \frac{1}{2C}$ for different values of C affect the results.

6.1 Limitations

There are some limitations to this project, which can be described as:

- Building and training Millions of graphs with millions of nodes and edges will require time and resources such as more RAM space and a potent processor to handle such a workload.
- Large dimension vectors generated from graph embedding methods can confuse the machine learning models as an input variable and cause the models' poor performance.

6.2 Threats to Validity

This section describes the summary and classification of the threat and also answers the mitigation strategy for it [69], [70], [71]. Further, it describes the factors which got affected by this mitigation.

- **Internal Validity**: It refers to the question whether the Treatment indeed causes the effect in the outcome or whether something not considered in the research design is causing the effect on the outcome.

  This would, for instance, be the case when the experiment yields different results depending on the current status of the computer, the time of day, or additional input data that is not considered in the design.

  We have mitigated this effect by (a) using a design and implementation of the experiment that is commonly used in similar research projects and (b) reviewing the design of the experiment for additional factors that might effect the outcome, unintentionally.

- **Construct Validity**: It refers to the question whether the treatment correspond to the actual cause or the outcome we are interested in.

  The final checkpoint was to predict the outcome of the graphs using machine learning models and evaluate the R2 score. For a better R2 score, ML models try to mitigate the over-fitting and under-fitting. Dimensions can threaten a better R2 score, as large dimensions confuse the ML models for better performance. Data should have correlation and optimal dimensions of input variables so that model can distinguish and perform better.

- **External Validity**:It refers to the question whether the cause and effect relationship we have shown valid in other situations and can we generalize our results. Do the results apply in other contexts?

  Here, talking about the dataset, the household dataset is used for carrying out this research. Although the research is generalised for using any data related to graphs, trying with another dataset can affect the performance of the graph embedding methods or machine learning models. It can be mitigated by trying with another dataset so that the models and methods used can be tested and improved.

Chapter 7

Conclusions & Future Work

7.1 Conclusion

This research aimed to search for techniques that could implement simulation models or other feature engineering tasks using less time and resources. Here the simulation models were household models for which fan cost was to be predicted; therefore, the prediction task to be done by the machine can be performed through machine learning. However, the problem was converting those simulation models into low-dimensional vector form so the machine could understand and be trained. Graphs were introduced then to reform the simulation models, and graph embedding techniques were applied to convert graphs into vectors.

Data was present in JSON format for the simulation models to be built, and that data was used to construct graphs. Python has some tools for structuring and manipulating the graph, and NetworkX was used in this research. It is a framework with many graph methods that can be used for graphs and also support the Karate Club library that has graph embedding methods built-in for different level tasks - node, link, cluster or whole graph level. Graphs were constructed using the house's rooms as nodes of the graph and links between them as edges. Other properties of the nodes (rooms) were also added, which were mentioned in the JSON file; length, width, specific port, and weights were added to the edges for the type of connection, if it is one-way or two-way.

Different graph embedding techniques were applied to the graphs, such as node level - DeepWalk and Node2Vec, whole graph level - Graph2Vec, Gl2Vec and NetBSD and the output were generated in vector form.

Machine Learning models were applied to predict the fan cost value. Linear models (linear regression, Lasso and Ridge), Decision Tree models (Random Forest, AdaBoost) and Neural Networks (MLP) were applied, and vectors generated from graphs were trained. Results were evaluated based on the different regression models' mean square errors and r2 scores.

7.2 Future Work

7.2.1 Dataset

In this research, the dataset is manually built, and this dataset was compatible with performing simulation on MATLAB. This dataset can be improved, adding more information while performing graph embedding techniques and machine learning models to train this data. Datasets can be improved and should correlate with differentiating between different data points. More information can help generate vectors, and correlation can help lower the variance and bias for machine learning models. With correlation, bias and variance become high, and the model becomes confused in learning the data as no learning is gained by the model.

7.2.2 Tool for Structuring Graphs

In this research, the NetworkX framework is used for structuring graphs, which is a good and accessible tool. Also, it has supported the Karate Club library for applying different graph embedding techniques, and one can easily apply and understand things.

Also, there are other frameworks, libraries and tools for structuring graphs, such as Pytorch Geometric, Spektral, and Deep Graph Library. Pytorch Geometric is a mostly rated library for structuring graphs and is widely used by many researchers. Also, it is built in Pytorch, a popular python framework. Maybe it can help structure graphs better with more methods and functions than NetworkX and supports mixed graphs (having undirected and directed edges (not in NetwrokX)) which can help generate the vectors with more information.

7.2.3 Graph Embedding Methods

Here, the graph embedding methods used were DeepWalk, Node2Vec, Graph2Vec, Gl2Vec, and NetLSD. NetLSD method was discarded, after which only undirected edges were supported. While performing the operations on different graph embedding techniques, it has been known that this research can be efficient only if subgraph embedding and whole graph embedding techniques are applied. Therefore other methods related to mentioned techniques should be used and evaluated using this dataset. Using different features of the whole graph embedding can help in generating vectors helpful for machine learning models.

7.2.4 Machine Learning

While training the ml models with a vector with large dimensions or the same data points, it becomes difficult for the ml models to differentiate between the data points to make the correct predictions. Therefore, hyper-parameter tuning is required to choose the optimal feature and parameter to optimise the machine learning models' performance. In this case, our vector scale can be significant because of the dimension parameter of the graph embedding methods. Therefore, training data with large vectors can be a considerable task, and hyper-parameter tuning such as **GridSearchCV** can help differentiate between the critical data points to increase the performance of the model.

Machine learning models such as Linear Regression, Lasso/Ridge, Random Forest, AdaBoost and MLP regressor were used to train this regression problem. Other models, such as Support Vector Machine with regression problem (SVR), can also be used to experiment with this problem.

Bibliography

[1] Zhiyuan Liu, Yankai Lin, and Maosong Sun. Network representation. In *Representation Learning for Natural Language Processing*, pages 217–283. Springer, 2020.

[2] Graph nets - github. `https://github.com/deepmind/graph_nets`.

[3] Tradition machine learning methods for graphs. `http://web.stanford.edu/class/cs224w/`.

[4] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[5] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[6] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[7] Song Chen, Samuel Owusu, and Lina Zhou. Social network based recommendation systems: A short survey. In *2013 international conference on social computing*, pages 882–885. IEEE, 2013.

[8] Wen Zhang, Xinrui Liu, Yanlin Chen, Wenjian Wu, Wei Wang, and Xiaohong Li. Feature-derived graph regularized matrix factorization for predicting drug side effects. *Neurocomputing*, 287:154–162, 2018.

[9] Wen Torng and Russ B Altman. Graph convolutional neural networks for predicting drug-target interactions. *Journal of chemical information and modeling*, 59(10):4131–4149, 2019.

[10] Lucy Skrabanek, Harpreet K Saini, Gary D Bader, and Anton J Enright. Computational prediction of protein–protein interactions. *Molecular biotechnology*, 38(1):1–17, 2008.

[11] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514, 2021.

[12] Stefan Edelkamp and Stefan Schrödl. Route planning and map inference with global positioning traces. In *Computer science in perspective*, pages 128–151. Springer, 2003.

[13] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[14] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[15] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[17] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

[18] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[19] Mark Heimann and Danai Koutra. On generalizing neural node embedding methods to multi-network problems. In *KDD MLG Workshop*, 2017.

[20] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[21] Hong Chen and Hisashi Koga. Gl2vec: Graph embedding enriched by line graphs with edge features. In *International Conference on Neural Information Processing*, pages 3–14. Springer, 2019.

[22] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

[23] Nathan de Lara and Edouard Pineau. A simple baseline algorithm for graph classification. *arXiv preprint arXiv:1810.09155*, 2018.

[24] Saurabh Verma and Zhi-Li Zhang. Hunt for the unique, stable, sparse and fast feature learning on graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[26] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[27] Harm Derksen and Jerzy Weyman. Quiver representations. *Notices of the AMS*, 52(2):200–206, 2005.

[28] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[29] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[30] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[31] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[32] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral [application notes]. *IEEE Computational Intelligence Magazine*, 16(1):99–106, 2021.

[33] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[34] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[35] John Boaz Lee, RA Rossi, S Kim, NK Ahmed, and E Koh. Attention models in graphs: A survey. arxiv. *arXiv preprint arXiv:1807.07984*, 2018.

[36] Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. Community preserving network embedding. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 203–209, 2017.

[37] Xiaohan Zhao, Adelbert Chang, Atish Das Sarma, Haitao Zheng, and Ben Y. Zhao. On the embeddability of random walk distances. *Proc. VLDB Endow.*, 6(14):1690–1701, sep 2013.

[38] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250, 2008.

[39] Shiyu Chang, Wei Han, Jiliang Tang, Guo-Jun Qi, Charu C Aggarwal, and Thomas S Huang. Heterogeneous network embedding via deep architectures. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 119–128, 2015.

[40] Hanwang Zhang, Xindi Shang, Huanbo Luan, Meng Wang, and Tat-Seng Chua. Learning from collective intelligence: Feature learning using social images and tags. *ACM transactions on multimedia computing, communications, and applications (TOMM)*, 13(1):1–23, 2016.

[41] Xue Geng, Hanwang Zhang, Jingwen Bian, and Tat-Seng Chua. Learning image and user features for recommendation in social networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4274–4282, 2015.

[42] Juzheng Li, Jun Zhu, and Bo Zhang. Discriminative deep random walk for network classification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1004–1013, 2016.

[43] Cunchao Tu, Weicheng Zhang, Zhiyuan Liu, Maosong Sun, et al. Max-margin deepwalk: Discriminative learning of network representation. In *IJCAI*, volume 2016, pages 3889–3895, 2016.

[44] Ruobing Xie, Zhiyuan Liu, Maosong Sun, et al. Representation learning of knowledge graphs with hierarchical types. In *IJCAI*, volume 2016, pages 2965–2971, 2016.

[45] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[46] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023. PMLR, 2016.

[47] Tuan MV Le and Hady W Lauw. Probabilistic latent document network embedding. In *2014 IEEE International Conference on Data Mining*, pages 270–279. IEEE, 2014.

[48] Han Xiao, Minlie Huang, Lian Meng, and Xiaoyan Zhu. Ssp: semantic space projection for knowledge graph embedding with text descriptions. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

[49] Liang Yao, Yin Zhang, Baogang Wei, Zhe Jin, Rui Zhang, Yangyang Zhang, and Qinfei Chen. Incorporating knowledge graph embeddings into topic modeling. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[50] Zhigang Wang, Juanzi Li, Zhiyuan Liu, and Jie Tang. Text-enhanced representation learning for knowledge graph. In *Proceedings of International joint conference on artificial intelligent (IJCAI)*, pages 4–17, 2016.

[51] Cheng Li, Jiaqi Ma, Xiaoxiao Guo, and Qiaozhu Mei. Deepcas: An end-to-end predictor of information cascades. In *Proceedings of the 26th international conference on World Wide Web*, pages 577–586, 2017.

[52] Zhilin Yang, Jie Tang, and William Cohen. Multi-modal bayesian embeddings for learning social knowledge graphs. *arXiv preprint arXiv:1508.00715*, 2015.

[53] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[54] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia-a crystallization point for the web of data. *Journal of web semantics*, 7(3):154–165, 2009.

[55] Basma Alharbi and Xiangliang Zhang. Learning from your network of friends: A trajectory representation learning model based on online social ties. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 781–786. IEEE, 2016.

[56] Qing Zhang and Houfeng Wang. Not all links are created equal: An adaptive embedding approach for social personalized ranking. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 917–920, 2016.

[57] Shirui Pan, Jia Wu, Xingquan Zhu, Chengqi Zhang, and Yang Wang. Tri-party deep network representation. *Network*, 11(9):12, 2016.

[58] Mukund Balasubramanian and Eric L Schwartz. The isomap algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[59] Xiaogang Su, Xin Yan, and Chih-Ling Tsai. Linear regression. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(3):275–294, 2012.

[60] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 18(6):275–285, 2004.

[61] Yan-Yan Song and LU Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130, 2015.

[62] Abraham J Wyner, Matthew Olson, Justin Bleich, and David Mease. Explaining the success of adaboost and random forests as interpolating classifiers. *The Journal of Machine Learning Research*, 18(1):1558–1590, 2017.

[63] Mariana Belgiu and Lucian Drăguţ. Random forest in remote sensing: A review of applications and future directions. *ISPRS journal of photogrammetry and remote sensing*, 114:24–31, 2016.

[64] Gunnar Rätsch, Takashi Onoda, and K-R Müller. Soft margins for adaboost. *Machine learning*, 42(3):287–320, 2001.

[65] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[66] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Karate club: an api oriented open-source python framework for unsupervised learning on graphs. In *Proceedings of the 29th ACM international conference on information & knowledge management*, pages 3125–3132, 2020.

[67] Ego splitting framework, 2022. `https://www.eecs.yorku.ca/course_archive/2017-18/F/6412/reading/kdd17p145.pdf`.

[68] Oliver Kramer. Scikit-learn. In *Machine learning for evolution strategies*, pages 45–53. Springer, 2016.

[69] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[70] Bernd Freimut, Teade Punter, Stefan Biffl, and Marcus Ciolkowski. State-of-the-art in empirical studies. *Report: ViSEK/007/E, Fraunhofer Inst. of Experimental Software Engineering*, 2002.

[71] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012.