NOVEL REPRESENTATION LEARNING TECHNIQUE USING GRAPHS FOR PERFORMANCE ANALYTICS

by

Tarek Ramadan, B.E.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2022

Committee Members:

Tanzima Islam, Chair

Apan Qasem

Ziliang Zong

FAIR USE AND AUTHOR’S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

I want to thank my thesis advisor, Dr. Tanzima Islam, for her guidance and support throughout my master's degree and the work of this thesis.

I also appreciate Dr. Apan Qasem and Dr. Ziliang Zong taking the time to ensure the successful completion of my thesis, and I am thankful for them for being on my thesis committee.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	**Description**
ML	Machine Learning
GNN	Graph Neural Network
GCN	Graph Convolution Network
GCN-MP	Graph Convolution Network – Message Passing
CUDA	Compute Unified Device Architecture
AI	Artificial Intelligence
GAT	Graph Attention Network
HPC	High-Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MLP	Multi-Layer Perceptron
ARML	Automated Relational Meta-learning
RMSE	Root Mean Squared Error
MSE	Mean Squared Error
NLP	Natural Language Processing
SSL	Self-Supervised Learning
SSGNN	Self-Supervised Graph Neural Network
SSBGNN	Self-Supervised Batched Graph Neural Network

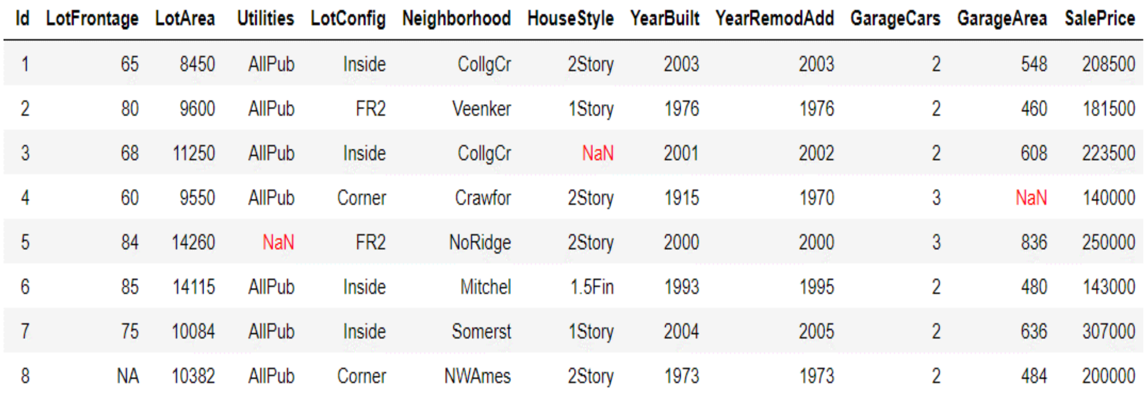
ABSTRACT

Most publicly available datasets are in a tabular format. It is one of the common data formats used for machine learning (ML) applications, especially in HPC, where models solve regression problems, such as predicting the execution time. Existing ML techniques leverage the correlations among features given tabular datasets, disregarding any relationship between the samples. Moreover, the success of the downstream analysis techniques depends on how well information is extracted from the raw features. For high-quality embeddings, existing methods rely on extensive feature engineering and preprocessing steps, which come at a high cost and require a human in the loop. To fill these two gaps, we propose a novel idea of transforming performance data into graphs to leverage the advancement of graph neural network-based (GNN) techniques in capturing complex relationships between features and samples. In contrast to other ML application domains such as social networks, the graph is not given; instead, we need to build it. To address this gap, we propose graph building methods where nodes represent samples, and the edges are automatically inferred iteratively based on the similarity between the features in the samples. We evaluate the effectiveness of the generated embeddings from GNNs based on how well they make even a simple feed-forward neural network perform for regression tasks compared to other state-of-the-art representation learning techniques. Our evaluation demonstrates that even with up to 25% random missing values for each dataset, our method outperforms commonly used graph and deep neural network (DNN)-

based approaches and achieves up to 51.77% improvement in MSE loss over the DNN baseline.

1. INTRODUCTION

Most publicly available datasets that researchers use on machine learning models are in a tabular format due to the user-friendly interfaces that make it easier to collect data from various applications (e.g., google forms and online questionnaires). In a tabular dataset, each row contains a data sample, and each column either contains a feature or a target that needs to be predicted. An example of a tabular dataset is the House Prices dataset [1], where the columns contain features including Lot Area, Year Built, Utilities, etc. The machine learning (ML) task is to predict each house's Sale Price.



Id	LotFrontage	LotArea	Utilities	LotConfig	Neighborhood	HouseStyle	YearBuilt	YearRemodAdd	GarageCars	GarageArea	SalePrice
1	65	8450	AllPub	Inside	CollgCr	2Story	2003	2003	2	548	208500
2	80	9600	AllPub	FR2	Veenker	1Story	1976	1976	2	460	181500
3	68	11250	AllPub	Inside	CollgCr	NaN	2001	2002	2	608	223500
4	60	9550	AllPub	Corner	Crawfor	2Story	1915	1970	3	NaN	140000
5	84	14260	NaN	FR2	NoRidge	2Story	2000	2000	3	836	250000
6	85	14115	AllPub	Inside	Mitchel	1.5Fin	1993	1995	2	480	143000
7	75	10084	AllPub	Inside	Somerst	1Story	2004	2005	2	636	307000
8	NA	10382	AllPub	Corner	NWAmes	2Story	1973	1973	2	484	200000

Figure 1. A sample for the House Prices Tabular dataset contains numerical and categorical features in the first eleven columns, and the last column is the prediction target (Sale Price). Note that values colored in red (NaN) are missing values.

To use machine learning (ML) algorithms on tabular data, we need to split the data into training and testing sets, where both sets contain input features and a target value. Splitting the dataset is necessary to evaluate the model's accuracy and its ability to generalize to unseen data samples during testing. The downstream ML models can only operate on a vector representation of each sample, called **feature representation**, so the raw data needs to be transformed in that format.

1.1 Motivation

HPC is critical for solving many complex problems, and big tech companies are interested in predicting the time to develop a new product. HPC configurations (e.g., number of threads, nodes, power cap, thread binding) significantly impact resource utilization and execution time. It is critical in the HPC systems to predict the runtime precisely since scheduling systems allocate job time based on these predictions, which means under prediction will cause the jobs to exceed the time limit; therefore, the job will need to be resubmitted. Over-predict the runtime causes the system to be idle, which means system underutilization. Hence, over and under predictions of runtime waste both user time and expensive system resources.

To enable precise runtime prediction, the performance analytics research area in HPC leverages ML techniques. The problem of interest is extracting the most meaningful information from seemingly unrelated user inputs (i.e., configurations and algorithms) to preserve the correlation between input features and target performance metrics. Effectively learning to represent information is a fundamental problem because the accuracy of downstream ML models heavily depends on the quality of information captured.

While the existing performance analytics research captures user inputs and the respective application performance in tabular formats, this format only allows most of the downstream ML models to exploit relationships across the features within each sample. Here, a feature is the ML terminology for describing a user input, and a sample is the instance of an application run. This thesis proposes investigating a novel approach of transforming tabular data into a graph data structure. Specifically, we hypothesize that using a graph structure to describe a performance dataset (spanning many samples) enables implicitly describing similarities between samples and, thus, constructing the idea of neighborhoods. This novel idea of building a graph from a performance dataset [2] with many samples enables learning from similarities across samples and features to infer the edges and their weights across samples (or nodes). In contrast, the existing tabular data representation approaches require explicit distance measures to define similarity, which is challenging for multimodal data. Moreover, for a streaming performance analytics system, where finding labeled data is scarce and rebuilding models when new samples arrive is too expensive, new unlabeled samples can be easily placed in the existing dataset based on their inherent neighborhood similarity.

1.2 Representation Learning

Representation learning is a technique that allows the model to automatically learn the representations from raw input data that is needed for feature extraction. Then the model uses the features extracted to predict the required task. Representation learning brought a lot of breakthroughs in image processing and Natural Language Processing by learning representations within the data with various levels of abstraction [3]. Researchers use many methods, such as preprocessing, feature engineering, dimensionality reduction, etc., to encode the salient information from these features into the representation vector, also known as embeddings. Depending on their types, some of these methods are easy to use, others not. One of the requirements in the HPC community is to provide explainability to the downstream models. So, feature engineering methods such as dimensionality reduction using Principal Component Analysis (PCA) [4] that obfuscate the features cannot be used. On the other hand, techniques like Long-Short-

Term-Memory (LSTM) do not support multimodal data. They require an explicit distance function to find the correlation between samples making HPC performance prediction even more difficult.

Finding an effective solution for tabular data is an active research problem, where researchers proposed a lot of work to enhance the model's performance. Tree-based models like XGBoost [5] have achieved encouraging enhancement in performance prediction for real-world applications. However, when the data has a lot of missing values in the features or if the data sample is unlabeled, existing supervised algorithms perform poorly or become inapplicable. To overcome these two issues, existing literature has proposed Attentive Interpretable Tabular Learning like the TabNet model [6]. Nonetheless, the TabNet model also uses deep learning techniques which leverage the correlations among features within each sample given a tabular dataset but disregard any relationship between the samples. Existing methods based on tabular data format rely on extensive feature engineering and preprocessing steps to overcome this shortcoming, which come at a high cost and require a human in the loop.

To leverage correlations across features and samples without investing time and effort in the manual feature engineering process, we suggest a new representation learning technique that automatically encodes the input into the best possible representation to address the previously mentioned problems. We transform tabular data into a graph structure which allows relationships between the data to be implicitly modeled. To improve the graph, we automatically infer edges based on similarity, which makes graph transformation autonomous and results in explainable models downstream. To validate the performance of our hypothesis, we would use High-Performance

Computing (HPC) data to test how we could leverage representation learning. Our representation learning technique is effective for a streaming performance analytics system, where finding labeled data is scarce and rebuilding models when new samples arrive is too expensive. New unlabeled samples can be easily placed in the existing dataset based on their inherent neighborhood similarity. The experimental results show that our representation learning technique combined with automated edge inference improves performance predictions in the HPC domain compared to deep learning modeling.

1.3 Summary of Contributions

- To capture relationships between samples, we transform tabular data into a graph data structure, where each node represents a sample, and each edge represents a relation between two samples.
- We propose a novel representation learning technique that can automatically refine the edges based on feature and sample similarities and use Graph Neural Networks (GNN) to build effective embeddings.
- We develop an end-to-end framework for autonomous graph transformation and representation learning that results in explainable models downstream.

2. BACKGROUND

Since the ML model does mathematical operations on the input data, the categorical data in the tabular dataset needs to be converted into numerical values. This process is called categorical encoding. There are multiple algorithms like One-hot Encoding, Label Encoding, etc. For example, Label encoding detects unique values in the categorical columns and assigns classes to them. The numerical values range between 0 and the number of unique values -1, and based on that, a look-up table is created where each categorical value maps to a numerical number. If the value is repeated, it converts it to the same value assigned previously in the look-up table.

2.1 Data Description

We need to specify the learning type of the algorithm based on the task given. For example, the input data to a supervised learning algorithm, which is an algorithm that learns to predict a target value based on the input features, are a feature and a target vector. In our tabular format case, the input feature vector after transformation will be N x M size, where N is the count of samples and M is the number of columns -1. The target vector size is N x 1, where N is the number of samples, and 1 is the predicted value.
On the other hand, in an unsupervised learning algorithm, the input is only a feature vector. The algorithm attempts to find relationships in the data and groups them together. The input feature vector size is N x M where N is the number of samples and M is the number of columns.

2.2 Preprocessing

Data preprocessing makes the data samples consistent, making it easier for the model to find relationships and patterns. Thus, it is necessary to improve the model's performance, which can be divided into three main steps: data cleaning, data reduction, and data transformation.

In data cleaning, the primary focus is handling missing data, which is common in datasets and usually happens due to the user or program errors. The ML model does not understand missing values, so they must be cleaned and replaced with numerical values. We can use imputation techniques like simple imputer provided by the sklearn library [7]. The simple imputer finds the missing values. Based on the strategy, it replaces all missing values in the column for the given strategy (i.e., it replaces all missing values with the mean value of the column). Other imputation algorithms are KNNimputer [8], IterativeImputer [9, 10], and DataWig [11], which is a Deep Neural Network (DNN) model for learning the imputation of missing values.

Data reduction techniques are commonly used with large datasets, increasing the computation time of the model. Usually, large datasets cause memory errors (e.g., out-of-memory errors) due to the massive amount of data that needs to be running in the memory. A common technique is Principal Component Analysis (PCA) [4] and t-Distributed Stochastic Neighbor Embedding (t-SNE) [12]. Both are used for dimensionality reduction using a mathematical algorithm that reduces complexity and can plot figures in 2-D or 3-D to notice patterns and better visualize the data.

Data transformation enhances the model performance dramatically when the input features have a different scale. For example, if one feature ranges from 0 to 1, and another feature ranges from 1 million to 100 million, the model will understand that the second feature is more critical, giving it more weight in the prediction decision. The most common data transformation technique is data normalization, where all features are normalized into a specific range using a standard scale. Normalization transforms the data into new values while maintaining the correlation and main distribution between data samples. The most popular normalization algorithms provided by sklearn are Standard Scaler [13], Min-Max Normalization [14], and Robust Scaler [15].

2.3 Feature Engineering

Feature engineering is the process of inspecting the dataset and converting the raw data into more fine-grained features. Most feature engineering techniques have a similar approach to preprocessing techniques except for feature creation. Feature creation is the procedure of creating new columns in the tabular format that can help the model correlate and enhance its performance. For example, suppose we are predicting the price of a TV in an electronic store. In this case, a useful Boolean feature can be created, indicating if today is a weekend or a holiday. The model can correlate the current day with two features considering the price prediction.

The ML researchers have used many feature extraction models to understand the data better and achieve the required predictive task. In recent years, approaches like deep neural networks have been commonly used for modeling. Feature extraction models have shown a considerable improvement in predicting specific tasks compared to neural networks and have been the focus ever since.

2.4 Neural Networks and Deep Learning

Neural networks have been a topic of research for over 70 years. It started in 1943 when McCulloch and Pitts provided a way to describe brain functions in abstract terms. They proposed the idea of a neuron where simple elements connected in a neural network can have immense computational power [16]. In 1949 Introduced the Hebb's low, which explained how close neurons fire together, making coupling tighter, illustrating the basis of neural pathways and synaptic transmission. It was not until 1959 that we had the first computation model represented in the Mark 1 Perceptron, a machine designed for image recognition; it had an array of 400 photocells, randomly connected to the neurons. Weights were encoded in potentiometers, and an electric motor performed weight updates during learning. At this time, Frank Rosenblatt went into a public academic debate with Marvin Minsky, who was an MIT professor researching neural networks as Rosenblatt claimed that any classification for which a solution exists will always yield a solution in finite time using his perceptron. Minsky proved that the function was too simple to map underlying distributions, which turned other researchers in this field away from the topic of neural networks in the research community. In 1986, Rumelhart, Hinton, and Williams popularized gradient calculation for the multi-layer network. Until this point, no one could train a multiple-layer network consistently, and the algorithm was popularly called Back-Propagation. It won the pattern recognition prize in 1993 and became the de-facto machine learning algorithm in the 1990s. In 1989 Yann LeCun used back-propagation to learn the Convolutional Kernel Coefficients directly from images of hand-written numbers. Learning was thus fully automatic, performed better than manual coefficient design, and solved a broader range of image recognition problems. In 1995 Sepp

Hochreiter proposed Long Term Short Memories (LSTM), which, unlike feed-forward neural networks, LSTM has feedback connections. It can process single data points (such as images) and entire sequences of data (e.g., text, speech, and videos). In the early 2000s, other tree-based models like Random Forest and Support Vector Machines with kernels gained much traction over neural networks. It was not until 2004 that Hinton secured funding from CIFAR and published a paper in 2006 on using pre-training and Restricted Boltzmann Machines (RBMs), followed by another paper in 2007 claiming that deep networks are more efficient than RBMs. In 2009 Hinton's lab along with Andrew Ng's lab, started using GPUs for training networks which decreased the training time by 70 folds. In 2012, Hinton's lab entered the ImageNet competition and won by over 10% from the second place using convolutional neural networks, and the computer vision community adopted it en masse.

Existing tabular approaches require many labeled data; they cannot self-regulate based on unlabeled samples and need retraining when slightly different data samples are introduced to the model. Additionally, these approaches consider each sample or batch as input without leveraging the similarities or dissimilarities between the input samples regarding the predictive task.

Tabular datasets used in the production environment require extensive preprocessing and feature engineering since both are the key to improving the ML model's performance. This process also requires time, resources, and human involvement. Nevertheless, finding the best representations of the input data is necessary to save resources by choosing less complex models that will still perform well and generate reliable results because the input is well structured. Another commonly used technique is feature importance algorithms (e.g., the Random Forest algorithm) [17]. The Random Forest algorithm is a supervised ML algorithm that automatically ranks features based on their importance, allowing the removal of the poor or irrelevant features. However, when it has empty values in the features, it performs poorly, and if the data sample is unlabeled, it will not work since it is a supervised algorithm.

2.5 Graph Data Structure

A graph is a special kind of data structure consisting of a set of vertices and edges to connect vertices. A graph can be cyclic where at least one path of the starting vertice is the ending vertice or can be acyclic and does not contain any cycles. The edges of the graph can be directed or undirected. A directed graph has one-way relationship connectivity, e.g., node A is connected to node B; however, node B is not connected to node A. Undirected graph means every edge connects the two vertices using a two-way relationship. Graphs are heavily used in real-world applications. For example, on a social network like Facebook, a user can be a node, and when they become a friend with another user, there is an undirected edge between them. Another application is google maps, where graphs are used to find the path from the source to the destination where the data are represented in a graph structure. We ran the shortest path from source to destination to find the fastest available path for the user. We plan to explicitly model the relationships between the input data samples and leverage the findings on a graph-based ML model.

3. RELATED RESEARCH

This chapter will discuss research related to graph building methods for different tasks.

3.1 Deep Modeling Explain-ability using Graphs

Researchers have previously used graphs in model explain-ability where tabular data was transformed into a graph to produce explainable predictions. This model is called TableGraphNet [18]. TableGraphNet architecture builds a single graph or multiple graphs for each data sample. The graph node contains the feature attributes, and the graph edges contain the distance between the feature attributes. Using the generated graphs for the input samples, they extracted node and attribute-centric features for every attribute. They fed the extracted features into a deep neural network to get the desired prediction task. Researchers evaluated their proposed architecture using three classification datasets and eight regression data sets. Their experiments concluded that TableGraphNet performs similarly to a regular Deep Neural Network model. However, the TableGraphNet is robust towards missing data, providing consistency and explainability behind the predictions. TableGraphNet resulted in better predictions due to better explainability, improving the overall model's performance. Their architecture did not perform better than DNN. On the other hand, our architecture gets evaluated with a DNN as a baseline, and the results show significant improvement. For more information about the results, please refer to the results chapter.

3.2 Tabular Data Prediction using Multiplex Graphs

Recently researchers at 4Paradigm published "TabGNN" [19], a new framework architecture in 2021, to leverage relations between samples by building Multiplex graphs

[20]. Multiplex graphs were constructed to model relations between people that can be used in many applications (e.g., friend suggestions on Facebook). Multiple graphs explore the links between two people based on similarity and suggest new connections.

TabGNN framework takes tabular data as input, constructs a directed multiplex graph based on the table columns referred to as features, and encodes each sample for an initial node embedding. Researchers used a graph neural network to produce latent feature embedding for each data sample. After getting the final representation embedding, they use Auto Feature Engine (AutoFE) to choose significant features and then use a Multi-Layer Perceptron to get the final prediction. They evaluated the proposed approach by doing experiments on nine private and two public datasets. The framework heavily relies on AutoFE and DeepFM [21] to become scalable, two popular feature engineering methods used for tabular data. AutoFE is a feature engineering tool developed by 4Paradigm [22] that provides automated feature generation for tabular data and has been used by various users in different fields. DeepFM is a deep neural network architecture based on factorization machines and deep feature learning that automates the feature engineering process and requires only raw data. The experiments showed that utilizing both AutoFE+TabGNN and DeepFM+TabGNN outperforms using AutoFE or TabGNN alone. In contrast, our approach leverage the graph data structure without requiring any complex feature engineering DNNs, significantly improving the training/testing time.

3.3 Knowledge Graph for Efficient Meta-learning

Researchers explored the idea of graph structure to improve Meta-learning [23] in recent work like Automated Relational Meta-learning (ARML) [24]. ARML generates multiple prototypes from training samples based on tasks given. Each task has a prototype feature vector representing it and is utilized to build a knowledge graph called the “Meta-knowledge graph”. The meta-knowledge graph then computes the similarity between the prototypes and itself to update the graph. The output graph, which they call SuperGraph, contains information regarding previous samples and successfully adapts when a new task is introduced into the model by finding the most similar structure in the meta-graph.

The Researchers evaluated their proposed approach on a 2D toy (random numbers) for regression and image classification. They concluded that ARML leverages the meta-knowledge graph to obtain fine-grained structure more than other gradient-based meta-learning implementations. The single limitation of this implementation is that it can only be used with meta-learning algorithms and has been evaluated by image classification datasets that are not in tabular format. The ARML architecture uses DNN to transform the input data into a better representation. One idea to leverage the ARML architecture is to use our approach for the data transformation to produce more fine-grained representation leveraging both feature extraction and the relationship between the input data.

4. METHODOLOGY

This section will discuss the core concepts of our proposed approaches, data preprocessing, and embedding generation.

We propose transforming tabular data as a graph, where nodes represent samples and edges represent relationships between the samples. We hypothesize that holistically capturing the relationships across samples and parameters enabled by the graph formulation can improve the effectiveness of the downstream ML tasks.

Unlike other research domains where the graph structure is explicitly provided, the graph from tabular data needs to be constructed. While data samples can directly map to nodes, edges between samples must be defined explicitly or inferred automatically. The rationale for organizing data as a graph compared to the state-of-the-art dictionary-based approach is that a graph structure can inherently describe relationships among features and samples, allowing downstream ML tasks to capture information from relevant neighbors by exploiting these interrelations. On the other hand, the dictionary-based approach only leverages feature extraction to generate the representation vector. In contrast, the graph structure leverages both feature extraction and the relationship between the samples in generating the representation vector.

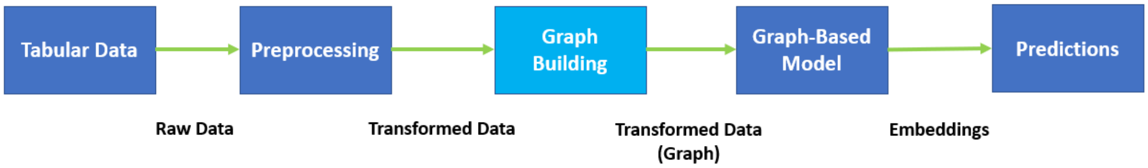


Figure 2. Our proposed ML pipeline.

The graph design adopts the architecture of an undirected weighted graph G = (V, E, A) to represent a performance dataset, where each node V_i ∈ V denotes a sample, and edge ∈ E denotes a relationship with another measurement. A ∈ $R^{N\times N}$ is an adjacency matrix that specifies the weights on the edges, where $A_{i,j}$ corresponds to the edge weight between nodes V_i and V_j that can be calculated based on their similarities.

For data that belongs to the Euclidean space, A_{ij} can be calculated using a distance metric between V_i and V_j. However, for non-Euclidean or dense data such as a tree, building A_{ij} using Euclidean distance is not meaningful. Hence, instead of using explicit distance measures to construct the graph, we propose to learn $E_{ij}$and A_{ij} through automated graph edge inference method using self-supervised learning, which starts from a fully connected graph with measurements as node features and then iteratively refines node and edge features based on their similarities without any explicit distance measure.

4.1 Self Supervision

Supervised learning is a bottleneck for building more intelligent models that represent the world and can do multiple tasks without massive amounts of labeled data. Practically speaking, it is impossible to curate everything in the world and give it a label. We propose that using Self-supervised learning (SSL) enables ML systems to learn from the order of magnitude more unlabeled data, which is essential to recognize and understand patterns of more subtle, less common representations of the world. The main idea of SSL is to automatically generate a helpful representation of the input data and learn using a supervisory signal from the model that helps achieve the desired task. The advantage of SSL is that it learns from unlabeled data. On the other hand, supervised learning algorithms require extensive labeled data, which is time-consuming and hard to collect.

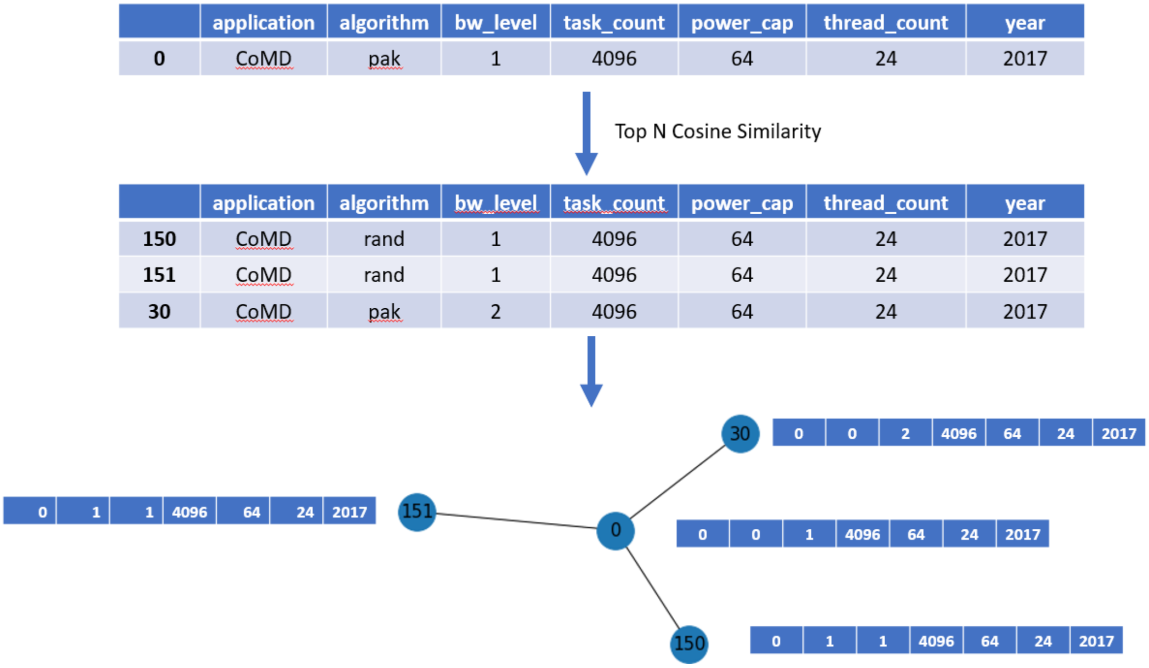
SSL algorithms have been heavily used in recent years due to their promising results and the breakthrough of developing Transformers, which was a massive milestone in Natural Language Processing (NLP) and resulted in state-of-the-art models in NLP like OpenAI GPT2 [25] and Google's BERT [26]. We effectively use self-supervised learning to develop an automated edge-inference-based method that optimizes the graph edge weights.

4.1.1 Automated Edge-inference-based

The automated edge-inference-based method has been shown to create effective embeddings that improve the predictive performance of the downstream analysis algorithms [27]. After building the graph, we will use graph-based models like Graph Neural Network (GNN) [28], Graph Convolutional Network (GCN) [29], or Graph Attention Layer (GAT) [30] for the embedding generation and use a simple model, e.g., Linear regression for the predictive task.

4.2 Neighborhood Similarity

Our hypothesis is to leverage input features and relationships between the input samples. So In order to determine which samples are related or similar to each other, we use similarity distance algorithms to detect the relevance between multiple samples. The most popular algorithms regarding distance calculation in 2-D data are Euclidean, Manhattan, and Minkowski distances. Since most data scientists use multi-dimensionality datasets and have many features, we chose the Cosine and Mahalanobis algorithms since they function better in higher dimensionality. We found that Cosine Similarity performs better, so that was our choice. The cosine similarity works by plotting the data on an N-dimensional plot based on the number of features and then measuring the cosine of the angle between the current sample and all other samples. The high cosine similarity indicates that the angle between the two samples is slight. Therefore, the two samples are similar to each other. To further understand, let us look at figure 3. We take sample 0, calculate the cosine similarity between all other samples, and then choose the top N similar samples. N is a hyperparameter that the user can change, and it determines the number of edges in the graphs.



	application	algorithm	bw_level	task_count	power_cap	thread_count	year
0	CoMD	pak	1	4096	64	24	2017

	application	algorithm	bw_level	task_count	power_cap	thread_count	year
150	CoMD	rand	1	4096	64	24	2017
151	CoMD	rand	1	4096	64	24	2017
30	CoMD	pak	2	4096	64	24	2017

Figure 3. Example of building a graph from cosine similarity. We utilize the Top N approach, which is the number of edges.

In figure 3, N is equal to three, and after finding the top three samples (150, 151, and 30) based on the cosine similarity, we create a node for each of them where each node contains the sample's feature then build the edges accordingly. This process happens for each sample within the dataset, so the final output is a large graph representing the dataset set and the relationship between the samples.

4.3 Graph-based models

Recent research has found that neural network-based learning on graphs [31] accommodates unlabeled data and outperforms traditional convolutional neural networks. The graph can learn to represent graph nodes, edges, and sub-graphs in low-dimensional vectors, which captures relationships across samples and parameters to embed the whole dataset effectively and learn from unlabeled samples by exploiting their similarities. The graph will be used in multiple tasks like node classification, link prediction, community detection, or graph classification.

The most common graph models are GNN and GCN, so let us discuss their differences. GNN is a specific kind of Neural Network (NN) [32] that takes graph data structure as an input. The main advantage of the GNN over the NN is that the learning equation includes the neighborhood of the current node vector representation based on the edges connected with the current node. This operation is known as message passing. Researchers kept developing the GNN architecture until GraphSage [33] was introduced in 2017. GraphSage optimized the learning equation by learning the vector representation of each node inductively and having more complex aggregation functions for the message passing. That makes the GNN perform better and generalize for unseen samples during the deployment of the model by representing the unseen sample to the neighboring nodes. GCN is a specific kind of Convolution Neural Network (CNN) [34]. The GCN behaves similarly to the GNN, where we include the message passing perspective in the learning equation. The main difference is the hidden layers architecture. The GCN stacks several graph convolution layers to extract a high-level node vector. On the other hand, the GNN contains Multi-Layer Perceptron (MLP) [35] for the hidden layer representation to extract the node feature vector.

Later on, graph-based models gained much interest, explicitly improving the GCN architecture, and many GCN architecture variations have been proposed. The GAT model was one of the top-performing variations. GAT has a unique architecture where the model finetunes the neighborhood message using self-attention layers. The purpose of the self-attention layer is to generate the attention coefficient α_{ij} which acts like a weight that defines how $node_j$ related to $node_i$. The attention coefficient effectively improves edge connections and generates overall more fine-tuned message passing embeddings, making GAT a state-of-the-art algorithm broadly used in the research and production industries.

This thesis uses the graph-based model to generate a vector representing each node. Then apply semi-supervised node regression, which produces a regression value for each node.

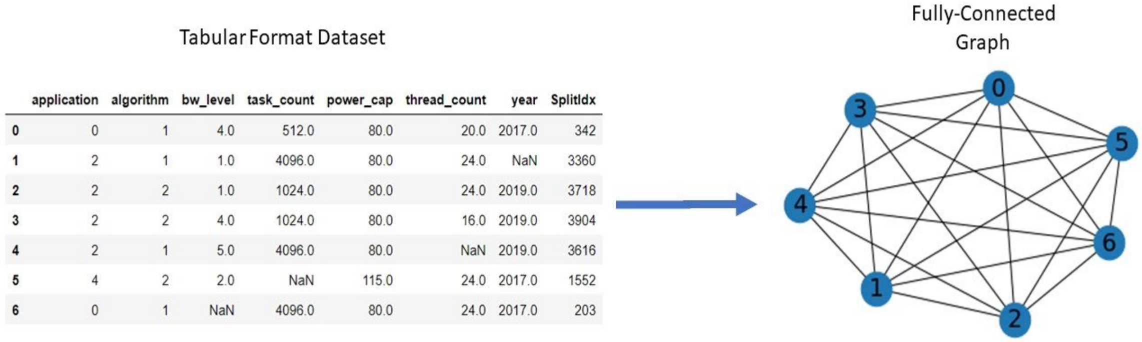
Based on a deeper understanding of the problem, we developed two main approaches for graph generation.

4.4 Graph Generation

This sub-section explains how to generate the graph from the tabular data and provides a general discussion about each approach and how to optimize it further.

4.4.1 First Approach: Single Graph

Building a single graph for the whole dataset where each node is a data sample and the edges can be fully connected. The edge weights will be initialized by one value and updated using self-supervision learning. The model will encode the input data to discover valuable representations, giving higher edge weight to similar data and lower edge weight to dissimilar data. This approach has high computation cost, can cause limitations by the memory based on the dataset size, and can cause averaging effect since the graph is fully connected, resulting in having an edge between two unrelated nodes.



	application	algorithm	bw_level	task_count	power_cap	thread_count	year	SplitIdx
0	0	1	4.0	512.0	80.0	20.0	2017.0	342
1	2	1	1.0	4096.0	80.0	24.0	NaN	3360
2	2	2	1.0	1024.0	80.0	24.0	2019.0	3718
3	2	2	4.0	1024.0	80.0	16.0	2019.0	3904
4	2	1	5.0	4096.0	80.0	NaN	2019.0	3616
5	4	2	2.0	NaN	115.0	24.0	2017.0	1552
6	0	1	NaN	4096.0	80.0	24.0	2017.0	203

Figure 4. Transforming tabular data into a fully connected graph.

The number of nodes represents the number of samples in the fully connected graph. In Figure 4, the Fully connected graph contains seven nodes and 42 edges representing the dataset. To optimize this approach further, we used a cosine distance measure to generate the edges between N similar nodes, where N is the number of edges connected to each node. N is treated like a hyperparameter that the user can adjust for better model performance or specify based on memory-specific constraints.

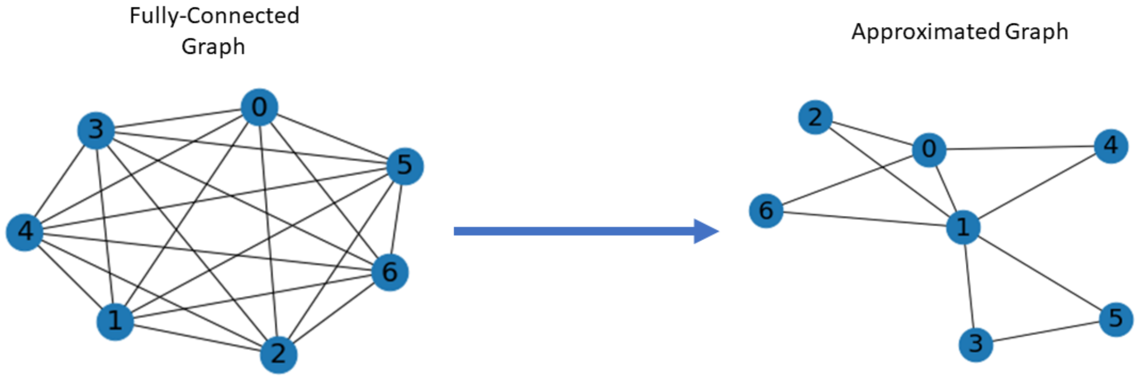


Figure 5. Reducing the graph into a more approximated graph.

In the above figure, we chose N =2 for easier visualization, and we can see that each node has only two output edges connected to the nodes with the highest similarity.

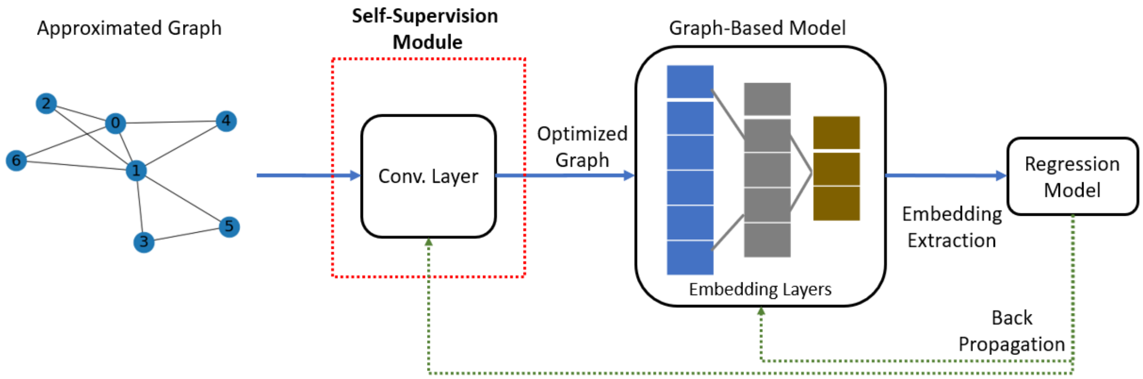


Figure 6. The framework uses a self-supervision module to generate more fine-tuned edge weights iteratively through the convolution layer. Then we feed the optimized graph to the graph-based model and extract the embeddings from the last hidden layer of the model to evaluate its effectiveness on a simple regression model.

The figure above explains our ML pipeline. We extracted the embedding from the hidden layer of the graph-based model to get embeddings that represent the features and their relationship. Then fed the embeddings to a simple model to see the improvement percentage.

4.4.2 Second Approach: Batched Graph

Generating clusters using unsupervised learning to find groups within the data based on similarity to build M graphs, where M is the number of clusters. This approach is memory efficient since we have batches of samples connected together and positively leverage the averaging effect. We still have high computation cost due to N*N-1 edges being fully connected. To further optimize this approach, we can reduce the number of edges using top N similar nodes to each node as explained in the first approach.

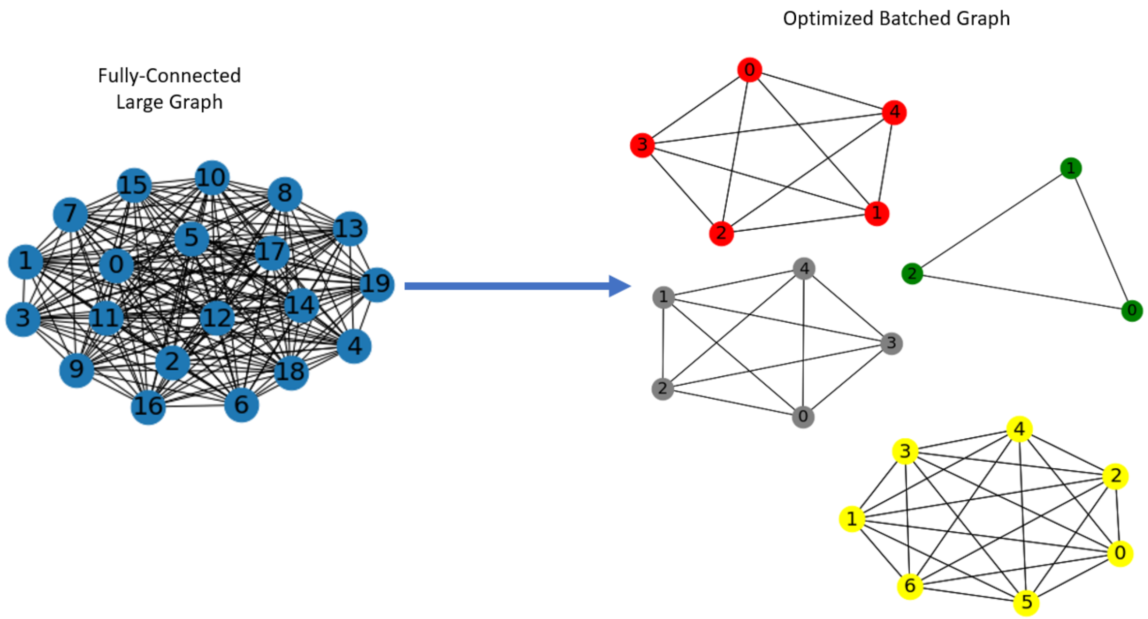


Figure 7. Transforming the graph into smaller batched graphs based on clusters found. In the figure above, there are 5 clusters found, and each cluster has a different number of samples. For example, the red cluster has 5 data points while the green cluster has 3.

This approach's performance heavily relies on the chosen unsupervised learning algorithm. Based on several experiments, we found that some clustering algorithms, e.g., K-means clustering [36], are sensitive to noise and tend to group one large cluster and multiple small clusters that are usually outliers. We chose Hierarchical Density-based Spatial Clustering of Applications with Noise (HDBscan) [37] due to its parameter flexibility, where we can adjust parameters like "min_samples" and "min_cluster_size" to control minimum cluster size and when to split up one large cluster into two or more clusters. HDBscan encodes the input samples into transformed space according to their density and then builds a minimum spanning tree (MST) using standard MST algorithms like Prim's [38] to find the cluster hierarchy. The final step in HDBscan is to extract the generated cluster, so we first condense the cluster using the minimum cluster size parameter and then extract the remaining parts.

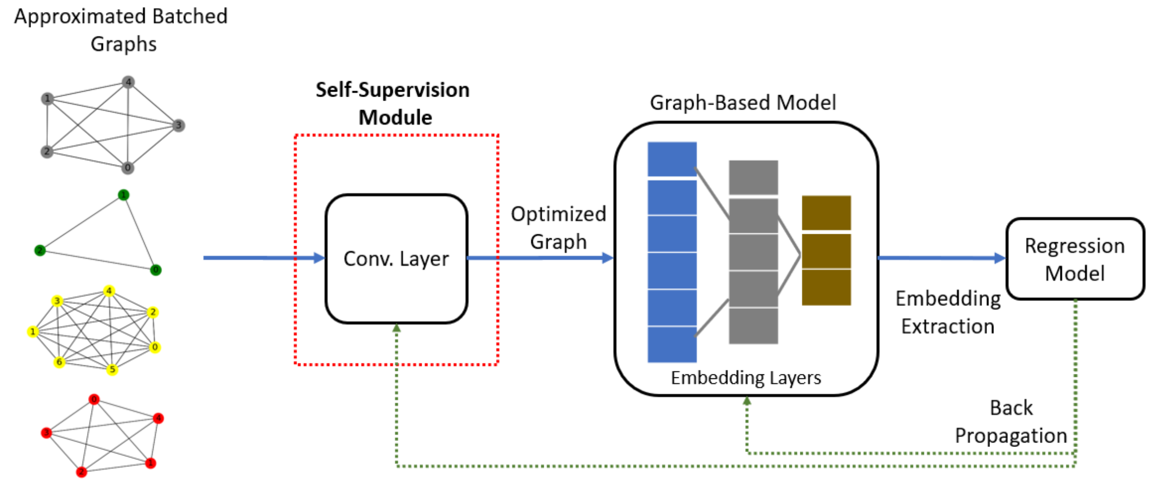


Figure 8. The Approximated batched graphs get fed to the pipeline one graph at a time. The framework uses a self-supervision module to generate more fine-tuned edge weights iteratively through the convolution layer. Then we feed the optimized graph to the graph-based model and extract the embeddings from the last hidden layer of the model to evaluate its effectiveness on a simple regression model.

We used the same top N technique as the first approach based on cosine similarity to optimize this approach further and prevent the averaging effect. In approaches one and two, we used the graph-based model as GNN, specifically, the deep graph library implementation called GraphSage [39, 40].

4.5 Preprocessing

Since the core of our approach is to find a better representation of the input data automatically, we used minimal processing, precisely three preprocessing techniques, categorical encoding, standardization, and imputation of missing values.

Categorical Encoding: Since the ML model does mathematical operations on the input data, the categorical data in the tabular dataset needs to be converted into numerical values. This process is called categorical encoding. There are multiple algorithms like One-hot Encoding, Label Encoding, etc. We used the One-hot Encoding implementation by Pandas library [41]. One-hot encoding detects unique values in the categorical columns and creates a binary column for each unique value. This column will have a value of 1 for each sample corresponding to its original column and assign a 0 otherwise.

Standardization: To standardize the input data into a common scale, we transform the data using the standard scaler algorithm provided by sklearn [13]. The standard scaler removes the mean and scales the data based on the unit variance so that the scaling of each feature is independent of the other features, but eventually, they are all scaled in the same way and have equal weights.

Imputation: To address the missing values in the input data, we used Simpler Imputer provided by sklearn [42]. The simple imputer finds the missing values and replaces all missing values in the column based on the selected strategy. We used the strategy parameter as the mean, replacing the missing value with the feature column's mean.

4.6 Classifier Optimization

4.6.1 Generalization

We used the drop-out layers concept to prevent overfitting and assure model generalization.

Drop-out Layer: Randomly dropping neurons in the hidden layer is a well-designed technique developed in 2013 [43] to counter the overfitting problem. It works by randomly dropping some of the edges to the subsequent layer neurons during the training phase. We used the technique effectively on the input and hidden layers of the model and chose the drop-out probability using a hyper-parameter optimization tool. The drop-out technique does not affect the testing set since it is only used during training.

4.6.2 Optimizer

The loss gets calculated based on the loss function in every epoch, and the optimizer tweaks the model parameters to minimize the loss. In other words, the loss function is the guide for the optimizer, making sure it is working in the right direction. This makes the optimizer a key parameter in the model performance. We used the Adam optimizer [44]. Adam is commonly used in the ML field due to being computationally efficient, memory-optimized, and robust against noisy gradients. Adam is built based on two algorithms, The Root Mean Square Propagation (RMSProp) [45] and the Adaptive Gradient Algorithm (AdaGrad) [46], taking advantage of both of them and continuing on it. The unique idea of Adam is that it takes the exponential moving mean of the squared gradient and the gradient. Adam also has additional parameters like beta1 and beta2 to adjust the decay rates of the moving mean. Based on that, Adam is the ML default optimizer, and it outperforms most of the other optimizers [47].

4.6.3 Activation Function

The activation function is critical for designing the model since it decides how to propagate the weighted sum of the current layer into the next layer neuron. It is being used in each neuron node in the neural network. Based on multiple experiments, we chose Rectified Linear Unit ReLU. ReLU became widely used due to its simple implementation, faster training of the model, overcomes the vanishing gradient problem, and usually outperforms other activation functions, thus achieving better performance for the model. The ReLU function removes any negative value and replaces it with a zero, and for all positive values, it returns the same value. The following equation explains the implementation of the ReLU:

$$\mathrm{ReLU}(x) = \max(0\ ,\ x)$$

5. IMPLEMENTATION AND EXPERIMENTAL SETUP

5.1 Implementation

We Implemented our code using two main libraries in python:

- Networkx (NX) [48]
- Deep Graph Library (DGL) [49] with Pytorch [50] backend

5.1.1 Networkx

Networkx is an open-source python library that is commonly used for building, visualizing, and studying complex graphs to understand and manipulate the structure of the data.

We developed our code to generate the graph using NX based on similarity measures and initialized edge weight of ones. Then we can extract the adjacency matrix network.linalg.graphmatrix.adjaceny_matrix() function that returns a SciPy [51] sparse matrix of the agency. We also used the nx.draw() function to visualize the graph and understand the ML model's input.

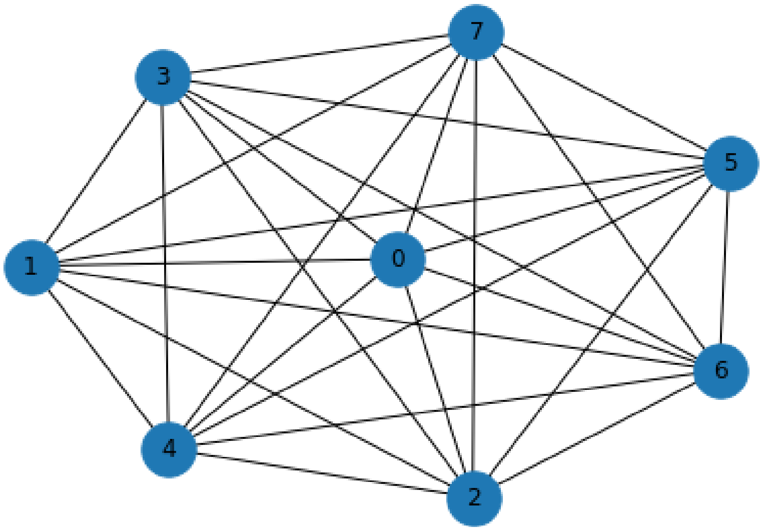


Figure 9. A sample for the NX visualizing a fully connected graph with eight nodes and

56 edges where each node has seven edges.

5.1.2 Deep Graph Library

DGL is an open-source python library developed specifically to provide an end-to-end application for the graph data structure. DGL delivers high-performance and scalable ML algorithms integrated with the back end of major ML frameworks like TensorFlow [52], PyTorch, and Apache MXNet [53]. DGL is efficient regarding memory parallelization due to the usage of message passing primitives that scales complex graphs using distributed training and GPU acceleration infrastructure.

We extracted the NX graph using DGL.from_networkx() function, which returns a DGLGraph object. We then can leverage DGL to do all the heavy lifting and parallelization for the model. DGL provides more than twenty state-of-the-art graph algorithms that can be used in different domains. It has examples and detailed documentation that helps manipulate the algorithms to achieve the required task.

5.2 Experimental Setup

We ran out experiments in parallel using two systems: Penguin Computing On Demand (POD) [54] and Google Colab [55].

5.2.1 Penguin Computing On-Demand

We used a machine with 96 AMD EPYC CPUs. Each CPU has 8GB RAM and is based on the x86 64 architecture. We used 8 MI50 AMD GPUs for the computing power with 512GB total memory.

5.2.2 Google Colab

We used a machine with Intel(R) Xeon(R) CPU with 26.75GB RAM and one NVIDIA Tesla P100 GPU with 16GB memory. The GPU is equipped with CUDA 11.2 toolkit by NVIDIA. The CPU consists of 2.30GHz Intel (R) CPU with two cores based on the x86 64 architecture.

5.3 Performance Metrics

Since we are working with a regression dataset where the label is a numerical value, we cannot calculate the model's accuracy. We estimate the model's performance by seeing the prediction value close to the actual label value. There are multiple techniques following this idea, and let us talk about Mean Squared Error (MSE). MSE is commonly used in statistics and, based on the name, it calculates the error by getting the average square of the two numbers, and the output is always positive. Let us take an example to understand, if our model predicts a value of 50 and the label is 25, then the mean squared is $(\text{predicted} - \text{actual})^2$ which yields to $(50 - 25)^2 = 625$, which means the model error is high because the prediction value is double the actual value for just one sample. The model tries to update its weights to lower that the loss is in the direction of zero loss which is the best case. We do the equation above N times, where N is the number of samples to calculate the loss. More specifically, the average MSE after an epoch (one epoch is where the model passes through all samples one time) is:

$$\frac{1}{N}\sum_{k=1}^{N}(predicited_{\ k} - actual_{\ k})^2$$

In our thesis, we will investigate other metrics to quantify the performance of the ML methods, e.g., using average MSE, Root Mean Squared Error (RMSE), and training time.

We compared the suggested approach using four datasets from different domains. Using DNN and graph-based models, we generate coarse-grained embeddings and evaluate them using a Linear Regression Model as a baseline. The Linear regression model achieves the target prediction by finding a linear relationship between the input and the output. The model adjusts the weights during the training phase to decrease the loss of the learning function and get better predictions. It is widely used in simple ML tasks like predicting the weather or time series analysis due to being fast, efficient, and having less complex architecture than other models. We used the implementation of the linear regression model by sklearn [56].

5.4 Hyper-Parameter Tunning

To achieve the best hyperparameters, we utilized the Optuna framework [57]. Optuna is a widely used hyperparameter optimization framework that does an automatic search for the best parameters and does that efficiently by using state-of-the-art searching algorithms. Optuna also handles large searching spaces and straightforward parallelization and creates trials history that can be visualized easily.

The DNN and graph-based model parameters are optimized in all experiments using Optuna. Specifically, the tuned model parameters are in the following table: the number of hidden neurons, convolution layers, drop-out rate, and learning rate.

Table 1. Hyperparameters search summary

Model	Hyper-parameters	Search range
DNN and	Hidden Layer Dimension	[25, 600]

Graph-Based	Number of Hidden Layers	[2,6]
	Learning Rate	[$1e^{-5}$, $1e^{-1}$]
	Dropout	[0,1]
	Optimizers	[Adam, Adagrad, Adadelta, SGD, RMSprop]
	Activation function	[relu, elu, leaky_relu]
Graph-Based	Hidden Layer Aggregation	[pool, mean, gcn]
	Self Supervision Conv Layer activation function	[relu, elu, leaky_relu, sigmoid, tanh]
	Number of Attention Heads (GAT model only)	[1,2]

5.5 Datasets Description

We used four regression datasets to evaluate our two approaches and the new representation learning technique. We chose the datasets from the HPC domain. Specifically, three datasets were collected from supercomputers in our previous research teamwork.

Catalyst

The dataset was introduced in SC'19 [2], and it was the first dataset collected from multiple systems and various user-level parameters on a production HPC cluster. The features contain numerical values like thread count and categorical values like the application used. The dataset has various information since changes in the number of nodes, cores per node, and service levels were introduced during data collection across five benchmarks applications.

Vulcan

This dataset captured all information being used by the system and was collected from a supercomputer in a production environment. The label of the dataset is the process's runtime. The features contain numerical values, including the number of threads running, reload collisions and input size, and categorical values like the application used. The dataset has various information since adjustments in the input size were introduced during the data collection process.

Cab

The dataset was collected from a supercomputer in a production environment, capturing rich features that decide the process's runtime. The features contain numerical values, including the number of faults, cache references and input size, and categorical values like the application used. The dataset has various information since changes in the input size were introduced during the data collection process.

Table 2. Datasets description

Dataset	Samples	Features	Domain	Target	Evaluation
Catalyst	3992	10	HPC	Runtime	Regression (MSE)
Cab	320	148	HPC	Runtime	Regression (MSE)
Vulcan	321	172	HPC	Runtime	Regression (MSE)

5.6 Cross-Validation

Researchers frequently use the cross-validation technique to estimate the model generalization over the whole dataset by randomly splitting the train/test sets multiple times and evaluating the model by averaging the results. In all experiments, we used K-fold, where K is equivalent to five, meaning we split the data into five train/test splits, generating five different results. Those five splits are randomly chosen to cover the whole dataset without changing anything in the actual data. Then we calculate the mean and standard deviation to get the generalized model performance and range.

6. RESULTS

This section will compare our approach using graph-based models with the DNN approach.

6.1 Model and Embedding Assessment

We assess the model from multiple perspectives to validate our approach. First, to validate the model's ability to learn, we train the model to predict the value, look out for the training loss, and compare it to the testing loss to see if the model can learn and do feature extraction on the input. Then we notice the difference between training and testing losses and see the model's ability to generalize. If the model is overfitting, we apply drop-out layers and L2 regularization. Second, after making sure the model can learn, we use the same approach as the first step. Still, instead of noticing the model performance, we extract the model embeddings through the hidden layer before the final prediction layer. Then we use those embeddings on a simpler model like linear regression to see the change in performance and how effective the generated embeddings are.

6.2 Experiments

6.2.1 Experiment One

To evaluate the quality of embeddings generated by our graph representation learning method. We use a simple regression model which takes the generated embeddings as an input and outputs the predicted task. We measure the performance using the improvement percentage in the MSE loss using DNN as a baseline. Also, to compare our two approaches, Self-Supervised GNN (SSGNN) and Self-Supervised

Batched GNN (SSBGNN), with other existing graph-based models like GAT and GCN-MP.

Table 3. Evaluation of embeddings

Dataset	Catalyst		Vulcan		Cab	
	MSE	Improvement	MSE	Improvement	MSE	Improvement
DNN	21.69	-	617.64	-	140.96	-
GCN-MP	22.28	-2%	510.12	17.4%	142.16	-0.8%
GAT	29.05	-33.9%	536.05	13.2%	133.85	5%
SSGNN	16.39	24.4%	**455.69**	**26.2%**	131.52	6.6%
SSBGNN	**10.46**	**51.7%**	477.49	22.6%	**122.73**	**12.9%**

As shown in table 3, one of our two approaches consistently outperforms all other models. Specifically, SSGNN achieved up to 51.7% performance improvement on the Catalyst dataset, and SSBGNN achieved up to 26.2% performance improvement on the Vulcan dataset.

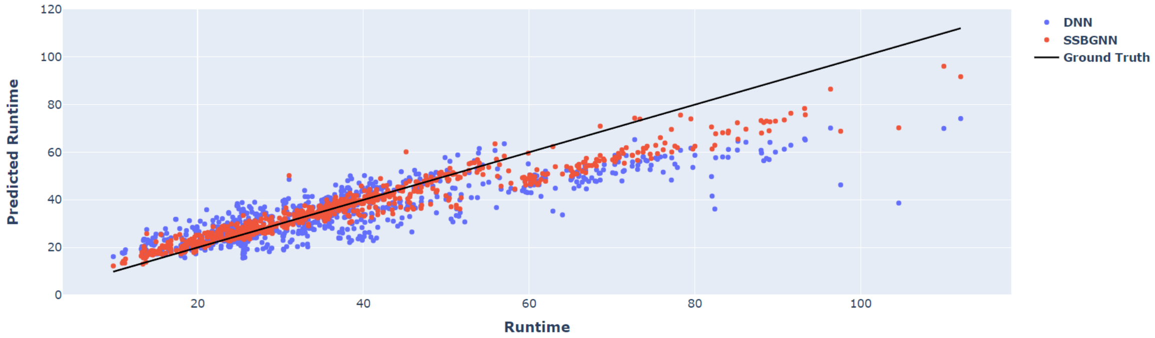


Figure 10. Comparison of DNN and SSBGNN in performance prediction. The x-axis represents the runtime range, and the y-axis represents the predicted runtime.

In figure 10, it is visually apparent that the SSBGNN model (represented in red) is closest to the best fit line of the ground truth compared to the DNN model (represented in blue) and is more robust toward outliers. Also, the DNN usually under-predicts higher runtimes, which will cause the jobs to be resubmitted and waste system resources.

6.2.2 Experiment Two

Since most publicly available tabular data contain a lot of missing values, we concluded that to evaluate the robustness and generalizability of the embeddings generated, we need to make multiple manipulations to the data. Based on that, we did five experiments where we randomly injected a specific missing values ratio on each dataset, and this ratio is 5%, 10%, 15%, 20%, and 25%. After manipulating the input, we evaluate and compare the generated embeddings on multiple models like DNN, GNN, GCN-MP, and GAT.

As shown in figure 11, we introduced the three HPC datasets with different missing values percentages. The results show that our approach one, SSGNN, and approach two, SSBGNN, consistently improve the prediction performance better than all other models. More specifically, they performed the best in 16 out of 18 experiments.

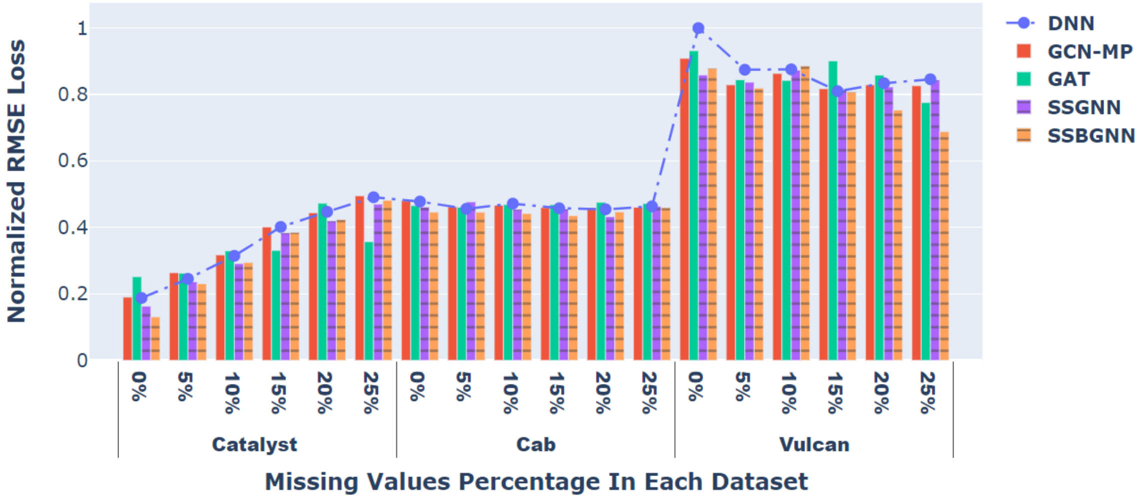


Figure 11. Summary of evaluation of the five models on the HPC datasets: the y-axis represents the normalized RMSE value. The x-axis represents the Dataset name and the missing value per dataset percentage.

6.3 Impact of Graph Representation

As shown in the experiments, we can notice that most of the time, the graph-based representation learning approach outperforms the DNN. The graph-based models reveal the proposed representation learning method's effectiveness and confirm the usefulness of the relations between the data samples for the tabular data. It can be leveraged during the model prediction.

The automated edge-inference-based method using self-supervision combined with the graph representation, which is the core of our two approaches, outperformed all the other models in 88.8% of the experiments.

6.3.1 Online Prediction

Another significant benefit of graph representation is online prediction. New samples fed to the model in an online matter will first see the similar samples in previous data that the model has trained on. Then create edges between them to build the graph and get enhanced predictions since the model leverages their relationship.

6.4 Computational Overhead

We evaluate the computational overhead to ensure the feasibility of the graph representation learning method and its usage in the production environments. Figure 12 evaluates the average of multiple training runs for 500 epochs.

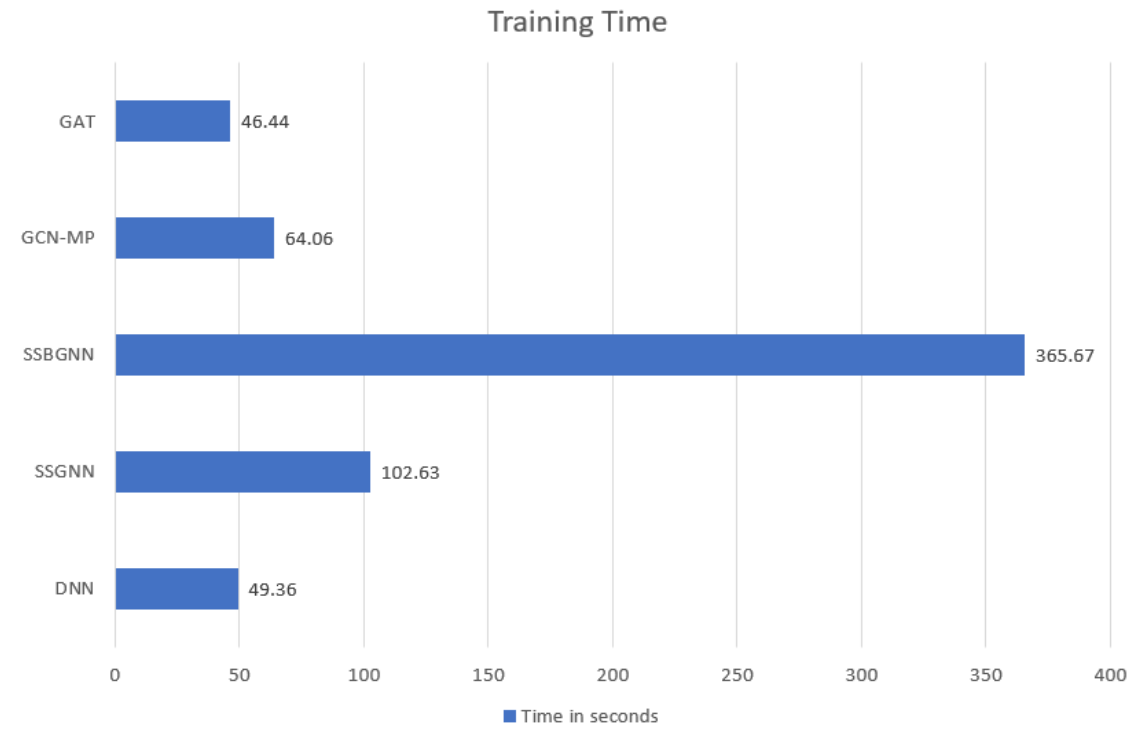


Figure 12. Training time for each model based on 500 epoch, the y-axis represents the model name, and the y-axis represents time in seconds.

We can notice that, on average, the graph-based models are taking more time to train because of the extra time to load the graph into memory and do operations on it. This extra time can be acceptable considering the performance improvement mentioned in Figures 10 and 11. Compared to the other solution, manual feature engineering needs a human to provide more fine-tuned input. SSBGNN takes significantly more time than all other models because it loads batches of graphs, and DGL implementation of graph batch loader API is a bottleneck for the training phase. However, since SSBGNN uses batch training, we can optimize the training even further by using multiple GPUs, which will dramatically decrease the training time.

7. CONCLUSION

In this thesis, we proposed a new representation learning method and developed two models that use an automated edge-inference-based method using self-supervision. Our primary motivation for the thesis is to enhance tabular datasets that are commonly used in real-world ML production systems, such as runtime prediction for the HPC domain. The critical element in the graph representation is that it leverages samples relationships and input features together to create more fine-grained embeddings. On the other hand, most deep learning approaches only use input features without leveraging the sample relations.

Given the tabular data as an input, we construct a graph out of it by using a cosine similarity distance measure where the number of edges is a parameter that can be defined by the user and reflects on the samples similar to the current sample. Then we feed this graph to the self-supervised graph-based model, and in the last layer, we extract the embeddings to assess it on a linear regression model.

We evaluated our approach using multiple graph-based models with a DNN model on four datasets with a total of 18 experiments. On average, graph-based models outperform the DNN model. We integrated an automated edge-inference-based method using self-supervision with the GNN model to optimize the graph-based model architecture (approaches one & two) to achieve the best performance in 88.8% of the experiments. With that said, graph representation learning significantly enhances the embeddings produced and all four datasets. It reveals the effectiveness of the relationship between samples in tabular datasets and the resilience toward missing values in the dataset.

REFERENCES

[1] "House Prices - Advanced Regression Techniques," [Online]. Available: https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview.

[2] T. Patki, J. J. Thiagarajan, A. Ayaka and T. Islam, "Performance optimality or reproducibility: That is the question.," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. (2019).*

[3] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *nature, 521(7553), 436-444.,* (2015).

[4] M. E. Tipping and C. M. Bishop, "Probabilistic principal component analysis," *Journal of the Royal Statistical Society: Series B (Statistical Methodology),* pp. 611-622.

[5] T. Chen, T. He, M. Benesty, V. Khotilovih, Y. Tang, H. Cho and K. Chen, "Xgboost: extreme gradient boosting," in *R package version 0.4-2, 1(4)*, 1.4 (2015): 1-4.

[6] S. O. Arik and T. Pfister, "Tabnet: Attentive interpretable tabular learning.," in *In AAAI (Vol. 35, pp. 6679-6687).*, (2021).

[7] F. Pedregosa, "Scikit-learn: Machine learning in Python," *the Journal of machine Learning research 12 (2011),* pp. 2825-2830.

[8] O. Troyanskaya, "Missing value estimation methods for DNA microarrays," *Bioinformatics 17.6 (2001),* pp. 520-525.

[9] S. Van Buuren and Karin Groothuis-Oudshoorn, "mice: Multivariate imputation by chained equations in R," *Journal of statistical software 45 (2011),* pp. 1-67.

[10] S. Buck, "A method of estimation of missing values in multivariate data suitable for use with an electronic computer.," *Journal of the Royal Statistical Society: Series B (Methodological) 22.2 (1960),* pp. 302-306.

[11] F. Biessmann, "DataWig: Missing Value Imputation for Tables," *J. Mach. Learn. Res. 20.175 (2019),* pp. 1-6.

[12] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research (2008).*

[13] "Sklearn, Preprocessing, StandardScaler," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[14] "Sklearn, Preprocessing, MinMaxScaler," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler.

[15] "Sklearn, Preprocessing, RobustScaler," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.RobustScaler.

[16] W. S. McCulloch and P. Walter , "A logical calculus of the ideas immanent in nervous activity.," *The bulletin of mathematical biophysics 5.4 (1943),* pp. 115-133.

[17] L. Breiman, "Random forests," *Machine learning 45.1 (2001),* pp. 5-32.

[18] G. Terejanu, J. Chowdhury, R. Rashid and A. Chowdhury, "Explainable Deep Modeling of Tabular Data using TableGraphNet," arXiv preprint arXiv:2002.05205 (2020).

[19] "TabGNN: Multiplex Graph Neural Network for Tabular Data Prediction," arXiv preprint arXiv:2108.09127 (2021).

[20] L. M. Verbrugge, "Multiplexity in adult friendships," Social Forces 57.4 (1979): 1286-1309.

[21] H. Guo, R. Tang, Y. Ye, Z. Li and X. He, "DeepFM: a factorization-machine based neural network for CTR prediction," arXiv preprint arXiv:1703.04247 (2017).

[22] "4Paradigm," [Online]. Available: https://en.4paradigm.com/index.html.

[23] R. Vilalta and Y. Drissi, "A perspective view and survey of meta-learning," Artificial intelligence review 18.2 (2002): 77-95.

[24] H. Yao, X. Wu, Z. Tao, Y. Li, B. Ding, R. Li and Z. Li, "Automated relational meta-learning," arXiv preprint arXiv:2001.00745 (2020).

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language models are unsupervised multitask learners.," *OpenAI blog 1.8 (2019),* p. 9.

[26] J. Devlin, M. W. Chang, K. Lee and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805 (2018).

[27] J. Kim, T. Kim, S. Kim and C. Yoo, "Edge-labeling graph neural network for few-shot learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019.

[28] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, "The graph neural network model," in *IEEE transactions on neural networks 20.1* , (2008): 61-80.

[29] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," arXiv preprint arXiv:1609.02907., (2016).

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio and Y. Bengio, "Graph attention networks," arXiv preprint arXiv:1710.10903, (2017).

[31] M. Wu, S. Pan, L. Du and X. Zhu, "Learning graph neural networks with positive and unlabeled nodes," in *ACM Transactions on Knowledge Discovery from Data (TKDD), 15(6), 1-25*, 2021.

[32] R. Hecht-Nielsen, "Theory of the backpropagation neural network," *Neural networks for perception. Academic Press,* pp. 65-93, 1992.

[33] W. Hamilton, Z. Ying and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems 30* , (2017).

[34] J. Schmidhuber, "Deep learning in neural networks: An overview," in *Neural networks 61*, (2015): 85-117.

[35] S. Haykin and R. Lippmann, "Neural networks, a comprehensive foundation," *International journal of neural systems 5.4,* pp. 363-364, (1994).

[36] S. Lloyd, "Least squares quantization in PCM," in *IEEE transactions on information theory 28.2* , (1982): 129-137.

[37] L. Mclnnes, J. Healy and S. Astels, "hdbscan: Hierarchical density based clustering," in *J. Open Source Softw., 2(11), 205*, 2017.

[38] R. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal 36.6,* pp. 1389-1401, (1957).

[39] W. Hamilton, Z. Ying and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems, 30.*, 2017.

[40] "Deep Graph Library - GraphSage," [Online]. Available: https://github.com/dmlc/dgl/tree/master/examples/pytorch/graphsage.

[41] "Pandas," [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html.

[42] "Sklearn, Impute, SimpleImputer," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html.

[43] G. E. Dahl, T. N. Sainath and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout.," in *IEEE international conference on acoustics, speech and signal processing.*, 2013.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980., 2014.

[45] G. Hinton, "Neural Network for Machine Learning - Slide 15," [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[46] J. Duchi, E. Hazan and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research 12.7,* (2011).

[47] S. Ruder, "An overview of gradient descent optimization algorithms.," arXiv preprint arXiv:1609.04747 , (2016).

[48] A. Hagberg, D. Schult and P. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pp. 11–15, Aug 2008.

[49] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li and X. Song, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," arXiv preprint arXiv:1909.01315., (2019).

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan and T. Killeen, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems 32*, (2019).

[51] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy and D. Cournapeau, "SciPy 1.0: fundamental algorithms for scientific computing in Python," in *Nature methods 17.3*, (2020): 261-272.

[52] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean and X. Zheng, "{TensorFlow}: A System for {Large-Scale} Machine Learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, (2016), In (pp. 265-283).

[53] T. Chen, M. Li, Y. Li, M. Lin, N. Wang and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," arXiv preprint arXiv:1512.01274., (2015).

[54] "Penguin Computing On Demand (POD)," [Online]. Available: https://pod.penguincomputing.com/.

[55] "Google Colab," [Online]. Available: https://colab.research.google.com/.

[56] "sklearn, Linear Model, Linear Regression," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html.

[57] T. Akiba, S. Sano, T. Yanase, T. Ohta and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *25th ACM SIGKDD international conference on knowledge discovery & data mining* , (2019), (pp. 2623-2631).