Predictive Business Process Monitoring using Graph Neural Networks

Phuc Tran

Hamburg University of Applied Sciences, Berliner Tor 7, 20099 Hamburg, Germany
thienphuc.tran@haw-hamburg.de

Abstract. Deep learning methods have gained attention in predictive process mining tasks. The most popular architectures include Recurrent Neural Networks and Convolutional Neural Networks. Inspired by the fact that Graph Neural Networks can operate directly on graph data, this work proposes a novel predictive business process monitoring method using Graph Neural Networks to learn the underlying graph representation of business processes and make predictions about the next activity of running cases. The embeddings for the events and their attributes can be learned in an end-to-end manner. Moreover, the proposed method employs a similarity function to guide the learning process and reduce overfitting. The model was evaluated using the BPIC'12 dataset and outperformed state-of-the-art methods by a large margin.

Keywords: Predictive Business Process Monitoring · Graph Neural Networks.

1 Introduction

In the era of information technology, organizations nowadays are becoming increasingly aware of the business value hidden in event logs. Process mining is a discipline that discovers and monitors business processes by analyzing event logs. Process mining allows to extracting knowledge from event logs and understand the happenings around the business processes.

Predictive analytics is gaining momentum in process mining. While other subfields examine event logs to find potential improvements in a post-mortem manner, predictive business process monitoring makes predictions about how a running process will unfold, and thus, helps to improve the decision-making process and mitigate risks.

Some predictive monitoring approaches require an explicit process model such as a Petri Net [3], or a Probabilistic Finite Automaton [9, 10]. Meanwhile, other approaches extract features from the data and learn to make predictions without an explicit process model. These approaches include Support Vector Machine [4], Clustering Analysis [5], and Deep Learning. Over the past five years, Deep Learning methods have gained more popularity in the field of predictive monitoring. Due to the sequential nature of event logs, Convolutional Neural Networks (CNN) [8] and Recurrent Neural Networks (RNN) [1, 2] have been widely applied and have successfully achieved state-of-the-art results. However, CNNs and RNNs often require the variable-sized input to be padded to the same length, which may have negative impacts on the performance.

This work proposes a novel predictive monitoring method using Graph Neural Networks (GNN) to predict the next activity of a running process, which accepts input of arbitrary length thanks to the nature of GNN. One of the major challenges is to construct a meaningful graph representation for the trace of a running case. The graph representation should be able to preserve not only the process flow characteristics (e.g. cycles, self-loops) but also the attributes of individual events. In addition, case-specific attributes may have significant influences on the process. Therefore, these attributes should be incorporated into the model. In order to evaluate the effectiveness of the proposed method, the proposed method is compared to state-of-the-art Deep Learning-based methods using the dataset of the Business Process Intelligence Challenge 2012 (BPIC'12)[12].

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 gives a brief introduction to Graph Neural Networks. The proposed approach is introduced in Section 4. Section 5 describes the experiments and the evaluation results. Finally, Section 6 concludes the paper and discusses the potential for future work.

2 Related work

Weber et al. [10] applied Probabilistic Finite Automaton (PFA) to the α-algorithm. Breuker et al. [9] proposed RegPFA, a grammatical inference method to predict the next event based on PFA and Bayesian Regularization.

Evermann et al. used a recurrent model with Long Short-Term Memory (LSTM) cells. Analogous to many-to-many sequence prediction in Natural Language Processing, this approach generates embeddings for the events and uses two LSTM cells to predict the next event of a running case. Tax et al. proposed a similar architecture with improvements in feature engineering so that the architecture can be applied for other related tasks. The main idea is to extract the features that can be consumed by multiple classification/regression heads.

Pasquadibisceglie et al. constructed spatial data by mapping the traces into 2D image-like data structures, which is used to train a CNN model. This method achieved slightly better results than the RNN methods.

Mehdiyev et al. [11] used Autoencoders to learn the representation of the data. The encoders are forced to compress the input into a lower-dimensional representation, while the decoders try to reconstruct the input based on the learned representation. This method reduces the dimensionality of the input. However, this method requires the autoencoders to be trained separately.

Esser and Fahland [6] discussed the potential of modeling event logs using the graph data structure. Most importantly, it was showed that the graph data model enables process mining over multiple inter-related entities. Venugopal [7], inspired thereby, proposed a Deep Learning model using GNN to predict the next event. However, the method operates on a subgraph of the Directly-Follows

Graph (DFG) generated based on the training set, resulting in several disadvantages. The size of the input matrix does not depends on a particular trace, but on the whole dataset, which causes poor scalability. If an event occurred more than once, the input only records the latest event. Therefore, the process flow characteristics of a particular trace can hardly be learned.

To overcome this problem, this work aims to develop a GNN-based method that constructs the input graph based on individual traces and preserves the process flow characteristics. As mentioned in Section 1, the proposed method considers not only the events and their properties but also other properties of the running case that have influences on the result.

3 Background

This section introduces the fundamentals of Graph Neural Networks in general, and Graph Convolutional Networks (GCN) in particular.

3.1 Graph Neural Networks

Common Deep Learning approaches have been successful on working with Euclidean data, i.e., data with a fixed structure such as text, speech, or images. These approaches, however, struggle when dealing with non-Euclidean data [13]. Graphs are one of the most popular non-Euclidean data structures. A graph consists of vertices (or nodes), and edges that describe the connectivity between the nodes. In order to work with graph data, common Deep Learning approaches require a pre-processing step to project the input graphs into an Euclidean space. This pre-processing step is lossy because the structural information are generally lost.

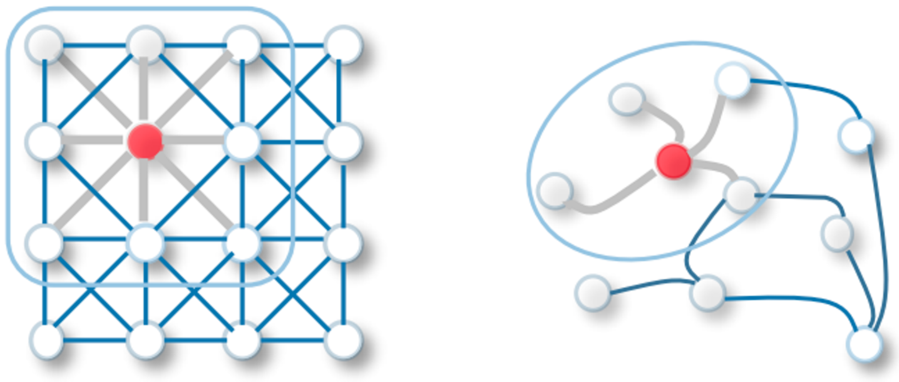


Fig. 1. The similarity between 2D convolution and graph convolution [15].

Graph Neural Network (GNN) is a type of neural network in Geometric Deep Learning that can operate directly on graph data without projecting them back into an Euclidean space. Similar to Convolutional Neural Network, GNN has relational inductive bias [14] since relationships among entities serve as a guidance for the learning process. GNN exploits the local connectivity and extracts meaningful features by performing graph convolutions. Figure 1 shows the similarity between 2D convolution and graph convolution. 2D convolution can be de facto considered as a special case of graph convolution.

Various architectures for GNN have been proposed over the past decade and most of them share the same underlying concept. Therefore, Gilmer et al. [16] proposed a framework called Message Passing Neural Network (MPNN) to unify the diversity of GNNs.

Consider a graph $G = (V, E)$, where V denotes the set of nodes and E denotes the set of edges. $v_i \in V$ is a node and e_{ij} is an edge between v_i and v_j. h_i^t is the feature vector of v_i at time step t. MPNN consists of two phases, namely message passing and readout.

Message Passing (Aggregation) After time step t, each node v_i aggregates the feature vector of its direct neighbors (and itself) to create an "aggregated message" m_i^{t+1} by applying a message function M_t. The new feature vector (or hidden state) of the node h_i^{t+1} is calculated based on m_i^{t+1} and the previous state h_i^t using an update function U_t. This phase can be summerized by the following equations, where $N(i)$ denotes the set of neighboring nodes of v_i:

$$\begin{aligned} m_i^{t+1} &= \sum_{j \in N(i)} M_t(h_i^t, h_j^t, k_{ij}) \\ h_i^{t+1} &= U_t(h_i^t, m_i^{t+1}) \end{aligned}$$

Readout After the message passing phase, the hidden state of the nodes can be used for the downstream tasks. In the case of graph-level tasks, the readout function performs a kind of pooling operation to create a single feature vector for the entire graph.

3.2 Graph Convolutional Networks

The most popular architecture is Graph Convolutional Network (GCN) [17] thanks to its simplicity and effectiveness. Instead of iterating over nodes and performing aggregations, GCN utilizes graph signal processing techniques and symmetric normalization to perform a fast approximation.

Let D and A denote the degree matrix and the adjacency matrix, respectively. $\tilde{A} = A + I$ is the new adjacency matrix with self-loops and $\tilde{D}$ is the corresponding degree matrix for $\tilde{A}$. The next hidden state for every node in the graph can be computed using a single equation:

$$H^{t+1} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^tW^t\right)$$

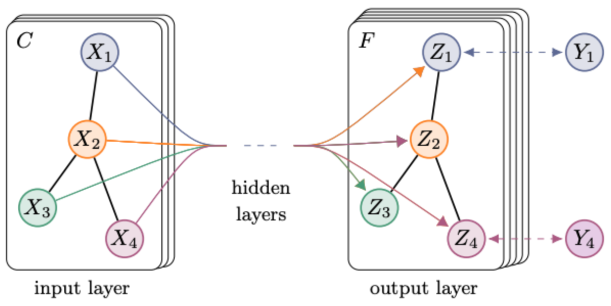


Fig. 2. An illustration of Graph Convolutional Network

where σ denotes the non-linearity function such as ReLU, LeakyReLU and W denote the learnable weight matrix.

3.3 Mini-batching

Due to the nature of GNNs, they cannot work with mini-batches in a new dimension like common neural networks. A mini-batch for GNNs is created by merging all input graphs into one large graph, wherein the subgraphs are not connected to each other. Mathematically, the mini-batching process builds a block diagonal sparse matrix from the adjacency matrices and stacks the feature matrices on top of each other.

$$A = \begin{bmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_n \end{bmatrix}, X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}$$

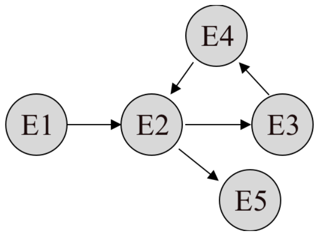
4 Method

This section describes the proposed method and discusses the decisions that were made in the development process.

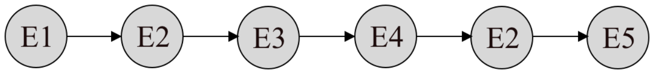
4.1 Input

A good input graph representation is one of the crucial factors for the success of GNN. Figure 3 illustrates the two extremes the author considered when designing the input.

The first extreme (see Figure 3a) is to fully utilize the graph topology to describe event logs. The main advantage of this approach is that the process flow characteristics such as self-loops or cycles are preserved. However, when



(a) The graph representation that prioritizes the process flow characteristics



(b) The graph representation that prioritizes the event features.

Fig. 3. The illustration of the two extremes in input graph construction

the same event type occurred twice but with different attributes, this approach cannot record the attributes of both occurrences. If a transition happens twice or more, it can only be counted as once.

Figure 3b shows the second representation that models event logs as sequences. This approach can record all the event attributes but fails to preserve the topology, which also discards the major advantage of GNN.

Inspired by Esser and Fahland [6], this work models event logs similarly to the approach of graph databases. An input graph in this work consists of two node types, namely type nodes, and attribute nodes. The type nodes represent event types and are connected according to the process flow, resulting in a graph representation as shown in Figure 3a. The attribute nodes represent the event attributes and are connected to the corresponding type node; each attribute node also represents an event occurrence. Figure 4 illustrates the graph representation used in this work.

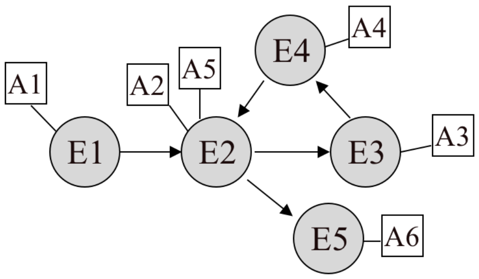


Fig. 4. The graph representation used in this work consists of two node types, namely attribute nodes and type nodes.

4.2 Model architecture

The input graphs are heterogeneous since there are two different types of nodes. For the sake of simplicity, this work utilizes embeddings for both types of nodes, projecting them to a higher-dimensional space, so that homogeneous graph neural network operators like GCN can be applied.

The meaning of the projection for type nodes is different from that for attribute nodes. Therefore, two different embeddings are used, although the resulting vectors have the same number of dimensions. Another possible approach is to separate the node types and utilize heterogeneous graph neural network operators.

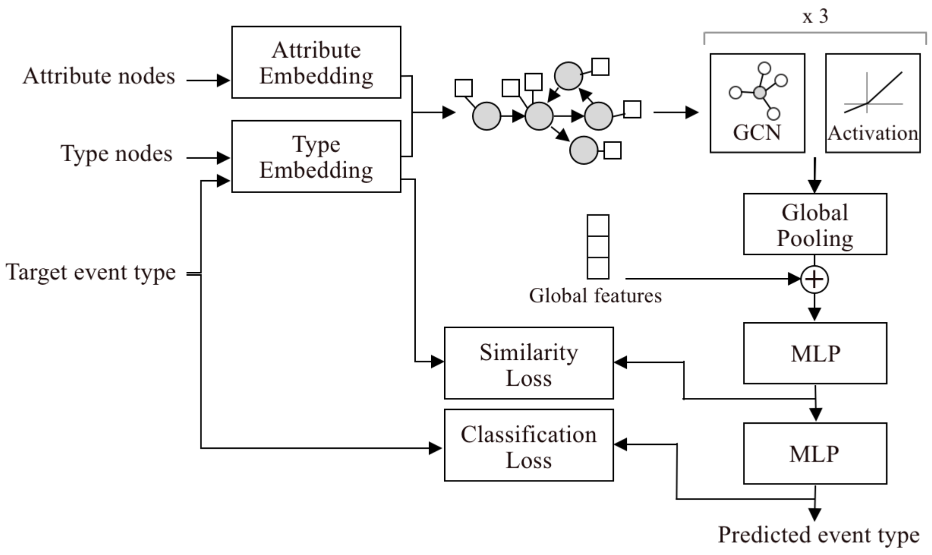


Fig. 5. The overview of the model architecture and the training procedure.

Multiple GCN layers extract high-level features of the input graph. A global pooling is performed to create a single feature vector that represents the entire graph (readout). The vector for global features is concatenated with the former. Afterward, a multi-layer perceptron (MLP) processes the concatenated vector and returns the embedding for the next event type as output. The second MLP block takes the generated embedding and returns the probability distribution of the possible next event types (using softmax).

The generated embedding is compared to the embedding of the ground truth using a similarity function, and the probabilities in the output are used to compute the classification loss. This work uses the cosine similarity function and categorical cross entropy function. Formally, the loss function can be described as follows:

$$L_{sim} = -\frac{\xi(y) \cdot \widehat{\xi(y)}}{|\xi(y)||\widehat{\xi(y)}|}$$

$$L_{class} = -\sum_{i}^{C} y_i \cdot log \hat{y}_i$$

$$L = L_{sim} + L_{class}$$

One can argue that using only the similarity function is sufficient since one can compute the similarity between all possible next events and the output. However, this approach requires a well-trained embedding to work efficiently. This work solely uses the cosine similarity function as additional guidance for the learning process. The guidance helps the model to learn the embedding faster and reduces the risk of overfitting. The model is not forced to produce the exact embedding for the next event. Hence, another MLP with softmax is needed to produce the final output.

5 Experiments

The dataset of Business Process Intelligence Challenge 2012 (BPIC'12) [12] was used to evaluate the performance of the proposed model since it is one of the most popular datasets in predictive business process monitoring. The dataset contains real-life event logs taken from a financial institute. The process in the dataset is an application process for a loan.

Table 1. Overview of the BPIC'12 W dataset

Attribute	
Number of events	72413
Number of cases	9658
Number of event types	6
Average number of events per case	7.498
Max case length	74

Each case has a global attribute that indicates the amount requested by the customer. The application process is a mix of three sub-processes. Similar to other methods, this work narrows down to a sub-process of the work items (BPIC'12 W), so that the performance of the methods can be compared.

The requested amount and the time since the case started were used to create the global feature vector. The attribute nodes are modeled by the time until completion of the event and the one-hot encoding of the day of the week of the event.

Different numbers of dimensions for the embedding and the hidden state are examined. The model was trained for 100 epochs using the Adam optimizer. The batch size is 64, the learning rate is 1×10^{-3} and the weight decay is 5×10^{-4}.

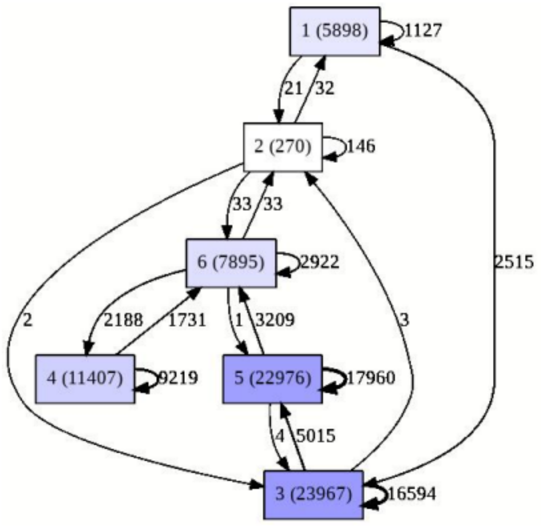


Fig. 6. The Directly-Follows Graph for the BPIC'12 W dataset [7].

Table 2 shows the generalization ability of the proposed method. As the number of dimensions grows, the training accuracy and validation accuracy improve until the peak is reached. The performance dropped significantly when using an overabundant number of dimensions. However, the validation accuracy is always comparable to the training accuracy.

Table 2. The performance of the model using different numbers of dimension

Embedding dim.	Hidden dim.	Training accuracy	Validation accuracy
8	8	0.8054	0.8043
16	16	0.8471	0.8498
32	32	0.8473	0.8521
64	64	0.8541	0.8593
128	128	**0.8611**	**0.8679**
256	256	0.8090	0.8091
512	512	0.5625	0.5710

Figure 7 shows the learning process of the model with 128-dimensional embedding and hidden state. The model converged quickly in the first epoch, achieved 79.32% accuracy. In addition, one can observe from Figure 8 that the poor performance of the model with 512-dimensional embedding and hidden state was not caused by the lack of training epochs. The performance saturated in the first 30 epochs, then converged slowly to its best. The results confirmed the

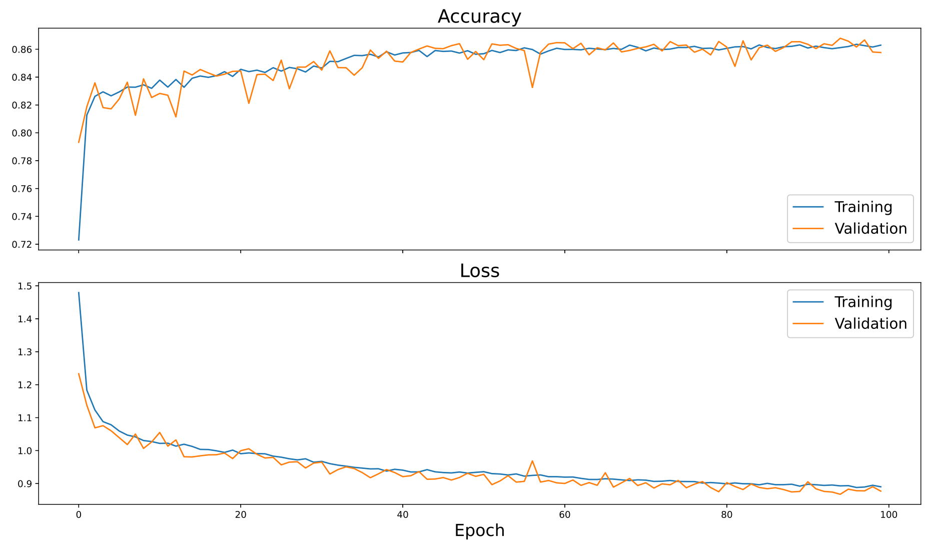


Fig. 7. The training process of the proposed method with 128-dimensional embedding and hidden state.

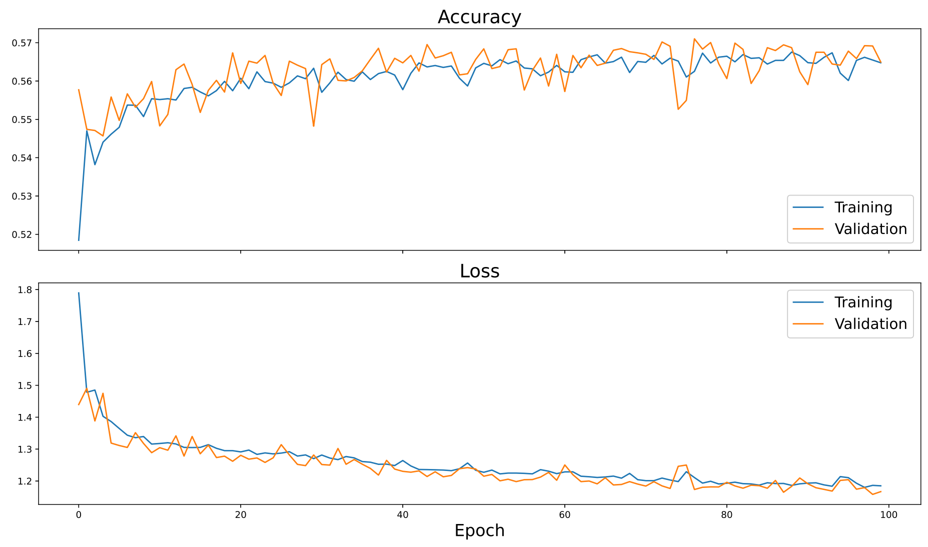


Fig. 8. The training process of the proposed method with 512-dimensional embedding and hidden state.

idea mentioned in the previous section that utilizing the cosine similarity function can help to learn faster and reduce overfitting. Although the model with a high number of dimensions can learn by heart the training data to maximize the probability of the next event, the embedding generated by the type embedding block and the embedding generated by the first MLP block will hardly match when doing so.

Table 3 compares the results of the proposed method and the state-of-the-art methods, including the methods that use PFA, CNN, RNN, and GNN. The proposed method outperforms the other methods by a large margin.

Table 3. Comparison of the results obtained from the models on the BPIC'12 W dataset

Method	**Accuracy**
Evermann et al. [2]	0.623
Tax et al. [1]	0.76
Breuker et al. [9]	0.719
Pasquadibisceglie et al. [8]	0.7817
Venugopal [7]	0.6758
The proposed method	**0.8679**

The source code and the supplementary material to reproduce the experiment results are available at: `https://git.haw-hamburg.de/acf530/predictive-monitoring-gnn`.

6 Conclusion

The major contribution of this paper is the technique to construct the input graph and the application of a similarity function to guide the learning process and reduce overfitting. The input graph is constructed as a heterogeneous graph with two node types, allowing not only to preserve the process flow characteristics but also to record the number of occurrences of each event type.

The experiments showed that the proposed method has a high generalization ability. Although an overabundant number of embedding and hidden state dimensions can cause performance degradation, the training accuracy is always comparable to the validation accuracy.

The model achieved 86.79% accuracy on a subset of the BPIC'12 dataset (BPIC'12 W), and thus, outperformed state-of-the-art methods that use PFA, CNN, RNN, and GNN. The proposed method can be extended to other tasks such as outcome prediction, or next event time prediction.

This work has proved the potential of GNNs in the field of predictive business process monitoring. An interesting topic for future work is to embed the event transitions into the edges or to use heterogeneous graph convolution operators to improve the performance further. Another topic for future work is the explainability of the model. Since the input graph preserves the process flow, explanation methods for GNN such as GNNExplainer [18] or SubgraphX [19] can be used to identify the events that have the most influence on the result.

References

1. Tax, Niek, et al. "Predictive business process monitoring with LSTM neural networks." International Conference on Advanced Information Systems Engineering. Springer, Cham, 2017.
2. Evermann, Joerg, Jana-Rebecca Rehse, and Peter Fettke. "A deep learning approach for predicting process behaviour at runtime." International Conference on Business Process Management. Springer, Cham, 2016.
3. Rogge-Solti, Andreas, and Mathias Weske. "Prediction of business process durations using non-Markovian stochastic Petri nets." Information Systems 54 (2015): 1-14.
4. Kang, Bokyoung, Dongsoo Kim, and Suk-Ho Kang. "Periodic performance prediction for real-time business process monitoring." Industrial Management & Data Systems (2012).
5. Bevacqua, Antonio, et al. "A Data-adaptive Trace Abstraction Approach to the Prediction of Business Process Performances." ICEIS (1). 2013.
6. Esser, Stefan, and Dirk Fahland. "Multi-dimensional event data in graph databases." Journal on Data Semantics (2021): 1-33.
7. Venugopal, Ishwar. "Quartile-based Prediction of Event Types and Event Time in Business Processes using Deep Learning." arXiv e-prints (2021): arXiv-2102.
8. Pasquadibisceglie, Vincenzo, et al. "Using convolutional neural networks for predictive process analytics." 2019 international conference on process mining (ICPM). IEEE, 2019.
9. Breuker, Dominic, et al. "Comprehensible Predictive Models for Business Processes." MIS Q. 40.4 (2016): 1009-1034.
10. Weber, Philip, Behzad Bordbar, and Peter Tino. "A principled approach to mining from noisy logs using Heuristics Miner." 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM). IEEE, 2013.
11. Mehdiyev, Nijat, Joerg Evermann, and Peter Fettke. "A multi-stage deep learning approach for business process event prediction." 2017 IEEE 19th conference on business informatics (CBI). Vol. 1. IEEE, 2017.
12. van Dongen, B. F. "Real-life event logs–a loan application process, 2012." Second International Business Process Intelligence Challenge (BPIC'12). DOI:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.
13. Bronstein, Michael M., et al. "Geometric deep learning: going beyond euclidean data." IEEE Signal Processing Magazine 34.4 (2017): 18-42.
14. Battaglia, Peter W., et al. "Relational inductive biases, deep learning, and graph networks." arXiv preprint arXiv:1806.01261 (2018).
15. Wu, Zonghan, et al. "A comprehensive survey on graph neural networks." IEEE transactions on neural networks and learning systems 32.1 (2020): 4-24.
16. Gilmer, Justin, et al. "Neural message passing for quantum chemistry." International conference on machine learning. PMLR, 2017.
17. Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph con- volutional networks." arXiv preprint arXiv:1609.02907 (2016).
18. Ying, Rex, et al. "Gnnexplainer: Generating explanations for graph neural networks." Advances in neural information processing systems 32 (2019): 9240.

19. Yuan, Hao, et al. ”On explainability of graph neural networks via subgraph explorations.” arXiv preprint arXiv:2102.05152 (2021).