Trace-Based Runtime Prediction of Reoccurring Data-Parallel Processing Jobs

Trace-Based Runtime Prediction of Re-occurring Data-Parallel Processing Jobs

Alireza Alamgiralem

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the
Technical University of Berlin
in partial fulfillment of the requirements for the degree
Master of Computer Science

Berlin, Germany
September 15, 2021

First Supervisor:

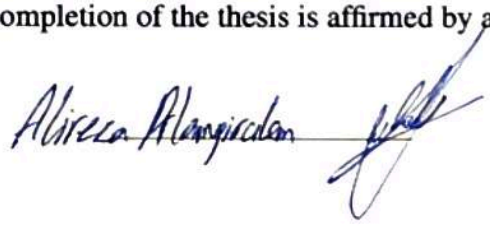
Prof. Dr. Odej Kao, Technical University of Berlin

Second Supervisor:

Prof. Dr. Volker Markl, Technical University of Berlin

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, September 14, 2021 

Zusammenfassung

Die Cloud-Rechenzentren bewältigen täglich zahlreiche Jobs mit aufwändiger Parallelisierung. Um eine so hohe und komplizierte Arbeitslast zu planen und eine effiziente Ressourcennutzung zu erreichen, ist die Laufzeitvorhersage für datenparallele Jobs entscheidend. Darüber hinaus kann eine genaue Laufzeitvorhersage Cloud-Benutzern dabei helfen, ihre benötigten Ressourcen intelligenter auszuwählen. Trotz der Bedeutung der Laufzeitvorhersage ist eine genaue Vorhersage nicht einfach, da die Ausführungszeit von Jobs in komplizierten Cloud-Umgebungen von vielen Faktoren beeinflusst wird, z. B. vom Status eines verwendeten Clusters oder den jeweiligen Benutzeranforderungen.

Die vorliegende Studie schlägt einen neuartigen und vollständigen Workflow vor, um die Laufzeit eingehender Jobs basierend auf Ähnlichkeiten wiederkehrender Jobs abzuschätzen. Für die Erreichnung dieses Ziels werden neueste Techniken aus dem Bereich des Deep-Learning genutzt, um die Jobabhängigkeiten angemessen zu repräsentieren. Anschließend werden verschiedene Clustering-Techniken verwendet, um Gruppen wiederkehrender und ähnlicher Jobs zu identifizieren. Basierend auf den Ähnlichkeiten innerhalb der Stichproben der Gruppen werden schließlich die wahrscheinlichen Laufzeiten vorhergesagt. Die Entwicklung und Evaluation des vorgeschlagenen Ansatzes basiert auf einem kürzlich veröffentlichten Trace-Datensatz, wodurch Informationen aus mehr als 200.000 realen Jobs Berücksichtigung finden.

Die ausgeführten Experimente bestätigen, dass die Struktur von Jobabhängigkeiten einen prädiktiven Wert hat. Tatsächlich kann der entwickelte Ansatz den mittleren absoluten Fehler der Baseline-Vorhersage um 27% reduzieren. Darüber hinaus zeigen die Auswertungen, dass der Ansatz zur Identifizierung und Gruppierung wiederkehrender Jobs sehr zuverlässig ist. Somit kann die vorgeschlagene Methodik zukünftigen Arbeiten für die weitere Erforschung verschiedener Anwendungsfälle, wie der Optimierung der Ressourcennutzung, dienen.

Abstract

The cloud data centers should daily handle numerous jobs with complex parallelization. In order to schedule such a heavy and complicated workload and reach efficient resource utilization, runtime prediction is critical. Moreover, accurate runtime prediction may assist cloud users in choosing their required resources more intelligently. Despite the importance of runtime prediction, achieving an accurate prediction is not straightforward because the execution time of jobs in complicated environments of clouds is affected by many factors, e.g., cluster status, users' requirements, etc.

The present research proposes a novel approach to estimate incoming job's runtime based on similarities of reocurring jobs. To achieve this goal, we utilize the latest achievements in neural network techniques to embed the job dependencies. Subsequently, we perform multiple clustering techniques to form meaningful groups of reoccurring jobs. Finally, based on the similarities within the groups of samples, we predict runtimes. A recently published trace dataset allows us to develop and evaluate our contribution with more than $200,000$ complex and real-world jobs.

Our experiments confirm that the structure of job dependencies carries predictive value. Indeed, we could overcome the mean absolute error of baseline prediction by 27%. Moreover, the evaluations reflect that our approach to identify and group reocurring jobs is highly reliable. Thus, it can serve future works for further exploration for various use cases like resource usage prediction and optimizing resource utilization.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Dominik Scheinert for the continuous support of my research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance and expertise were invaluable in formulating the research questions and methodology.

My sincere thanks also go to Dr. Lauritz Thamsen and Prof. Dr. Odej Kao for offering me the opportunities to work in such an innovative and productive research group of distributed and operating systems.

Last but not least, I would like to thank my family: My parents Nazila and Hamid, and my siblings: Mylad and Negar, for their spiritual support, their wise counsel, sympathetic ear and emotional support.

Contents

List of Figures

List of Tables

1
Introduction

This chapter first discusses the topic and scope of the present thesis and defines the aspects and framework of our contribution. We explain why the runtime prediction matters by illustrating the use cases and drawing the scope of the thesis. The following section discusses the problems which this thesis is aiming to solve. The third section abstracts how our contribution addresses the problem. Finally, we present the structure of the thesis in the last section.

1.1 Motivation

Cloud computing provides various advantages for the users, e.g., cost-effectiveness, business efficiencies, competitive advantages, etc. Thanks to incorporating researchers and service providers, cloud services in quantity and quality are not comparable with their state in the earliest years of establishment. Current clouds are highly merged with the concepts of data parallelization. Data parallelization allows the cloud providers to utilize the resources efficiently. Nevertheless, scheduling and management of clouds, along with the highly complex and increasing data-parallel jobs are among the controversial issues.

In order to achieve optimal application performance and more efficient resource usage, the scheduler should be aware of several levels of information, e.g., user-specific requirements, the current status of the cluster, estimation of incoming jobs runtime, etc. Accuracy of the aforementioned information results in efficiency, productivity, latency reduction, scalability, optimal utility, cost-effectiveness, and throughput improvement of cloud platforms. Contrastingly, inaccurate information results in inefficient resource management and poses a negative influence on cluster functionality, performance, and costs [1]. Among aforesaid information that incorporates the scheduler to optimize the cluster, the runtime prediction is the decision-maker parameter. Being aware of the future runtime of jobs significantly improves the performance of the scheduler and assists the cloud users and the third parties by providing informative data to purchase the cost-efficient and appropriate resources selection regarding requirements of different circumstances [2, 3]. Despite the importance of runtime prediction, there exist many obstacles to produce accurate predictions, namely, the dynamic nature of incoming jobs and customers' demands, unstable availability of resources, etc. Evidently, the runtime of data production jobs is affected by various factors and, as a result, is hard to maintain.

Moreover, many works outline that a considerable proportion of production jobs are recurring [4, 5, 6]. In recurring jobs, similar operations are usually applied to different datasets and are executed regularly in terms of time, e.g., in hourly or daily fashion. While the target dataset of recurring jobs differs, they have similar dependency structures and characteristics [7]. The present thesis proposes a novel and a complete framework aiming to correct the runtime predictions, utilizing common properties of reoccurring jobs and the patterns within the group of reoccurring jobs.

It is noteworthy to mention that the discovered patterns have other use cases rather than runtime prediction. One can consider the identified patterns as means of resource allocation and cluster schedulers improvement. Nonetheless, for the reason of space, discussion of other use cases falls outside the scope of this thesis.

1.2 Problem Description

Most jobs in the cluster environment include complex data-parallel tasks. A standard way to represent job dependencies is through Directed Acyclic Graphs (DAGs). The structure of DAGs carries a meaningful feature and extensively impacts optimizing resource provision and scheduling. Although the critical role of DAGs structure in analyzing data-parallel workloads, it failed to attract much attention, as cluster providers usually do not publish the dependencies information. To the best of our knowledge, there are only a few available trace datasets like Borg cluster traces from Google [8] and CMU OpenCloud [9] that provide information about the structure of dependencies. Unfortunately, even for these published datasets, the researcher should deal with unwanted customization like data concealment of the raw data due to the companies' policies. Consequently, it is hardly feasible for the researchers to investigate real-world job dependencies and many works are limited to develop and evaluate their approaches based on randomly generated job dependencies [10, 11, 12]. Recently, Alibaba published a cluster trace from real-world production, including job dependencies, which allows the researchers to achieve a deeper insight into the job dependencies and their characteristics in the real-world clusters with significant quantities [13]. By utilizing the published trace dataset, it is possible to evaluate the proposed algorithms in the real world and not by randomly generated DAGs. The released trace dataset additionally provides the possibility to research reoccurring jobs and study their runtime variations based on their corresponding DAG [14].

Despite the capability of DAGs to represent job dependencies, they are not easy to study. Because most of the machine learning and statistical models are optimized to study fixed-size data structures and not discrete data types of graphs. Indeed, although the connections between the tasks are a vital part of job dependencies, they are not easy to explore. Thus, it is reasonable to convert them into a more machine-readable format. During the transformation step, extra caution must be taken to keep the structural feature of DAGs to the fullest extent.

1.3 Contribution

As stated before, the published production traces usually include no dependency information. Consequently, the prior works typically design and evaluate their proposed technique on randomly generated DAGs. Our main contribution is to explore the published trace dataset, characterized by low entropy, aiming to investigate the structure of job dependencies and identify

recurrent jobs based on their common properties. Then, we use the identified groups of reoccurring jobs to estimate the runtime of incoming jobs.

To this end, we apply data integration methods and extract the DAGs by harmonizing the DAGs so that a comparison metric could rightly measure the differences of the DAGs structures. For representation learning of DAGs, we deploy the state-of-the-art achievements in Graph Neural Network (GNN) to propose a model, which is optimized to capture the critical feature of job dependencies. The proposed GNN can process DAGs with arbitrary sizes and shapes and output fixed-size vectors, holding their main features. Our steps proceed by clustering the job dependencies based on their associated representations. Afterward, we develop multiple clustering techniques to measure the predictability of clusters in comparison to the baseline and identify an appropriate way of representation. It is worthy of mentioning that our contribution is not a one-way pipeline and depending on the evaluation results, we optimize the graph representation learning model regarding the evaluation results in the clustering phase.

1.4 Framework and Structure

The present thesis reports our contribution besides the evaluation and is organized as follows:
The second chapter expresses the environment and names the required domain knowledge in section 2.1. The following sections explain the necessary knowledge to understand the motivation, problem and methodology. For this purpose, section 2.2 explains parallelism in the cluster domain. Additionally, section 2.3 expresses the preliminary concepts of graph theory and puts extreme focus on DAGs, as they are a standard way of representing DAGs. The section 2.4 demonstrates the concepts of graph embedding and graph neural network, which is used to learn the representation of DAGs. The idea behind utilized clustering techniques in this thesis is explained in section 2.5. The last section of this chapter (2.6) discusses the idea behind the linear regression model, which we used to generate predictions.

The third chapter is an overview of our contribution and how we merge multiple components to estimate runtimes. First We discuss the abstract of our approach in section 3.1. The next section expresses the preprocessing steps and data integration steps, including transforming the data into DAGs object (3.2). Lastly, the section 3.3 gives an overview of our central methodology, which is graphs clustering. This section discusses our methodology to generate graph embeddings and identifying the clusters using different techniques.

The fourth chapter emphasizes the details of our contribution and clarifies the techniques we used. Section 4.1 analyzes the graph embedding process and discusses how we developed the graph neural network model and robust the model to generate more reliable graph embeddings. The section 2.4 illustrates the GNN model, which is designed to generate embeddings in detail. The third section (4.3) of this chapter describes the clustering techniques and the way we applied them to our target dataset. This section contains two subsections to clarify clustering processes completely.

The next chapter evaluates our contribution. To do that, we first illustrate the target dataset in detail in section 5.1 by representing the statistics and multiple distribution graphs. The forthcoming section (5.2) demonstrates a rule-based classification method that is considered as our baseline in detail. This section also discusses how the baseline techniques are runtime predictive. The section 5.3 measures the performance of the proposed GNN model and defines the metric we use for evaluating the embeddings. The next three sections (5.3,5.4,5.5), examine

the clustering techniques and study their performance in different circumstances. We point out the limitations and discussions regarding our contribution in sections 5.7 and 5.8.

The sixth chapter represents the related works to our contribution. Since there exists no work that follows the similar pipeline as ours, we discuss the similar jobs to different components of our workflow, namely graph embedding (6.1), runtime and resource usage prediction (6.2), reoccurring jobs(6.3) and the works that used the recently published trace dataset of Alibaba (6.4). The final chapter wraps up the thesis by summarizing our contribution and evaluation results.

2
Background

This chapter expresses the required information to understand the problem and contribution of the thesis. The first section is an overview of the target environment and the framework of our methodology. The following sections provide the necessary background to understand the problem and different aspects of our approach.

2.1 Use Case Description

The main objective of this thesis is to extract useful information through patterns inside similar jobs. Accordingly, we need to perform multiple aspects of analysis and developing multiple mathematical approaches as follows:

1. **Trace Dataset Analysis:** the trace dataset publishers provide information regarding the methodology of jobs scheduling. To study the dependencies, one should have a deeper insight into data-parallel jobs and how DAGs illustrate the dependencies. To this end, we provide the necessary background to study job dependencies.

2. **Graph Objects Analysis:** having the data and understanding the representation's methodology enables us to transform the job dependencies to the graph objects. The graphs can effectively represent the data-parallel jobs, as they contain the dependencies besides the task features. The characteristics of graphs, and their ability to represent the relations between the tasks, make them a great candidate for representing job dependencies. Furthermore, many techniques developed over the last several decades to analyze graph objects and extract meaningful information. Consequently, to analyze the job dependencies, basic knowledge regarding graph theory is required.

3. **Graph Embedding:** despite the capability of graphs in representing job dependencies, they are considered as an unfitting data type for many machine learning models. Thus, another step of transformation is required to make the graph objects more machine-readable. Moreover, extra caution should be taken during the transformation step to hold as much information as possible. To do this, we transform the graph objects to fixed-size vectors using a Graph Neural Network (GNN). A well-designed GNN enables summarizing the

graph information without smoothing essential features of the graph. We discuss the idea behind GNN in the following sections.

4. **Clustering:** having the graph embeddings, one can detect patterns and cluster the reoccurring jobs. The data providers do not publish information regarding recurring and reoccurring jobs. Consequently, an unsupervised machine learning technique is needed. There are many proposed techniques to cluster similar samples. In this research, we utilize multiple clustering techniques aiming to answer our requirements in different aspects.

5. **Prediction:** the detected clusters and the common properties between the identified clusters of reoccurring jobs, it is possible to predict incoming jobs runtime. To do that, we perform multiple linear regression models on all the extracted clusters. The linear regression model enables the detection of the patterns between explanatory variables and runtime. Finally, we use linear evaluation techniques to measure the correctness of the predicted runtimes.

In the following, we discuss the essential scientific background to make our proposed approach comprehensible.

2.2 Data Parallel Processing Workloads

A cluster is either a group of servers or other resources that can act as a single system and be accessed via the internet. Clouds provide virtual machines at the request of customers for computational tasks. The cloud resources are geographically distributed.

Modern cluster platforms guarantee high availability, reliability, and scalability. To achieve these goals, schedulers should allocate resources at least in a partial optimal way. Furthermore, the scheduler assigns a plan to any specific job to produce final results effectively and efficiently. The jobs could be an aggregation query from an end-user or a forecasting task to predict the weather through various data sources.

2.2.1 Cloud Computing Scheduler

In the cloud domain, resource allocation and the method of scheduling have a significant impact on performance. As the number of jobs is numerous at the provider's side, scheduling should be done automatically. A good scheduler plays a vital role in efficiently handling accessing different resources and load balancing. The task scheduling problem can be seen as the searching for an optimal mapping or assignment of a set of subtasks over the available set of resources to computational nodes to achieve the desired results. During past years, many scheduling algorithms developed. The algorithms mostly assign a proper job plan based on cost, time, bandwidth, priority, etc. Most of the proposed algorithms tried to ensure the quality of services while keeping the costs down. An example of the scheduler is *improved cost-based task scheduling algorithm*, which considers computational performance and cost of resources as decision-maker factors [15]. The algorithm uses computation and communication ratio to find a partial optimal plan for resource allocation [16, 17].

2.2.2 Task Dependencies

In the cloud domain, task dependencies represent the map of parallelization and how the tasks are connected. In most cases, task dependencies should be available before the job can be started. The map can be shown as a DAG, where the computations tasks are shown as nodes, and the edges reflect the relations. Task dependencies indicate the strategy of parallelization. Two tasks in this scenario are either connected or disconnected, showing if the computation is dependent on results of the neighbor node. Typically, the task dependencies have a MapReduce-like structure and are composed of two components, namely, scatter and gather. The former breaks the computation into multiple tasks, and the latter aggregate the upstream task's computation results to form the final output. Tian et al. argue that the number of tasks and parallelism rate have direct relation. Additionally, the map of task decencies can carry other information such as the number of instances, predicted CPU and memory usage, etc. [14].

2.2.3 Recurrent Jobs

A considerable part of cloud consumers has a demand on a regular manner query. This could be a daily report on selling products, an hourly report on the weather condition, or even a weekly backup. Such jobs are a significant proportion of the total workloads of clouds. The evidence for that is the diurnal pattern of resource usage in the released trace dataset of Alibaba, showing that lots of jobs are executing in a specific time of days periodically. Since the queries of this kind have similar operators and possibly even similar target data, the cloud-side execution strategy also has similar characteristics. For instance, the related task dependencies of recurrent jobs should be similar, as they are raised from the same requirements and, as a result, the same parallelism strategy and task dependencies [14].

2.3 Graph Theory

As mentioned earlier, the job dependencies are mostly representable through DAGs. In this thesis, the central part of our contribution is analyzing the structure of graphs to detect similarities between them. This section discusses the preliminaries to the concept of graph theories, and specifically DAGs.

In discrete mathematics, a graph G is a pair of ordered and disjointed sets (V,E) such that E is a subset of V. The former represents a set of edges, and the latter is a set of vertices. The sets of V and E are finite. An edge $\{x,y\}$ shows the relation between two nodes and is denoted by xy. If xy exists in the set of E, it means the x and y are adjacent or neighbors. It is not feasible to think of graphs as these two subsets, and thus it is a natural step and more readable to draw graphs [18]. Besides the sets of G and E, other sets could also be included in a graph structure to represent the attributes of nodes, vertices, and sub-graphs.

2.3.1 Directed Acyclic Graphs

In graph theory, a *directed graph* is a graph in which the edges are directed, meaning the direction of edges matters. More officially $\{x,y\} \neq \{y,x\}$. A *cycle graph* is a graph that some number of vertices are connected in a closed chain. The cycle graph with n vertices is called C_n. In C_n, n equals the number of vertices and edges because every vertex has exactly two

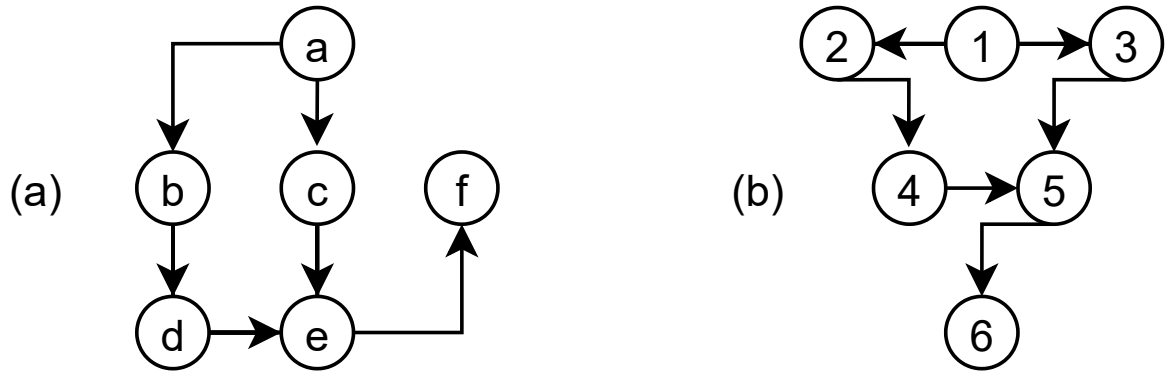


Figure 2.1: Toy example of two isomorphism DAGs. The graphs do not have identical labels, but hold identical dependencies and node numbers.

edges incident with. A *directed cycle graph* is a cycle graph in which the edges are directed and all edges oriented in the same direction. A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles. A DAG comprises vertices and directed edges in a way that following those directions will never form a closed loop. In other words, a directed graph is a DAG if and only if, by arranging the vertices in a linear order, it stays consistent, having all edge directions. DAGs can define relations in many scientific domains such as biology, sociology, and computation (scheduling). The figure 2.1 illustrates two examples of DAGs.

2.3.2 Definitions

In the domain of graph theory, the following definitions should be considered.

Graph Isomorphism: two graphs are isomorphism if they have an equal number of vertices, edges and same edge connectivity. Note that, in the above definition, the labels of edges and nodes are irrelevant. To check isomorphism of two graphs, a one-to-one mapping of edges and vertices of two graphs should be founded in a way the relation of nodes being preserved. In figure 2.1, the graphs (a) and (b) despite the dissimilar node names are isomorphism as they have equal numbers of nodes and there exist a one-to-one map of edges as follows: $\{(a,1) \to (b,2), (a,1) \to (c,3), (b,2) \to (d,4), (c,3) \to (e,5), (d,4) \to (e,5), (e,5) \to (f,6)\}$.

Average Shortest Path Length: is the average of the shortest distance in terms of the number of nodes between all possible pairs of nodes. More officially, it equals:

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}, \tag{2.1}$$

where V is the set of all nodes in the graph G and $d(s,t)$ represent the shortest path between nodes s and t, and n equals with the number of nodes.

Adjacency Matrix: is a square matrix that represents the relation within a finite graph. In simple graphs, the elements in the adjacency matrix show if they are through an edge connected (1) or not (0). The adjacency matrix for an undirected graph is always symmetric, but it can also be an asymmetric matrix for directed graphs.

The Longest Path: in a given graph, the longest path equals the longest existing distance between all pairs of nodes.

Number of Cliques: is a subset C of V in an undirected graph in a way that C is fully connected. It means that in C, every two vertices are connected with a distinct edge. Thus, on a clique C with k vertices, the number of edges equals to:

$$n = \frac{k(k-1)}{2}, \tag{2.2}$$

where k is the number of nodes. The number of cliques in a graph is the number of existing distinct C holding the above condition.

Degree Centrality: within an undirected graph is simply the count of edges associated with every node. For a directed graph, the incoming edges are positives, and the outgoing is negative. In other words, degree centrality equals the node degree, and it can be easily calculated through an adjacency matrix. Assuming $M = (a_{i,j})$ is the adjacency matrix of a directed graph. The in-degree centrality x_i of node calculated by $x_i = \sum_k a_{k,i}$ and the out-degree centrality y_i is $y_i = \sum_k a_{i,k}$

Average Node Degree: equals to sum of nodes degrees divided by number of nodes. More officially, average node degree equals to $\frac{1}{n}\sum_i a_i$ where n denotes the number of nodes and a_i represents the degree of node i.

Graph Density: is a measure to determine the number of edges between vertices compared to maximal possible number of edges. In other words, the edge density measures that how a graph is with edges populated. The density for undirected graphs calculated as follows:

$$d = \frac{2e}{n(n-1)}, \tag{2.3}$$

and for directed graphs, equals:

$$d = \frac{e}{n(n-1)}, \tag{2.4}$$

where n denotes the number of nodes, and e is the number of edges in a graph [19].

2.4 Graph Embedding

Graphs show up in various applications like social analysis, bioinformatics, etc. The remarkable capability of graphs to capture the relations among data entities makes them a superior candidate in comparison with isolated data. In any case, it is regularly challenging to unravel the learning issues on graphs because of two main reasons: (1) many data types are not originally organized in a graph structure, e.g., pictures and text data, and (2) within the graph-structured data, the underlying network designs are regularly complex and assorted which make them not easily interpretable for computers.

On the other hand, embedding learning could achieve great success in different fields. Consequently, in this thesis, we aimed to transform the graphs to finite, fixed-size embeddings

before clustering. Graph embedding is the process of learning the representation of a graph in a low-dimensional metric space, such that the graph characteristics are preserved. Various works address the problem of graph embedding, but the methodologies in this domain are still in their way of evolution. Recently, Graph Neural Network (GNN), which is a deep learning model performed specifically on graphs, has been developed and could outperform other techniques [20].

2.4.1 GNN

GNN models can be divided into different fields. There is a conceptual difference between transductive and inductive GNNs. The former is highly interconnected with the semi-supervised techniques, meaning that the induction on a given graph can not be repeated, and the model should be trained from scratch for incoming new data. DeepWalk [21] and Node2vec [22] are examples of this kind. Contrastingly, the latter is a type of GNN that, utilizing the graph information, tries to extract some patterns that are generalizable for new graph samples [23]. In this thesis, we use GNN of inductive type because we assume that the whole data is not available at first, and we purpose to build a model that enables generating embeddings for incoming DAGs. Another conceptual difference between GNN models is concerning the idea in which the graph data is summarized. The summarizing techniques can be broken into two main groups: spectral and spatial. In Spectral-based approaches, the summarization process is performed through an Eigen decomposition in a Fourier space. The extracted decomposition carries information regarding the structure and underlying features of the graph. The idea behind spatial convolution, on the other hand, is similar to the idea of image convolution layers. This approach is performed to study the spatial properties of the nodes in relation to the neighbors' nodes that are k steps away and the information transformed through the edges. In this thesis, we apply spatial-based approaches because the methods belonging to this category are outperforming spectral techniques and are more computationally efficient [24].

Conceptually, the idea behind GNN is to aggregate nodes of neighbors using a message passing mechanism. In other words, the neighbors of the nodes send their properties, e.g., attributes and their neighbor information, to each other. Any specific node will be notified regarding a great part of the whole graph in a few aggregation steps. The information would be collected cumulatively. That is to say, concerning the size of the graph, a decent number of steps (k) should be defined to avoid a phenomenon called *over smoothing*. Over smoothing in GNN means the process of message passing and aggregating does not match the size of the graph, and the aggregated data is over-mixed such that it is not informative anymore. GNN, in its initial version, assigned random embeddings to the nodes, and through superficial neural network layers, as shown below, capture the information of neighbors node. In the basic version of GNN, the attributes of the nodes are not considered. The neural network layers have the structure of:

$$h_k^i = \sum_{v_j \in N(v_i)} \sigma(W h_j^{k-1} + b), \tag{2.5}$$

where W represents the weights and b stands for bias parameters and are trainable, σ is nonlinearity function [25]. Many aggregation approaches during past years are proposed; each one has its feature and works better in certain domains [26].

Duvenaud et al. proposed a convolution-based propagation rule on graphs [27] called *MFConv*. The primary purpose of this work was to classify molecular data, where the nodes represent atoms and edges are bonds. The authors used a function that computes graphs' fingerprints in the first layer of the neural network. A similar local filter applies to all nodes. The encoding process is based on the Morgan algorithm [28], which makes the relabeling of the nodes and edges irrelevant to extracted vectors. The present thesis utilizes *MFConv* layers, as they are optimized to capture information of small graphs and can capture the nodes feature effectively.

2.4.1.1 Backpropagation

Gradient Descent (GD) is a widely used algorithm to train the neural networks so that the deviation of output minimizes from the target variable. GD identifies a local minimum of function $f(\theta)$ parameterized by $\theta \in R^d$ and categorized in first-order iterative optimization methods. GD finds the local minimum by iteratively taking steps in the opposite direction of the gradient of the objective function $\bigtriangledown_\theta f(\theta)$. The equation below describes how GD detects the local minimum:

$$\theta_b = \theta_a - \gamma . \bigtriangledown f(\theta_a), \tag{2.6}$$

where θ_b is the next point in the opposite direction of the gradient, θ_a is the current point, and γ denotes the learning rate. The learning rate γ indicated the magnitude of the steps to reach the local minimum. The equation 2.6 calculates the gradients for the whole samples. The process of updating the weights through GD is called backpropagation.

Stochastic Gradient Descent(SGD) is a variant of GD that update the parameter for *each* training sample of $x^{(i)}$ and label of $y^{(i)}$ as follows:

$$\theta_b = \theta_a - \gamma . \bigtriangledown f(\theta_a; x^{(i)}; y^{(i)}). \tag{2.7}$$

SGD performs a single update at a time and skips the redundancy of recomputing gradients for similar samples in every update. Therefore, SGD is more computationally efficient in comparison with GD [29].

2.4.1.2 Activation Function

In order to enable training in neural networks, activation should be applied. Indeed, without activation functions, the existence of hidden layers becomes irrelevant. The activation functions add non-linearity to the network, and thus enable the evolution of the neural network in learning the complex patterns. More precisely, the output of the previous layer is first converted through the activation function and then becomes the input of the next layer. There are various activation functions, e.g., sigmoid, softmax, Exponential Linear Unit (ELU), etc. [30]. A major drawback of many activation functions is the problem of *vanishing gradient*. This phenomenon happens when multiple layers of a neural network have a specific activation function. Consequently, the calculation of the gradient approaches zero and slows the learning process [31]. In the present thesis, we aim to select the activation function to avoid the vanishing gradient problem. Following, we discuss the popular activation functions.

Sigmoid Function: is a differentiable function that contains positive derivatives with a degree of smoothness. Sigmoid is primarily used in prediction tasks because the output ranges from

$[0,1]$. The sigmoid function is calculated as follows:

$$f(x) = \left(\frac{1}{(1+exp^{-x})} \right). \tag{2.8}$$

The sigmoid has some drawbacks, e.g., the output is not centered with zero, gradient saturation, lazy convergence, and vanishing gradient problems [30].

Hyperbolic Tangent Function: or *Tanh* is a zero centered activation function that returns a value in range of $[-1,1]$. *Tanh* calculated as follows:

$$f(x) = \left(\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \right). \tag{2.9}$$

The *Tanh* in comparison with sigmoid in many scenarios preferred, as it results in faster convergence. Nevertheless, *tanh*, similar to sigmoid, is unable to solve the problem of vanishing gradient [30].

Softmax Function: calculates the probability distribution of a n-dimensional vector. The output similar to sigmoid is in range of $[0,1]$ and is represented as:

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}. \tag{2.10}$$

The softmax function is commonly used in problems that there exist multi classes of the target value. The softmax calculates the probability of each class, and the class with the highest score will be selected. The major difference between softmax and sigmoid is their use cases; sigmoid is applicable for the binary classification problem, whereas softmax is capable of solving multivariate classification tasks. The softmax is unable to solve the problem of vanishing gradients [30].

Exponential Linear Unit: ELU is differentiable and continuous for all points [32]. For positive points, ELU returns simply the value itself, whereas the output is equal to $\exp(x) - 1$ for negative values. More formally, the output of ELU is calculated as follows:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \exp(x) - 1, & \text{otherwise} \end{cases}. \tag{2.11}$$

ELU offers some advantages over other types of activation functions, e.g., fast training time, avoiding dying neurons problem, avoiding the problem of exploding or vanishing gradients, etc. Thus, it satisfies our requirements regarding graph embedding.

2.4.1.3 Global Pooling Layers

In many cases, the network structure in the last layer of the stack has a fully connected layer. The graph convolutional neural network at lower layers has some convolution or possibly pooling layers to extract patterns. A global pooling layer is placed in the last layers of the network to summarize extracted features from all the nodes in the graph. In other words, the global

pooling layer is a function that aggregates the output of previous layers into the desired size of final embedding. This layer is beneficial as the size of the graphs is not fixed, and with a global pooling mechanism, we can squeeze any arbitrary size of the input graphs into a deterministic fixed size.

2.4.1.4 Regularization Methods

Complexities in the neural network structure prevent the model from generalizing well for unseen data and results in overfitting. In general, regularization techniques prevent overfitting or variance reduction in the convolutional network model by penalizing the model for relatively large weights. Here, we discuss briefly two stand-of-the-art regularization techniques namely, *Droput* [33], and L_2 *regularization* [34].

Dropout: is simply disabling some neural networks randomly with a certain frequency. By performing this technique, we prevent the model from having highly correlated neurons by turning them off once in a while [35]. The dropout mechanism should activate only during the training phase.

L_2 **regularization:** also called weight decay, reduces the complexity of a neural network by turning the weights to a value close to zero by calculating the squared magnitude of all weights. This technique applies to all the weights equally, and thus the model, after regularization, is still sensitive to outliers.

2.5 Data Clustering and Classification

A common task of machine learning algorithms is categorizing the samples in search space and solving the problem of separating the objects employing labels, called classification. Classification is considered a supervised technique. This section highlights rule-based classification, which is one of the basic approaches to the classification problem. Clustering, in contrast, is the unsupervised version of the classification problem. The purpose of clustering is to group similar and unlabeled observations. Each group or cluster should contain similar objects, and they should be dissimilar to objects of other clusters [36]. There are numerous proposed approaches to solve the clustering problem. Here, we discuss three approaches, namely, K-Means [37], Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [38], Ordering Points To Identify the Clustering Structure (OPTICS) [39]. We employ the OPTICS to select proper settings of GNN regarding the aggregation function of the global pooling layer, and the dimensionality of embeddings.

2.5.1 Rule-Based Classification

Rule-based classification refers to the classification scheme which uses the *IF-THEN* rules to identify the classes. An *IF-THEN* rule is usually an expression that consists of one or multiple conditions. Rule-based classification, making use of this expression to divide the search space into classes in such a way that all members of a class satisfy identical *IF-THEN* expressions. We employ rule-based classification to produce the predictions of our baseline. [40]

2.5.2 K-Means

K-Means is among the most popular clustering methods. In this approach, we select K randomly centroids for the K cluster in metric space. Then, using centroids, the objects are assigned to clusters. In the next step, the new centroids are recalculated. This process of reassigning centroids and clusters repeats till some criterion is satisfied. This criterion is usually the stability of clusters after repeating the above loop. To be more precise, this approach is to minimize the following objective function:

$$F = \sum_{j=1}^{k} \sum_{i=1}^{n} |x_i^{(j)} - c_j|^2, \tag{2.12}$$

where c is the centroid, k denotes the number of clusters, and n represents the number of objects in the cluster k. The above function minimized with the following algorithm:

1. Select randomly c centroids for each K cluster.
2. Assign the nearest objects to each c in cluster K.
3. Recalculate the centroids of clusters.
4. Repeat second and third step till convergence.

Clustering using K-means offers many advantages. e.g., the ability to cluster the whole dataset and find the global optimal centroids, but it has two critical issues. First, it is computationally expensive ($O(n^2)$), and second, the number of clusters should be known before running the algorithm [41]. We perform K-means clustering to measure the performance of our methodology in identifying the reoccurring jobs compared to the baseline.

2.5.3 DBSCAN

DBSCAN is a member of density-based clustering approaches. In this family of methods, clustering is performed based on local density conditions. The idea behind these methods is to make borders based on the assumption that sparse objects exist between the cluster. These methods are able to detect clusters even with the different average distances. DBSCAN specifically assumes some objects to be in a cluster when at least $MinPts$ number of samples exists in distance of $radius$. To put it in the other way, the points in the cluster should be more than $MinPts$. This approach is not sensitive to outliers and can detect clusters of any size. Additionally, DBSCAN runtime is faster in comparison with K-Means and is more suitable for the extensive dataset. However, this technique has a big drawback, and it is the necessity of predefined hyperparameters. As mentioned earlier, the $radius$ and the minimum number of points ($MinPts$) should be given, and the model for clustering and outlier detection is susceptible to these parameters [42]. We generate the runtime predictions using the DBSCAN.

2.5.4 OPTICS

OPTICS does not build clusters in the first step; instead, it creates an augmented ordering of the objects in the data, representing the density-based structure. The idea behind OPTICS is very similar to DBSCAN, but it could solve one of the biggest issues of DBSCAN, which is

the weakness of the model to detect clusters with vary of densities automatically. OPTICS firstly order the data points linearly such that the points with less distance become close to each other. In the next step, a dictionary for every point built containing their distance is used to calculate density to form the clusters [39]. Clustering using OPTICS requires two parameters: *radius* and, $MinPts$ like DBSCAN. The key difference is that it also considers all spectrums of smaller *radiuses*. Consequently, OPTICS enable the detection of optimal *radius* compare to DBSCAN. It is also possible to set the default *radius* value to infinite. In this case, the algorithm can detect the clusters even if they have a substantial variation in the average distance of clusters. Compared with DBSCAN, OPTICS puts a heavy workload on the CPU and is memory intensive, which is the main drawback of the algorithm. Hence, we utilize OPTICS to determine the hyperparameters of the graph embedding process.

2.5.5 Clustering Evaluation

The evaluation of clustering algorithms is not straightforward, as the ground truth is almost always unavailable. As a result, standard evaluation techniques, e.g., confusion matrix-related metrics, are not applicable in the absence of ground truth. Nevertheless, during past years, many techniques developed to evaluate the performance of the clustering technique in different aspects, e.g., homogeneity and completeness, where the latter evaluate if the samples of a class are not clustered in different groups and the former check if all the samples of a class are considered as a cluster [43]. It should be noted that these metrics are applicable when the ground truth regarding the classes is available. There are also evaluation techniques, which provide a metric to measure the similarity of two clustering techniques. Based on domain knowledge, one can consider a clustering approach as baseline, and measure the similarities with other clustering techniques. Indeed, the evaluation of clustering techniques is a comparison of multiple clustering approaches.

It is worth mentioning that, during the evaluation, the absolute values of cluster labels are irrelevant, and the evaluation should verify how well the clustering technique could identify the clusters. One of these measurement techniques is the *rand index* that calculates the similarity of two lists of labels regardless of their permutations [44]. The rand index equals to:

$$RI = \frac{a+b}{\binom{n}{2}}, \tag{2.13}$$

where a is the number of pairs of samples assigned to the same cluster in both of the label's list of length n, and b denotes the number of pairs of samples in both lists in different clusters. The rand index for two lists of labels with similar groups equals one and zero for completely different clusters. To ensure that rand index for random labelling obtains zero, the *adjusted rand score* proposed [45]. The correction for the chance in the adjusted version of rand score achieved by comparison of the expected similarity of all possible pairs of samples which specified through a random model and calculated as follows:

$$ARI = \frac{RI - Expected(RI)}{max(RI) - Expected(RI)}. \tag{2.14}$$

The adjusted Rand index can obtain negative values if the RI is less than the expected RI. Note that ARI is symmetric, i.e., $ARI(A,B) = ARI(B,A)$. The ARI score can obtain values between $[-1,1]$. The ARI value for random labeling equals zero and for full matches equals one.

2.6 Linear Regression

Linear regression is a linear model that finds a relationship between a scalar variable (response variable) and one or more explanatory variables. In the case of the availability of one single explanatory variable, the model is called *simple linear regression* and in the case of multiple variables called *multiple linear regression*. The relationships in this technique are defined through a linear predictor functions. The extracted relationships are used to predict unseen response variables. There are many proposed approaches to define the relationship using multiple mathematical functions. The basic version of linear regression fits the model with weights (or coefficients) $W = (W_1, W_2, ..., W_p)$ to minimize the residual sum of squares between the seen responses, and then uses the identifies coefficients to generate a line. Then the generated line is utilized to predict response value of an incoming sample as follows:

$$\hat{y} = W_1 x_1 + W_2 x_2 + ... + W_p x_p, \tag{2.15}$$

where x_i is the explanatory variable, and $\hat{y}$ denotes the unseen target variable. The objective of the model is to find coefficients so that the residual sum of squares (RSS) minimizes. RSS calculated by:

$$\sum_{i=2}^{n} (y_i - \bar{y})^2, \tag{2.16}$$

where $\bar{y}$ is the mean value of the observed response values. In other words,

$$RSS(W) = \sum_{i=1}^{n} (y_i - W_0 - W_1 x_{i,1} - \cdots - W_p x_{i,p})^2, \tag{2.17}$$

where X is the matrix of explanatory variables [46].

2.7 Evaluation Metrics in Linear Models

The linear model's output is one or multiple continuous values, thus to measure the performance of the models, the confusion matrix-related metrics like false positive and true negative rates are not applicable. Instead, it is desired to measure the distances of predicted values to the real ones. Two widely-used metrics are Mean Absolute Error (MAE) and Mean Squared Error (MSE). Following, we discuss these metrics.

MAE: is a measure of distances between the observations and predicted values. MAE categorized in L_1 norm losses and calculated as follows:

$$MAE(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} |y_i - \hat{y}_i|, \tag{2.18}$$

where $\hat{y}$ denotes the predicted value and y is the observations [47].

MSE: corresponds to the quadratic distance of predicted and values and observations. MSE categorized in L_2 norm and calculated as follows:

$$MSE(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (y_i - \hat{y}_i)^2. \tag{2.19}$$

Comparison: the MSE because of a quadratic function in calculation error reflects the model failure by higher error values. The MAE, however, regardless of error magnitude, calculates simply the distance. One can consider MAE as a better evaluation metric for samples without outliers and the scenarios that the cost of distance to actual values is equally distributed. The MSE is a more appropriate candidate for the samples that include outliers or cases where the cost of errors increases exponentially.

3
Approach

This chapter discusses the framework of the thesis and illustrates our approach in merging a variety of machine learning models aiming to identify reoccurring jobs and predict runtimes. To this end, first, we describe the complete framework at a high level. Second, we explain the functionality of the components in our methodology. Then, we discuss how our methodology enables the estimation of incoming jobs' runtime.

3.1 Overview

As shown in figure 3.1, our approach is broken down into three major phases. In the first phase, we apply data integration and cleaning methods to hinder incorrect traces negatively affecting our methodology. Then, we transform the traces to DAGs objects. This step proceeds by harmonizing the DAGs so that a comparison metric could rightly measure the differences of the DAGs structures. We finalize the first phase by clustering the DAGs based on their similarities. Extreme caution must be taken to detect reoccurring jobs, as their representative attributes may arise from two irrelevant job productions despite their similarities in various aspects. In the second phase, the middle part of the figure, we perform a linear model to estimate the runtime of incoming jobs. To this end, we utilize the patterns inside the detected clusters. Finally, in the third phase, we evaluate our results using MAE and MSE. The workflow is not a one-way pipeline, and depending on the evaluation results, we optimize the previous configuration and hyperparameters of machine learning techniques to make the predictions as precise as possible.

3.2 Data Cleaning and Integration

The target trace dataset provides traces of the Alibaba cluster within a time interval of ca. 214 hours. Job dependencies are a mixture of multiple tasks that are dependent on each other. In the released trace dataset, each task can be identified with a corresponding identification number. The released dataset provides other information for the tasks, which we briefly describe as follows:

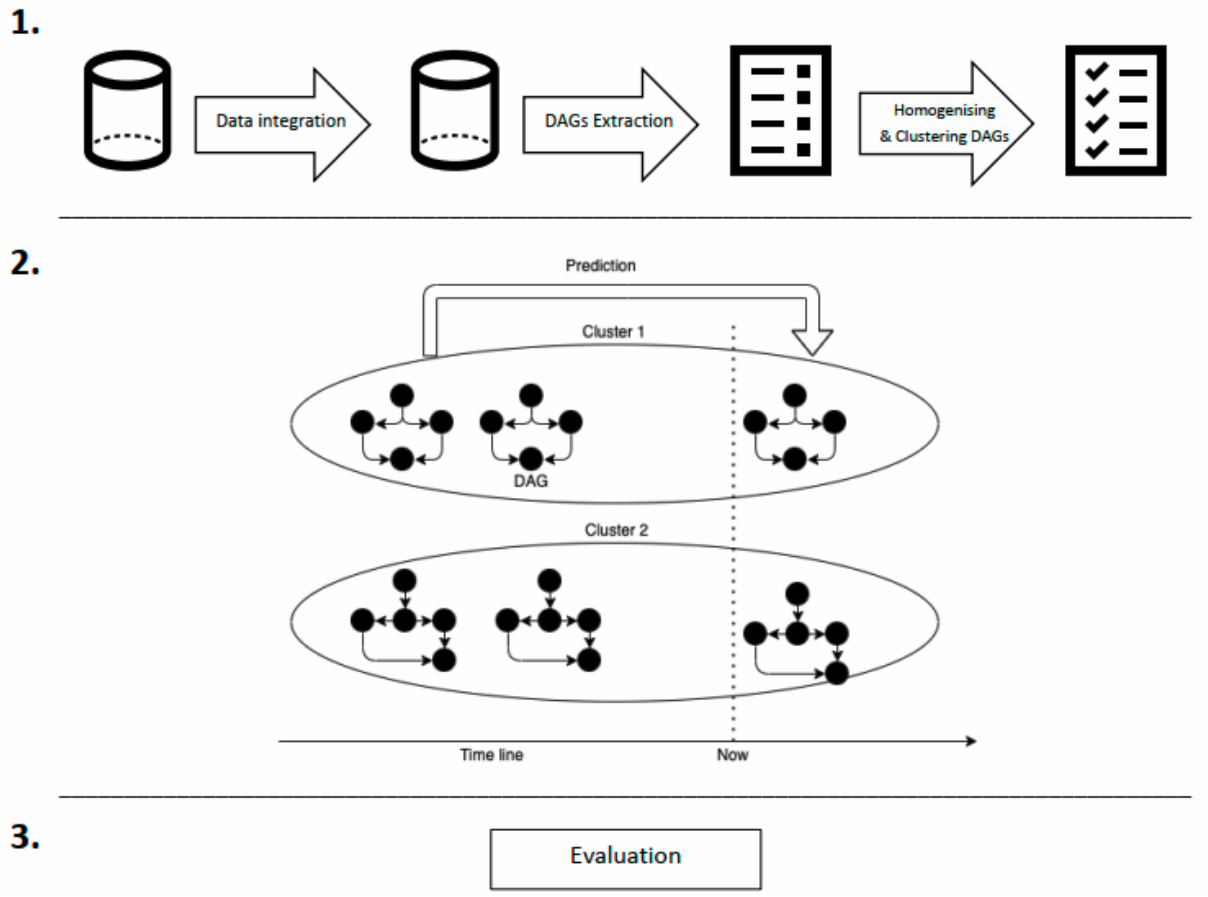


Figure 3.1: Workflow of investigation in three steps. The first step includes data preprocessing and clustering DAGs. In the second step, we predict the runtime of incoming jobs according to their assigned cluster. The ovals in the figure indicate a cluster of DAGs. Finally, in the third phase, we evaluate the results. Based on the evaluation results, we adjust the settings of the previous steps.

- ◇ *Task name:* consisting of one or multiple alphabets, which represent the name of the task and the way that it is dependent on other tasks.
- ◇ *Job name:* showing the job that each task corresponds to.
- ◇ *Start time:* starting time of each task.
- ◇ *End time:* ending time of each task.
- ◇ *Status:* status of a task, e.g., if it is terminated or waiting.
- ◇ *Planned CPU:* number of CPUs that should be assigned to each task. In this dataset, 100 means one core, 200 means two cores, etc.
- ◇ *Planned memory:* size of memory which is needed. The sizes are normalized to a range of [0,100].
- ◇ *Instance number:* the number of instances required for each task.

According to the meta-information of the released dataset, the required data for generating DAGs are stored in *Task name* feature. We utilize the traces and generate the DAGs based on the dataset descriptions, which are available in the dataset repository [13].

3.2.1 Data Cleaning

Although the beneficial usability of released traces due to their low entropy, we observe a degree of incompleteness and inconsistency in the dataset. To capture such the samples in traces, we operate data cleaning in two steps as follows:

1. The published traces contain samples that syntactically do not match the dataset description. The traces of this kind hinder the process of DAGs generation. For instance, the tasks that contain undefined attributes or the occurrence of not interpretable characters in task names. We eliminate these samples before the generation of DAGs object.

2. The other group of samples that should be eliminated are semantically not definable. More specifically, the chain of dependencies indicates a degree of contradiction, e.g., the job tasks that are reciprocally dependent on each other. To observe such contradictions, interpreting the dependencies is required. Thus, this step of data cleaning occurs after DAGs generation.

3.2.2 DAGs Extraction

The published dataset contains information on more than four million jobs. The jobs are composed of one or multiple tasks that are recognizable through their associated numbers. Indeed, the tasks within the jobs are, through unprecedented numbers, distinguishable. Every task has one or multiple dependencies on other tasks. Moreover, the dependencies are stored in the *task name* column of the data set.

In order to generate DAGs, first, we group the dataset by job names, which can be considered as a unique identifier of jobs. Then we gather all the information of dependencies and assign them as a concatenated mixture of strings to a table of job dependencies. Having the strings of tasks and based on the dataset description, we build the DAG objects. The details of the DAGs generation process are discussed in the implementation chapter. In addition to the structure of the DAGs, two important types of features are among the area of our interest, as follows:

1. The *node-level* features, i.e., provided information for the tasks such as the number of instances and memory usage. These types of features are stored within the DAG objects, and,

2. the *graph-level* features, i.e., the features that indicate the characteristics of the whole job. e.g., runtime, the total number of tasks, etc.

To make our contribution robust, we consider both the levels of features.

3.3 Graph Embedding and Clustering

The generated DAGs enable further analysis of structural dependencies. The current thesis aims to cluster reoccurring jobs based on their similarity in structural properties of DAGs. In other words, we aim to form a cluster of reoccurring jobs. To this end, the graph should be represented in a fixed-size data structure. This step is mandatory, as many clustering machine learning models do not accept the discrete data type of DAGs as input. The embeddings should hold three main conditions: (1) they have to represent the structural features of DAGs and the information regarding node- and graph-level attributes. (2) The representations must be definable in Euclidean space to make the comparison for clustering models possible, and (3) the representation learning phase should be categorized in the inductive models since the graph embedding phase in real-world scenarios should be repeatable to enable generating of comparable embeddings.

3.3.1 Graph Embedding

To generate embeddings holding the aforementioned conditions, we employ a Graph Neural Network (GNN). The GNN is a type of neural network optimized for processing graph data structures and categorized in supervised learning. The input of GNN is graph objects, and the output is fixed-size vectors representing the DAGs. The proposed GNN in our thesis is categorized in supervised machine learning techniques, which means that target variables accomplish the model training. The target variable should hold the essential features of graphs. In the following subsection, we define how we define the target variables. The GNN contains multiple convolutional layers and an output layer. The size of the output layer equals the dimensionality of embedding in Euclidean space. The GNN in every epoch learns how to separate the DAGs in Euclidean space regarding the associated target variable of DAGs.

3.3.1.1 Target Variable Definition

The GNN learns how to represent DAGs based on target variables; thus, we should sound a note of caution for selecting the target variables. The target variables should be a member of graph-level attributes. Every DAG holds numerous characteristics, e.g., number of nodes, average shortest path, etc. It is possible to extract countless features from DAGs by applying multiple aggregation functions and defining various structural-related metric. Some features could define the graph correctly, and the rest may lead to an unsuccessful training process of GNN by constraining it to learn irrelevant and distracting patterns.

Additionally, the training phase with such variables results in the overfitting of the model. Thus, we need a mechanism to select the most relevant informative features. To reach this goal, we firstly extract logically reasonable features of DAGs. This step extracts DAGs features based on our domain knowledge, regardless of their predictive ability. In the next phase, we utilize ensemble machine learning and statistical methods to filter irrelevant features—the details of this step discussed in 4.1.3.

3.3.1.2 Test and Train Data

After collecting DAG objects and the target variables, we can train the GNN model to produce graph embeddings. We split the DAGs collection into train and test set to ensure the model

performance for unseen samples. The model evolution succeeded through the train set alone. In addition, we gather information regarding the model performance on unseen samples by calculating losses in every epoch. To put it in another way, the calculation of losses for the test set indicates how generalizable our model is. We assign the size of the train set to 70% of the data.

Besides that, the job dependencies have a high variation in terms of runtime. Therefore, randomly splitting data into train and test set cause imbalanced set of data and negatively affect the process. In other words, the GNN learns to represent a proportion of data, which may not be generalizable. To avoid this phenomenon, we need to split the train and test set so that they have similar runtime variations. This type of splitting, called *continuous data stratification*.

Moreover, to robust the training speed, we get the assistance of a data-parallel training technique. Note that parallelization here refers to training the model with a multicore processing environment and is not related to parallelization concepts in the clusters. In order to attain the objective of parallelization, the training process accomplished by *batches* , which are fixed-size matrices. These matrices hold the diagonal adjacency matrices of single or multiple DAGs beside the node attributes and target variables. Logically, the size of the batches should be large enough to hold the data of multiple DAG objects, and at the same time small enough to be stored in memory. To satisfy both conditions, we select the batch size of 128 and hold the exact batch sizes for all training, validation, and embedding generation phases.

3.3.1.3 Whole Graph Embedding with GNN

After defining train and test sets, we are at the point to generate embeddings through the GNN model. From this phase on, it is necessary to review the model performance, as many hyperparameters and variables significantly impact the graph representation learning process. Following, we discuss some critical hyperparameters that need to be chosen with extra caution.

- ◇ *Loss function:* calculates the distance of the current performance of the model in generating informative embedding to the desired point.
- ◇ *Graph convolutional layers:* defines the mechanism of aggregating DAGs information.
- ◇ *Non-linearity function:* which adds non-linearity into the GNN and enables the process of learning.
- ◇ *Global pooling layer:* the strategy to aggregate the results of the final convolution layer.
- ◇ *Structure of neural network:* number of layers and their sizes.
- ◇ *Number of epochs:* that determines how many iterations should be executed to train the model and, at the same time, overfitting avoided.

We distinguish between the different combinations of parameters by observing the evolution process of the GNN and calculating the losses in different scenarios to reach maximal convergence.

Afterward, we utilized the trained model to generate the graph embeddings. The model in this phase should hold constant weights and biases. Hence, we disable the gradient descent process. Moreover, we disable the *dropout* mechanism, since all the neurons should be activated. Further details of the GNN model are discussed in section 4.2.

3.3.2 Clustering

After representation learning of DAGs, we can group the DAGs based on their associated representations or embeddings. It is worth mentioning that the embeddings contain no labels indicating which group they belong to; thus, this phase is categorized in unsupervised learning.

During the past decades, many techniques have been developed aiming to cluster points in the Euclidean space, and every clustering technique has its costs and benefits. To select a proper clustering technique, we need to consider the characteristics of generated DAGs embeddings, e.g., size of embeddings (dimensionality of the data), the range of values, etc. Furthermore, the following parameters should be considered: (1) An appropriate metric for calculating distances between points in highly multidimensional space. (2) Minimum required number of samples to consider as a cluster, and (3) the way of labeling points as outliers (the points that do not belong to any cluster).

By observing the quality of identified clusters, we verify the graph representation learning process and adjust its hyperparameters accordingly. To determine the quality of a cluster, we calculate the runtime variation within the detected clusters. The less variation indicates a better methodology of clustering technique and hyperparameter setting. Besides that, we consider the percentage of labeled samples as other measurements of clustering quality. The clustering techniques used in this work and their application are explained below.

3.3.2.1 Clustering with K-means

K-means is able to detect clusters accurately as long as two pre-assumptions are satisfied: (1) The density of clusters stays steady among all the clusters, and (2) the number of clusters (k) is known. With that being said, in our framework, the assumptions mentioned above are not satisfied because, in the collection of embeddings, there is no guarantee of constant densities among clusters. Additionally, we are not aware of the number of clusters of reoccurring jobs beforehand. There are techniques developed to determine the optimal k value, e.g., elbow method [48]. Nevertheless, identifying the optimal value using these methods is computationally highly expensive, as the dataset should be clustered multiple times with different k values through K-means. Consequently, because of the enormous number of embeddings, applying such techniques is not feasible. As a result, we utilize the K-means to observe the quality of embeddings in labor alone; more specifically, we compare them to the baseline with the same settings, i.e., the number of groups in baseline determines the k value. With k-value, K-means is able to assign all the samples to the clusters, which makes it a suitable candidate for evaluating the generated embeddings. Since K-means with the unknown value of k cannot solve real-world problems in runtime prediction, we do not attempt to predict the runtimes with this method and utilize it alone to measure the quality of embeddings in comparison with the baseline. Further details of K-means implementation fall into evaluation chapter, section 5.5.

3.3.2.2 Clustering with DBSCAN

DBSCAN can complete the learning process without having the number of clusters beforehand, but still suffers from the essential determination of two important hyperparameters, namely *radius*, and $MinPts$, where *radius* is a distance measure that determines the neighborhood of any points, and $MinPts$ indicates the minimum number of points to form a cluster. To set these parameters correctly, we operate a grid search methodology, meaning we firstly set

MinPts to a fixed random value and then check the quality of DBSCAN by assigning different values of *radius*. After detecting an optimal value for the first variable, we repeat the process by fixing the value of *radius* to find the optimal value of *MinPts*. To measure the quality of DBSCAN, we need to calculate the mean-variance of the runtime of clusters and the percentage of clustered samples. The proportion of clustered samples should also be considered, since the DBSCAN algorithm does not cluster the samples that cannot meet the conditions for a cluster, and consequently a number of samples remain unlabeled. Note that, to assure clustering task by DBSCAN, we standardize the embedding data beforehand. Further discussion regarding implementation of DBSCAN fall in section 4.3.2. We use the clusters identified by DBSCAN to generate the final predictions.

3.3.2.3 Clustering with OPTICS

OPTICS is another type of density-based clustering technique, and it has one primary advantage compared to DBSCAN, which is the unnecessity of the definition of *radius* beforehand. This advantage comes at the price of more complicated calculations and higher computational complexity. Like DBSCAN, OPTICS can detect clusters with different shapes and sizes besides the ability of outlier detection. Since OPTICS can cluster the unknown dataset (due to its ability to automatically detect the *Radius* value), we use it to measure the quality of the embeddings by training the model with multiple generated embeddings, i.e., embeddings of different sizes generated by different global pooling strategies. Before operating OPTICS, the data should be standardized. The section 4.3.3 discusses the details of implementation and methodology.

3.3.3 Runtime Prediction

Ultimately, we can predict the runtime of incoming jobs by utilizing the characteristics of cluster members. It is reasonable to consider other information rather than relying on the runtime of reoccurring jobs alone. Hence, we get the aid of other graph-level attributes to estimate runtimes—the prediction task in our thesis applied by the linear regression model for every single cluster. As the size of clusters varies, we define the train and test sets based on a simple rule: the last job in terms of beginning time is considered as the test sample, and the other members of clusters as the train set. The linear regression model determines the predictability of variables and accordingly assigns the coefficients. Then we evaluate the predicted runtimes using MAE and MSE. We utilize both metrics because the squared error can point out the errors with greater magnitudes. Figure 3.2 illustrates our framework to predict runtimes.

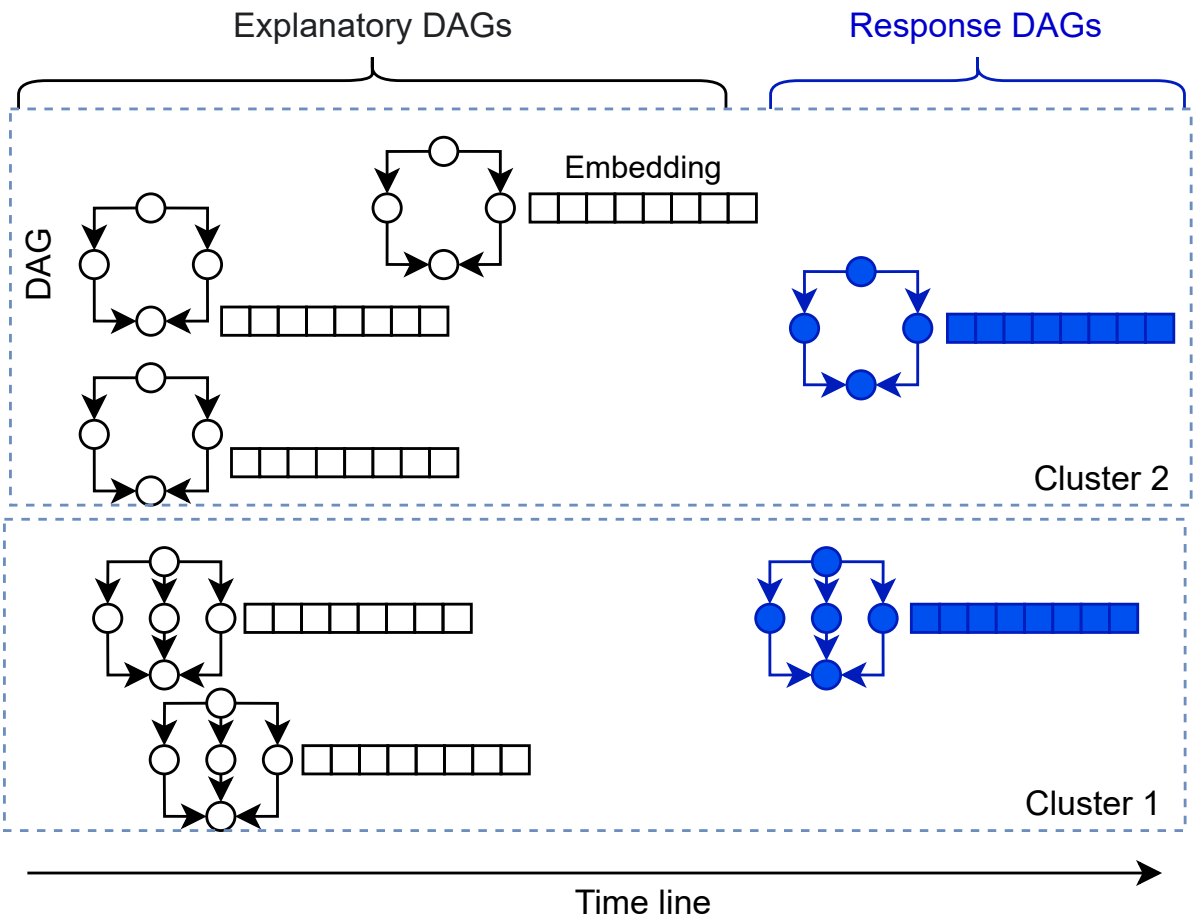


Figure 3.2: The illustration of our framework to predict runtimes for two toy examples of clusters. The blue DAGs represent the most recent job of each cluster and consider as an unseen job. Next, we train a linear regression model based on explanatory DAGs for every cluster to predict the runtime of response DAG. Note that, for the sake of simplicity, the structure of DAGs, as well as the size of embeddings are simplified.

4

Implementation

After formally describing the main components of our approach in the previous section, this chapter discusses the details of our approach and the implementation. This chapter is divided into four sections: DAGs extraction, graph embedding, data clustering techniques, and runtime prediction. In the DAGs extraction section, we explain our approach to creating DAGs objects along with node attributes. The graph embedding section describes the model that learns the graphs' representation and assigns a point to graph samples in multidimensional Euclidean space. The following section elaborates the clustering techniques that we use to identify reoccurring and similar jobs, and finally, the runtime prediction section defines our methodology to predict runtimes.

4.1 DAGs Extraction

There are multiple approaches to represent job dependencies. In our target dataset, the associated DAGs to job dependencies should be generated by concatenating the dependencies of different tasks. Furthermore, we apply some data integration and cleaning steps. After generating graphs, to create embeddings, we need to extract and evaluate target variables. The following section presents the implementation of the DAGs extraction phase in detail.

4.1.1 Data Preprocessing

Many datasets' publishers fail to standardize the data. The mistypes and lack of defining units and explanations are among the common issues of published datasets. Moreover, the errors in the data coupling phase will inevitably be an issue. The published trace dataset of Alibaba is not an exception and suffers from minor inconsistencies and incompleteness, which should be handled before generating DAG objects. The tasks in the dataset hold different statuses, namely *terminated*, *failed*, *running*, and *waiting*. The table 4.1 represents the percentage of jobs with different statuses. The table indicates that a considerable proportion of jobs are completely terminated. The runtime of jobs that hold other statuses rather than *terminated* is unclear, e.g., it is not determinable if a job with *failed* status will be eliminated or repeated. Thus, we skip

Table 4.1: Percentage of different statuses

Status	Terminated[1]	Failed[2]	Running[3]	Waiting[4]
Percentage	**97.076%**	1.980%	1.187%	0.003%

[1] The jobs with all *terminated* tasks.
[2] The jobs with at least one *failed* task.
[3] The jobs with at least one *running* task.
[4] The jobs with at least one *waiting* task.

the jobs containing unfinished jobs, which, according to the table 4.1, are not a considerable part of the whole dataset.

In addition, according to Robinson, for the randomly generated DAGs with fewer than six unlabeled nodes, the expectation of the occurrence of at least one pair of isomorphism graphs equals one [49]. In other words, the fewer number of individual tasks results in increasing of chance by the occurrence of accidentally isomorphism graphs [14]. We consider only jobs with at least ten individual tasks to avoid similarity by chance in job dependencies structure.

Furthermore, in the available dataset, some tasks are not interpretable using publisher descriptions. For example, the tasks that start with characters and not with tasks identifications. We also exclude the job dependencies that their associated DAGs are not inexplicable due to the following reasons:

- ◇ *Not Fully connected:* a subset of nodes is not connected to other nodes. This is not explainable because the whole pipeline of job parallelization should work together to provide the final result. The disconnected graphs indicate either the jobs consist of two separated jobs or the data is incomplete.

- ◇ *Cycle in the graphs:* the graph contains a cycle; thus, the associated graph does not form a DAG. The job dependencies must be explainable through DAGs because a cycle inside the graph contradicts parallelization concepts. For example, in an arbitrary DAG of G, if task A depends on the output of task B and task B is dependent on task A, the job output will not be finalized.

- ◇ *Multigraphs:* two nodes are connected with more than one edge. Consequently, they do not form a DAG. Parallel relations between tasks are not allowed in the structure of job dependencies.

- ◇ *Parallel description of tasks:* the graphs that contain nodes with two different attributes and dependencies. We omit this type of jobs, as every task must have a unique identifier and unique attributes according to the publisher description.

Besides the earlier mentioned categories of exceptions, we exclude the jobs with more than one-hour runtime because the mean runtime of jobs is ca. 264 seconds (less than five minutes) and runtime of less than 1% of jobs are more than one hour, but they still highly effect on results of prediction's tasks.

4.1.2 DAGs Objects

The combination of characters of tasks of a job carries the structure of its related DAGs. More specifically, the order of tasks number indicates the direction of edges, the "» between them represents the edges, and "," after a task number shows no further dependencies. For example, $M5\_2\_3$ means that the identifier of this specific task is 5 and depends on the results of tasks 2 and 3. In other words, task 5 can only be started when the output of task 2 and 3 are generated. Other attributes of tasks like *instance number* are also defined in the same dataset. To make graph objects, we delete nonnumerical values except "_" which shows the dependencies. Then we transform the characters into comprehensive graph objects which contain information regarding node features and the structure of DAGs. The algorithm 1 shows the procedure of generating DAGs. The first loop of the algorithm removes irrelevant characters from task names, and the second loop iterate over the jobs and collects information regarding the structure of DAGs and the value of attributes for the nodes. The DAGs are stored in a dictionary that the key indicates job number, and the corresponding value is the DAG object.

Algorithm 1: Generating objects of DAGs based on the data description

```
Result: DAG objects
DAGs_dictionary = {}
// Removing irrelevant characters from task_name coloumn
for i in batch.task_name do
    remove all characters except numbers and "_"
end
// DAGs generation
for j in batch.job_name do
    job_dependencies = concanecatation of tasks of job j
    // The DAG_creator generates graph objects assuming
       that the "_" represents an edge and that the
       characters around "_" are the nodes.  The order of
       characters define the direction
    DAG_object = job_dependencies.DAG_creator()
    // Gathering attributes of tasks and assigning them to
       feature of nodes
    DAG_object.plan_CPU = j.plan_CPU;
    DAG_object.plan_memory = j.plan_memories;
    DAG_object.instance_number = j.instance_numbers;
    DAGs_dictionary.update( j:DAG_object);
end
Return DAGs_dictionary
```

To elaborate the process, one can see the provided data for job number 34 in table 4.2. The second column of the table determines the structure of the DAG for job number 34. The rest of the columns representing the features of the nodes. In addition, the total runtime of the job

Table 4.2: Provided data for a toy example job

job_name	**task_name**	**plan_cpu**	**plan_mem**	**instance_num**	start_time	end_time
j_34	1	100	0.30	1.0	669954	669955
j_34	2_1	100	0.39	1.0	669954	669962
j_34	3	100	0.30	100.0	669954	669965
j_34	4	100	0.30	1.0	669954	669955
j_34	5_4	100	0.39	1.0	669954	669956
j_34	6	100	0.30	1.0	669954	669955
j_34	7_6	100	0.39	1.0	669954	669956
j_34	8_5_7	100	0.30	141.0	669954	669976
j_34	9_2_3_8	100	0.59	243.0	669954	669988
j_34	10_9	100	0.39	243.0	669954	670005

can be calculated by subtracting the minimum value of start_time from the maximum value of end_time. The figure 4.1 represents the generated graph after performing the algorithm 1 on the data.

We used two classes of graphs to reach different purposes. To create graph objects, we make use of two python libraries: NetworkX [50] and PyTorch-Geometric [51]. NetworkX library provides many functions to analyze the structure of graphs. We utilize PyTorch-Geometric to generate DAGs objects that are compatible with the GNN model.

4.1.3 Target Variables Definition

It is essential to identify target variables among the graph-level attributes to generate meaningful embeddings and enable the GNN to learn the representation of the DAGs correctly. The feature extraction consists of two steps. First, we extract 20 features that are informative enough to consider as candidates of target variables based on our substantiated assumptions and domain knowledge. After that, we get the assistance of a voting mechanism of different machine learning models and statistical techniques to select the most relevant features. The list of candidate features and a short description are presented in table 4.3.

After generating the features, it is vital to realize to what extent they are predictive. To do this, we used three multiple approaches, which aim to rank the features based on their importance. Following, we discuss the details of the implementation of our approaches.

Principal Component Analysis: Principal Component Analysis (PCA) enables dimensionality reduction based on linear algebra techniques. PCA transforms the nodes of a higher-dimensional space into a new space with lower desired dimensionality without losing much information. The procedure is done by converting the set of features, which may be correlated, into a set of linearly uncorrelated features [52]. Thus, PCA in its nature defines the importance of each dimension or feature. It is noteworthy to mention that it is necessary to standardize the data before applying PCA. To do that, we use the Standardize features by removing the mean and scaling to unit variance. More formally, standardized vector of x calculated by $z = \frac{x-u}{s}$, where u is the mean and s is the standard deviation.

We select the number of components (the desired dimension of new Euclidean space) as one, and we train the PCA with the standardized data. We operate PCA on our data using

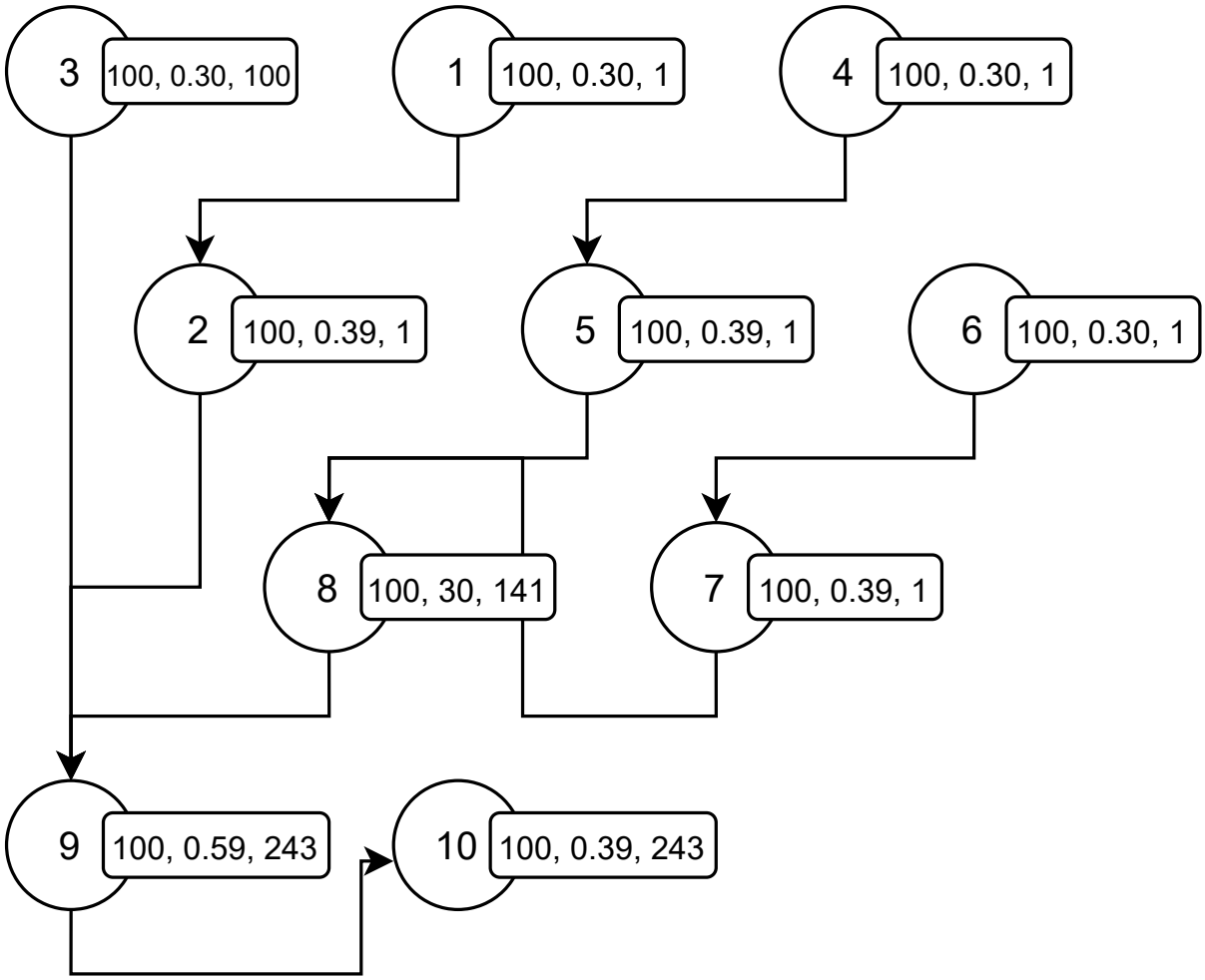


Figure 4.1: The created DAG object is based on provided data in table 4.2. Note that, in attributes vector, the first element is plan_cpu, the second element stands for plan_memory and the third element is instance_num.

Scikit-learn libraries with default values [53]. The output is the weight of each feature in a one-dimensional space. We consider the magnitude of weights as feature importance.

Extra-trees Classifier: is an ensemble learning method for classification, regression, and other problems based on multiple decision trees. Extra-trees tries to minimize overfitting by randomizing decisions and data subsets. In this method, every decision tree is trained with all available data, but the splitting nodes in decision trees are randomized, helping to avoid overfitting. These advantages of Extra-trees can fit well in our case, as all the decision trees consider all the data and none of the data subsets could be randomly ignored, and the randomized nature of selecting splitting points yield to considering all features within their different intervals. Note that, unlike the PCA, this method is categorized in supervised learning methods. Here, we use Extra-trees to predict runtime, so the runtime is excluded from the search space. We operate Extra-trees on our data using Scikit-learn preprocessing libraries, setting the number of the estimator to 10 and the maximum decision trees depth by 15. After the training phase, the learned model provides the feature importance based on their impurity.

Table 4.3: The extracted graph attributes and their descriptions

Feature	Description
Runtime	The execution time of job
Num-edges	The number of edges within the DAG
Num-nodes	The Number of nodes within the DAG
AVG-shortest-path	The average of distances to reach other nodes
Longest-path	The length of the longest path within DAG
Num-cliques	The number of cliques within the graph
SUM-center	The Summation of the degree of centrality of all nodes
MAX-center	The centrality degree of most connected node
MAX-center-instance-num	The instance number of the most connected node
MAX-center-plan-CPU	The plan CPU of the node with the highest centrality
MAX-center-plan-memory	The plan memory of the most connected node
AVG-node-degree	The average of node degree of all nodes
AVG-instance-num	The average of instance number of all tasks
AVG-plan-CPU	The average of plan CPU of all tasks
AVG-plan-memory	The average of instance number of plan memory
SUM-instance-num	The summation of instance number of all tasks
SUM-plan-CPU	The summation of plan CPU of all tasks
SUM-plan-memory	The summation of plan memory of all tasks
MAX-instance-num	The average of instance number of all tasks
MAX-plan-CPU	The average of plan CPU of all tasks
MAX-plan-memory	The maximum of plan memory of all tasks

Recursive Feature Elimination: the Recursive Feature Elimination (RFE) tries to recursively remove features and builds an arbitrary model with the remaining ones. The accuracy of the model with different combinations of features indicates which variables are more relevant to predict the target variable [54]. This method is a supervised procedure. Hence, we define *runtime* as the target variable and exclude it from search space. Moreover, we specify *linear regression* as the predictor model, which is a linear approach to determine the relationship between a scalar value (in our case, runtime) and other features [55]. To operate this technique, we utilize the RFE library from Scikit [53] with default parameters and a linear regression model as the predictor. RFE returns the ranking of features based on their predictability

Table 4.4 represents the results of the models as mentioned earlier. To choose the final target variables, we select the best five features of each feature selection technique—the highlighted rows are the selected target variables.

4.2 Graph Embedding

After identifying the target variables, the current section discusses our approach to generate embeddings that hold the key information of DAGs. In other words, the graphs with similar structures and similar node attributes should have similar representations. This section describes the model that represents the graph in Euclidean space in detail. The input of the model

Table 4.4: The results of feature selection techniques. The best five features of each technique are selected as the target variable

Feature	PCA	RFE	Extra-trees
Runtime	**0.0209**	-	-
Num-edges	0.0001	12	0.0426
Num-nodes	0.0001	11	0.0433
AVG-shortest-path	0.0	**5**	**0.0719**
Longest-path	0.0	13	0.0587
Num-cliques	0.0001	14	0.0383
SUM-center	0.0	**2**	0.0373
MAX-center	0.0	**4**	0.0616
MAX-center-instance-num	0.0004	15	0.0153
MAX-center-plan-CPU	0.0	6	0.0392
MAX-center-plan-memory	**0.0663**	20	0.0517
AVG-node-degree	0.0	**1**	0.0460
AVG-instance-num	**0.0517**	17	**0.0952**
AVG-plan-CPU	0.0003	9	0.0081
AVG-plan-memory	0.0	7	**0.0660**
SUM-instance-num	**0.8245**	18	**0.0962**
SUM-plan-CPU	0.0156	16	0.0345
SUM-plan-memory	0.0001	8	0.0626
MAX-instance-num	**0.559**	19	**0.0790**
MAX-plan-CPU	0.0004	10	0.0093
MAX-plan-memory	0.0	**3**	0.0425

is the collection of DAGs, and the output is their corresponding embedding. In general, it is possible to transfer the graph in Euclidean space with arbitrary dimensions. The size of embeddings determines the dimensionality of the Euclidean space. Following, we discuss the details of implementation.

4.2.1 Model Structure

Assuming the desired embedding size is n, our proposed GNN consists of one initial convolutional layer of size three by n, three convolutional layers of size n by n, and a final global pooling layer which outputs the embeddings. We add nonlinearity to the GNN using Exponential Linear Unit (ELU) [32]. Figure 4.2 shows the overview of our proposed GNN. The graph shows three main components of GNN, namely input or DAGs' collection, the combination of convolutional layers, and finalizing the embedding using transformation functions. We train the GNN using batch-wise fitting, making it possible to split the data and fit each split in the cache, resulting in faster training. In batch-wise fitting, the model is trained via multiple instances of input data at once; consequently, in reality, each layer has another dimension that equals the size of batches. Here, we set the size of the batch to 128, i.e., data of 128 objects are processed at once. For ease of simplicity, the figure follows the node-wise operation. The second

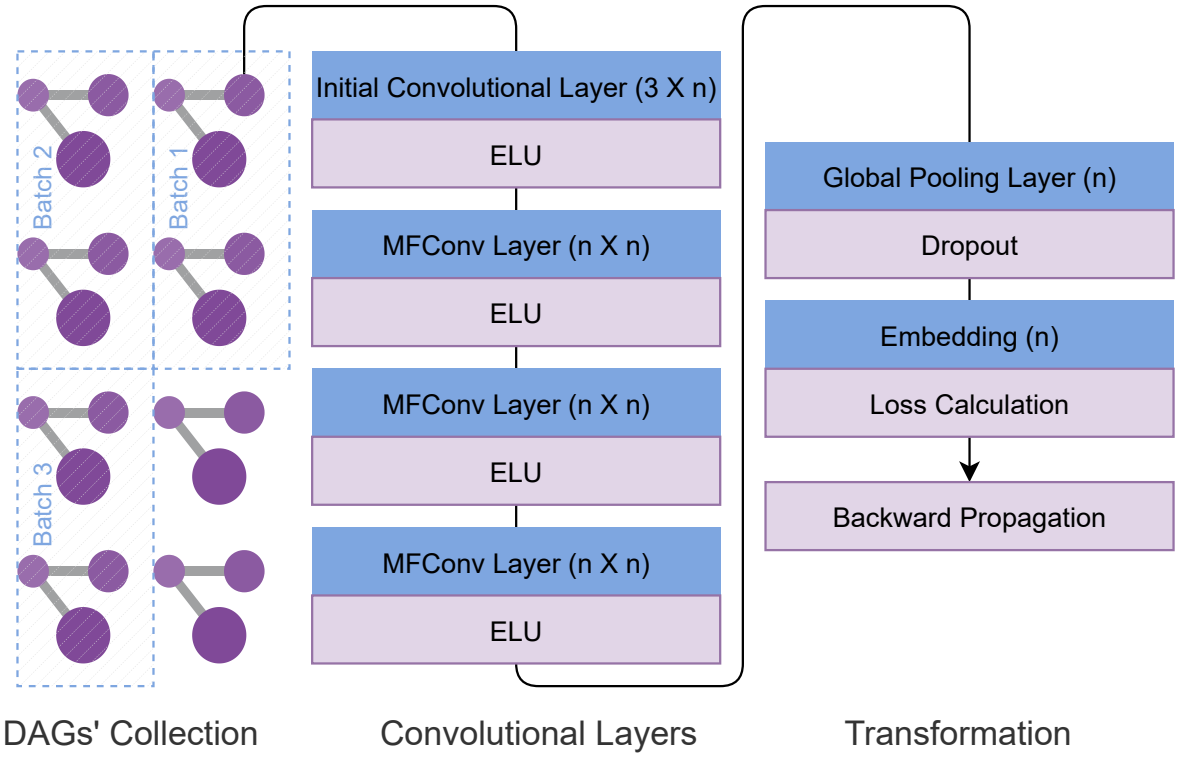


Figure 4.2: The GNN's structure for graphs embedding. Note that the n represents the size of the embedding.

component of GNN applies the convolutional operations, and finally, the nodes' data is aggregated in the last component of the GNN. In the training phase of GNN, the distance of GNN to true states is calculated, and the corresponding weights and biases are adapted in the backward propagation phase. In addition, we avoid the overfitting of GNN by applying the dropout and L_2 regularization technique on the final layer. We utilize the PyTorch-Geometric library to generate the GNN model [51]. PyTorch-Geometric provides multiple classes of convolutional layers, backpropagation methodology, global pooling layer, etc. Following, we overview the components of the GNN in more detail.

4.2.1.1 Convolutional Layers

The convolutional layers are the main components of the GNN model. There exist many proposed techniques aiming to capture graph information as precise as possible. Neural networks typically expect a fixed input size, but the size and shapes of graphs vary within our dataset. Therefore, we need a mechanism to handle the different shapes of graphs. The idea of the convolutional layer is similar to techniques of the convolutional layer in image processing, i.e., breaking the input data into multiple segmentation using a fixed-size window. The key difference in the graph processing domain is the arbitrary structure of input data. Indeed, the adjacency matrix of graphs should be permutation invariant, which is not valid for the graph as the adjacency matrix is sensitive to nodes order. Thus, the adjacency matrix can not directly be used as input.

The fundamental idea of the GNN is the aggregation of structural information in addition to the node features. The nodes' embedding carry information regarding the structure of neigh-

borhood nodes and their attributes besides their own attributes in a compressed format. To maintain such a compressed vector, we gather the information of the neighbor nodes and combine it to get new embeddings, and iteratively repeat the process to enable the GNN to capture the structure of the whole graph. This procedure is done simultaneously for all nodes and is the main idea behind the message passing mechanism, as the states can be seen as the messages that passing between the nodes. The first layer gathers information of direct neighbors of every node, and in the second layer, the knowledge of neighbors' neighbors is collected, and so on. Indeed, the number of layers indicates the distance that the structural information should be gathered. As the DAGs of job dependencies are usually small graphs, the GNN can learn the representation with a small number of message-passing layers. Message passing is formally defined as:

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^k, \forall \upsilon \in \mathscr{N}(u)\})), \tag{4.1}$$

where h is the node representation, u represents the node, and k indicates the time step. The $AGGREGATE$ operation uses all neighbors nodes of u and aggregates them. Multiple types of aggregation functions are applicable here. The $UPDATE$ operation uses the current state in time step k and aggregates it with the neighbor's state. The aforementioned operation is the general structure of message passing layers [56]. There are multiple strategies to define $UPDATE$ and $AGGREGATE$ functions. The mathematical operation 4.1 can be expanded as:

$$h_u^{(k+1)} = U_{(k+1)}(h_u^{(k)}, \sum_{v \in \mathscr{N}(u)} M_{(k+1)}(h_u^k, h_v^k, X_{uv}^e)), \tag{4.2}$$

where $U_k(.)$ denotes the update function with learnable parameters, $M_k(.)$ is the message passing function, and v is the neighbor node of u. The fixed vector of h should be extracted for every node [24]. The equation 4.2 consists of the update and aggregate function. Indeed, the information of node (h_u^k) along with neighbors information (h_v^k) and the features (X_e^{uv}) are gathered together and generate the message information.

The present thesis operates the similar message passing layer, which is mainly proposed for handling learning molecular fingerprints (MFConv) [27]. The MFConv is optimized for learning the representation of small graphs and generating embeddings for prediction tasks. The formal definition of MFConv is as follows:

$$h'_u = W_1^{(\deg(u))} h_u + W_2^{(\deg(u))} \sum_{v \in \mathscr{N}(u)} h_v, \tag{4.3}$$

where W indicates the learnable parameters and assign a distinct W for each possible node degree. To put it another way, MFConv calculates substructure feature vectors via message passing and sums them to generate overall representations.

Graph representation learning, using GNN, is the state-of-the-art approach but still suffers from some issues. For example, the disproportionate number of message passing layers significantly worse the performance for GNNs. Li et al. show that the convolutional operation in GNN is a type of Laplacian smoothing [57]. Thus, after iteratively performing Laplacian smoothing, the feature of nodes will converge to a value that carries no further information. This phenomenon is called over-smoothing and can be avoided by selecting the proper number of hidden layers in GNN. To hinder the over-smoothing effect, we select three message passing layers [58].

4.2.1.2 Global Pooling Layer

As we aim to learn the representation of the whole graph, we need to define an aggregation method to combine the learned node-level embeddings. This procedure will be applied by utilizing a global pooling layer that iteratively compresses the node embeddings into a fixed-size representation. For instance, a global mean pooling layer, which outputs the graph-level embedding by averaging node features across the node dimension over a batch of $\mathbf{b} \in \{0,\dots,B-1\}^N$ equals to:

$$\mathbf{r}_i = \frac{1}{N_i} \sum_{n=1}^{N_i} \mathbf{x}_n, \tag{4.4}$$

where N denotes the number of dimensions and $\mathbf{X} \in \mathbb{R}^{(N_1+\dots+N_B)\times F}$ is the feature matrix. We operate a variety of aggregation techniques in global pooling layers, namely, ADD, MEAN, and MAX. Then, we measure the quality of embeddings based on the evaluation results of OPTICS clustering.

4.2.1.3 Optimization Methods

The proposed GNN should not get overly complex because it results in overfitting. To avoid overfitting in the GNN, we perform two optimization techniques. These techniques operate mathematical operations that hinder the GNN from learning generalizable and global patterns. We optimize our proposed GNN by using the following optimization techniques.

Dropout: the dropout function disables some elements of the hidden layer. The idea behind the dropout function is to force the learning process to find the general patterns and not be influenced by some specific values. The elements are disabled randomly by chance of p, using samples from a Bernoulli distribution. The p is independent of previous forward passes. Additionally, the embeddings are scaled by a factor of $\frac{1}{1-p}$ during the training phase, i.e., during model validation, the dropout becomes an identity function [33]. We operate dropout technique by setting $p = 0.5$.

L_2 regularization: the L_2 regularization hinders the overfitting by assigning small weights to the GNN parameters. To perform L_2-regularization, we utilize *weight decay* that apply a L_2 penalty to the cost. Weight decay effectively results in smaller model weights by applying the following operation for the calculation of gradients:

$$G^{(i)} = \frac{dL}{dW^i} + \lambda W^{(i)}, \tag{4.5}$$

where $\frac{dL}{dW^i}$ is the gradient, and λ denotes the weight decay for the GNN weights $W^{(i)}$. We set the weight decay to 0.0001.

4.2.2 Model Training

As mentioned before, we train and validate the GNN with stratified data. We exclude the already mentioned outliers before splitting the data. Then, to employ parallelization benefits and to the roust training phase of GNN, we transform the train data into batches of size 128. We

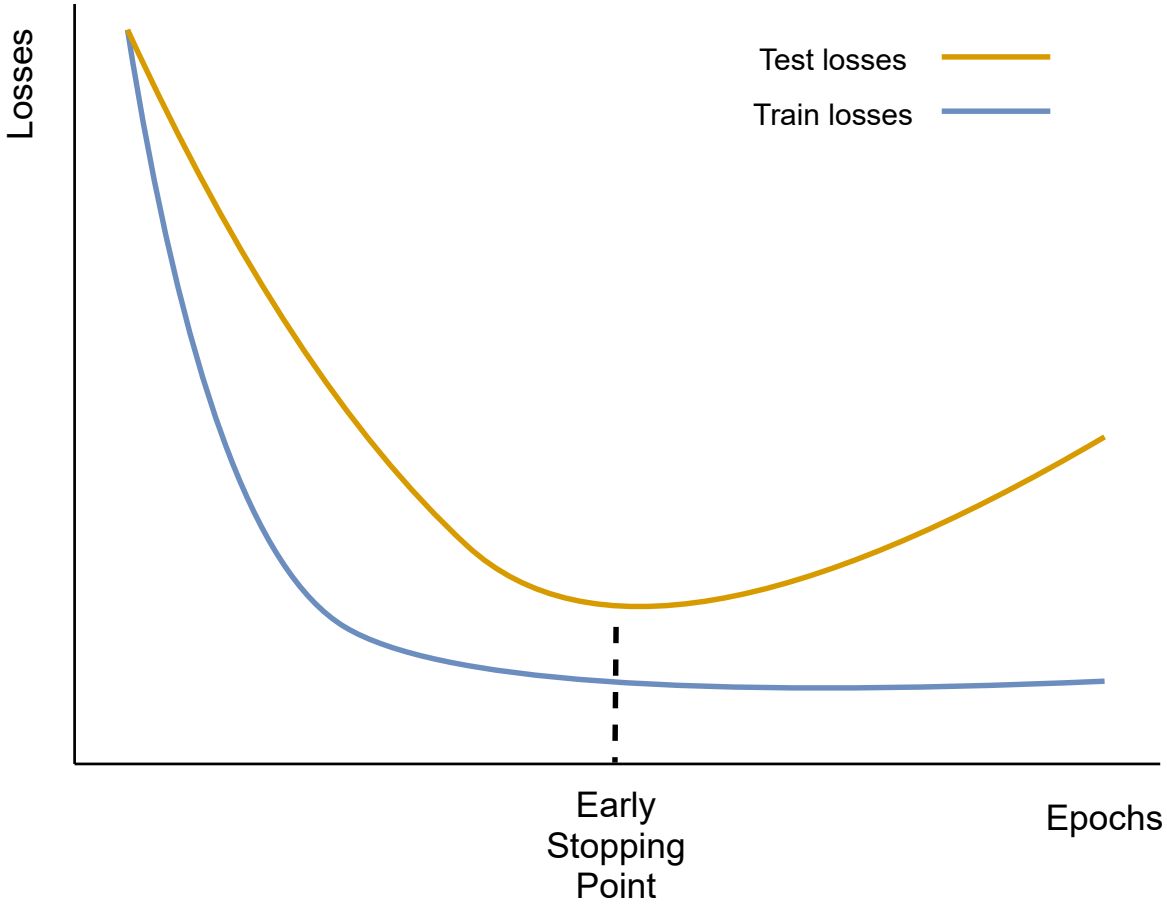


Figure 4.3: Illustration of early stopping approach.In early stopping, the model training procedure will be stopped when the losses of unseen samples start increasing.

operate the technique proposed by [59], which is an extension to stochastic gradient descent to update network weights iteratively, and we set the learning rate to 0.001. To train the model, we mined the embeddings and calculated the loss function using triplet margin loss. We disable the gradient descent procedure and optimization techniques to evaluate the model performance via the test set. Following, we discuss the mining procedure as well as calculating losses.

4.2.2.1 Early Stopping

The GNN models could be overfitted by evolution to lonely define the dataset patterns, not global ones. To do this, we observe the losses not only for train data but also for test data, as the increasing value of losses on unseen samples is a suitable indicator of model overfitting. Afterward, we define a threshold that stops the learning procedure at an appropriate point, where the GNN stops learning generalizable patterns. This strategy is called early stopping. The figure 4.3 illustrates the idea behind the early stopping approach. To apply this strategy, we define a flag that indicates if the losses are increasing. Three positive flags in a row force the model to stop [60].

4.2.2.2 Triplet Loss

The intuition behind triplet loss is that elements belonging to the same category or, in our case, jobs with similar target values possess similar characteristics. Triplet loss reduces the distance between similar objects and increases the distance between dissimilar ones in target values. To achieve this goal, triplet loss considers a triplet containing an anchor, a positive sample, and a negative sample. The triplet loss function, defined as:

$$L_0 = \sum_{1}^{N} [d_0^{ap} - d_0^{an} + m_0], \tag{4.6}$$

where N denotes the number of samples (equals to batch size), m is the margin that should exist between positive and negative sample, d^{ap} and d^{an} are the Euclidean distance between positive and negative samples to the anchor and calculated as:

$$d_0^{ap} = \|f_0(x_i^a) - f_0(x_i^p)\|_2^2, \tag{4.7}$$

$$d_0^{an} = \|f_0(x_i^a) - f_0(x_i^p)\|_2^2, \tag{4.8}$$

where $f_0(x_i)$ is the feature vector of x_i with dimension of D. The mechanism of triplet loss can answer our requirements, as, within generated embeddings, similar jobs are close to each other in Euclidean space and vice versa. To apply triplet loss, we use the published library of PyTorch metric learning [61]. We calculate the triplet loss for all the target values. The total loss of the GNN is calculated by the summation of all target variable losses.

4.2.2.3 Miner

It is reasonable to apply a mining procedure to find the most relevant baseline (Anchor), positive, and negative samples before calculating triplet losses. The miner identifies valid triplets for each batch of input. i.e., for a given batch of B after generation of embeddings, we identify the embeddings with the highest anchors values. The miners categorize into two domains, namely, offline and online. The latter extracts the appropriate triplets with already sampled vectors, and the former begins the mining process before batches generation. We used the *MultiSimilarityMiner* from PyTorch Metric learning [62], which is an online miner and finds the appropriate triplet within every batch and submits the indices of them to the loss function. More specifically, to make the loss function robust and allow efficient computation of losses, the *MultiSimilarityMiner* returns the triplets by finding and merging positive and negative pairs with similar anchors.

4.2.2.4 GNN Evaluation

The generated embeddings are not directly inferable. Thus, we evaluate and adjust the results with two main approaches. First, we directly observe the training phase of the GNN, i.e., monitor how well the GNN is able to distinguish the jobs with different target values and if the convergence happened in the training phase. Second, we take advantage of the utterly unsupervised method of OPTICS clustering to check the ability of identified clusters in runtime prediction. In other words, we consider all the embeddings that could satisfy the first evaluation phase in our methodology. Then we adjust the structure of GNN based on the results of OPTICS clustering, which determines the quality of embeddings in terms of predictability.

4.3 Data Clustering Techniques

To this end, we achieve the representation of graphs in a fixed format. Interpretation of the generated embeddings through the GNN is not straightforward. Because in the convolutional layer, the graph information breaks down into different compositions, and after the aggregation step the origin can not be specified. This leads us to select different clustering approaches, namely OPTICS [39], and DBSCAN [38]. The OPTICS used to measure the quality of embeddings themselves without setting requirements of *radius* in cost of longer runtime. Thus, utilizing OPTICS, we choose the partial optimal global pooling strategy and vector size of embeddings. Then we apply the DBSCAN to form the final clusters. We define our contribution to select hyperparameters of DBSCAN in the following sections. Many clustering methods, including the clustering approaches in this thesis, require preprocessing, i.e., data normalization and standardization. We first perform these preprocessing steps on embedding data. Afterward, we train the clustering models with the data. The details of the clustering phase are discussed in the following sections.

In addition, the trace dataset publisher does not provide any information regarding the recurrent job dependencies, meaning that no ground truth is available. Hence, we specify the runtime of job dependencies as an indicator of the quality of the clustering step.

4.3.1 Preprocessing

To cluster the data, we consider the Euclidean distance of embeddings. Data normalization is required to eliminate the redundancy of the data and ensure the quality of the clustering process, as the Euclidean distance can highly be affected by changes in differences [63]. We force the embeddings to be normally distributed by centering and scaling the embeddings' data. For this, we apply the following operation on all embeddings independently:

$$z_i = (\frac{h_i - m_i}{s_i}), \tag{4.9}$$

where h_i is the value of feature i, m_i is the mean, and s is the standard deviation. Afterward, as in the clustering task, the measure of similarity of samples matters. We normalize the embeddings' data using the least squares or $l2$ normalization method, i.e., we modify the values so that in each row, the sum of the squares will always be up to 1. We apply these transformations to standardize the data both row-wise and column-wise. The second transformation does not remove the mean and scale by deviation, but scales the whole row to unit norm. We apply the two transformations using Scikit library [64].

4.3.2 DBSCAN

To cluster data, DBSCAN considers the densities of data points, meaning the area with high density is detected as a cluster, and the border of the clusters are areas with low density. This enables DBSCAN to detect clusters with arbitrary shapes, unlike K-means, which is only able to detect clusters with convex shapes. In the DBSCAN method, The concept of *core samples* has considerable importance. Core samples are the data points within a high-density area. The combination of core samples forms a cluster in which the data point are close to each other.

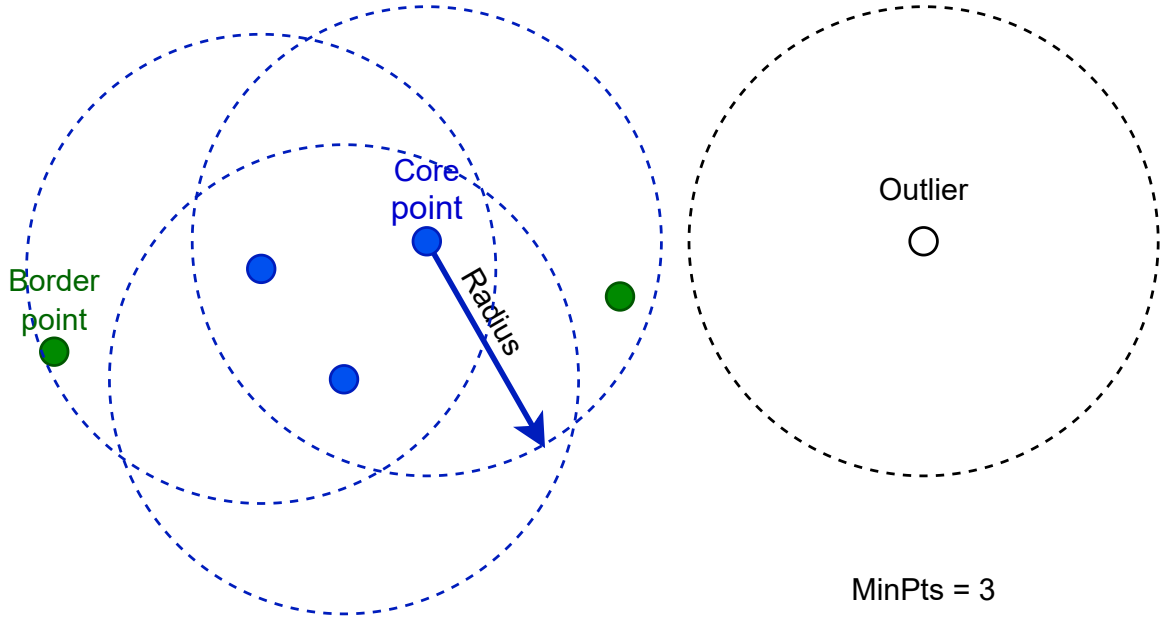


Figure 4.4: An example of clustering with DBSCAN. In the figure, the blue points consider as core points, as there exist at least three *MinPts* in the distance of *radius* (including the points themselves). Here, all the blue and green points consider as one cluster because they are reachable from one another. The green points do not have at least 3 points in *radius* distance, but they are in *radius* distance of core points. As a result, they identified as border points of the cluster. The white point is an example of an outlier, as there are no further points in the *radius* distance.

To measure the distance between data points, we used Euclidean distance, which is defined as below:

$$d(E_1, E_2) = \sqrt{\sum_{i=1}^{n} (E_1 - E_2)^2}, \tag{4.10}$$

where E_1 and E_2 are two embeddings in Euclidean n-space. The n is the dimensionality of the space vector size of the embeddings. DBSCAN also considers non-core data points, which are the points close to core points. As mentioned before, two parameters greatly impact the efficiency of DBSCAN, namely, *radius* and *MinPts*. The combination of these parameters formally defines the density. The higher values of *MinPts* and *radius* cause the DBSCAN to consider areas with higher density as clusters. In other words, a data point is labeled as a core sample when at least *MinPts* data points within the distance of *radius* exist. Therefore, the algorithm defines the clusters as a combination of core samples formed by recursively detecting core samples and their neighbor points. In this definition, every cluster consists of core and non-core data points which are located close to the border of cluster areas. Besides that, the value of *MinPts* shows the degree of tolerance toward noises. Additionally, selecting a small value for *radius* forces DBSCAN to consider most of the data as noises. On the other hand, a too large value of *radius* causes a cluster with many points as the clusters merge together. Note that the DBSCAN is a deterministic algorithm, meaning having the same parameters and the same data, the algorithm constantly forms identical clusters. Changing the order of the data changes the order of the cluster's labels and non-core data points. Because a non-core sample is assigned to the first identified cluster, and by changing the order of the extracted clusters, a non-core point can be assigned to a neighbor's cluster [38, 65].

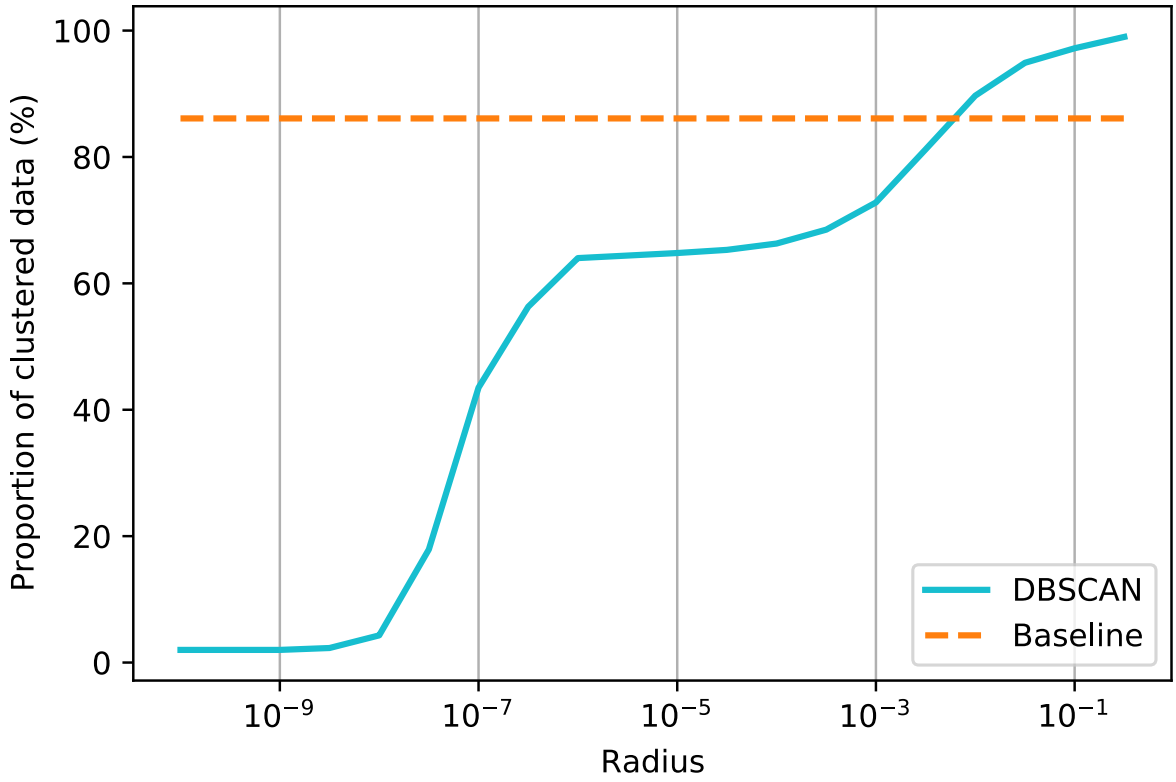


Figure 4.5: Percentage of clustered data using DBSCAN with different settings of *radius*.

To avoid generating a full distance matrix of all embeddings, we used a K-dimensional tree (KD-tree). The KD-tree is a member of binary trees that recursively partitions the dimensionality of space and defines nested orthographic spaces for data samples. Generating Kd-tree is significantly fast as the partitioning is executed singly using the dimensionality of data samples. Hence, it is not necessary to compute distances in the space with full dimensionality. After construction of KD-tree the complexity of computing distances becomes $O(\log(n))$ [66].

To implement the algorithm, we used the library of DBSCAN from Scikit library [53]. Afterward, we discuss how we define the parameters of DBSCAN. To measure the efficiency and to have a deeper insight into the performance of the clustering phase, we also compare our results with classification using the rule-based methodology. We inspired this approach from work by Tian et al. that assumes the recurrent jobs have isomorphic DAGs and are happening within certain repetitive time intervals [14]. The baseline of our thesis detects recurring jobs using a rule-based classification. Thus, all the members of a group of recurring jobs have an isomorphic structure. The details of the generation of groups are discussed in section 5.2.

4.3.2.1 Radius

To determine an appropriate value for *radius*, we first set the *MinPts* value to the fixed value of 3. Then we generate clusters using the fixed-value of *MinPts* and *radius* of $10^{\frac{i}{2}}$, where i is the integer numbers in the range of $[-20,-1]$ in ascending order. Particularly, we search for an appropriate value of *radius* by generating clusters using 19 different values in a loop. In every iteration, two critical metrics collected as follows:

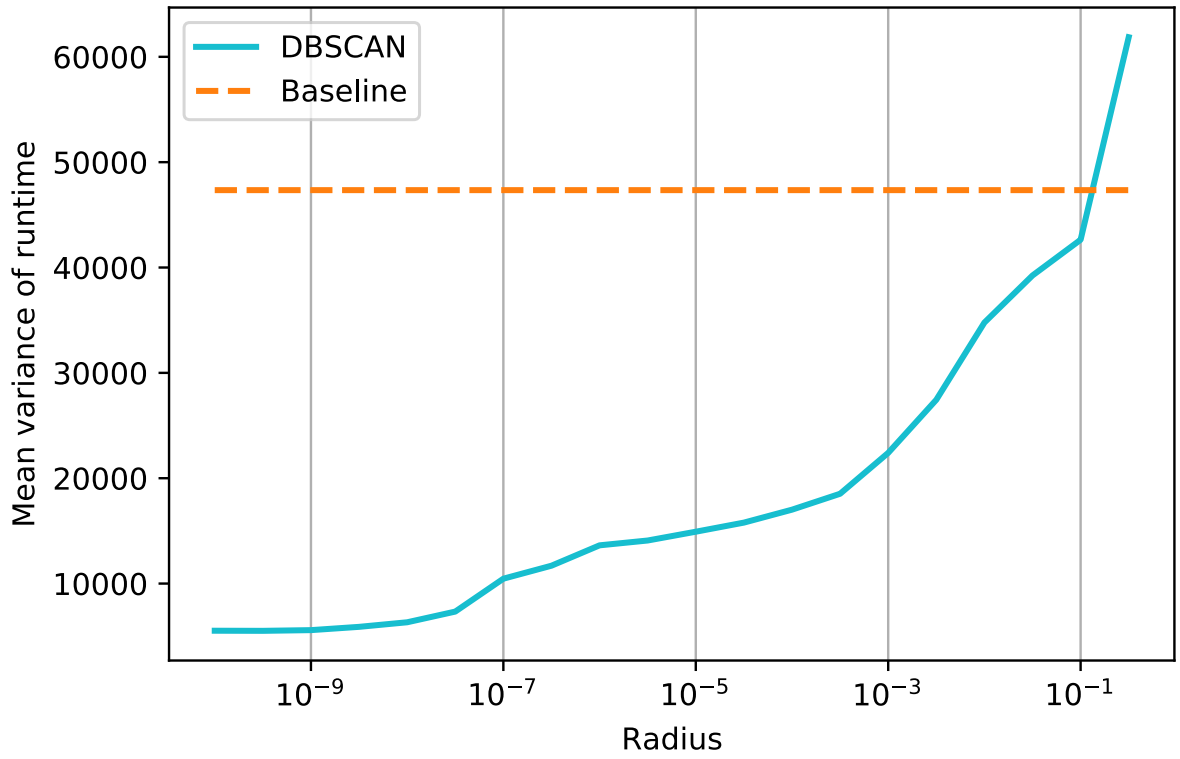


Figure 4.6: Mean runtime variation of jobs withing the detected clusters generated through different settings of *radius*

- ◇ *Runtime-variation:* the variation of runtime of jobs within every cluster.
- ◇ *Percentage-clustered:* the proportion of clustered data.

Indeed, we consider the combination of these metrics as the indicator of the quality of clusters. The figure 4.5 illustrates the proportion of clustered data for different values of *radius*. As mentioned before, by assigning a small value to *radius*, DBSCAN considers many points as outliers because they can not satisfy core and non-core samples conditions. As the *radius* value increases, more and more jobs are assigned to clusters. Setting the *radius* big enough, the DBSCAN assigns all the points to a single cluster, and none of the samples is considered as outlier. The figure also presents the promotion of clustered data using baseline contribution.

Apart from the number of clustered data, it is reasonable to check the performance of DBSCAN in detecting similar job. Figure 4.6 presents the mean of runtime variation for different settings of *radius*. As shown in the figure, the quality of clusters decreases by increasing the value of *radius*. Since DBSCAN by the smaller value of *radius* is stricter in the way of forming clusters. By minimal values of *radius*, the DBSCAN considers only highly similar embeddings as clusters. In consequence, as we expected, the jobs within a cluster have very low variation in terms of runtime. To this end, we illustrate two aspects of DBSCAN, namely quantity and quality of clusters. As we expected, to select an appropriate value of *radius*, it is necessary to trade-off between these two aspects, in other terms, it is desired to cluster as many jobs as possible, and at the same time, hold the runtime variance within the clusters as low as possible. In our approach, there is no guarantee that the detected value is the global optimal *radius*, but

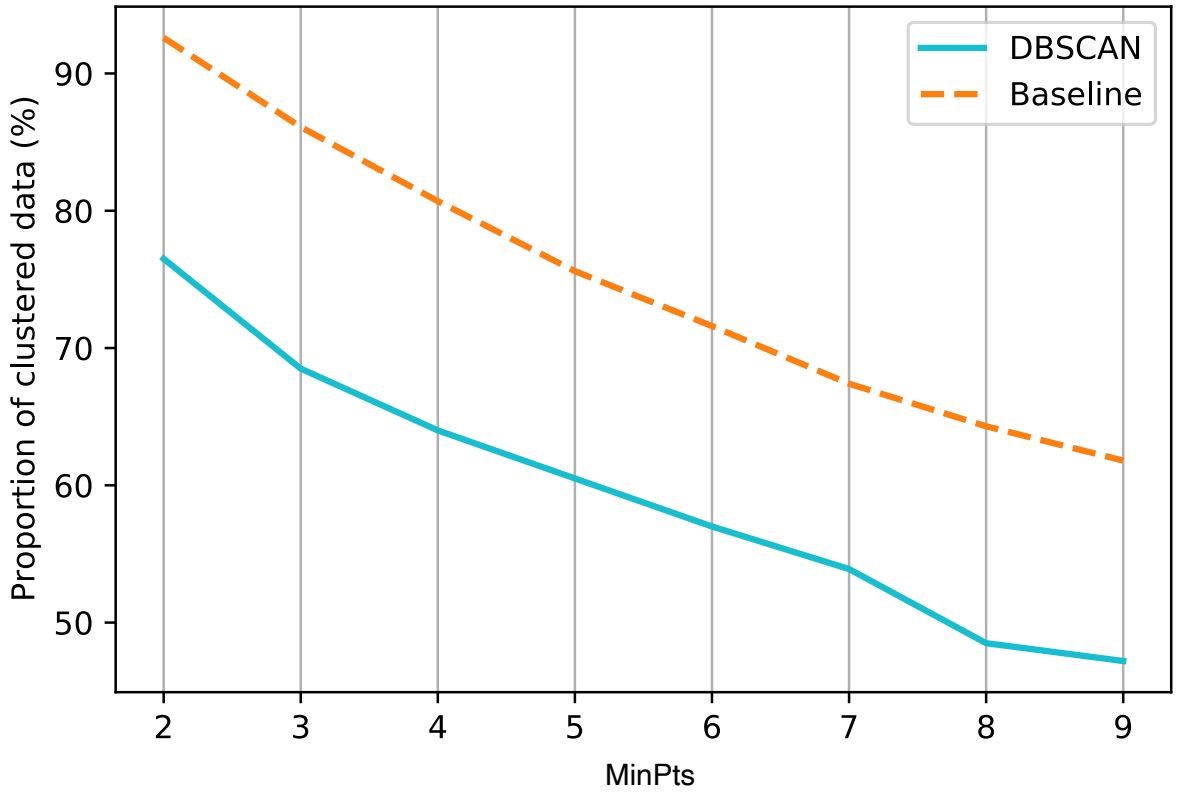


Figure 4.7: Percentage of clustered data using DBSCAN with different values of *MinPts*.

it is highly close to optimal. Searching for the global optimum is not only worth the cost, as we have to train a countless number of DBSCAN models. and despite the computational costs, we expect not much better results.

After assessment of the results, we determine *radius* to $10^{\frac{-3}{5}}$, as DBSCAN with this *radius* could cluster ca. 65% of jobs, and simultaneously, the variation of runtime stays under 20000. Note that the selected value is close to an optimal global value.

4.3.2.2 MinPts

After detecting the *radius* value, we operate the same approach to detect *MinPts*. We assign a fixed value to *radius* and observe clusters' quality by changing the value of *MinPts*. We consider the already detected value of *radius* as an anchor and narrow our search space to the basket size of natural numbers in the range of $[2,9]$. We do not consider *MinPts* of greater than 9, as the percentage of identified clusters by such values dramatically decreases. Afterward, for the range of values of *MinPts* in a loop, a DBSCAN model is generated. In every iteration, similar to our approach in finding the appropriate value of *radius*, some important information was gathered, namely runtime variation, percentage clustered, and the total number of groups. We also compare the results with baseline. The difference, here, is that we dynamically set the minimum required size of groups deterministically to the value of *MinPts*. Because comparing the identified clusters through DBSCAN with the baseline is not valid when they have different minimum basket sizes.

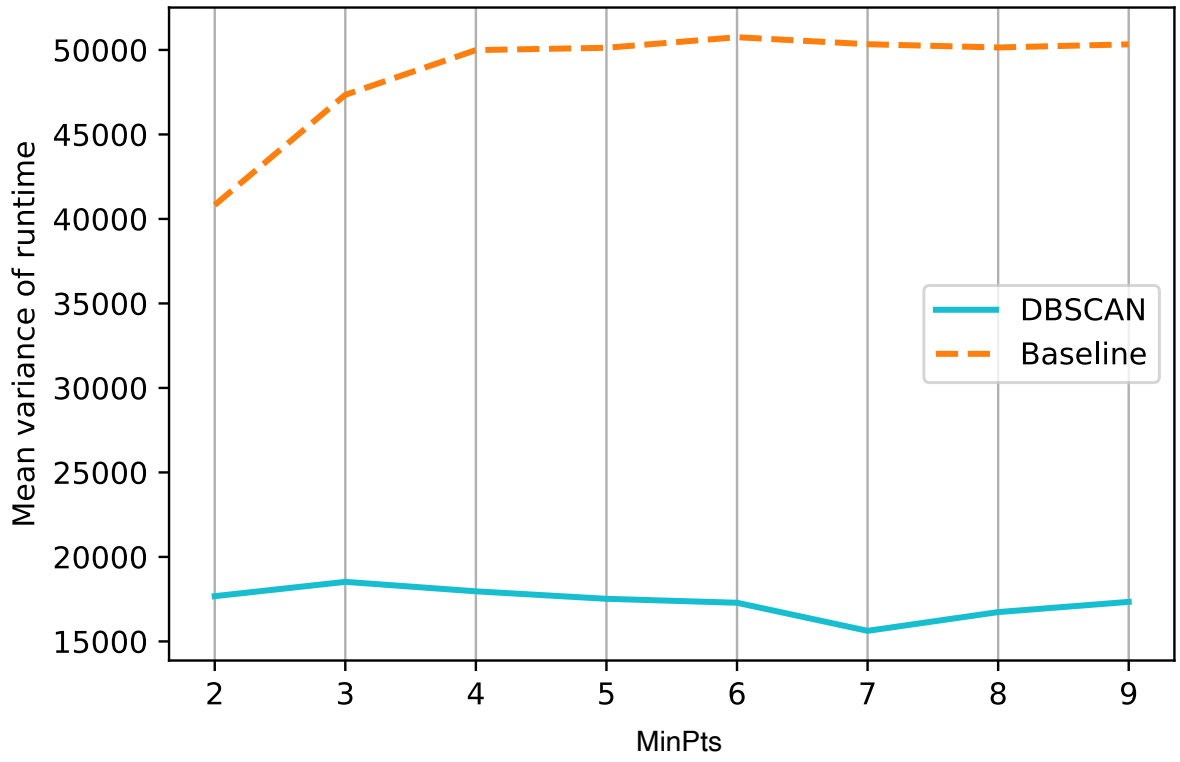


Figure 4.8: Mean runtime variation of jobs within the detected clusters generated through different values of *MinPts*.

Figure 4.7 represents the percentage of clustered data with different values of *MinPts*. As expected, by increasing the minimum size of the basket, the number of clustered points strongly decreases. Because by assigning a larger value to *MinPts*, we add a tighter constraint on the formation of clusters. This constraint is also applied to the baseline groups. It is also visible that DBSCAN, compared with baseline clusters, approximately clusters 20% fewer data points. Similar to the previous section, we measure the quality of clusters with different settings of *MinPts*. The figure 4.8 illustrates the changes in the mean of runtime variation. Unlike changes of other parameters, the mean of runtime variation does not have a constant tendency, and it does not change dramatically by different *MinPts*. Because changing this parameter affects not only the quality of clusters, but also the total number of groups. One should expect noticeable changes in the mean of runtime by considering a higher value of *MinPts*. We set the value of this parameter to two, since at this point the percentage of clustered samples is the highest.

4.3.3 OPTICS

The generated embeddings can not directly be interpreted, as they are an artificial compound of the nodes and edges information within the graph. Thus, we used OPTICS to select the proper embeddings. To do that, we trained multiple OPTICS models with different embeddings. The embeddings vary in the number of dimensions and the strategy of aggregation in global pooling. The OPTICS algorithm is highly similar to the DBSCAN. The key difference is the *radius* value that can automatically be detected through the reachability graph. In the reachability graph, the reachability distance and a spot within the ordered clusters for every point are calculated. In

order to define the reachability distance, the calculation of core distance is required. The core distance is the minimum value of *radius* necessary to mark a sample as a core, by a finite value of $MinPts$. The reachability distance is the shortest distance of a sample to the closest core sample.

It is worthwhile to mention that the reachability distance must be greater than the core distance of a sample. If we do not set a maximum value for *radius*, the OPTICS will consider all the distances to find the most proper distance for generation of different clusters [39]. Furthermore, unlike the DBSCAN, the value of the $MinPts$ parameter does not have a significant impact on the performance of the algorithm because all the distances are automatically scaled at a very similar rate [39]. In other words, OPTICS is able to form clusters even in the absence of domain knowledge. The automated assignment of hyperparameters leads us to use OPTICS to evaluate the generated embeddings. We used the Scikit-learn library to operate the OPTICS $MinPts$ [53]. In the implementation, we set *radius* to infinite, aiming to form the optimum clusters. Besides that, we cluster the data using $MinPts$ of two, to identify all possible clusters. For every group of embeddings, we collect the following information:

- ◇ *Runtime-variation:* the variation of the runtime of jobs within every cluster.
- ◇ *Percentage-clustered:* the proportion of clustered data.
- ◇ *Group-number:* the number of detected clusters.
- ◇ *Avg-group-size:* the average size of groups.
- ◇ *Largest-group-size:* the members of the largest cluster.
- ◇ *Outliers-number:* the number of points that do not fall in any cluster.
- ◇ *Outliers-runtime-variation:* the variation of runtime of all outliers.

Having the information and after analyzing them, it is possible to select the appropriate embeddings. The results, as well as the methodology of selection of embeddings, are discussed in the section 5.4.

4.4 Runtime Prediction

Throughout the previous sections, we discussed our methodology to identify reoccurring jobs. Utilizing the identified groups, we are able to predict runtimes. The group label of jobs should be considered as a categorical value, as their order is unrelated to runtimes. In other words, increasing or decreasing the group labels do not provide and meaning. Hence, we predict the runtime of every group independently. To do that, we assume that the most recent job in every group is an unseen sample, then using the properties of other group members and performing a linear regression[55], the runtime will be predicted. The linear regression model assigns coefficients $W = (W_1, ..., W_p)$ to p explanatory variables and generate a pattern aiming to minimize the residual sum of squares (RSS) between the explanatory variables as follows:

$$RSS = \sum_{i=1}^{n} (y_i - W_0 - W_1 x_{i,1} - \cdots - W_p x_{i,p})^2, \tag{4.11}$$

where $x = (x_1, ..., x_p)$ are the explanatory variables. We run multiple linear regression models with multiple sets of explanatory variables to determine which graph-level attributes have the most predictive value. To train a linear regression model, we utilize the graph-level attributes, e.g., runtime, mean of instance numbers, mean of plan CPU, etc., to make the predictions as accurate as possible. The algorithm 2 shows our methodology for predicting runtimes. It is worth to mention that we apply an identical algorithm for the baseline approach.

Algorithm 2: Runtime Prediction; The algorithm categorizes the data based on the cluster labels and train a linear regression model to predict the most recent job

Result: Runtime predictions

```
// Creating the set of clusters
clusters_list = set(DAGs.cluster_label) Prediction_list = list()
for cluster in clusters_list do
    // Select the recurring jobs
    Recurring_jobs = DAGs.cluster_label == cluster
    Recurring_jobs.order_by(start_startTime)
    // Defining the train and test set
    train_set = Recurring_jobs.iloc[:-1 , :]
    test_set = Recurring_jobs.tail()
    // Model training
    model = LinearRegression().fit(train_set)
    // Prediction
    prediction = model.predict(test_set) Prediction_list.append(prediction)
end
Return Prediction_list
```

5
Evaluation and Discussion

This chapter discusses how our proposed contribution achieves its goal of detecting reoccurring jobs and prognosticating the runtime of incoming jobs. Accordingly, we first explain the dataset we are working with, then discuss our baseline methodology in predicting runtimes. Afterward, we present the result of the performance of OPTICS and how it assists us in improving the graph embeddings. Next, utilizing K-means, we compare the ability of our methodology in the detection of reoccurring jobs to the baseline. Eventually, we report the performance of DBSCAN and evaluate the predicted runtimes.

5.1 Dataset

Alibaba cluster trace dataset [13], consists of trace data of about 4000 machines for ca. 8 days. The dataset is composed of 6 tables as follows:

- ◇ *Machine_meta:* the meta and event information of target machines.
- ◇ *Machine_usage:* the rate of resource usage of target machines.
- ◇ *Container_meta:* the meta and event information of containers.
- ◇ *Container_usage:* the rate of resource usage of containers.
- ◇ *Batch_instance:* the information regarding instances of the workloads.
- ◇ *Batch_task:* the information about instances of the workloads (also include DAGs).

All the required information for this thesis pipeline, including the structure of job dependencies and runtimes, is accessible in the batch_task table. The batch_task table has eight features, as discussed in section 3.2. The Batch_task table includes 14,295,731 tasks of 4,201,014 jobs. The data is published in Comma Separated Value (CSV) format. We transform the data to a *pandas* data frame [67] that is an open-source data analysis and manipulation tool built on top of the Python programming language. Additionally, to calculate basic statistics, we use *NumPy* [68] which is one of the main packages for computing in Python. NumPy enables multidimensional data and object handling, e.g., masking the data, matrices, etc.

Table 5.1: Number of exception jobs belonging to different categories.

Exception category	# jobs	Percentage (rounded)
Jobs containing less than 10 tasks	2,914,563	69%
Jobs without any dependencies	1,339,890	32%
Jobs with parallel description of tasks	396	$< 1\%$
Unfinished jobs	122,825	3%
Jobs with undefined values	36,589	3%
Weakly connected graphs	9714	$< 1\%$
Graphs that form no DAG	25	$< 1\%$
Runtime over 1 hour	2255	$< 1\%$

5.1.1 Exceptions Handling

We eliminate the samples that according to the constraints defined in the section 4.1.1. The number of jobs that fall into the different outlier categories is presented in the table 5.1. Note that, to avoid unnecessary calculations, we eliminate the exceptions in two steps which are separated in the table through the horizontal line. The exception that located above the second horizontal line of the table are calculated after eliminating the first exception categories. Moreover, because highly time-consuming jobs affect the training phase of the GNN and the prediction results, we eliminate the jobs with a runtime longer than one hour. After the data cleaning step, the total number of remaining jobs is $232,809$. It should also be noted that some jobs fall into more than one exception category. To determine weakly connected graphs and graphs that do not form DAGs, we use the methods of the NetworkX package, which is designed to handle and study graph objects [50].

5.1.2 Data Characteristics

This section provides some statistical facts of the dataset. It should be borne in mind that the exception jobs discussed in the previous section are already eliminated. Analyzing the data in all aspects is not feasible; thus, we visualize and provide metrics related to the goal of the present thesis. Most of the jobs were executed within five minutes. Figure 5.1 presents the distribution of runtimes of jobs. As it is visible, 75% of the jobs have a runtime of fewer than 335 seconds. The figure also indicates the fact that less than 5% of jobs take longer than half an hour. The orange plot represents the Kernel Density Estimate (KDE) of runtimes. KDE shows the density of the probability function (PDF) variables by using Gaussian kernels [69].

It is also reasonable to realize the level of parallelization in the target dataset. The number of tasks is insufficient to estimate the parallelization, and the way tasks are dependent should also be considered. Nevertheless, according to work by Tian et al., the number of tasks is a good indicator of parallelization level [14]. The distribution of task numbers of jobs is illustrated in figure 5.2. It is worth reminding that, to avoid accidentally labeling jobs as reoccurring, we eliminate jobs containing fewer than ten nodes. The figure shows that half of the jobs have less than ten tasks, and more than 95% of jobs consist of less than 30 tasks. We visualize the plot by using the Matplotlib package, which is a two-dimensional graphics library for Python [70].

Moreover, to achieve a deeper insight into the distribution of structure of DAGs, figure 5.3 represents the distribution of DAGs' densities. All of the DAGs in the dataset have less than

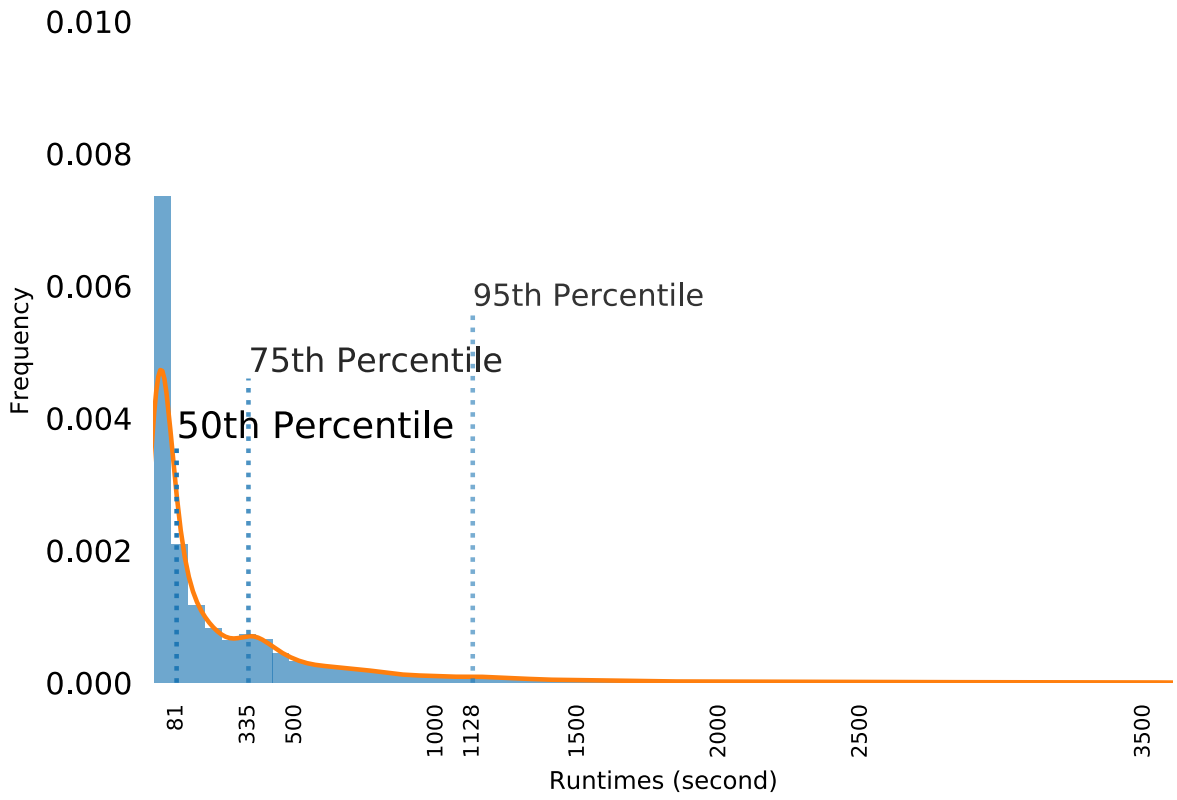


Figure 5.1: Distribution of runtimes. According to the figure, the runtimes are highly variant. Most runtimes are executed in less than 30 minutes, and more than 50% of jobs have a runtime of less than one minute.

20% of possible edges. Indeed, the DAGs of production jobs are relatively sparse, as most of the tasks are only dependent on the results of a limited number of other tasks.

To maintain an in-depth perspective of the data, we present some statistical facts for other essential features. For different features described in table 5.2 the calculated statistics are mean, the STandard Deviation (STD), minimum and maximal values, as well as 25th, 50th, and 75th percentiles. Note that all the features are in the category of graph-level attributes. For instance, *Dependencies AVG* is the average number of dependencies of tasks within the jobs.

5.2 Rule-Based Classification

The main idea of this thesis is a novel approach; thus, to the best of our knowledge, only a limited number of works exist that could be considered as baseline. Nevertheless, there is a similar work proposed by Tian et al. that defines an approach to detect recurrent jobs [14]. In order to do that, the authors argue that two or multiple jobs should be assumed as recurrent when they fulfill two following conditions:

1. The associated DAG of jobs should be an isomorphism.
2. The start time of jobs should happen within periodical time intervals.

Although this work defines a framework to extract recurrent jobs, the authors do not use them to predict runtimes. To define our baseline, we first detect recurrent jobs by repeating the

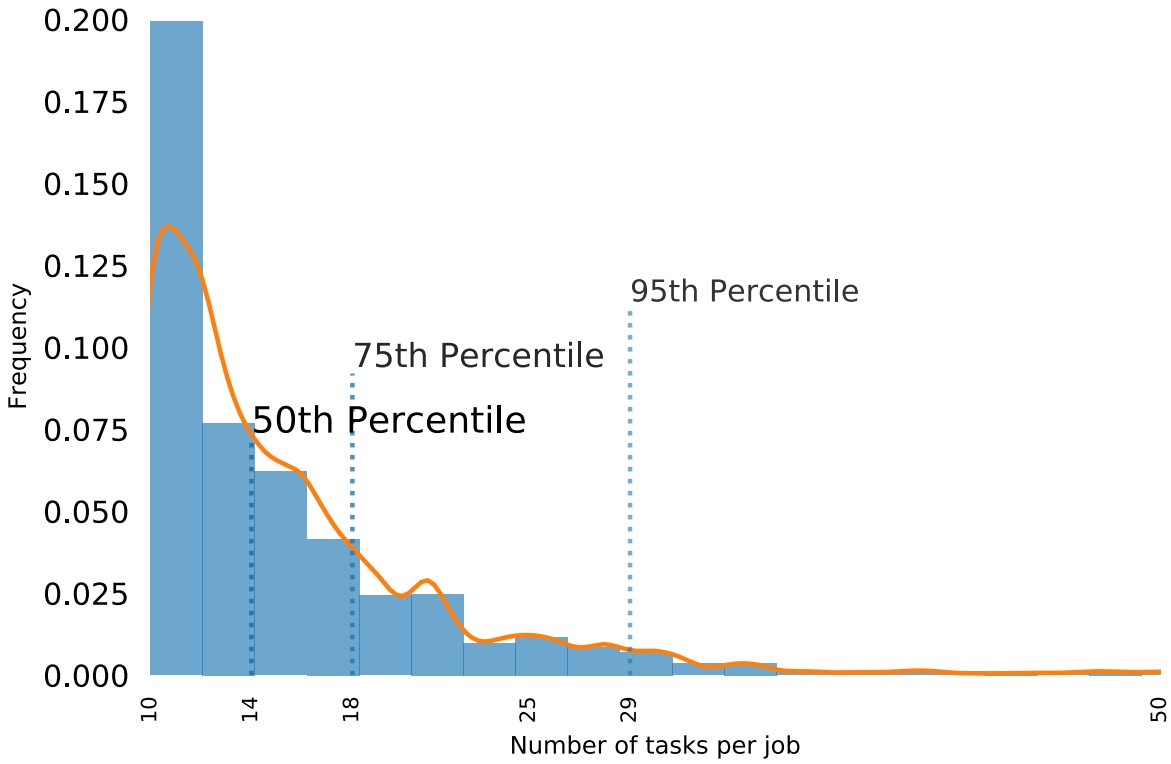


Figure 5.2: Frequency distribution of task number of jobs. According to the figure, the number of tasks and the number of jobs within the category have an inverse relationship.

Table 5.2: Statistical description of dataset

Feature	Mean	STD	Min	25%	50%	76%	Max
Runtime	264.72	443.19	0	20	81	335	3600
# Tasks[1]	15.75	7.42	10	11	14	18	134
# Dependencies[2]	16.92	10.34	9	11	14	19	218
Dependencies AVG[3]	1	0.17	0.90	0.92	1	1.1	5.28
Instance number AVG[4]	105	371	1	3.57	20.54	81.25	13346.90
Planed CPU AVG[5]	78.13	25.43	50	50	100	100	708.96
Planed memory AVG[6]	0.35	0.08	0.15	0.27	0.39	0.42	1.84

[1] Number of tasks within the jobs.
[2] Number of dependencies within the jobs (edge number of DAGS).
[3] Average number of dependencies along the tasks within the jobs.
[4] Average of instance numbers within the jobs.
[5] Average of planed CPU within the jobs.
[6] Average of planed memory within the jobs.

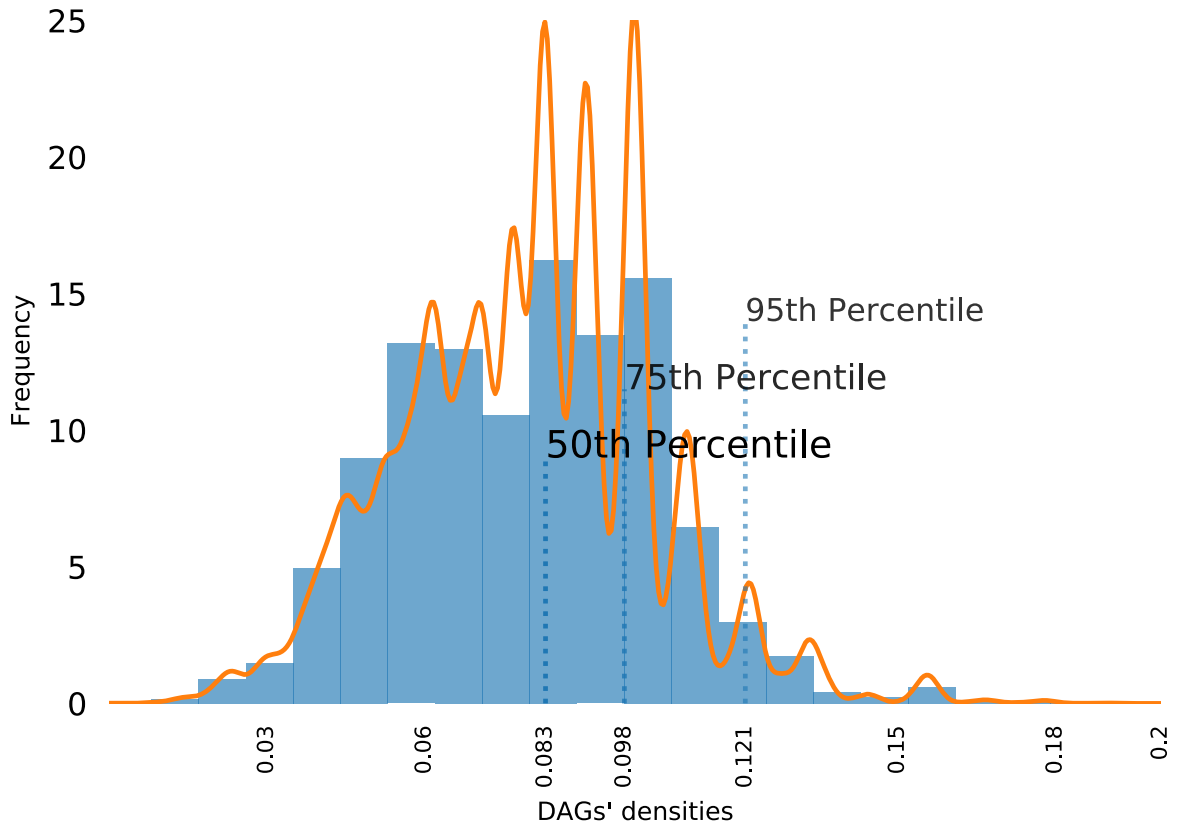


Figure 5.3: Frequency distribution of DAGs' densities. No DAGs have a density of more than 0.2, which means that the job dependencies produce sparse DAGs.

explained process in this thesis. Note that, to check the isomorphism of the graphs, we operate a different approach compared to the work from Tian et al. Indeed, the authors in their work identify the isomorphic DAGs object by comparing the images of DAGs. In the present thesis, however, thanks to the DAGs extraction step, we can check for isomorphism of graphs more efficiently by analyzing the structure of DAGs. Next, using the characteristics of recurrent jobs, we predict the runtimes and evaluate our results. Further details of the approach are explained in the following sections.

5.2.1 Recurring Jobs Detection

Our baseline considers two jobs as recurring when they fulfill the two above-mentioned conditions. Here, we discuss the approach of detection of recurrent jobs in more detail.

5.2.1.1 Isomorphism Check

The associated DAGs of recurrent jobs should hold the identical structure, or in other words, should be an isomorphism. To check isomorphism order and labels of nodes is irrelevant, e.g., two graphs with different node labels are still considered isomorphic if the number of nodes are equal and have the same connectivity. In this stage, we do not consider the attributes of nodes, as only the way that tasks are dependent on each other matters.

To check isomorphism, we first generate DAGs objects using the approach described in section 4.1.1. Checking the isomorphism of graphs is by its nature computationally expensive, and due to high numbers of graphs, an appropriate algorithm should be selected. Consequently, we compare the graphs using *bliss* approach to extract isomorphic graphs [71]. The bliss is a tool that solves the problem of checking isomorphism by coloring the graphs. A *colored graph*, in addition to sets of edges and vertices, consists of a set that assigns positive integers of N (colors) to vertices. The colored graph of G is generated using the following equation:

$$G^{\gamma} = (V, \{\{v^{\gamma}, w^{\gamma}\} | \{v, w\} \in E\}, c^{\gamma}), \tag{5.1}$$

where $c^{\gamma} : V \to N$ is defined for all vertices by $c^{\gamma}(v^{\gamma}) = c(v)$, and c^{γ} is a permutation of V denotes by v^{γ}. The bliss considers two graphs as isomorphic if a permutation of $\gamma : V \to V$ exists such that the colored graph of G_1 equals the G_2. The complexity of this approach is still exponential but in practice, for small graphs of jobs dependencies, performs efficiently. We apply the bliss approach on the graphs using the igraph package, which is a library collection for creating and manipulating graphs and analyzing networks [72].

5.2.1.2 Temporal Intervals

It is mostly the case that the recurring jobs run periodically. This is the reason that Tian et al., in their work, consider only the jobs as candidates of recurring when they begin in periodic time intervals [14]. Microsoft reported that there are three routine periods in production jobs, namely, 15 minutes, 1 hour, and 24 hours [73]. Therefore, we prune the search space by considering only the jobs that begin in the above-mentioned periodic times. As Alibaba traces do not guarantee the exactness of submission time, we expand those three intervals by adding 3% of intervals after and before every period. To achieve this goal, we calculate for every candidate the beginning time of interest; then we enlarge the time by producing an incremental list of seconds that begins with $T - 3\%P$ and ends with $T + 3\%P$, where T denotes the periodic times of interest, and P is the time interval. The algorithm 3 illustrates this procedure.

Note that the 6% margin of the different intervals is relational. Hence, the hourly time intervals are by far smaller than daily intervals. The algorithm 3 returns the seconds that a recurring job might happen; thus, it decreases the search space.

5.2.2 Rule-Based Classification

The jobs that meet the two aforementioned conditions group together; in other words, these two conditions define each class's rules. To avoid unnecessary computational overhead, we first filter the jobs that their start-time does not begin within extracted seconds of algorithm 3. Then, we check the isomorphism of graphs. In addition to job classes, we distinguish between different periodic time intervals by assigning three flags. The Flags indicate in which time intervals the jobs start. It is worth mentioning that one job may run at a time that satisfies more than one time intervals.

After classification, it is observed that 91.97% of jobs have at least one recurring job. In table 5.3 the results of some queries over the results are presented. Moreover, figure 5.4 illustrates the average size of groups by different categorized task numbers. As expected, the figure shows that the size of groups partly decreases by more complicated job dependencies with a higher number of tasks. We highlight the fact that the dataset does not include any information

Algorithm 3: Periodic time extractor

```
Result: target seconds of jobs
GolbalMaxTime = MAX(batch.start-time);
MinInterval, HourInterval, DayInterval = 900, 3600, 86400;
// Converting intervals to seconds
for t in batch.start-time do
    MinitueCenters = range(begin=t, end=GolbalMaxTime, step=MinInterval)
    for center in MinCenters do
        extract all seconds from 3% before to 3% of MinInterval after center
    end
    HourCenters = range(begin = t, end = GolbalMaxTime, step = HourInterval )
    for center in HourCenters do
        extract all seconds from 3% before to 3% of HourInterval after center
    end
    DayCenters = range(begin = t, end = GolbalMaxTime, step = DayInterval)
    for center in DayCenters do
        extract all seconds from 3% before to 3% of DayInterval after center
    end
end
eturn extracted seconds
```

Table 5.3: Queries over rule-based classification

Query	Result
Percentage of jobs that assigned to a group	**92.08%**
# groups	**26980**
Mean of group sizes	**8.53**
# recurring jobs with time interval of 15 Minutes	**97233** (42.22%)
# recurring jobs with hourly time interval	**115879** (50.31%)
# recurring jobs with daily time interval	**139445** (60.55)%

regarding actual recurring jobs, which means that no ground truth is available. Consequently, we assume the runtime variation of groups as an indicator of recurring jobs.

5.2.3 Runtime Prediction

After identifying the recurring jobs, we are at the stage of making use of groups and their similarity to predict runtime. To achieve this goal, we first split the jobs by their group labels and then train a multiple linear regression model for every group of recurring jobs. We assign the most recent job of every group to the test set and calculate the accuracy of the model using Mean Absolute Error (MAE) and Mean Squared Error (MSE). Additionally, we check how good the attributes namely, *instance_number*, *planned_CPU*, *planned_memory* could improve the accuracy of the predictions. One should be aware that the mentioned attributes are node-

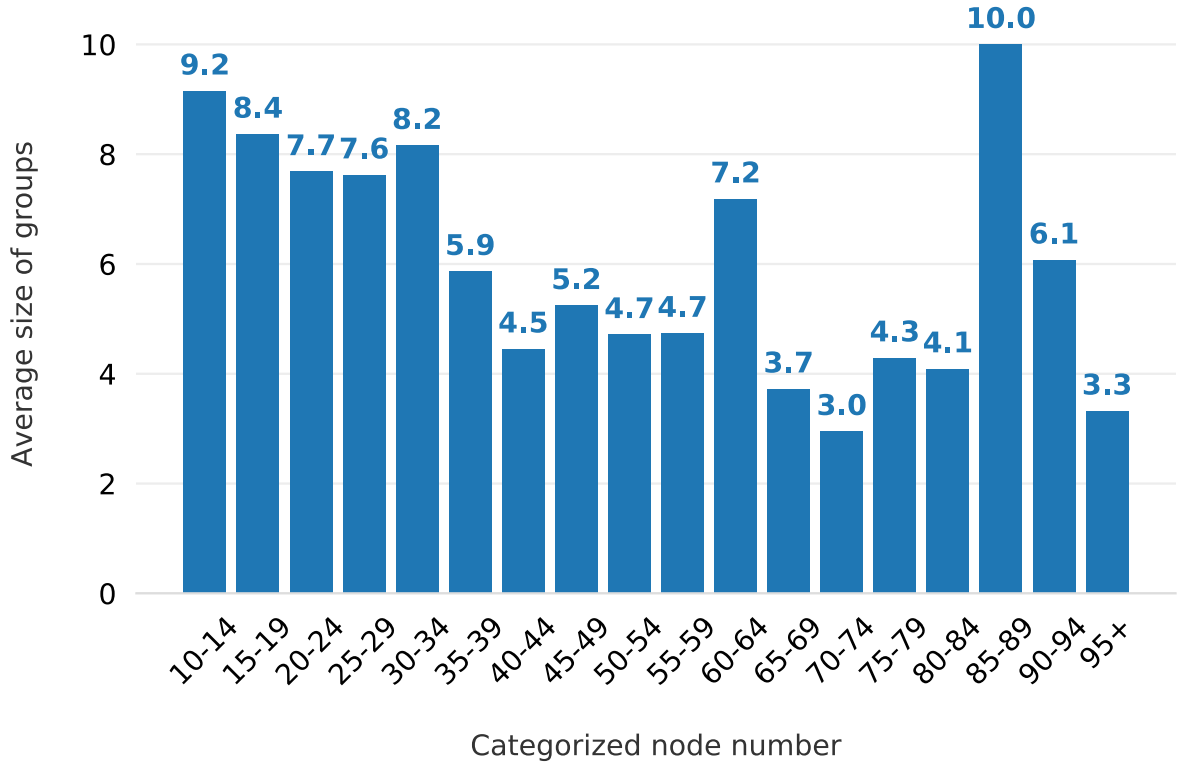


Figure 5.4: The average size of groups is detected through rule-based classification by different categories of node numbers (number of tasks).

level, and we need to transform them to graph-level using some multiple aggregation functions. To do that, we define six scenarios as follows:

1. Three explanatory variables selected. Aggregation is done by calculating *variance* of the three node-level attributes.

2. Three explanatory variables selected. Aggregation is done by calculating *maximum* of the three node-level attributes.

3. Three explanatory variables selected. Aggregation is done by calculating *summation* of the three node-level attributes.

4. Three explanatory variables selected. Aggregation is done by calculating *mean* of the three node-level attributes.

5. All the mentioned variables selected. In this scenario, the linear regression model fits with all aggregated variables.

6. None of variables considered. The model fits with a constant value of one. In other terms, the prediction is made by calculating the runtime mean of previous jobs.

Table 5.4: Predictions evaluation using rule-based classification in different scenarios.

Scenario	MSE	MAE
1^{st} scenario (variance)	$> 10^{38}$	$> 10^{17}$
2^{nd} scenario (maximum)	$> 10^{28}$	$> 10^{12}$
3^{rd} scenario (summation)	$> 10^{32}$	$> 10^{32}$
4^{th} scenario (mean)	$> 10^{33}$	$> 10^{15}$
5^{th} scenario (all)	$> 10^{41}$	$> 10^{18}$
6^{th} scenario (none)	$> 10^{4}$	$> 10^{2}$

5.2.4 Evaluation

As stated before, to measure the prediction accuracies in different scenarios, we calculate MAE and MSE. For this purpose, the number of jobs that could be predicted is equal to the number of groups with more than one member. The table 5.4 presents the results. According to the table, the graph-level attributes are not able to improve runtime predictions. Although MAE and MSE can measure the prediction task, they are susceptible to outliers, as one bad prediction can greatly adjust the measurement value. The sensitivity to outliers is even worse for the MSE metric because it calculates the squared distances. To overcome sensitivity to outliers, we visualize the distribution of absolute error between predicted values and runtimes. Figure 5.5 shows the distribution of absolute error in all the scenarios through box plots. To make the distributions more readable, we do not show the fliers in the distribution, which are the values greater than $Q_3 + 1.5(IQR)$, where Q_3 is the third quartile of the distribution and IQR is the interquartile range. The graph generated with pandas package [67].

Not only the distributions but also MAE and MSE metrics indicate that the graph-level attributes do not provide predictive information, as the results in the absence of them have the highest accuracy. The results show that the architecture of dependencies in comparison with graph-level attributes is more predictive. Thus, we consider the sixth scenario as the baseline, which means that the incoming job's runtime equals the mean runtime of previous jobs.

5.3 GNN Evaluation

As outlined above, we split the DAGs' dataset with a stratified sampling method. The GNN trained singly via the trainer set, which is 70% of the whole data. Afterward, in every epoch, we calculate the losses separately for the train and test set. We apply such an approach to make sure the GNN is not overfitted and measure the performance of the model in case of unseen samples. The calculation of losses for the test set is similar to the methodology discussed in section 4.2.2. Furthermore, the early stop strategy discussed in the section is applied to test set losses.

The figure 5.7 shows the results of losses of train and test set with different settings of global pooling layer. The loss changes in all the cases indicate that the GNN can find a pattern to learn the representation of graphs based on the defined target values. Except for the one in the bottom left of the figure, all the graphs represent the losses for the GNN with a global pooling layer of average but with different sizes in the output layer. The graph in the lower left of the figure represents the losses of a GNN that has a mixture of multiple aggregation functions

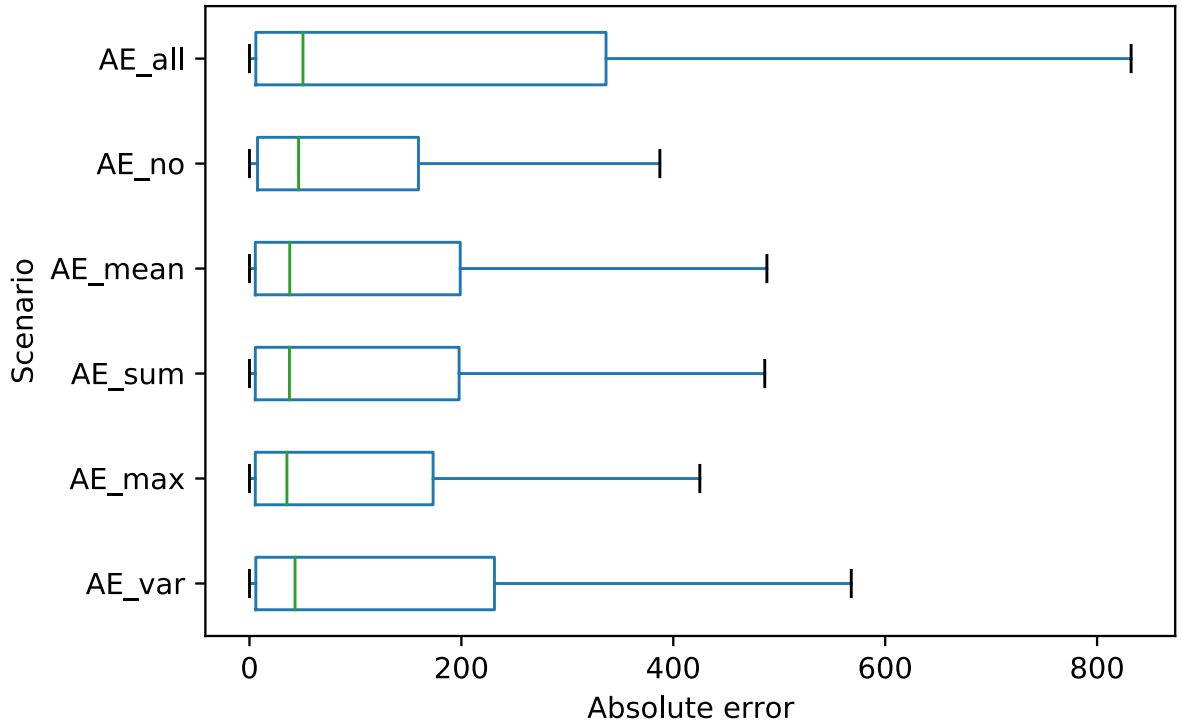


Figure 5.5: Distribution of absolute error in different scenarios. The recurring jobs are detected using rule-based classification.

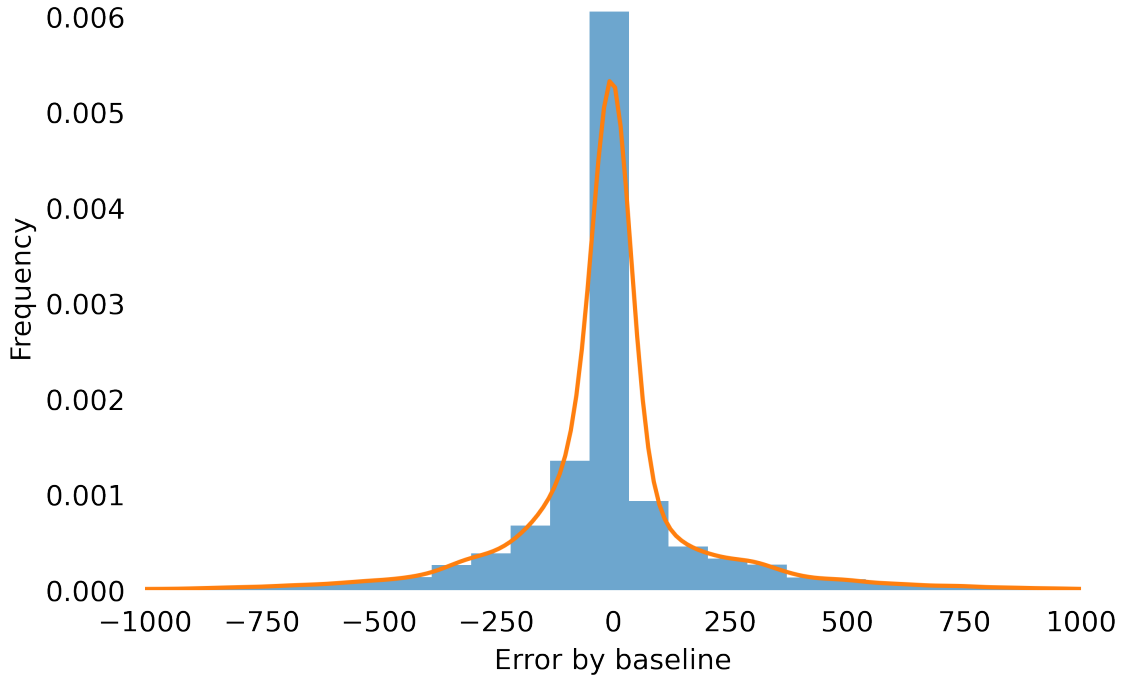


Figure 5.6: The distribution of prediction error generated by baseline.

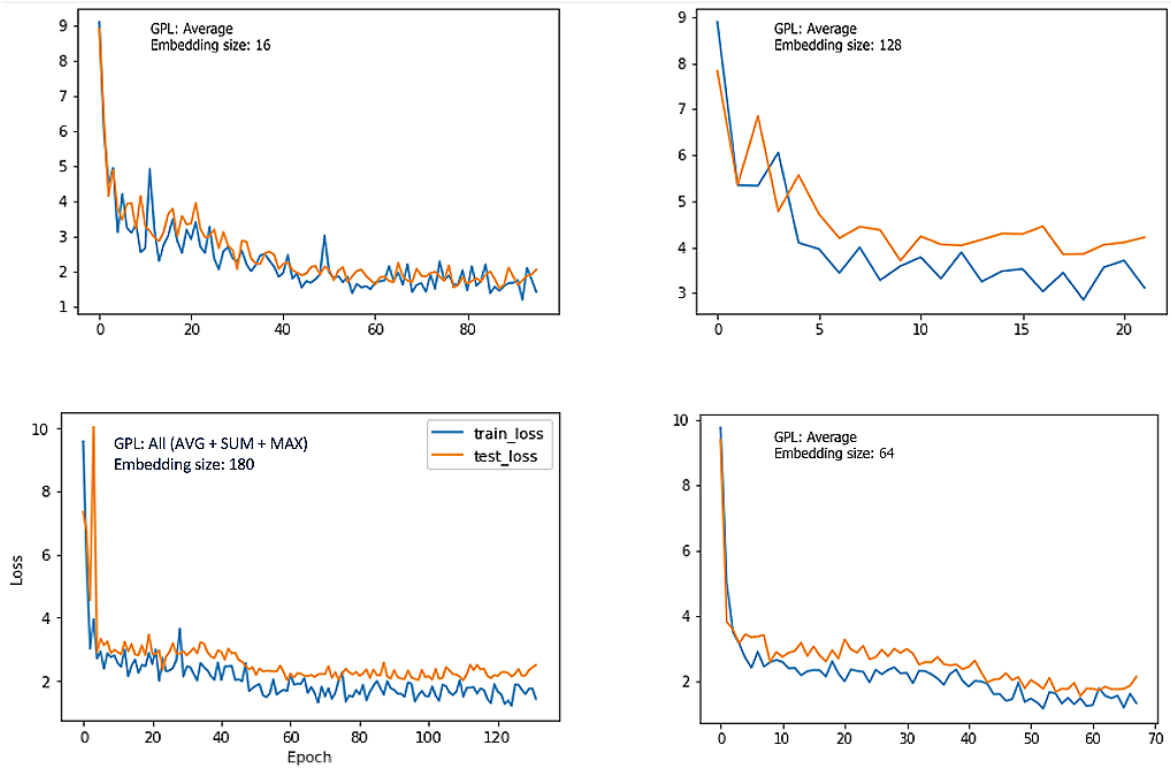


Figure 5.7: The GNN model evolution with different settings of Global Pooling Layer (GPL) and output layer size. (1) GPL of the average and embedding size of 16 (upper left). (2) GPL of the average and embedding size of 128 (lower right). (3) GPL of the average and embedding size of 64 (lower right). (4) Three GPLs, namely, average, maximum, and summation. The embedding size of 180 is the Concatenation of three GPLs (lower left).

within the global pooling layer, namely mean, sum, and max. Here, every aggregation approach summarizes the DAGs information in a vector of size 60, and the final embedding is their concatenation. All the graphs show an amplification of test losses in the latest epochs, which is the reason that early stopping approach is activated and prevents the model from overfitting. We consider all the generated embeddings as candidates, and we check their quality regarding their ability to predict runtimes using the approach discussed in the section.

5.4 OPTICS Clustering

This section discusses our methodology to explore the GNN generated embeddings and ranking them through the OPTICS algorithm. Needless to say, that the candidate structure of GNNs is selected base on our domain knowledge, and thus there is no guarantee to have global optimal GNN among the candidates.

The stochastic nature of generated embeddings leads us to study their quality so that no prior information regarding the embeddings is required. As mentioned before, OPTICS is highly independent of settings of hyperparameters beforehand. One can use OPTICS to cluster data efficiently without defining the number of clusters, the minimum size of clusters, and *radius* to define the distances.

To train the OPTICS models, first, we perform the preprocessing steps discussed in section 4.3.1. To train the OPTICS model, we used the Scikit package and made an OPTICS object

Size[1]	GPL[2]	# groups[3]	AVGsizes[4]	# outliers[5]	Clustered variance[6]
60	MAX	16583	13	25209 (10.8%)	43020
60	AVG	19672	8	66927 (28.7%)	37046
60	ADD	19693	8	66599 (28.6%)	36408
96	ALL	14510	15	10040 (4.3%)	36408
16	AVG	22130	8	50798 (21.8%)	30622
64	AVG	22384	8	60205 (25.8%)	**30465**
128	AVG	17379	10	66263 (17.2%)	39983

Table 5.5: OPTICS evaluation with multiple groups of embeddings.

[1] Size of embeddings vector.
[2] Aggregation function in Global Pooling Layer.
[3] Number of formed clusters.
[4] Average size of clusters.
[5] Number of non-clustered samples.
[6] Mean of runtime variance of jobs within clusters.

with default parameters as follows: class OPTICS *(min_samples = 5, max_eps = inf, metric = minkowski, p = 2, cluster_method = xi, eps = None, xi = 0.05, predecessor_correction = True, min_cluster_size = None, algorithm = auto, leaf_size = 30, n_jobs = None),* where the parameter *eps* has the same meaning as *radius*, which we discussed before, and *metric* is the approach of distance computation. The *Minkowski* distance is the generalization of the Euclidean distance and calculated via the following equation [64]:

$$D(E,\acute{E}) = \left(\sum_{i=1}^{n} |E_i - \acute{E}_i|^p \right)^{\frac{1}{p}}, \tag{5.2}$$

where E and $\acute{E}$ are two arbitrary embeddings with the equal dimension of n, and p is an integer. Scikit uses the equation with default values of 2. Moreover, by setting the *max_eps* to inf, the OPTICS model checks all the possible distances to find optimal global and local distances.

In the next step, we train the model with different embeddings generated through multi-structural GNNs. Table 5.4 illustrates some vital statistics after forming the clusters, which help us to have a deeper insight into the embeddings' quality and their ability to predict runtimes. The embeddings represented in the table differ in two aspects, namely vector size of embeddings, and aggregation strategy in the global pooling layer. For each group of embeddings, we calculate the statistics discussed in section 4.3.3. The table shows a negative coefficient between the number of groups and the mean-variance of runtimes. The same relation is visible in the case of outliers proportion. This proves our assumption that the model's ability to detect clusters can positively affect runtime predictions. It is worthwhile noting that the *clustered variance* (sixth column of the table) reflects the mean of runtime variance of all the groups, excluding the outliers. Based on the primary goal of this work, we select the candidate embeddings that enable the clustering model to generate groups with the least runtime variance.

We highlight that we used OPTICS to explore the quality of embeddings with different settings of the GNN. Consequently, to make the results reliable, all the clustering tasks are

performed with identical OPTICS settings.

5.5 K-means Clustering

K-means offers many advantages, e.g., the guarantee of convergence, the ability to detect clusters with different shapes, etc. This leads us to apply such a clustering method to the dataset. Nonetheless, the essential definition of the number of clusters beforehand made k-means an unsuitable approach for predicting runtimes in real-world scenarios, as the number of clusters can not be determined beforehand. Thus, we use k-means to compare rule-based classification and the ability of embeddings to identify reoccurring jobs.

5.5.1 Model Setup

To apply K-means, we used Sklearn package with default parameter except for the k as follows: class KMeans *(n_clusters = k, init = k-means++, n_init = 10, max_iter = 300, tol = 0.0001, precompute_distances = deprecated, verbose = 0, random_state = None, copy_x = True, n_jobs = deprecated, algorithm = auto)* where $n\_cluster$ determines the number of clusters (k), *init* is the initializing setting. In the default setting, this parameter is set to $k\_means++$, enabling the model to find cluster centroids more intelligently. n_init is the times that different centroid seeds will be used. Max_iter is the maximum allowed number of iteration for a single run. This parameter prevents the algorithm to operate exponential number of computations [64].

5.5.2 Evaluation

For the next step, we trained the K-means object with DAGs' embeddings. We use the embeddings of size 64 generated via the global pooling layer and operate the standardization and normalization steps. Table 5.6 provides some statistics to make the comparison between K-means and baseline. One should consider that K-means algorithm has exponential complexity. Thus, it is not applicable for big datasets like the collection of embeddings. To get over this issue, we split the data based on the criterion described in the first two columns of the table. We select the criteria in a way that we can analyze DAGs with different properties. The first two criteria target the job dependencies with different parallelization structures, namely, the average shortest path within associated DAGs and the number of nodes equal to the number of tasks. The following two criteria, though, consider average instance number and average planned memory, which target job dependencies with various resource usage plans. To make the comparison certain and fair, we set the parameter of k to the number of detected groups through rule-based classification. The result specifies that in all the cases, K-means outperforms the baseline by a large margin. Moreover, the embeddings could enable the K-means to generate more sizable clusters in comparison with rule-based classification.

Moreover, table 5.7 represent the similarity of identified cluster through K-mean to rule-based classification using rand index [44]. We consider the subsets of jobs as in table 5.6, and we calculate the adjusted rand index using Scikit [64]. According to the rand scores, it is in sight that the similarity of clusters of K-means and Baseline are more comparable when the splitter is based on *average shortest path*. Since this criterion, similar to the rule-based classification, addresses the DAGs with similar structures, both the classifiers vote on similar recurring jobs.

Table 5.6: Comparison of K-means with baseline

				Baseline			**K-means**		
Splitter (Y)	Condition	Data size	K[1]	Mean[2]	Max[3]	Runtime variance[4]	Mean	Max	Runtime variance
Average Shortest Path	1<Y⩽ 1.25	19229	2802	245.50	1904	30674	357.91	2010	12360
	1.25<Y⩽ 1.5	4772	809	51.64	316	17647	69.70	337	3979
	2<Y⩽ 3.5	575	148	12.64	43	36134	28.85	113	18717
Number of Nodes	Y==15	14916	2983	33.55	255	32841	92.78	482	13028
	Y==20	4400	1070	28.46	168	29148	57.16	306	14383
	45<Y⩽ 50	1198	335	13.73	58	60115	13.80	57	24438
Average instance number	1<Y⩽ 1.5	16459	3544	54.04	479	12394	28.68	203	9092
	2<Y⩽ 3	13407	3851	26.59	377	8775	19.27	151	2711
	4<Y⩽ 5	12745	2430	95.16	659	7101	43.06	221	1915
Average planed memory	Y⩽ 0.22	25697	5072	37.34	255	19248	96.28	843	8085
	0.25<Y⩽ 0.3	26307	6662	28.08	311	36500	48.10	574	18817
	Y>0.45	19032	4364	56.24	565	46704	110.35	861	15482

[1] Number of clusters, determined through the number of groups found through baseline.
[2] The average size of detected groups.
[3] Size of the largest detected group.
[4] The average of runtime variance within detected groups.

Table 5.7: Adjusted Rand score of different dataset's subset.

Splitter	**Average Shortest Path**			**Number of Nodes**		
Condition	1<Y<=1.25	1.25<Y<=1.5	2<Y<=3.5	Y==15	Y==20	45<Y<=50
AR_Score[1]	**0.494**	**0.742**	0.174	0.168	0.259	0.082
Splitter	**Average instance number**			**Average planed memory**		
Condition	1<Y<=1.5	2<Y<=3	4<Y<=5	Y<=0.22	0.25<Y<=0.3	Y>0.45
AR_Score	0.127	0.183	0.105	0.190	0.285	0.482

[1] Adjusted Rand (AR) index of clusters in comparison with the baseline.

5.6 DBSCAN Clustering

DBSCAN is a member of density based on clustering techniques. Thus, no determination of the number of clusters beforehand is required. Nonetheless, unlike the OPTICS, settings of *radius* and *MinPts* significantly affect final clusters. Furthermore, by assigning these hyperparameters, DBSCAN, compared to OPTICS, can identify clusters computationally more efficiently. This leads us to select this technique to generate final runtime predictions. We use the pre-defined package of Sklearn to apply DBSCAN with the parameters as follows: class DBSCAN *(eps = pow(10, -3.5), min_samples = 2, metric = Euclidean, algorithm = auto, leaf_size = 30, p = 2, n_jobs = None)*, where the value of eps and $min\_samples$ are equal to *radius* and *MinPts* that are determined through the semi-grid search approach [64]. We consider the runtime variation of groups and the percentage of clustered data to select these parameters. Moreover, by setting $algorithm$ to $auto$, the model uses $KD-tree$ approach to compute the pairwise distances. Additionally, the p parameter denotes the *power* value in the Minkowski metric.

Our steps proceed by feeding the DBSCAN model with preprocessed embeddings data. The table 5.8 represent the results of some queries over identified clusters. The DBSCAN labels ca.

Table 5.8: Queries over DBSCAN clustering. The table shows that the clusters are more sizable in comparison with rule-based classification. DBSCAN identified 15.61% fewer recurring jobs in comparison with baseline.

Query	Result
# clusters	22278
Mean size of clusters	166.16
Mean of runtime variance in clusters	17662.75
Percentage of jobs that assigned to a group	76.47%

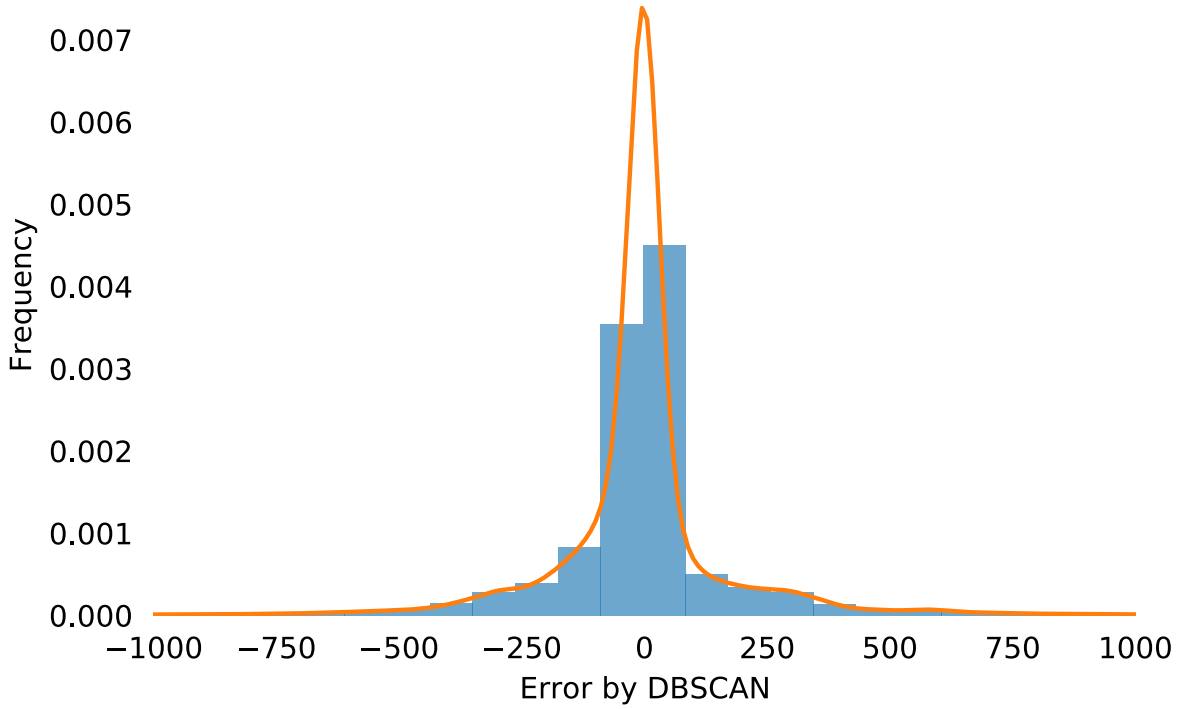


Figure 5.8: Distribution of absolute errors prediction generated using DBSCAN clustering.

23% of the data as outliers and groups the embeddings in 22278 clusters. The average size of clusters is ca. 166, which highlights that the DBSCAN can make larger clusters compared to the baseline. The adjusted rand score for DBSCAN compared with the baseline equals 0.3676, which indicates that the generated clusters are barely similar to the baseline classes.

5.6.1 Runtime Prediction

After the extraction of clusters, we predict the runtime based on the runtime of the other group members. We assume the most recent job as an unseen sample and apply a linear regression model to predict runtime for all the clusters. The approach is identical with the runtime prediction in the baseline approach (see section 5.2.3). The figure 5.8 illustrates the distribution of prediction errors. The figure shows that a great proportion of errors are close to zero.

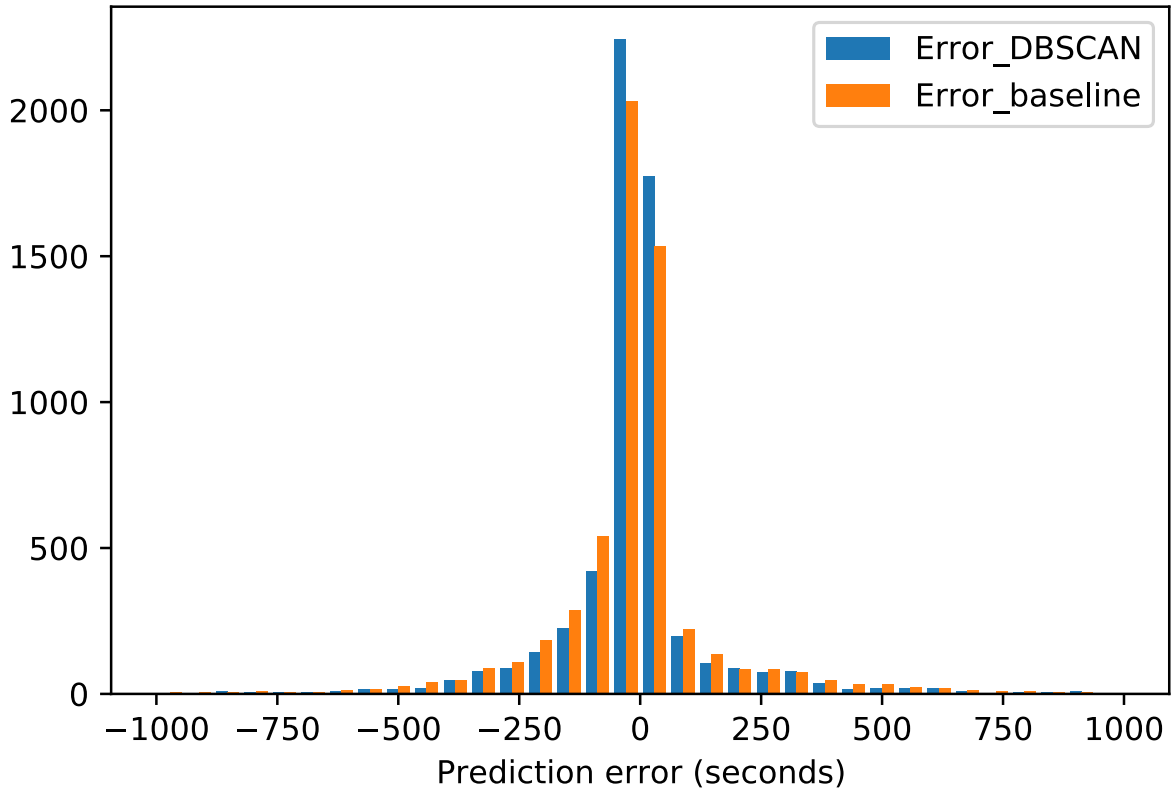


Figure 5.9: The distribution of prediction error generated by DBSCAN and baseline in comparison.

5.6.2 Comparison with the Baseline

DBSCAN and rule-based classification form clusters of different shapes, which becomes an obstacle in operating a complete comparison. To overcome this issue, we only consider the jobs that both techniques could perform a prediction. i.e., the jobs in both rule-based classification and DBSCAN are temporary, the most recent ones. There are 5809 jobs that could satisfy both conditions. The figure 5.9 shows the distribution of errors for both the prediction methods. The figure shows that our method could outperform the baseline with a large margin. The table also shows the mean absolute and mean squared error of DBSCAN compared to the baseline for all the predictable samples. Both the metrics indicate that our approach outperforms the baseline.

5.7 Limitations

We suggest that our contribution has three main limitations, as follows:

Reliability of detected clusters: the lack of ground truth impacts the reliability of clustering tasks. The released dataset does not contain any indicator to help the researchers figure out which jobs are truly reoccurring. This leads us to define two assumptions: (1) DAGs isomorphism and periodically time intervals for the baseline, and (2) similarity of job's representations for the current thesis's contribution. Even though these assumptions are reasonable, we could not verify if the reoccurring jobs of our target dataset are indeed recurrent. One can argue

Table 5.9: MAE and MSE of baseline and DBSCAN in comparison

	Shared		Total	
	MAE	MSE	MAE	MSE
Baseline	110.55	61556.19	140.64	8927.02
DBSCAN	83.50	38050.65	100.97	46429.59
Percentage Difference	27.88 %	47.19%	32.83%	135.49%

[1] The jobs that are through baseline and DBSCAN predictable (5809 jobs).
[2] The jobs that are through baseline and DBSCAN separately predictable, which equals to 26980 and 22278.

that the similarity in detecting clusters' runtime is sufficient to rely singly on the assumptions; however, the clusters are still not verified as reoccurring jobs. Because of lack of ground truth, our choices in detection of reoccurring jobs techniques are limited to unsupervised ones. In other words, this phase could be a classification rather than a clustering task. We believe that the knowledge regarding actual reoccurring jobs and their characteristics could significantly increase the correctness of runtime prediction, and the results might have been more persuasive if we had access to a labeled cluster dataset.

Definition of a globally optimal clustering method: the results of the K-mean clustering phase indicate that the generated embeddings can potentially cluster the data meaningfully. Unfortunately, the lack of computational resources and stochastic nature of generated embeddings become an obstacle to cluster the embeddings in a globally optimal way. To overcome these obstacles, we perform a semi-grid search approach. Thus, we argue that the clustering models are close to their optimal performance. Although a great caution in selecting clustering methods is taken, it remains unanswered whether the utilized clustering algorithm is globally optimal.

Globally optimal GNN: despite the ability of GNN in learning the representation of graphs, during our experiments, we encounter different behaviors of the GNN in training phase. The performance of GNN is highly affected by various hyperparameters, e.g., the structure of the neural network, the methodology of convolutional layers, the loss function definition, the activation function, etc. The defined GNN in this thesis could generate high-quality embeddings and succeed in learning the representations. However, further customization of GNN results in more precise embeddings. For instance, in view of the fact that the DAGs have many common properties, we believe that a specific convolutional layer should be developed.

5.8 Discussions

The most remarkable result to emerge from our contribution is the occurrence of a correlation between the structure of job dependencies and runtime. After evaluating the results, both our methodologies and the contribution from the baseline agree that the structure of job dependencies carries meaningful information, as the runtime variation of clustered jobs is with a high

margin lower than the runtime variation of the rest of the jobs. Our results also suggest that the structure of job dependencies compared to other job features, i.e., graph-level attributes, is more capable of predicting runtimes. Using the GNN, we could provide new insight into the relationship between job dependencies' structure and their runtime.

It needs to be reminded that our experiments agree on prior works, e.g., Tian et al. [14], that the prediction of the runtime is not a trivial task as they are highly variant. Chen et al. also reported that even for recurring jobs, other parameters, such as CPU cores and memory, results in variation in runtime [74]. The evaluation results show that, in contrast with our assumption, there is barely a correlation between different graph-level attributes and runtime. Our experiment also shows that even the combination of attributes fails to predict the runtimes correctly. Surprisingly, the predictions based on the structure of job dependencies alone outperform the prediction results in using any other combination of graph-level attributes. We suggest that the cluster scheduler on allocating the resources emphasize other factors rather than the possible runtime of incoming jobs. In fact, there are probably more critical decision-maker factors in real-world scenarios than the historical similar job's runtime, for instance, the cluster configuration and user's related parameters, which can highly affect the runtime. One can also take advantage of this information besides the historical behavior of jobs to make the predictions more accurate.

The other interesting aspect of our analysis is the results of data clustering with K-means. Indeed, K-means is not applicable in predicting runtime but still is a reliable indicator of generated embeddings' quality. The K-means in all the scenarios could outperform the baseline in terms of the ability of runtime prediction. Moreover, the K-means could mostly form more sizable clusters. The K-means evaluation indicates that the embeddings themselves are highly predictive. We suggest that the outstanding results of K-means are because of three main reasons: first, the number of clusters in all the scenarios is big enough, which forces the k-means to distinguish between embeddings precisely. Second, the K-means, in its nature, consider all the samples and do not assume any data point as an outlier. And not least of all, the ability of the GNN in learning the representation of job dependencies structure.

Furthermore, the low score of rand index score (ca. 0.36) of identified clusters through DBSCAN and baseline is debatable. The rule-based classification (the baseline methodology) can consider the structural identical job dependencies as recurring jobs, but it fails to find highly structural similar jobs. It is worth mentioning that not all recurring jobs have identical structural job dependencies. This results in the failure of the baseline method to capture all the reoccurring jobs. The baseline method has another disadvantage, which is its disability to capture tasks attributes. More specifically, the rule-based classification considers two jobs with identical job dependencies as recurrent, regardless of the task's attributes. Furthermore, during our analysis, we verify that a significant proportion of job dependencies do not have any parallelization, i.e., each task is dependent singly on the result of the primary task. The rule-based classification considers the jobs of this kind as recurring if they have an equal number of tasks. The assumptions mentioned above result in the inaccurate results of the baseline. Because of the first assumption, the baseline methodology fails to detect all the reoccurring jobs (false negatives), and as a result of the second assumption, some jobs may be labeled as recurring without sufficient reason (false positives). We suggest these assumptions are the major drawback of our baseline. Despite that, our contributors do not rely alone on the structure of job dependencies, as the GNN produces learn to represent the DAGs such that they are distinct regarding the target variables. Thus, clustering the DAGs based on the embeddings outperform the baseline.

Furthermore, during our experiments, we notify that clustering methods, namely OPTICS, and DBSCAN mark a high proportion of the data points as outliers. There is a trade-off between the quantity of clustered samples and the quality of clusters. Because of the nature of these clustering algorithms, i.e., the points should satisfy both the *radius* and $MinPts$ related conditions to be clustered. One can set the parameters so that the models cluster all the data points, but our experiments show that clustering the embeddings in such a way results in inaccurate predictions.

6

State of the Art

The present thesis proposes a novel approach. Thus, to the best of our knowledge, there are no contributions that follow a similar framework's pipeline as ours, i.e., runtime prediction using structural analysis of job dependencies. Therefore, in this chapter, we discuss state-of-the-art contributions similar to different components of our framework: first, we study the techniques aiming to learn the representation of graphs. Second, we discuss the methodologies that address the problem of runtime as well as resource usage prediction in cloud computing domain. Third, we explain the works that proposed techniques to identify recurring jobs. Finally, we name the works that develop and evaluate their techniques using the trace dataset of Alibaba.

6.1 Graph Representation Learning

Recently, graph representation learning has received much attention, as many real-world scenarios are only representable through graphs. Learning the representation of a graph is more difficult in comparison with other data types, e.g., image, audio, etc. One can categorize the graph embedding technique as follows: (1) *dimension reduction based* methods that are classical approaches aiming to reduce the dimensionality of the graph and, at the same time, preserve the desired important characteristics of the graphs. To reduce dimensionality, various linear methods like PCA [75] and Linear discriminant analysis [76] are used. (2) *Random-walk-based* approaches sample a graph by multiple walks starting from random nodes like DeepWalk [21] and node2vec [22]. (3) *Matrix-factorization-based* or Graph Factorization (GF) [77] methods that are the first generation of techniques that could enable node embeddings in $O(|n|)$ time. To this end, GF uses adjacency matrix factorization, preserving the structural information of the graph by a dimensional reduction process. Node Proximity Matrix Factorization (NPMF) [78] and Text-associated DeepWalk [79] are among the variation of this method. (4) *Neural-network-based* methods rely on achievements of Recursive Neural Network (RNN) and Convolutional Neural Network (CNN). Due to the success of these models, the researchers attempt to generalize techniques to enable neural networks to learn the representation of graphs. The input of such models is either the whole graph or the sampled path of the graph. In this thesis, we generate graph embeddings using models of this kind. For instance, in variational graph auto-encoders, the graph is summarized with an encoder mechanism, then the loss is cal-

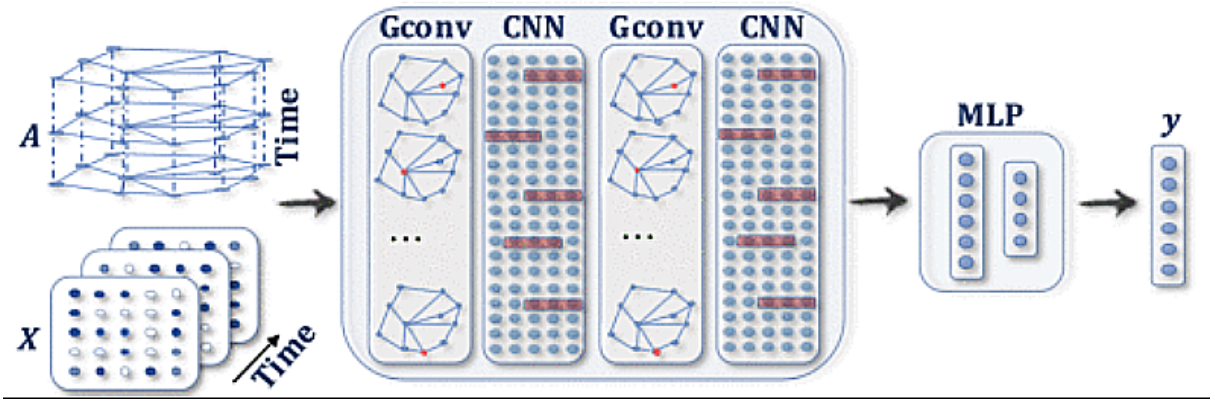


Figure 6.1: The architecture of STGNN for a toy example spatial-temporal graph. The convolutional layer capture the adjacency matrix A and the feature matrix X (spatial analysis). The one-dimensional CNN captures the historical changes of X. The MPL layer represents a multiLayer perceptron. The output layer predicts the feature matrix of the nodes using linear transformations. The figure is taken from the work by Wu et al. [85].

culated by distance of decoded vector with the graph itself [80]. The other example of neural network-based methods is GraphSAGE [81] that iteratively samples and aggregates, aiming to learn the representation of the nodes. In GraphSAGE, the aggregation functions are trained to enable nodes integration while holding the associated attributes. There exist other methods that are adjusted to handle specific kinds of graphs. For example, graph partition neural networks (GPNN) [82] uses the idea behind GNNs to learn the representation of large graphs by alternating between local and global information message passing mechanism. GPNN avoids many deep computations by a randomized flood fill algorithm that selects the nodes with large out-degrees. There exist also methods that are optimized to handle more complicated graphs like social networks, e.g., hyper-graph neural network [83] which uses the idea of spectral convolutional layer on hyper-graphs [84].

In addition to the categories mentioned above, recently Spatial-Temporal GNNs (STGNNs) were developed aiming to analyze the dynamic graphs and capture their dynamicity. The STGNNs techniques capture as input dynamic graphs nodes and, at the same time, consider the interdependency between related nodes. In other words, STGNNs analyze the temporal and spatial connectivities together. Figure 6.1 shows the workflow of STGNN. STGNNs compose of a convolutional layer and a one-dimensional convolutional neural network. The former operates on the adjacency matrix A with the feature matrix of X to capture in the time of t, while the latter captures the attributes' matrix along with the time axis [85]. One can utilize the STGNN techniques to predict the values of a dynamic node [86]. Table 6.1 represents the taxonomy of neural network-based works, along with the related works of each category.

In light of the above explanations, the state-of-the-art graph embedding approaches are neural network-based. One can categorize GNNs into spectral- and spatial-based approaches. In spectral-based techniques, the presumption is that the graphs are undirected. A graph Fourier transformation converts the graph signal into an eigenvectors-based orthonormal space. These approaches use the normalized Laplacian matrix to calculate the eigenvectors in an orthonormal space [104]. An example of spectral-based methods is CayleyNet [105] that enables narrow frequency band capturing using parametric rational complex functions.

In spatial-based approaches, graph convolutions capture the node's spatial relations. Indeed, this approach tries to represent the nodes and the neighbors' information using a message

Table 6.1: Taxonomy and the related publication of GNN's categories according to work from Wu et al. [85].

Category		Contributions
Recurrent GNNs		[87], [88], [89]
Convolutional GNNs	Spectral methods	[90], [91], [92]
	Spatial methods	[93], [94], [95], [96], [97], [27]
Graph Autoencoders	Network Embedding	[98], [99], [100]
	Graph Generation	[101], [102], [103]
Spatial-temporal GNNs		[87], [88], [89]

passing mechanism. The idea of spatial-based GNN was introduced firstly in the neural network for graphs (NN4G) [106], which learns the graph representation using a compositional neural architecture. In this work, the model consists of independent parameters at each layer. One can consider Message Passing Neural Network (MPNN) [107] as a general framework of spatial-based graph embedding. The main difference between these kinds of neural networks is in aggregation and update functions. The work from Kearnes et al. [108] and Schütt et al. [109] proposed a graph embedding mechanism similar to the convolutional layer we used in the present thesis. The former develops a simple encoding mechanism to learn the small undirected graph's representation. In contrast, the latter enables the neural network to learn the representation of graphs, specifically molecular systems, using a statistical partitioning of molecular fingerprints.

6.2 Runtime and Resource Usage Prediction

Effective prediction becomes essential for cloud resource management; there were many efforts to apply prediction techniques and time series to forecast future resource allocation and runtime. These studies investigate different aspects and provide various solutions to improve the accuracy and efficiency of the prediction task. Liang et al. proposed a prediction model base on the fact that the correlated resources can accurately predict the future behavior of resource usage [110]. Taking advantage of time series and considering the correlation of resources could achieve a high prediction accuracy. There were also other contributions like the research by Gupta et al. [111] that impose techniques regarding the nature of cloud platforms. The authors in their study claimed that conventional time series are unsuitable for resource usage prediction. Hence, they propose a statistical approach, specifically multivariate long short-term memory (MLST), to generate accurate forecasts. Moreover, Rahmanian et al., in their paper [112] investigate to solve the problem by considering the nature of Learning Automata (LA) and applying an algorithm that is a combination of several prediction models. Using LA algorithms, it is possible to assign weights for individual constituent models. Although the authors did not group jobs, their proposed technique could outperform other ensemble prediction algorithms.

In addition to the aforementioned efforts to predict resource usage, there are techniques developed to enable runtime prediction. Tsafrir et al. believe that users' estimates are an accurate indicator of jobs' runtime [113]. Thus, they developed a model that predicts the runtime by correlating parallel jobs characteristics and the users' estimates. Gaussier et al. using the classical machine learning methods, develop a loss function that is customized for runtime pre-

diction [114]. The authors relied on three explanatory variables: (1) historical information, e.g., the last job's user runtime. (2) current state of the cluster, e.g., number of running jobs, and (3) job's related description. e.g., amount of resources requested by job. Then, using the variables, a L_2 regularized polynomial model is trained to estimate the incoming job's runtime. Similarly, Rauschmayr presents a framework to do the prediction task using the historical information [115] using a Maximum Likelihood Estimation (MLE). The model is trained to predict both the runtime and memory consumption. The author reported that the model could not achieve promising results for runtime prediction. Chen et al. believe that runtime prediction is a non-linear and complicated problem, as the runtime is affected by multiple factors. Thus, they utilize an ensemble machine learning approach. To get the advantage of multiple voters, the authors used the idea of Gradient Boosting Decision Tree (GBDT), which enables the ensemble learning of multiple regression models [74]. Zhu et al. proposed an adaptive matrix factorization model to perform runtime prediction in Quality of Service (QoS) scenarios [116]. The model in their framework was adjusted through techniques of data transformation and online adaptive weights. The data transformation techniques are applied to stabilize data variance and produce a normal distribution to robust the matrix factorization model. Additionally, to solve the real-world problems characterized by the continuous and incremental data model, the authors employ a stochastic gradient descent approach to update the model [116]. In their approach, unlike our thesis, the authors do not consider the structure of job dependencies as an explanatory variable for runtime prediction.

6.3 Reoccurring Jobs

In addition to approaches related to prediction tasks, there is still a need for grouping recurrent jobs, as the structure of task dependencies of production jobs themselves carry meaningful information and delivers a great impact on the accuracy of predictions. Thamsen et al. propose a system to monitor job execution, model the scalability of jobs using jobs' history, and finally select appropriate resources according to users' requirements [117]. The proposed system predicts runtime using parametric and non-parametric regression, allowing users to allocate capable resources regarding their runtime targets. Venkataraman et al. also consider the similarity of jobs. They argue that even utilizing a limited of jobs' sample make it possible to predict computation and communication of future jobs demand [118]. Through their statistical technique, the authors could train a predictor model that effectively allocate the resources. These works evaluated their system in a dedicated cluster, which is not customarily the case in real-world's scenarios.

6.4 Alibaba Cluster Dataset

Another perspective of this thesis is that we utilize a real-world trace dataset of the popular cluster of Alibaba. There are only limited works that develop and evaluate their approaches based on such the trace dataset. The researcher often relies on traces of a dedicated cluster or defines a method to generate artificial traces. This section briefly describes the works that, similar to ours, use the published Alibaba trace dataset. Tian et al., based on analysis of Alibaba trace dataset [13], report that the artificially generated traces are not capable of reflecting the

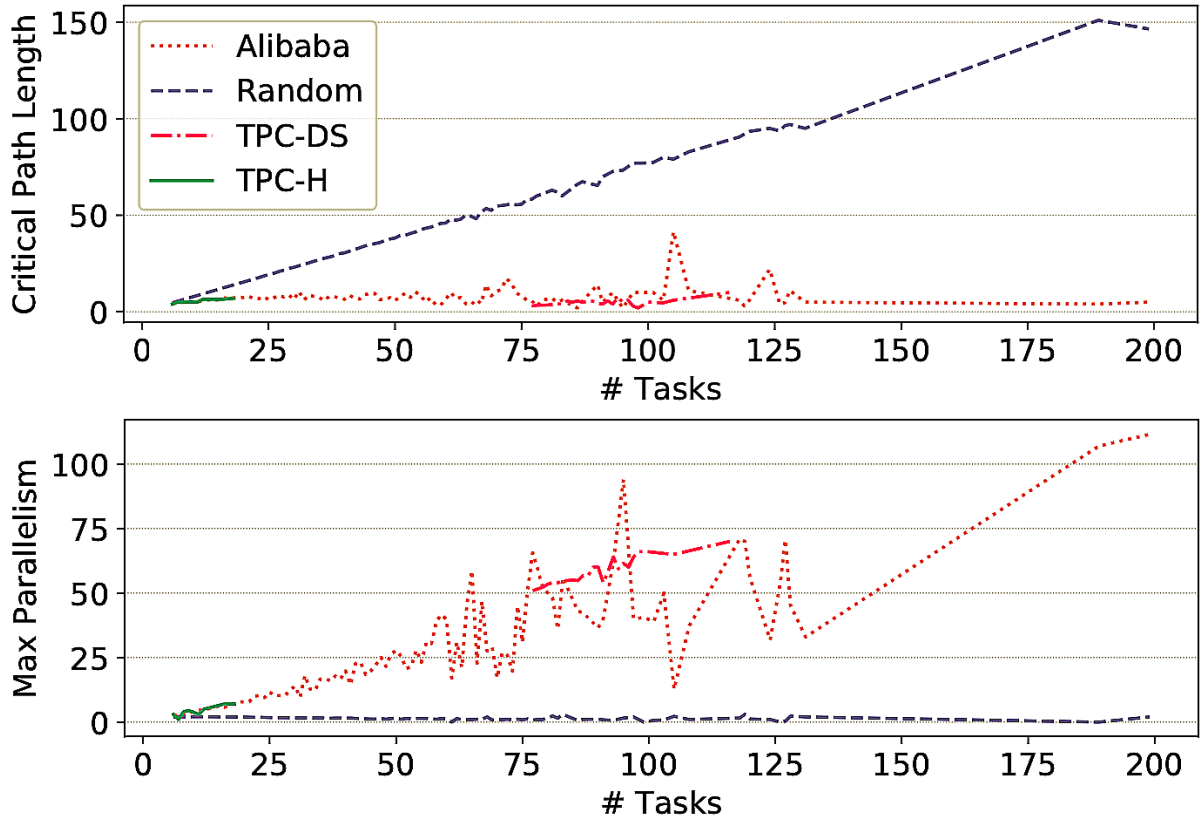


Figure 6.2: The relation of the number of tasks with a critical path length of the jobs and parallelization in four datasets. The random dataset is a dataset of randomly generated DAGs, whereas TPC-DS [119] and TPC-H [120] are standard benchmarks. The former figure illustrates the changes in the length of the longest path of DAGs by the various number of tasks. The latter shows the number of tasks that can be executed in parallel by different numbers of tasks. The figure is taken from the work by Tian et al. [14]

real-world scenarios and, therefore, are not reliable [14]. The figure 6.2 compares two critical characteristics of randomly generated DAGs with the trace dataset of Alibaba. The figure illustrates the relation of the length of the longest path within the DAGs and the number of nodes. The lower figure represents the degree of parallelism of jobs with various task numbers. The figure shows that the randomly generated DAGs are not representative of real-world data-parallel jobs. Guo et al., in their paper, analyze the trace dataset to detect bottlenecks of efficient resource utilization [121]. The authors report three main findings: First, memory is the main bottleneck in the Alibaba cluster. Second, latency-critical applications make batch-processing restricted, and third, Most of the latency-critical applications are written in Java. Li et al. propose DeepJS that is a job scheduling algorithm [122]. The authors first consider the scheduling problem as a bin packing problem. Then they enable DeepJS to calculate fitness to solve the problem of bin packing automatically. The fitness here means a measurement to calculate how a job and its associated machine match to increase the throughput. DeepJS can maximize the throughput based on reinforcement learning techniques. Their proposed technique was developed and evaluated on the Alibaba trace dataset. Wu et al. argue that the long-lived applications, like latency-sensitive web services, become a new bottleneck in clusters [123]. To schedule the cluster with such the applications, the scheduler should run multiple containers of an application on different machines. Thus, global parallelism is required. To

solve this problem, they proposed Aladdin, which is a global scheduler. The Aladdin maximizes resource efficiency and, at the same time, avoids constraint violation. Aladdin evaluated on cluster dataset of Alibaba.

7
Conclusion

Runtime prediction of data-parallel jobs is the key to maintain reliability and efficient utilization of cluster resources. An accurate runtime prediction is beneficial not only for the cluster scheduler but also for the cluster users. Nevertheless, maintaining correct predictions is not straightforward, as execution time is affected by many factors. In this thesis, we aimed to generate precise predictions by proposing a novel contribution that consists of two main stages: first, identifying and grouping reoccurring data-parallel jobs, and second, utilizing the common characteristics of groups to predict runtime. The real-world trace dataset of Alibaba provides the opportunity to research empirically and evaluate our contribution based on traces with low entropy.

To identify reoccurring jobs, we perform a cutting-edge solution technique. Indeed, we suggest that the structure of parallelization is a reliable indicator of reoccurring and similar jobs. This leads us to use these meaningful dependencies to group the jobs. Our experiments show that the best means to achieve this goal is converting the arbitrary structure of job dependencies into a fixed vector in Euclidean space with the help of state-of-the-art achievements in deep learning techniques. We developed a graph convolutional neural network that was highly capable of learning the representation of graphs. The evaluation results of the GNN model confirm its capability in representing the unseen samples correctly. Despite the achievements of the graph embedding phase, we believe the proposed GNN is not globally optimized. This is because multiple hyperparameters affect the quality of embeddings. Further work needs to be carried out to establish whether there exists a more calibrated approach to produce job dependencies embedding.

Our steps proceed by operating multiple clustering techniques to form the job's groups based on their representation, namely, K-means, OPTICS, and DBSCAN. We perform multiple clustering techniques to reach multiple goals. e.g., we used K-means to evaluate the quality of embeddings and DBSCAN to identify the clusters of reoccurring jobs for final prediction. The similarity of jobs within the groups, especially the identified groups through K-means, confirmed that the structure of parallelization is able to demonstrate clusters of similar jobs. Indeed, our experiment indicates that the detected clusters through our methodology have more similarities in terms of runtime than the recurring jobs identified by baseline. Nevertheless, our work in this phase had a limitation: the unavailability of ground truth. This limited us to

use unsupervised techniques alone. We strongly suggest that availability of a subset of labeled samples can positively adjust the clustering phase. Thus, we are confident that the proposed approach serves as a base for further studies with a labeled dataset.

Finally, we evaluate if the identified clusters have predictive value. To achieve this goal, we predicted the runtime of the last recent jobs of every cluster of reoccurring jobs. Besides our contribution, we redo the approach of Tian et al. to define our baseline [14]. The baseline specifies their approach to identify the recurring jobs based on two main rules. The first rule considers the parallelization structure, and the second rule filters the jobs that begin in the periodic time intervals. After comparing the baseline with our contribution, the results show that we could improve the MAEs by ca 27%. In addition to our promising results in terms of runtime prediction, we suggest that detected clusters can be beneficial in other aspects, e.g., resource usage prediction, and optimizing resource utilization in clusters.

Acronyms

PyG *PyTorch Geometric*

DAG *Directed Acyclic Graph*

GNN *Graph Neural Network*

GD *Gradient Descent*

SGD *Stochastic Gradient Descent*

DBSCAN *Density-Based Spatial Clustering of Applications with Noise*

OPTICS *Ordering Points To Identify the Clustering Structure*

MAE *Mean Absolute Error*

MSE *Mean Squared Error*

GPL *Global Pooling Layer*

CSV *Comma Separated Value*

STD *Standard Deviation*

GF *Graph Factorization*

NPMF *Node Proximity Matrix Factorization*

RNN *Recursive Neural Network*

CNN *Convolutional Neural Network*

MPNN *Message Passing Neural Network*

QoS *Quality of Service*

SPGNN *Spatial-Temporal Graph Neural Network*

Supplementary Material

All the milestones of the thesis, including data integration, graph embedding, clustering, and prediction, are developed in *Jupyter Notebook* environment [124]. Jupyter Notebook is a web-based and open-source application that enables creating and sharing of live codes and visualization. The programming language to develop all components of this thesis is Python in version 3.7.1 [125]. Additionally, we used multiple toolkits and libraries to manipulate the data and create a machine learning and statistical models. The table 7.1 provide the list of libraries and tools we utilized in the thesis. The main libraries of the thesis are PyTorch-geometric [51], and Scikit [64]. The former provides multiple libraries to create GNNs, and the latter is a collection of machine learning algorithms. The codes are executed in the CPU model of Quadcore Intel Xeon E3-1230 V2 3.30GHz and 16 GB of RAM. To robust the speed of the GNN training, we utilized Nvidia CUDA in version 10.2.

7.1 Provision of the Code

The codes and the methodology of generating all the models and statistical analysis are available in `https://git.tu-berlin.de/aala15/reoccurring-jobs_detection`. The codes in this repository should be considered as proof of the concepts of the thesis. Furthermore, the codes represent our contribution to evaluating the results of models. The repository also provides the prerequisite libraries and tools to reproduce the reported results.

Table 7.1: List of libraries and tools which used in the thesis

Tool	**Description**	
Jupyter Notebook	Web-based application for editing and running documents and codes	[124]
Python (versoin 3)	Object-oriented programming language with dynamic semantics	[125]
Pandas	Analysis and manipulation tools	[67]
Numpy	Manipulation of multidimensional arrays and mathematical functions	[68]
Matplotlib	Visualizing tools	[70]
Seaborn	Visualizing tools	[126]
Igraph	Network analysis tools	[72]
NetworkX	Network analysis tools	[50]
PyTorch	Open source machine learning library based on Torch	[127]
PyTorch Geometric	Tools for writing and training Graph Neural Networks (GNNs)	[51]
PyTorch Metric Learning	Implementation of various deep metric learning	[61]
Scikit	Various data analysis and machine learning tools	[64]
Scipy	Scientific and technical computing tools built on top of Python	[128]
Git	Version tracking tool	[129]

7.2 Provision of the Datasets

The necessary datasets to reproduce the results are available in `https://tubcloud.tu-berlin.de/s/YQffxrJXjmCM8P8`. All the intermediate datasets are generated through the published trace dataset of Alibaba [13].

Bibliography

[1] Dan C Marinescu. "Chapter 6–Cloud Resource Management and Scheduling". In: *Cloud Computing Theory and Practice* (2013), pp. 163–203.

[2] Mina Naghshnejad and Mukesh Singhal. "Adaptive online runtime prediction to improve HPC applications latency in cloud". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 762–769.

[3] Robert Heinrich et al. "Integrating run-time observations and design component models for cloud system analysis". In: (2014).

[4] Andrew D. Ferguson et al. "Jockey: guaranteed job latency in data parallel clusters". In: *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. Ed. by Pascal Felber, Frank Bellosa, and Herbert Bos. ACM, 2012, pp. 99–112.

[5] Robert Grandl et al. "Altruistic Scheduling in Multi-Resource Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 65–80.

[6] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 117–134.

[7] Nicolas Bruno et al. "Recurring job optimization in scope". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 805–806.

[8] John Wilkes. *Google cluster-usage traces v3*. Technical Report. Posted at `https://github.com/google/cluster-data/blob/master/ClusterData2019.md`. Mountain View, CA, USA: Google Inc., Apr. 2020.

[9] Kai Ran. *OpenCloud Hadoop cluster trace : format and schema*. Apr. 2020. URL: `http://ftp.pdl.cmu.edu/pub/datasets/hla`.

[10] Chen Chen, Wei Wang, and Bo Li. "Performance-aware fair scheduling: Exploiting demand elasticity of data analytics jobs". In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 504–512.

[11] Yang Liu, Huanle Xu, and Wing Cheong Lau. "Online job scheduling with resource packing on a cluster of heterogeneous servers". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1441–1449.

[12] Xiaoda Zhang et al. “COBRA: Toward provably efficient semi-clairvoyant scheduling in data analytics systems”. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 513–521.

[13] Alibaba. *alibaba/clusterdata: cluster data collected from production clusters in Alibaba for cluster management research*. Apr. 2020. URL: `https://github.com/alibaba/clusterdata`.

[14] Huangshi Tian, Yunchuan Zheng, and Wei Wang. “Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud”. In: *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 139–151.

[15] S Selvarani and G Sudha Sadhasivam. “Improved cost-based algorithm for task scheduling in cloud computing”. In: *2010 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE. 2010, pp. 1–5.

[16] Gopika Venu and KS Vijayanand. “Task Scheduling in Cloud Computing: A Survey”. In: 8 (2020).

[17] Ahmed A Hamed and Rabah A Ahmed. “AVAILABILITY EVALUATION OF DIFFERENT PLANNING AND SCHEDULING ALGORITHMS IN HYBRID CLOUD SYSTEM”. In: *Iraqi Journal of Information & Communications Technology* 3.4 (2020), pp. 47–59.

[18] Béla Bollobás. *Modern graph theory*. Vol. 184. Springer Science & Business Media, 2013.

[19] Stanley Wasserman, Katherine Faust, et al. “Social network analysis: Methods and applications”. In: (1994).

[20] Si Zhang et al. “Graph convolutional networks: a comprehensive review”. In: *Computational Social Networks* 6.1 (2019), pp. 1–23.

[21] 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*. 2014, pp. 701–710.

[22] 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*. 2016, pp. 855–864.

[23] Sergi Abadal et al. “Computing graph neural networks: A survey from algorithms to accelerators”. In: *arXiv preprint arXiv:2010.00130* (2020).

[24] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[25] Franco Scarselli et al. “The graph neural network model”. In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[26] William L Hamilton, Rex Ying, and Jure Leskovec. “Representation learning on graphs: Methods and applications”. In: *arXiv preprint arXiv:1709.05584* (2017).

[27] David Duvenaud et al. “Convolutional networks on graphs for learning molecular fingerprints”. In: *arXiv preprint arXiv:1509.09292* (2015).

[28] Harry L Morgan. "The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service." In: *Journal of Chemical Documentation* 5.2 (1965), pp. 107–113.

[29] Jack Kiefer and Jacob Wolfowitz. "Stochastic estimation of the maximum of a regression function". In: *The Annals of Mathematical Statistics* (1952), pp. 462–466.

[30] Chigozie Nwankpa et al. "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *CoRR* abs/1811.03378 (2018).

[31] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[32] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[33] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[34] Andrey Nikolayevich Tikhonov. "On the stability of inverse problems". In: *Dokl. Akad. Nauk SSSR*. Vol. 39. 1943, pp. 195–198.

[35] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[36] Osama Abu Abbas. "Comparisons between data clustering algorithms." In: *International Arab Journal of Information Technology (IAJIT)* 5.3 (2008).

[37] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[38] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[39] Mihael Ankerst et al. "OPTICS: Ordering points to identify the clustering structure". In: *ACM Sigmod record* 28.2 (1999), pp. 49–60.

[40] Anthony K. H. Tung. "Rule-based Classification". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 2459–2462. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_559. URL: https://doi.org/10.1007/978-0-387-39940-9_559.

[41] Amit Saxena et al. "A review of clustering techniques and developments". In: *Neurocomputing* 267 (2017), pp. 664–681.

[42] T. Soni Madhulatha. "An Overview on Clustering Methods". In: *CoRR* abs/1205.1117 (2012). URL: http://arxiv.org/abs/1205.1117.

[43] Andrew Rosenberg and Julia Hirschberg. "V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure". In: *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*. Ed. by Jason Eisner. ACL, 2007, pp. 410–420.

[44] William M Rand. “Objective criteria for the evaluation of clustering methods”. In: *Journal of the American Statistical association* 66.336 (1971), pp. 846–850.

[45] Lawrence Hubert and Phipps Arabie. “Comparing partitions”. In: *Journal of classification* 2.1 (1985), pp. 193–218.

[46] Francis Galton. “Regression towards mediocrity in hereditary stature.” In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 15 (1886), pp. 246–263.

[47] Cort J Willmott and Kenji Matsuura. “Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance”. In: *Climate research* 30.1 (2005), pp. 79–82.

[48] Robert L Thorndike. “Who belongs in the family?” In: *Psychometrika* 18.4 (1953), pp. 267–276.

[49] Robert W Robinson. “Counting unlabeled acyclic digraphs”. In: *Combinatorial mathematics V*. Springer, 1977, pp. 28–43.

[50] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[51] Matthias Fey and Jan Eric Lenssen. “Fast graph representation learning with PyTorch Geometric”. In: *arXiv preprint arXiv:1903.02428* (2019).

[52] Svante Wold, Kim Esbensen, and Paul Geladi. “Principal component analysis”. In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

[53] Matthias Feurer et al. “Auto-sklearn: efficient and robust automated machine learning”. In: *Automated Machine Learning*. Springer, Cham, 2019, pp. 113–134.

[54] Isabelle Guyon et al. “Gene selection for cancer classification using support vector machines”. In: *Machine learning* 46.1 (2002), pp. 389–422.

[55] George AF Seber and Alan J Lee. *Linear regression analysis*. Vol. 329. John Wiley & Sons, 2012.

[56] Jie Zhou et al. “Graph neural networks: A review of methods and applications”. In: *AI Open* 1 (2020), pp. 57–81.

[57] Qimai Li, Zhichao Han, and Xiao-Ming Wu. “Deeper insights into graph convolutional networks for semi-supervised learning”. In: *Thirty-Second AAAI conference on artificial intelligence*. 2018.

[58] Chen Cai and Yusu Wang. “A note on over-smoothing for graph neural networks”. In: *arXiv preprint arXiv:2006.13318* (2020).

[59] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[60] Rich Caruana, Steve Lawrence, and Lee Giles. “Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping”. In: *Advances in neural information processing systems* (2001), pp. 402–408.

[61] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. “Pytorch metric learning”. In: *arXiv preprint arXiv:2008.09164* (2020).

[62] Xun Wang et al. “Multi-similarity loss with general pair weighting for deep metric learning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 5022–5030.

[63] Deepali Virmani, Shweta Taneja, and Geetika Malhotra. “Normalization based K means Clustering Algorithm”. In: *arXiv preprint arXiv:1503.00900* (2015).

[64] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[65] Erich Schubert et al. “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN”. In: *ACM Transactions on Database Systems (TODS)* 42.3 (2017), pp. 1–21.

[66] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[67] Wes McKinney et al. “Data structures for statistical computing in python”. In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[68] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[69] Emanuel Parzen. “On estimation of a probability density function and mode”. In: *The annals of mathematical statistics* 33.3 (1962), pp. 1065–1076.

[70] John D Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in science & engineering* 9.03 (2007), pp. 90–95.

[71] Tommi Junttila and Petteri Kaski. “Engineering an efficient canonical labeling tool for large and sparse graphs”. In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 135–149.

[72] Gabor Csardi, Tamas Nepusz, et al. “The igraph software package for complex network research”. In: *InterJournal, complex systems* 1695.5 (2006), pp. 1–9.

[73] Sangeetha Abdu Jyothi et al. “Morpheus: Towards automated slos for enterprise clusters”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 117–134.

[74] Xiaomeng Chen et al. “Runtime prediction of high-performance computing jobs based on ensemble learning”. In: *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*. 2020, pp. 56–62.

[75] Shinji Umeyama. “An eigendecomposition approach to weighted graph matching problems”. In: *IEEE transactions on pattern analysis and machine intelligence* 10.5 (1988), pp. 695–703.

[76] Jieping Ye, Ravi Janardan, and Qi Li. “Two-dimensional linear discriminant analysis”. In: *Advances in neural information processing systems* 17 (2004), pp. 1569–1576.

[77] Amr Ahmed et al. “Distributed large-scale natural graph factorization”. In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 37–48.

[78] Ajit P Singh and Geoffrey J Gordon. “Relational learning via collective matrix factorization”. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008, pp. 650–658.

[79] Cheng Yang et al. “Network representation learning with rich text information”. In: *Twenty-fourth international joint conference on artificial intelligence*. 2015.

[80] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[81] William L Hamilton, Rex Ying, and Jure Leskovec. “Inductive representation learning on large graphs”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 1025–1035.

[82] Alison A Motsinger et al. “GPNN: Power studies and applications of a neural network method for detecting gene-gene interactions in studies of human disease”. In: *BMC bioinformatics* 7.1 (2006), pp. 1–10.

[83] Yifan Feng et al. “Hypergraph neural networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 3558–3565.

[84] Fenxiao Chen et al. “Graph Representation Learning: A Survey”. In: *CoRR* abs/1909.00958 (2019). arXiv: 1909.00958. URL: http://arxiv.org/abs/1909.00958.

[85] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Trans. Neural Networks Learn. Syst.* 32.1 (2021), pp. 4–24.

[86] Bing Yu, Haoteng Yin, and Zhanxing Zhu. “Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 3634–3640.

[87] Youngjoo Seo et al. “Structured Sequence Modeling with Graph Convolutional Recurrent Networks”. In: *Neural Information Processing - 25th International Conference, ICONIP 2018, Siem Reap, Cambodia, December 13-16, 2018, Proceedings, Part I*. Ed. by Long Cheng, Andrew Chi-Sing Leung, and Seiichi Ozawa. Vol. 11301. Lecture Notes in Computer Science. Springer, 2018, pp. 362–373.

[88] Ashesh Jain et al. “Structural-RNN: Deep Learning on Spatio-Temporal Graphs”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 5308–5317.

[89] Zonghan Wu et al. “Graph WaveNet for Deep Spatial-Temporal Graph Modeling”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 1907–1913.

[90] Joan Bruna et al. “Spectral Networks and Locally Connected Networks on Graphs”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014.

[91] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering”. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, pp. 3837–3845.

[92] Ruoyu Li et al. "Adaptive Graph Convolutional Neural Networks". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 3546–3553.

[93] Alessio Micheli. "Neural Network for Graphs: A Contextual Constructive Approach". In: *IEEE Trans. Neural Networks* 20.3 (2009), pp. 498–511. DOI: 10.1109/TNN.2008.2010350. URL: https://doi.org/10.1109/TNN.2008.2010350.

[94] James Atwood and Don Towsley. "Diffusion-Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, pp. 1993–2001.

[95] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1263–1272.

[96] William L. Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 1024–1034.

[97] Petar Velickovic et al. "Deep Graph Infomax". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[98] Shaosheng Cao, Wei Lu, and Qiongkai Xu. "Deep Neural Networks for Learning Graph Representations". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 1145–1152.

[99] Ke Tu et al. "Deep Recursive Network Embedding with Regular Equivalence". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. Ed. by Yike Guo and Faisal Farooq. ACM, 2018, pp. 2357–2366.

[100] Wenchao Yu et al. "Learning Deep Network Representations with Adversarially Regularized Autoencoders". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. Ed. by Yike Guo and Faisal Farooq. ACM, 2018, pp. 2663–2671.

[101] Martin Simonovsky and Nikos Komodakis. "GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders". In: *Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I*. Ed. by Vera Kurková et al. Vol. 11139. Lecture Notes in Computer Science. Springer, 2018, pp. 412–422.

[102] Tengfei Ma, Jie Chen, and Cao Xiao. "Constrained Generation of Semantically Valid Graphs via Regularizing Variational Autoencoders". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 7113–7124.

[103] Aleksandar Bojchevski et al. "NetGAN: Generating Graphs via Random Walks". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 609–618.

[104] David I Shuman et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains". In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[105] Ron Levie et al. "Cayleynets: Graph convolutional neural networks with complex rational spectral filters". In: *IEEE Transactions on Signal Processing* 67.1 (2018), pp. 97–109.

[106] Alessio Micheli. "Neural Network for Graphs: A Contextual Constructive Approach". In: *IEEE Trans. Neural Networks* 20.3 (2009), pp. 498–511.

[107] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1263–1272.

[108] Steven M. Kearnes et al. "Molecular graph convolutions: moving beyond fingerprints". In: *J. Comput. Aided Mol. Des.* 30.8 (2016), pp. 595–608.

[109] Kristof T Schütt et al. "Quantum-chemical insights from deep tensor neural networks". In: *Nature communications* 8.1 (2017), pp. 1–8.

[110] Jin Liang, Klara Nahrstedt, and Yuanyuan Zhou. "Adaptive multi-resource prediction in distributed resource sharing environment". In: *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*. IEEE Computer Society, 2004, pp. 293–300.

[111] Shaifu Gupta and Dileep Aroor Dinesh. "Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks". In: *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems, ANTS 2017, Bhubaneswar, India, December 17-20, 2017*. IEEE, 2017, pp. 1–6.

[112] Ali Asghar Rahmanian, Mostafa Ghobaei-Arani, and Sajjad Tofighy. "A learning automata-based ensemble resource usage prediction algorithm for cloud computing environment". In: *Future Gener. Comput. Syst.* 79 (2018), pp. 54–71.

[113] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. "Modeling User Runtime Estimates". In: *Job Scheduling Strategies for Parallel Processing, 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers*. Ed. by Dror G. Feitelson et al. Vol. 3834. Lecture Notes in Computer Science. Springer, 2005, pp. 1–35.

[114] Éric Gaussier et al. “Improving backfilling by using machine learning to predict running times”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. Ed. by Jackie Kern and Jeffrey S. Vetter. ACM, 2015, 64:1–64:10.

[115] Nathalie Rauschmayr. “A History-based Estimation for LHCb job requirements”. In: *Journal of Physics: Conference Series*. Vol. 664. 6. IOP Publishing. 2015, p. 062050.

[116] Jieming Zhu et al. “Online QoS Prediction for Runtime Service Adaptation via Adaptive Matrix Factorization”. In: *IEEE Trans. Parallel Distributed Syst.* 28.10 (2017), pp. 2911–2924.

[117] Lauritz Thamsen et al. “Selecting resources for distributed dataflow systems according to runtime targets”. In: *35th IEEE International Performance Computing and Communications Conference, IPCCC 2016, Las Vegas, NV, USA, December 9-11, 2016*. IEEE Computer Society, 2016, pp. 1–8.

[118] Shivaram Venkataraman et al. “Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics”. In: *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. Ed. by Katerina J. Argyraki and Rebecca Isaacs. USENIX Association, 2016, pp. 363–378.

[119] Meikel Pöss et al. “TPC-DS, taking decision support benchmarking to the next level”. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*. Ed. by Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki. ACM, 2002, pp. 582–587.

[120] Meikel Poess and Chris Floyd. “New TPC benchmarks for decision support and web commerce”. In: *ACM Sigmod Record* 29.4 (2000), pp. 64–71.

[121] Jing Guo et al. “Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces”. In: *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE. 2019, pp. 1–10.

[122] Fengcun Li and Bo Hu. “Deepjs: Job scheduling based on deep reinforcement learning in cloud data center”. In: *Proceedings of the 2019 4th international conference on big data and computing*. 2019, pp. 48–53.

[123] Heng Wu et al. “Aladdin: optimized maximum flow management for shared production clusters”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 696–707.

[124] Thomas Kluyver et al. “Jupyter Notebooks – a publishing format for reproducible computational workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[125] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[126] Michael Waskom et al. *mwaskom/seaborn: v0.8.1 (September 2017)*. Version v0.8.1. Sept. 2017. DOI: 10.5281/zenodo.883859. URL: https://doi.org/10.5281/zenodo.883859.

[127] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[128] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* (2020).

[129] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.