Benchmarking Node Outlier Detection on Graphs

Kay Liu[1,*], **Yingtong Dou**[1,8,*], **Yue Zhao**[2,*], **Xueying Ding**[2], **Xiyang Hu**[2],
Ruitong Zhang[3], **Kaize Ding**[4], **Canyu Chen**[5], **Hao Peng**[3], **Kai Shu**[5],
Lichao Sun[6], **Jundong Li**[7], **George H. Chen**[2], **Zhihao Jia**[2], **Philip S. Yu**[1]
[1] University of Illinois Chicago [2] Carnegie Mellon University
[3] Beihang University [4] Arizona State University [5] Illinois Institute of Technology
[6] Lehigh University [7] University of Virginia [8] VISA Research
benchmark@pygod.org

Abstract

Graph outlier detection is an emerging but crucial machine learning task with numerous applications. Despite the proliferation of algorithms developed in recent years, the lack of a standard and unified setting for performance evaluation limits their advancement and usage in real-world applications. To tap the gap, we present, (to our best knowledge) the first comprehensive unsupervised node outlier detection benchmark for graphs called UNOD, with the following highlights: (1) evaluating fourteen methods with backbone spanning from classical matrix factorization to the latest graph neural networks; (2) benchmarking the method performance with different types of injected outliers and organic outliers on real-world datasets; (3) comparing the efficiency and scalability of the algorithms by runtime and GPU memory usage on synthetic graphs at different scales. Based on the analyses of extensive experimental results, we discuss the pros and cons of current UNOD methods and point out multiple crucial and promising future research directions.

1 Introduction

Graph outlier detection (OD) is a key machine learning (ML) task with many applications, including social network spammer detection [69], sensor fault detection [24], financial fraudster identification [19], and defense on graph adversarial attacks [29]. Different from classical outlier detection on tabular data and time-series data, graph outlier detection faces additional challenges: (1) complex data structure carries richer information, and thus more powerful ML models are needed to learn informative representations and (2) with more complex ML models, it often incurs higher training difficulty and more extensive memory consumption [31], posing challenges for time-critical (i.e., low time budget) and resource-sensitive (e.g., limited GPU memory) applications.

With its importance and challenges in mind, we argue that *a comprehensive benchmark on graph outlier detection* is absent, and thus limits its advancement. A recent graph OD survey also calls for "system benchmarking" and describes it as "the key to evaluating the performance of graph OD techniques" [47]. Although there are benchmarks for general graph mining (e.g., OGB [28]), graph representation learning [23], graph robustness evaluation [82], graph contrastive learning [84], graph-level anomaly detection [75], as well as benchmarks for tabular [7], and time-series OD [40], *unsupervised node outlier detection in graphs* (UNOD) needs its own benchmark to address the existing limitations: (1) **the lack of a standard environment for fair comparison**: UNOD algorithms are often evaluated on few datasets with "randomly injected" synthetic outliers, where the performances across algorithms are not comparable and the performance on real-world benchmark datasets remains unknown; (2) **the limited understanding of how algorithms respond to different types of node outliers** limits better algorithm design for specific types of outliers and (3)

在伪造数据集和真实数据集上的

对于不同种类的异常模型的理解

[*] Equal Contribution

Preprint. Under review.

the deficiency of efficiency and scalability analysis: existing works mainly focus on the detection accuracy, while the understanding of runtime and memory consumption on graph OD is insufficient.

To bridge the gap, we design *the first comprehensive unsupervised node outlier detection benchmark* called UNOD, to better understand graph OD algorithms' performance with regard to both real-world and synthetic datasets and/or with different types of outliers. Among all types of graph OD tasks at different levels, *unsupervised node-level* detection is most prevalent and important [47], and UNOD thus focuses on this aspect. To include a large number of algorithms, we specifically create Python Graph Outlier Detection (PyGOD)[1], which provides more than ten latest graph OD algorithms; all with unified APIs and optimizations. Meanwhile, we also include multiple non-graph baselines, leading to a total of fourteen representative but diverse methods for graph node OD detection.

Our work has following highlights:

1. **The first comprehensive node-level graph OD benchmark**. We examine fourteen OD methods, including classical and deep ones, compare their pros and cons, on five benchmark datasets.
2. **Consolidated taxonomy on node outliers.** After reviewing the fruitful related works, we give a unified definition of two node outlier types on graph data, i.e., structural and contextual outliers. Our results show that most methods fail to balance the OD performance of two types of outliers.
3. **Additional performance measures**. In addition to common *effectiveness* metrics (e.g., ROC-AUC), we also measure and analyze algorithm runtime and GPU memory consumption as their *efficiency* and *scalability* measures, which is crucial in UNOD tasks.
4. **Novel future research directions**. Based on experiment result analyses, we point out a few key future directions for graph OD, including unsupervised model selection, scalability improvement by specialized OD quantization, detecting outliers by types, and more.
5. **Reproducible and accessible benchmark toolkit**. To foster accessibility and fairness for future algorithms, we create a UNOD benchmark toolkit (see Appx. D for more details) and open-source it at `https://github.com/pygod-team/pygod/tree/main/benchmark`.

In §2, we briefly describe existing approaches for graph OD. In §3, we first give the problem definition and taxonomy, and then present the rich coverage of UNOD, followed by the detailed experiment results in §4. In §5 and 6, we present the key insights from the results and then conclude the paper.

2 Related Work

In this section, we briefly introduce related work on UNOD. Please refer to [2] and [47] for more comprehensive reviews of non-deep and deep learning-based graph outlier detectors. For most of the discussed algorithms in this section, we have included them in the PyGOD library.

2.1 Non-deep Node Outlier Detection

Real-world evidence suggests that the outlier nodes are different from regular nodes in terms of structure or attribute. Therefore, early node outlier detection works employ graph-based features like centrality measures and clustering coefficient to extract the anomalous signals from graph [2]. Since learning-based methods are more flexible than handcrafted features, the matrix factorization (MF) methods [61, 42, 52, 3], density-based clustering methods [8, 67, 27], and relational learning methods [37, 56] have been leveraged to encode the graph information and spot outlier nodes. As most of the works above have constraints on graph/node types or prior knowledge, we only include SCAN [67], Radar [42] and ANOMALOUS [52] into UNOD to represent methods in this category.

2.2 Deep Node Outlier Detection

The rapid development of deep learning and graph deep learning also shifts the landscape of node outlier detection from traditional methods to neural networks [47]. The autoencoder (AE) [34], which is a neural network architecture devised to learn the data representation by reconstructing it, becomes an ideal and dominant model to detect node outliers [17, 21, 35, 57, 64, 71]. The heuristic behind AE-based outlier detection is to learn the representation of inlier data and use the reconstruction error of AE as the outlier indicator. Initially, the Multi-layer Perceptron (MLP) neural network is used as the encoder and decoder in AE. As the Graph Neural Network (GNN) attains superior performance

[1] PyGOD: `https://pygod.org/`

on many graph mining tasks [25, 36, 62], GNN has become a primary method to learn the node representation for node outlier detection [65, 77]. The advantage of GNNs is that they encode the node attribute and structure information simultaneously, thereby capturing more high-dimensional anomalous signals. Furthermore, some recent works [46, 68] integrate contrastive learning with GNNs to improve the capability to differentiate node outliers. Besides the above UNOD methods that leverage the GNN as the encoder and decoder of AE, a recent work [12] adopts Generative Adversarial Network (GAN) for node outlier detection. Since there are many deep node outlier detectors, we select nine (see Table 2) with diverse backbone types and investigate them in UNOD.

3 Graph Outlier Benchmark

The formal description of UNOD is organized as below. In §3.1 and 3.2, we provide the problem definition and our proposed consolidated taxonomy for UNOD. Later, we elaborate on the design of datasets (§3.3), UNOD algorithms (§3.4), and evaluation metrics (§3.5).

3.1 Problem Definition of Unsupervised Node Outlier Detection in Graphs (UNOD)

An outlier is defined as *a sample that is significantly different from the remaining data* [1]. Since the graph is capable of representing the relational data, the graph OD has become a unique research topic distinguishing from generic OD on tabular data [2, 47]. In the past two decades, graph OD has fostered fruitful works, from unsupervised to semi-supervised, from static to dynamic graphs, targeting different graph objects (e.g., nodes, edges, subgraphs), and leveraging various methods from non-deep methods to deep methods. Our benchmark concentrates on the *unsupervised node outlier detection on the static attributed graph* (UNOD) which has the richest literature due to its importance and applicability. We formally define our problem as follows:

Definition 1 *(Unsupervised Node Outlier Detection on the Static Attributed Graph) A static attributed graph can be defined as* $G = (V, E, \mathbf{X})$ *where* V *and* E *represent a set of nodes and edges, respectively;* $\mathbf{X}$ *is the node attribute matrix. To detect node outliers* $V_o \subset V$*, an unsupervised detector* $f : G \rightarrow O$ *takes* G *as input and outputs the outlier scores* $O \in \mathbb{R}$ *for every node in the graph. Generally, the top-k nodes with the highest outlier scores are regarded as outliers.*

3.2 A Unified Taxonomy

Besides the general node outlier definition, many previous works [2, 3, 17, 29, 42, 47] have defined fine-grained node outlier types from different perspectives. In this paper, we unify the previous node outlier taxonomies as two major types: *structural outlier* and *contextual outlier*, which are illustrated in Figure 1 and defined below:

Definition 2 *(Structural Outlier) Structural outliers are densely connected nodes in contrast to sparsely connected regular nodes.*

Structural outlier is prevalent in real-world graphs as members of organized fraud gangs usually interconnect with each other to carry out malicious activities [27]. Some papers [42, 47] also regard nodes that do not belong to any communities as structural outliers. Since there is no dedicated algorithm for the isolated structural outlier, we would not discuss it in our benchmark.

Definition 3 *(Contextual Outlier) Contextual outliers are nodes whose attributes are significantly different from their neighboring nodes.*

Contextual outlier depicts instances dissimilar to their neighbors, e.g., spammers and fraudsters in social network [19]. Its definition is similar to the assumption of the outlier in the classic proximity-based OD methods [1].

Some works define the node whose attribute is different from all other nodes as the global outlier [47] or contextual outlier [42]. We argue

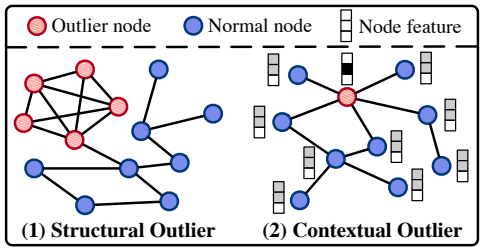

Figure 1: A toy example of two node outlier types.

that the above outlier resembles the tabular data outlier [1] without any connection to the graph data property and it can be detected using tabular data outlier detectors [80]. Therefore, we will not discuss this outlier type in this paper. The contextual outlier is also referred as attribute [17] or community outlier [42, 47] in previous work. We argue that the *contextual outlier* is a more accurate term to define the above outlier pattern, whereas *attribute* and *community* could mingle with the *global outlier* or *structural outlier*, respectively.

Note that the organic node outliers in real-world graphs may not strictly follow the above taxonomies and their anomalous signals may not be explicit. We should also note that *organic node outliers* may often be the mixture of different types of outliers, posing a greater detection difficulty. Experiment results (§4.2.1) and our analyses (§5) provide additional insights on detecting organic outliers.

Table 1: Statistics of real-world datasets in UNOD (where * denotes that outliers are injected).

Dataset	#Nodes	#Edges	#Feat.	Degree	#Con.	#Strct.	#Outliers	Ratio
Cora*	2,708	11,186	1,433	4.1	70	70	138	5.1%
Amazon*	13,752	515,872	767	37.5	350	350	694	5.0%
Flickr*	89,250	942,316	500	10.6	2,240	2,240	4,414	4.9%
Weibo	8,405	407,963	400	48.5	-	-	868	10.3%
Reddit	10,984	168,016	64	15.3	-	-	366	3.3%

3.3 Datasets in UNOD

To comprehensively evaluate the performance of existing UNOD algorithms, we conduct experiments on various real-world datasets whose statistics are listed in Table 1. Since there are a limited number of open-source graph datasets with organic node outliers, we need to inject the node outliers to help us investigate existing algorithms. To inject node outliers, we choose three real-world node classfication benchmark datasets (**Cora** [58], **Amazon** [59], and **Flickr** [72]) from three domains with different scales used by previous graph OD work. We also incorporate two real-world graphs (**Weibo** [76], and **Reddit** [39, 66]) with organic node outliers to evaluate the practical performance of UNOD algorithms. Besides the real-world graphs, we leverage a random graph generation algorithm from [32] to synthesize graphs to help benchmark UNOD algorithms' efficiency and scalability. More dataset details can be found in Appx. A.1.

We follow a widely-used outlier injection approach [17, 21, 12, 71] to inject two types of node outliers. For the *structural outlier*, we randomly select some nodes in a graph as outliers by making them fully-connected. For the *contextual outlier*, we randomly select some nodes in a graph as outliers by replacing each node's attribute with a dissimilar attribute from another node in the same graph. Please refer to Appx A.1 for more details about outlier injection.

3.4 Benchmarked Algorithms

Table 2: Representative OD algorithms evaluated in UNOD, where we list their characteristics: if designed graph (row 3), if a neural network method (row 4), and the backbones (row 5). MF denotes matrix factorization.

Alg.	LOF	IF	MLPAE	SCAN	Radar	ANOMA-LOUS	GCNAE	DOMI-NANT	DONE/ AdONE	Anomaly-DAE	GAAN	GUIDE	CONAD
Year	2000	2012	2014	2007	2017	2018	2016	2019	2020	2020	2020	2021	2022
Graph	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Deep	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Core	N/A	Tree	MLP+AE	Cluster	MF	MF	GNN+AE	GNN+AE	MLP+AE	GNN+AE	GAN	GNN+AE	GNN+AE
Ref.	[5]	[44]	[57]	[67]	[42]	[52]	[35]	[17]	[4]	[21]	[12]	[71]	[68]

Table 2 lists the fourteen algorithms evaluated in the benchmark and their properties; our principle for selecting the UNOD algorithms is to cover representative methods in terms of the published time ("Year"), whether using the graph structure ("Graph"), whether using deep neural networks ("Deep"), and the backbone model type ("Core"). By adding non-graph OD algorithms (LOF, IF, and MLPAE) into the benchmark, we can investigate the advantage and deficiency of graph-based OD algorithms in detecting node outliers. Selecting the non-deep methods follows a similar reason as above where the clustering method (SCAN) and MF methods (Radar and ANOMALOUS) are incorporated. For the GNN-based UNOD methods, we include methods from the classic DOMINANT to recent methods

leveraging GAN (GAAN) and contrastive learning (CONAD). Please refer to Appx. A.2 for more detailed introduction of the algorithms benchmarked in UNOD.

3.5 Evaluation Metrics

Detection quality measures. We follow the extensive literature in graph OD [16, 61, 77] to evaluate the UNOD detection quality: (1) ROC-AUC (2) Recall@k and (3) Average Precision. See Appx. A.3.

Efficiency and scalability measures. Another important aspect of graph-based algorithms is their high time and space complexity [18, 31], which imposes additional challenges for large, high-dimensional datasets on hardware like GPUs with limited memory (e.g., often facing the issue of out of memory). Therefore, we propose to measure (1) wall-clock time as an efficiency measure and (2) GPU memory consumption as a scalability measure. We provide more details in Appx. A.3.

4 Experiments

We design UNOD to understand the detection effectiveness, efficiency, and scalability of various graph-based node outlier detection algorithms. Specifically, we aim to answer: **RQ1** (§4.2.1): How effective of the algorithms on real-world data? **RQ2** (§4.2.2): How do algorithms perform under various types of outliers? **RQ3** (§4.2.3): How efficient and scalable of algorithms regarding larger graphs? Based on the analyses, we further summarize important research directions for future graph OD in §5.1 and future benchmark plans in §5.2. Due to the space limitation, we only report the results by ROC-AUC, while the AP and Recall@k results are consistent and available in Appx. C.

4.1 Experimental Settings

Model implementation and environment configuration. Most algorithms in UNOD are implemented via our newly released PyGOD package [45], non-graph OD methods are imported from our early work [80]. Although we tried our best to apply the same set of optimization techniques, e.g., vectorization, to all baselines, there might still be some space for optimization given the wide range of included algorithms. See more implementation details and environment configurations in Appx. B.

Hyperparameter Grid. In real-world settings, it is unclear how to do hyperparameter tuning and algorithm selection for unsupervised outlier detection due to the lack of ground truth labels and/or universal criteria that correlates well with the ground truth [47, 81]. For fair evaluation, we apply the same hyperparameter grid (see Appx. B) to each applicable algorithm and report its *avg. performance* (i.e., "algorithm performance in expectation"), along with the *standard deviation* (i.e., "algorithm stability") and the *max* (i.e., "algorithm potential"). See Tables 3 and 4 for illustration.

4.2 Experiment Results

4.2.1 Detection Performance on Real-world Datasets

Table 3 presents the results of algorithms on five real-world datasets (see §3.3). For each algorithm, we analyze its avg. performance on the hyperparameter grid as its expected performance, along with the standard deviation and max as measures of stability and potential. Below are the key findings:

Deep and graph-based OD methods are generally better than non-deep and non-graph ones at detecting injected outliers. The results on Cora, Amazon, and Flickr show that most graph-based methods are better than non-graph OD methods, i.e., LOF, IF, and MLPAE, justifying the importance of using structural information in graph OD tasks. Meanwhile, deep methods have better average performance than three non-deep methods (SCAN, Radar, and ANOMALOUS), demonstrating the superior learning capability of deep learning.

No graph OD algorithm outperforms on all datasets in expectation. Table 3 shows that only AnomalyDAE outperforms (by the avg. performance) on two datasets (i.e., Cora and Amazon), while it is far from the best on Flickr. Additionally, there is a huge performance gap between the best- and worst-performing algorithms, e.g., DONE is $2.06\times$ higher than LOF on Flickr. These observations corroborate our understanding of unsupervised OD algorithms that their effectiveness depends on whether the underlying algorithm assumption is consistent with the underlying data distribution.

Table 3: ROC-AUC (%) comparison among OD algorithms on five real-world benchmark datasets, where we show *the avg perf.* $\pm$ *the STD of perf.* (*max perf.*) of each. The best algorithm by expectation is shown in **bold**, while the max performance per dataset is marked with underline. OOM denotes out of memory with regard to GPU (_G) and RAM (_R). Clearly, no OD algorithm universally outperforms on all datasets; non-graph methods are generally worse than graph-based methods.

Algorithm	Cora	Amazon	Flickr	Weibo	Reddit
LOF	69.9±0.0 (69.9)	55.2±0.0 (55.2)	41.6±0.0 (41.6)	56.5±0.0 (56.5)	**57.2**±0.0 (57.2)
IF	64.4±1.5 (67.4)	51.3±3.0 (57.9)	57.1±1.1 (58.8)	53.5±2.8 (57.5)	45.2±1.7 (47.5)
MLPAE	70.9±0.0 (70.9)	74.2±0.0 (74.2)	72.4±0.0 (72.5)	82.1±3.6 (86.1)	50.6±0.0 (50.6)
SCAN	67.3±5.8 (73.0)	63.5±5.7 (71.3)	66.3±11.9 (75.3)	63.7±5.6 (70.8)	49.9±0.3 (50.0)
Radar	65.0±1.3 (65.9)	72.7±1.1 (73.5)	OOM_G	**98.9**±0.1 (99.0)	54.9±1.2 (56.9)
ANOMALOUS	59.9±9.7 (71.8)	72.8±1.8 (75.9)	OOM_G	**98.9**±0.1 (99.0)	54.9±5.6 (60.4)
GCNAE	70.9±0.0 (70.9)	74.2±0.0 (74.2)	70.8±3.9 (72.4)	90.8±1.2 (92.5)	50.6±0.0 (50.6)
DOMINANT	81.8±8.6 (85.8)	82.0±2.7 (83.5)	82.8±7.6 (85.5)	85.0±14.6 (92.5)	56.0±0.2 (56.4)
DONE	81.2±5.6 (88.0)	85.2±8.3 (95.3)	**85.5**±4.4 (88.4)	85.3±4.1 (88.7)	53.9±2.9 (59.7)
AdONE	81.1±5.8 (86.2)	83.1±11.5 (94.9)	82.1±4.8 (90.2)	84.6±2.2 (87.6)	50.4±4.5 (58.1)
AnomalyDAE	**84.4**±2.0 (86.7)	**87.8**±3.1 (93.0)	71.0±3.5 (74.7)	91.5±1.2 (92.8)	55.7±0.4 (56.3)
GAAN	75.1±0.7 (76.7)	82.2±0.3 (82.9)	72.4±0.1 (72.6)	92.5±0.0 (92.5)	55.4±0.4 (56.0)
GUIDE	75.1±0.7 (77.6)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	81.4±8.9 (85.4)	82.2±1.0 (83.5)	62.5±6.5 (68.4)	85.4±14.3 (92.7)	56.1±0.1 (56.4)

Knowledge and performance on synthetic outliers may not generalize to organic outliers. With a closer look, we find best performers on the datasets with synthetic outliers (e.g., AnomalyDAE) fail to achieve the best performance on the datasets with organic outliers (i.e., Weibo, Reddit). Moreover, the deep graph (graph resp.) methods are all worse than the non-deep graph (non-graph resp.) methods on Weibo (Reddit resp.). This disconnect reveals the gap in using synthetic outliers for graph OD evaluation. Extensive analysis shows that the outliers in Weibo are densely connected and their features differ from their normal neighbors, which are similar to our taxonomies defined in § 3.2. While the outliers on Reddit do not explicitly exhibit the above patterns, their abnormality has a higher dependency on the domain knowledge reflected by outlier annotations.

The trade-off between algorithm stability and potential. Certain graph OD methods, e.g., MLPAE and GCNAE, exhibit insensitivity to hyperparameters, where the performance variation is consistently below 1%. We credit their stability to the simple reconstruction loss, where these stable methods may suit the case when hyperparameter tuning is infeasible. In contrast, the high potential methods (the max performance highlighted with underline per dataset in Table 3) are often unstable methods with more complex loss terms (e.g., the weighted combination of multiple losses). For instance, DONE and AdONE collectively achieve the highest performance on three out of five datasets, while showing high performance variations. To sum up, we notice a trade-off between the algorithm stability and performance in graph OD, which may guide us for algorithm design.

4.2.2 Performance Variation under Different Types of Outliers

In Table 4, we compare the algorithm performance on three datasets with two types of injected outliers, i.e., contextual and structural outliers introduced in §3.3. Below are a summary of insights.

The reconstruction of the structural information, instead of the neighbor aggregation of GNN, plays the significant role in detecting structural outlier.. Specifically, the performance gap on structural outliers between GCNAE and DOMINANT is over 40%. With a closer look into these two algorithms, they only differ in DOMINANT is the structural decoder, which reconstructs the adjacency matrix, demonstrating the effectiveness of reconstruction of the structural information.

Low-order structure information (i.e., one-hop neighbors) are sufficient for detecting structural outliers. DOMINANT and DONE have a similar detection performance ($\sim$92%) on structural outlier on Cora and Amazon, while DOMINANT encodes 4-hop neighbor information but DONE only encodes 1-hop neighbor information. This observation could facilitate UNOD model design since encoding high-order information usually imposes a higher computational cost, and multi-hop neighbor aggregation may even lead to GNN's oversmoothing problem [9].

No method has a balanced detection performance on two outlier types. None of the methods reaches 85% detection AUC on both contextual outlier and structural outlier, suggesting the common practice that arbitrarily combines the contextual and structural loss with a fixed weight during the

Table 4: ROC-AUC (%) comparison among OD algorithms on three datasets injected with contextual and structural outliers, where we show *the avg perf.* ± *the STD of perf.* (*max perf.*) of each. The best algorithm by expectation is shown in **bold**, while the max performance per dataset is marked with underline. OOM denotes out of memory with regard to GPU (_G) and RAM (_R). Notably, reconstruction-based MLPAE and GCNAE perform best w.r.t contextual outliers, while there is no universal winner for both types of outliers.

	Cora		Amazon		Flickr	
Algorithm	Contextual	Structural	Contextual	Structural	Contextual	Structural
LOF	87.1±0.0 (87.1)	52.4±0.0 (52.4)	61.9±0.0 (61.9)	48.0±0.0 (48.0)	34.2±0.0 (34.2)	49.1±0.0 (49.1)
IF	77.5±2.2 (81.8)	51.4±2.3 (56.2)	51.8±6.0 (64.3)	50.9±0.8 (52.2)	63.1±2.1 (66.5)	50.9±0.3 (51.5)
MLPAE	**88.9**±0.0 (88.9)	52.5±0.0 (52.5)	**98.6**±0.0 (98.6)	49.0±0.0 (49.0)	**94.4**±0.1 (94.5)	50.0±0.1 (50.3)
SCAN	49.8±0.3 (50.6)	84.5±11.6 (95.9)	49.0±1.0 (49.9)	77.6±11.4 (94.0)	50.2±0.1 (50.3)	82.1±23.4 (99.7)
Radar	50.0±0.6 (51.0)	79.5±3.1 (81.6)	85.9±3.5 (88.2)	58.7±1.5 (61.3)	OOM_G	OOM_G
ANOMALOUS	51.1±1.6 (53.1)	68.7±19.2 (91.0)	84.8±0.9 (86.8)	60.1±3.4 (65.4)	OOM_G	OOM_G
GCNAE	**88.9**±0.0 (88.9)	52.5±0.0 (52.5)	**98.6**±0.0 (98.6)	49.0±0.0 (49.0)	91.2±7.7 (94.5)	50.0±0.2 (50.4)
DOMINANT	71.2±9.1 (82.6)	91.3±8.2 (95.4)	69.2±2.1 (71.3)	93.5±3.3 (94.3)	68.1±4.2 (70.9)	**96.2**±10.7 (99.0)
DONE	69.3±7.5 (77.9)	**92.0**±5.6 (97.8)	77.1±14.8 (95.6)	91.6±5.5 (99.4)	85.0±2.7 (89.0)	84.5±7.0 (88.5)
AdONE	70.2±7.5 (77.5)	90.9±4.4 (95.7)	78.0±13.1 (94.4)	86.7±14.2 (98.3)	78.7±4.6 (88.2)	84.3±8.1 (90.7)
AnomalyDAE	83.6±4.0 (87.5)	83.8±7.6 (95.9)	91.3±8.6 (98.3)	82.6±10.6 (94.3)	85.0±6.9 (93.1)	56.5±1.6 (59.8)
GAAN	88.7±0.1 (88.8)	60.8±1.4 (63.9)	98.4±0.0 (98.5)	64.7±0.6 (66.0)	94.2±0.2 (94.4)	50.2±0.2 (50.7)
GUIDE	87.4±4.5 (88.7)	62.0±5.6 (86.4)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	71.3±6.3 (74.4)	90.3±13.2 (95.6)	69.0±1.7 (71.3)	**94.1**±0.4 (94.3)	61.0±6.1 (67.0)	63.7±6.9 (69.4)

optimization fails to balance the performance well. The performance and preference of algorithms are vulnerable to the weights of different loss terms. How to detect different types of outliers well and consistently remains an open question.

4.2.3 Efficiency and Scalability Analysis

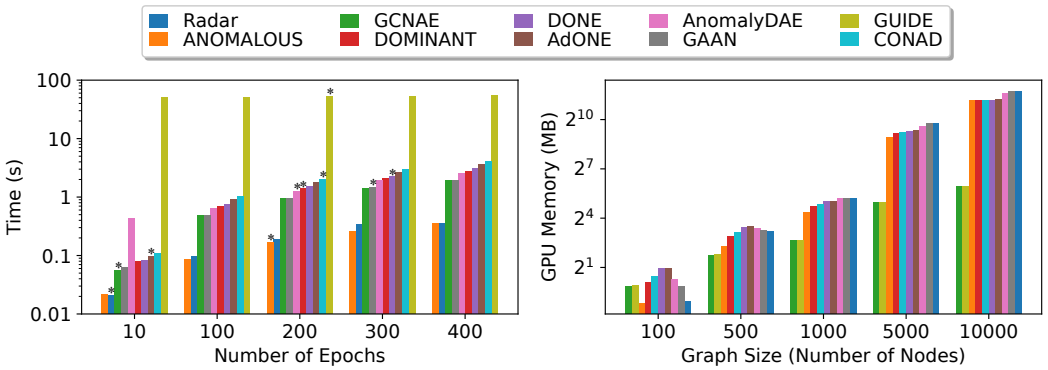

Figure 2: The runtime (left) and GPU memory consumption (right) of different methods. (∗ denotes the best performance of each methods among five different numbers of epochs. See Appx. B for details.)

We visualize the efficiency (i.e., runtime) and scalability (i.e., GPU memory consumption) of selected algorithms in Figure 2. The complete results are available in Appx. C. All algorithms are evaluated under randomly generated graphs with the same sets of injected outliers to guarantee fairness. Since all algorithms are unsupervised and transductive, the runtime in Figure 2 (left) represents the total time of training and generating outlier scores. We only measure the GPU memory consumption of the various methods because it is more expensive and often the bottleneck of OD algorithms [78]. Please find more details about the synthetic graphs and experimental settings in this section in Appx. A.1 and Appx. B, respectively. According to Figure 2, we have the following observations and insights:

Algorithm Efficiency. The non-deep methods are more efficient than deep ones, corroborating common sense. Among the GNN-based methods, GUIDE consumes far more time compared to others. The reason is that GUIDE leverages a graph motif counting algorithm (which is #P-complete [14]) to extract the structural features and consumes much more time on the CPU. CONAD has the second-lowest efficiency due to its utilization of contrastive learning. One should note that the algorithm efficiency depends on both the runtime per epoch and the convergence time.

Algorithm Scalability. According to Figure 2 (right), GCNAE and GUIDE consume much less GPU memory than other methods as graph size grows. GCNAE saves more memory due to its simpler architecture. GUIDE consumes more CPU time and RAM to extract low-dimensional node motif degrees, thereby saving more GPU memory. Though non-deep methods have the advantage in terms of runtime, most of them can not be deployed in a distributed fashion due to the limitation of "global" operators like matrix factorization and inversion. One advantage of deep models is that they can be easily extended to mini-batch and distributed training via graph sampling. Another advantage is that deep UNOD methods can be easily integrated with the deep-learning pipelines (e.g., graph pretraining module that obtains node embeddings), which conforms to modern applied ML fashion.

5 Discussion and Future Directions

5.1 Future Research Directions on Graph Outlier Detection

Unsupervised algorithm selection and hyperparameter tuning. As discussed in §4.1, it is a "black art" to pick a good graph OD algorithm without access to (at least a small set of) ground truth labels. Experiment results in §4.2.1 illustrate huge performance gaps among algorithms, and the lack of a universal winner, justifying the necessity of algorithm selection in graph OD. The performance of AnomalyDAE on Weibo shows that even for the same algorithm, its performance highly depends on the hyperparameters, and the variation can be as large as 14%, calling for the attention to hyperparameter tuning as well. How to automatically select a good graph OD algorithm and set its hyperparameter(s)? Our recent work on unsupervised OD model selection on tabular data [81] shows that under meta-learning frameworks, we may identify decent OD models for a new task/dataset based on its similarity to meta-tasks where ground truth and evaluation are possible. We argue the necessity of exploring similar approaches for graph OD algorithm selection and hyperparameter tuning, and designing methods to quantify task similarity among graph OD datasets, to prevent selecting algorithms blindly.

More robust and stable outlier detection algorithms. Experiment results also show that some algorithms yield better robustness to hyperparameters with slight performance variation (see Table 3), e.g., MLPAE, GCNAE, and GAAN, even though they are not necessarily the best performer among all. It is worth understanding the critical drivers of graph OD models' robustness to design more robust models, which is extremely meaningful for the unsupervised algorithms due to the challenges in hyperparameter tuning. In tabular OD tasks, researchers have already worked on designing more robust methods for detection, such as robust autoencoder [83] and RandNet [10] by the model ensemble. With such algorithms, the necessity of extensive hyperparameter tuning is reduced.

Focusing on high-value outliers and designing specialized algorithms. The results in §4.2.2 shows that outlier types can give great performance edge. For instance, we could pick the performing detection models given a specific type of outliers, e.g., MLPAE and GCNAE for structural outliers. In this way, practitioners only need to focus on high-value/interest types of outliers, e.g., illegal trades contributing most to revenue [33]. However, existing graph OD methods usually do not assume their specialized types, but assume some outlier characteristics and behaviors. Following the recent advancement in categorizing outlier types for tabular data [30], we call for attention to identify outlier types in graph OD as well as specialized algorithms for distinct types of outliers, which may unlock new opportunities in algorithm selection by anomaly types and ensemble learning (e.g., combining multiple OD models based on their specialty).

Improving the model efficiency and scalability. In §4.2.3, we observed certain algorithms like GUIDE requests high runtime (two degrees more) and extreme RAM, while GAAN needs excessive GPU memory consumption (up to $100\times$), and we thus suggest more research emphasis on improving the scalability on graph OD algorithms in addition to proposing new methods. Some existing works have discussed how to make the GNNs more scalable [31, 41], but more tailored work for graph OD considering its unsupervised and imbalanced nature, is currently absent. Some works provide a potential approximation of node motif degree in GUIDE, which can significantly reduce the computational cost in terms of runtime and RAM [11, 70]. Some of our recent works [79, 78] discuss the possibility of accelerating outlier detection on tabular data by distributed learning, data and model compression, and/or quantization, where we believe these techniques are also applicable to graph OD. For now, no work has focused on quantizing graph OD models and analyzing the impact.

5.2 Limitations of UNOD and Future Directions for Graph OD Benchmarking

Extending to detection tasks at different levels. In this work, we focus on the node-level detection with graphs due to its popularity, while there are more detection tasks at different levels of a graph. Recently, more graph OD algorithms are design for edge- [73], subgraph- [64], and graph-level [55, 75] detection. A more holistic view of graph OD can include these emerging graph OD tasks.

Designing more realistic and flexible synthetic outlier generation in graphs. Real-world outliers can be complex and often composed of multiple types of outliers. Experiment results in §4.2.1 point out that there is a generalization gap from the results on synthetic outliers to the results on organic outliers, calling for more realistic generation approaches. To improve existing generation methods in UNOD, we may use a generative model to fit the normal samples, and then perturb the generative model to generate different types of outliers, which has been successful in tabular OD [60].

Incorporating supervision in graph OD. Although UNOD focuses on unsupervised detection methods due to the abundance of applications, there can be cases with a small set of labels (either for OD or relevant tasks) available in graph applications with increasing attention to (semi-)supervised methods, e.g., [19, 74]. Future benchmarks may also incorporate supervision and understand its value and role in graph OD, which likely address the challenges in algorithm selection and robustness.

Curation of more real-world datasets. Although we have benchmarked fourteen graph OD algorithms on more than five real-world and synthetic datasets, it is helpful to gather more benchmark datasets for UNOD. The experiment results in §4.2.1 show that there is a disconnect between the performance of OD methods on real-world and synthetic datasets with specific types of injected outliers. Also, with enough datasets (e.g., ≥ 20), one can run pairwise and group statistical tests for comparison [15], which has been used in OD tasks with tabular datasets [43, 81]. How to enrich the list of graph OD datasets? Following the existing works [20], one can turn existing multi-classification graph datasets, e.g., ones from Open Graph Benchmark (OGB) [28], for OD by treating one or combining multiple small classes as outliers and the remaining classes as normal.

6 Conclusion

In this paper, we propose the first comprehensive benchmark—UNOD for unsupervised node outlier detection on graphs. We evaluate the detection performance of fourteen algorithms on five real-world datasets and benchmark their efficiency and scalability under a standard setting. We give in-depth analyses of extensive experiment results, discuss the pros and cons of existing methods, and share our insights on future directions in developing and applying UNOD algorithms.

References

[1] C. C. Aggarwal. An introduction to outlier analysis. In *Outlier analysis*. Springer, 2017.

[2] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29(3):626–688, 2015.

[3] S. Bandyopadhyay, N. Lokesh, and M. N. Murty. Outlier aware network embedding for attributed networks. In *Proceedings of the AAAI conference*, volume 33, pages 12–19, 2019.

[4] S. Bandyopadhyay, S. V. Vivek, and M. Murty. Outlier resistant unsupervised deep architectures for attributed network embedding. In *Proceedings of the WSDM*, pages 25–33, 2020.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[6] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *Proceedings of ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[7] G. O. Campos, A. Zimek, J. Sander, R. J. Campello, B. Micenková, E. Schubert, I. Assent, and M. E. Houle. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data mining and knowledge discovery*, 30(4):891–927, 2016.

[8] D. Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *European conference on principles of data mining and knowledge discovery*, pages 112–124. Springer, 2004.

[9] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3438–3445, 2020.

[10] J. Chen, S. Sathe, C. Aggarwal, and D. Turaga. Outlier detection with autoencoder ensembles. In *Proceedings of the 2017 SIAM international conference on data mining*, pages 90–98. SIAM, 2017.

[11] Z. Chen, L. Chen, S. Villar, and J. Bruna. Can graph neural networks count substructures? *Advances in neural information processing systems*, 33:10383–10395, 2020.

[12] Z. Chen, B. Liu, M. Wang, P. Dai, J. Lv, and L. Bo. Generative adversarial attributed network anomaly detection. In *Proceedings of the ACM CIKM*, pages 1989–1992, 2020.

[13] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y. Zheng. Nus-wide: a real-world web image database from national university of singapore. In *Proceedings of the ACM international conference on image and video retrieval*, pages 1–9, 2009.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[15] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[16] K. Ding, J. Li, N. Agarwal, and H. Liu. Inductive anomaly detection on attributed networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 1288–1294, 2021.

[17] K. Ding, J. Li, R. Bhanushali, and H. Liu. Deep anomaly detection on attributed networks. In *Proceedings of the SDM*, pages 594–602. SIAM, 2019.

[18] M. Ding, K. Kong, J. Li, C. Zhu, J. Dickerson, F. Huang, and T. Goldstein. Vq-gnn: A universal framework to scale up graph neural networks using vector quantization. *Advances in Neural Information Processing Systems*, 34, 2021.

[19] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the ACM CIKM*, pages 315–324, 2020.

[20] A. Emmott, S. Das, T. Dietterich, A. Fern, and W.-K. Wong. A meta-analysis of the anomaly detection problem. *arXiv preprint arXiv:1503.01158*, 2015.

[21] H. Fan, F. Zhang, and Z. Li. Anomalydae: Dual autoencoder for anomaly detection on attributed networks. In *Proceedings of the IEEE ICASSP*, pages 5685–5689, 2020.

[22] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[23] S. Freitas, Y. Dong, J. Neil, and D. H. Chau. A large-scale database for graph representation learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[24] A. Gaddam, T. Wilkin, M. Angelova, and J. Gaddam. Detecting sensor faults, anomalies and outliers in the internet of things: A survey on the challenges and solutions. *Electronics*, 9(3):511, 2020.

[25] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

[26] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[27] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *KDD*, 2016.

[28] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22118–22133. Curran Associates, Inc., 2020.

[29] V. N. Ioannidis, D. Berberidis, and G. B. Giannakis. Unveiling anomalous nodes via random sampling and consensus on graphs. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5499–5503. IEEE, 2021.

[30] C. I. Jerez, J. Zhang, and M. R. Silva. On equivalence of anomaly detection algorithms. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2022.

[31] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of the MLSys*, 2:187–198, 2020.

[32] D. Kim and A. Oh. How to find your friendly neighborhood: Graph attention design with self-supervision. *arXiv preprint arXiv:2204.04879*, 2022.

[33] S. Kim, Y.-C. Tsai, K. Singh, Y. Choi, E. Ibok, C.-T. Li, and M. Cha. Date: Dual attentive tree-aware embedding for customs fraud detection. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2880–2890, 2020.

[34] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[35] T. N. Kipf and M. Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[36] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the ICLR*, 2017.

[37] D. Koutra, T.-Y. Ke, U. Kang, D. H. P. Chau, H.-K. K. Pao, and C. Faloutsos. Unifying guilt-by-association approaches: Theorems and fast algorithms. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 245–260. Springer, 2011.

[38] H.-P. Kriegel, P. Kroger, E. Schubert, and A. Zimek. Interpreting and unifying outlier scores. In *Proceedings of the SDM*, pages 13–24. SIAM, 2011.

[39] S. Kumar, X. Zhang, and J. Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1269–1278, 2019.

[40] K.-H. Lai, D. Zha, J. Xu, Y. Zhao, G. Wang, and X. Hu. Revisiting time series outlier detection: Definitions and benchmarks. In *Advances in neural information processing systems*, 2021.

[41] G. Li, M. Müller, B. Ghanem, and V. Koltun. Training graph neural networks with 1000 layers. In *International conference on machine learning*, pages 6437–6449. PMLR, 2021.

[42] J. Li, H. Dani, X. Hu, and H. Liu. Radar: Residual analysis for anomaly detection in attributed networks. In *IJCAI*, pages 2152–2158, 2017.

[43] Z. Li, Y. Zhao, X. Hu, N. Botta, C. Ionescu, and G. Chen. Ecod: Unsupervised outlier detection using empirical cumulative distribution functions. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2022.

[44] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):1–39, 2012.

[45] K. Liu, Y. Dou, Y. Zhao, X. Ding, X. Hu, R. Zhang, K. Ding, C. Chen, H. Peng, K. Shu, et al. PyGOD: A python library for graph outlier detection. *arXiv preprint arXiv:2204.12095*, 2022.

[46] Y. Liu, Z. Li, S. Pan, C. Gong, C. Zhou, and G. Karypis. Anomaly detection on attributed networks via contrastive self-supervised learning. *IEEE transactions on neural networks and learning systems*, 2021.

[47] X. Ma, J. Wu, S. Xue, J. Yang, C. Zhou, Q. Z. Sheng, H. Xiong, and L. Akoglu. A comprehensive survey on graph anomaly detection with deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[48] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 43–52, 2015.

[49] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *Proceedings of the ICML Workshop on Graph Representation Learning and Beyond*, pages 1–11, 2020.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Proceedings of the NeurIPS*, 32, 2019.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[52] Z. Peng, M. Luo, J. Li, H. Liu, and Q. Zheng. Anomalous: A joint modeling approach for anomaly detection on attributed networks. In *IJCAI*, pages 3513–3519, 2018.

[53] J. W. Pennebaker, M. E. Francis, and R. J. Booth. Linguistic inquiry and word count: Liwc 2001. *Mahway: Lawrence Erlbaum Associates*, 71(2001):2001, 2001.

[54] L. Perini, V. Vercruyssen, and J. Davis. Quantifying the confidence of anomaly detectors in their example-wise predictions. In *Proceedings of ECML-PKDD*, pages 227–243. Springer, 2020.

[55] C. Qiu, M. Kloft, S. Mandt, and M. Rudolph. Raising the bar in graph-level anomaly detection, 2022.

[56] S. Rayana and L. Akoglu. Collective opinion spam detection: Bridging review networks and metadata. In *KDD*, 2015.

[57] M. Sakurada and T. Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA workshop on machine learning for sensory data analysis*, pages 4–11, 2014.

[58] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[59] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[60] G. Steinbuss and K. Böhm. Benchmarking unsupervised outlier detection with realistic synthetic data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(4):1–20, 2021.

[61] H. Tong and C.-Y. Lin. Non-negative residual matrix factorization with application to graph anomaly detection. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 143–153. SIAM, 2011.

[62] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *ICLR*, 2017.

[63] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[64] H. Wang, C. Zhou, J. Wu, W. Dang, X. Zhu, and J. Wang. Deep structure learning for fraud detection. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 567–576. IEEE, 2018.

[65] X. Wang, B. Jin, Y. Du, P. Cui, Y. Tan, and Y. Yang. One-class graph neural networks for anomaly detection in attributed networks. *Neural computing and applications*, 33(18):12073–12085, 2021.

[66] Y. Wang, J. Zhang, S. Guo, H. Yin, C. Li, and H. Chen. Decoupling representation learning and classification for gnn-based anomaly detection. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1239–1248, 2021.

[67] X. Xu, N. Yuruk, Z. Feng, and T. A. Schweiger. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 824–833, 2007.

[68] Z. Xu, X. Huang, Y. Zhao, Y. Dong, and J. Li. Contrastive attributed network anomaly detection with data augmentation. In *Proceedings of the PAKDD*, 2022.

[69] J. Ye and L. Akoglu. Discovering opinion spammer groups by network footprints. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 267–282. Springer, 2015.

[70] X. Yu, Z. Liu, Y. Fang, and X. Zhang. Count-gnn: Graph neural networks for subgraph isomorphism counting. *under review*, 2021.

[71] X. Yuan, N. Zhou, S. Yu, H. Huang, Z. Chen, and F. Xia. Higher-order structure based anomaly detection on attributed networks. In *Proceedings of the IEEE Big Data Conference*, pages 2691–2700, 2021.

[72] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[73] G. Zhang, Z. Li, J. Huang, J. Wu, C. Zhou, J. Yang, and J. Gao. efraudcom: An e-commerce fraud detection system via competitive graph neural networks. *ACM Transactions on Information Systems (TOIS)*, 40(3):1–29, 2022.

[74] G. Zhang, J. Wu, J. Yang, A. Beheshti, S. Xue, C. Zhou, and Q. Z. Sheng. Fraudre: Fraud detection dual-resistant to graph inconsistency and imbalance. In *2021 IEEE International Conference on Data Mining (ICDM)*, pages 867–876. IEEE, 2021.

[75] L. Zhao and L. Akoglu. On using classification datasets to evaluate graph outlier detection: Peculiar observations and new insights. *Big Data*, 2021.

[76] T. Zhao, C. Deng, K. Yu, T. Jiang, D. Wang, and M. Jiang. Error-bounded graph anomaly loss for gnns. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 1873–1882, 2020.

[77] T. Zhao, T. Jiang, N. Shah, and M. Jiang. A synergistic approach for graph anomaly detection with pattern mining and feature learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[78] Y. Zhao, G. H. Chen, and Z. Jia. TOD: Tensor-based outlier detection. *arXiv preprint arXiv:2110.14007*, 2021.

[79] Y. Zhao, X. Hu, C. Cheng, C. Wang, C. Wan, W. Wang, J. Yang, H. Bai, Z. Li, C. Xiao, Y. Wang, Z. Qiao, J. Sun, and L. Akoglu. SUOD: accelerating large-scale unsupervised heterogeneous outlier detection. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[80] Y. Zhao, Z. Nasrullah, and Z. Li. PyOD: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7, 2019.

[81] Y. Zhao, R. Rossi, and L. Akoglu. Automatic unsupervised outlier model selection. *Proceedings of the NeurIPS*, 34:4489–4502, 2021.

[82] Q. Zheng, X. Zou, Y. Dong, Y. Cen, D. Yin, J. Xu, Y. Yang, and J. Tang. Graph robustness benchmark: Benchmarking the adversarial robustness of graph machine learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[83] C. Zhou and R. C. Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 665–674, 2017.

[84] Y. Zhu, Y. Xu, Q. Liu, and S. Wu. An empirical study of graph contrastive learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

A Details on UNOD

A.1 Additional Dataset Information

A.1.1 Real-world Data

The **Cora** dataset [58] is a citation graph with machine learning papers representing nodes and edges representing papers' citation relationship. The node feature is a sparse bag-of-words (BoW) vector extracted from the paper document, and its label represents one of seven classes it belongs to.

The **Amazon** dataset [59] is a segment of the Amazon co-purchase graph [48], where nodes represent goods, edges indicate that two goods are frequently bought together, node features are BoW encoded product reviews, and class labels are given by the product category.

The **Flickr** dataset [72] originates from NUS-wide[13]. The SNAP website[1] collected Flickr data from four different sources including NUS-wide, and generated an undirected graph. One node in the graph represents one image uploaded to Flickr. If two images share some common properties (e.g., same geographic location, same gallery, comments by the same user, etc.), there is an edge between the nodes of these two images. The node feature is composed of a 500-dimensional BoW vector of the images provided by NUS-wide. Eighty-one tags of each image are manually merged into seven classes and each image belongs to one of the seven classes.

The **Weibo** dataset [76] is a user-posts-hashtag graph from a Tencent-Weibo, a Twitter-like platform. It has 8,405 users and 61,964 hashtags. We use the user-user graph [2] provided by the author, which connects users who used the same hashtag. Temporal information was used to label the users. If a user made at least five suspicious events, he/she is labeled as a suspicious user; if no suspicious event was made, he/she is a benign user. There are a total of 868 suspicious users and 7,537 benign users. The suspicious users are regarded as outliers in the graph. Since the ground truth was generated using time information, the timestamps are not used to create raw user features. Therefore, the raw feature vector has two parts: (i) For each user, the one-hot vectors of his/her posts are summed where each hot represents the location where a post was made. Then the #dimensions of the summed vector are reduced to 100 using SVD. (ii) For each user, the #dimensions of BoW vectors extracted from post texts are reduced to 300. The final node feature is the concatenation of the location vector and BoW vector. Note that Weibo is a directed graph; the remaining datasets used in our benchmark are undirected graphs.

The **Reddit** dataset [39, 66] is a user-subreddit graph extracted from a social media platform Reddit [3]. This public dataset consists of one month of user posts on subreddits [4]. The 1,000 most active subreddits and the 10,000 most active users are extracted as subreddit nodes and user nodes, respectively. This results in 168,016 interactions. Each user has a binary label indicating whether it has been banned by the platform. We assume that the banned users are outliers compared to normal Reddit users. The text of each post is converted into a feature vector representing their LIWC categories [53] and the features of user and subreddit are the feature summation of the posts they have, respectively.

A.1.2 Random Graph Generation Method

We leverage a random graph generation method used in [32]. Specifically, the implementation in PyG [5] is used with 2 classes, node_homophily_ratio=0.5, average_degree=5 and num_channels=64. We use the generated random graphs to benchmark algorithms' efficiency and scalability. We generate random graph **Gen_Time** with num_nodes_per_class=500 (1000 in total) as the graph data to test the runtime. To benchmark the scalability, we generate multiple random graphs **Gen_100**, **Gen_500**, **Gen_1000**, **Gen_5000** and **Gen_10000** with num_nodes_per_class equal to 50, 250, 500, 2500 and 5000, respectively.

A.1.3 Outlier Injection Details

The outliers are injected into the real-world graphs without organic outliers (i.e., Cora, Amazon, Flickr) and the generated random graphs.

Structural Outlier. We randomly select m nodes from the network and then make those nodes fully connected, and then all the m nodes in the clique are regarded as outliers. We iteratively repeat this process until a number of n cliques are generated and the total number of structural outliers is $m \times n$.

[1] http://snap.stanford.edu/

[2] https://github.com/zhao-tong/Graph-Anomaly-Loss/tree/master/data/weibo_s

[3] https://www.reddit.com/

[4] http://files.pushshift.io/reddit/

[5] https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html#torch_geometric.datasets.RandomPartitionGraphDataset

Contextual Outlier. To inject the same amount of contextual outliers, we randomly select another $m \times n$ nodes as the attribute perturbation candidates. For each selected node i, we randomly pick another m nodes from the data and select the node j whose attributes deviate the most from node i among the m nodes by maximizing the Euclidean distance $||x_i - x_j||^2$. Afterward, we change the attributes x_i of node i to x_j.

The parameters used in outlier injection is shown in Table 5. The m is set approximately as twice as the degree of the graphs. For real-world datasets, we keep similar outlier ratio (i.e., approximately 5%). For generated graph datasets of various size, we keep the similar number of outliers. The statistics of the generated graphs are shown in Table 6.

Table 5: Parameters used in outliers injection.

	Cora	Amazon	Flickr	Gen_Time	Gen_100	Gen_500	Gen_1000	Gen_5000	Gen_10000
Degree	4.1	37.5	10.6	5	5	5	5	5	5
n	70	350	2240	10	1	1	1	1	1
m	10	70	20	10	10	10	10	10	10

Table 6: Statistics of generated datasets in UNOD.

Dataset	#Nodes	#Edges	#Feat.	Degree	#Con.	#Strct.	#Outliers	Ratio
Gen_Time	1,000	5,746	64	5.7	100	100	189	18.9%
Gen_100	100	618	64	6.2	10	10	18	18.0%
Gen_500	500	2,662	64	5.3	10	10	20	4.0%
Gen_1000	1,000	4,936	64	4.9	10	10	20	2.0%
Gen_5000	5,000	24,938	64	5.0	10	10	20	0.4%
Gen_10000	10,000	49,614	64	5.0	10	10	20	0.2%

A.2 Description of algorithms in the benchmark

LOF [5]. LOF is short for the Local Outlier Factor. LOF computes the degree of an object is an outlier and the degree depends on how isolated the object is with respect to its surrounding neighborhood. Note that LOF only uses node attribute information and the neighborhood is composed of k-nearest-neighbors.

IF [44]. Isolation Forest (IF) is a classic tree ensemble method used in outlier detection. It builds an ensemble of base trees to isolate the data points and defines the decision boundary as the closeness of an individual instance to the root node of the tree. It only uses node attributes of data.

MLPAE [57]. The MLPAE is a vanilla autoencoder with MLPs as encoder and decoder. The encoder takes the node attribute as the input to learn its low-dimensional embedding and a decoder reconstructs the input node attribute from the node embedding. The outlier score of a node is the reconstruction error of the decoder.

SCAN [67]. SCAN is a structural clustering algorithm to detect clusters, hub nodes, and node outliers in a graph. Since the structural outliers exhibit clustering patterns on graphs, we use SCAN to detect clusters and the nodes in detected clusters are regarded as structural outliers in the graph. SCAN only takes the graph structure as the input.

Radar [42]. Radar is an anomaly detection framework for attributed graphs. It takes the graph structure and node attributes as the input. It detects node anomalies whose behaviors are singularly different from the majority by characterizing the residuals of attribute information and its coherence with network information. The outlier score of a node is decided by the norm of its reconstruction residual.

ANOMALOUS [52]. ANOMALOUS performs joint anomaly detection and attribute selection to detect node anomalies on attributed graphs based on the CUR decomposition and residual analysis. It takes the graph structure and node attribute as the input, and the outlier score of a node is decided by the norm of its reconstruction residual.

GCNAE [35]. GCNAE is the autoencoder framework with GCNs [36] as the encoder and decoder. It takes the graph structure and node attributes as input. The encoder is used to learn a node's embedding by aggregating its neighbor information. The decoder reconstructs the node attribute by applying another GCN to node embeddings and graph structures. Similar to MLPAE, the outlier score of a node is the reconstruction error of the decoder.

DOMINANT [17]. DOMINANT is one of the first works that leverage GCN and AE for node outlier detection. It uses a two-layer GCN as the encoder, a two-layer GCN decoder to reconstruct the node attribute, and an one-layer GCN and dot product as the structural decoder to reconstruct the graph adjacency matrix. The reconstruction errors of both decoders are combined as the node outlier score.

DONE [4]. DONE leverages a structural and an attribute AE to reconstruct the adjacency matrix and node attribute, respectively. The encoders and decoders are composed of MLPs. The node embeddings and outlier scores are optimized simultaneously with a unified loss function.

AdONE [4]. AdONE is a variant of DONE, which uses an extra discriminator to discriminate the learned structure embedding and attribute embedding of a node. The adversarial training approach supposes to better align the two different embeddings in the latent space.

AnomalyDAE [21]. AnomalyDAE also utilizes a structure AE and attribute AE to detect node outliers. The structure encoder of AnomalyDAE takes both adjacency matrix and node attribute as input; the attribute decoder reconstructs the node attribute using both structure and attribute embeddings.

GAAN [12]. GAAN is a GAN-based node outlier detection method. It employs an MLP-based generator to generate fake graphs and an MLP-based encoder to encode graph information. A discriminator is trained to recognize whether two connected nodes are from the real or fake graph. The outlier score is obtained by the node reconstruction error and real-node identification confidence.

GUIDE [71]. GUIDE is similar to DONE and AdONE with two different AEs, but it pre-processes the structure information before feeding it into the structure encoder. Specifically, node motif degree is used to represent the node structure vector which could encode high-order structure information.

CONAD [68]. CONAD is one of the UNOD methods that leverage the graph augmentation and contrastive learning technique. It imposes prior knowledge of node outliers via generating augmented graphs. After encoding the graphs using Siamese GNN encoders, the contrastive loss is used to optimize the encoder, and the node outlier score is obtained by two different decoders like DOMINANT.

A.3 Description of Evaluation Metrics

ROC-AUC (AUC). AUC computes the Area Under the Receiver Operating Characteristic Curve (ROC AUC) from predicted outlier scores. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. In the UNOD benchmark, we regard the outlier nodes as the positive class and compute the AUC for it. AUC equals 1 means the model makes a perfect prediction, and AUC equals 0.5 means the model has no class-separation capability. AUC is better than accuracy when evaluating the outlier detection task since it is not sensitive to the imbalanced class distribution of the data.

Recall@k. The outliers are usually rare in contrast to enormous normal samples in the data, and the outliers are of the most interest to outlier detection practitioners. We propose to use Recall@k to measure how well the detectors rank outliers over the normal samples. We set the k as the number of ground truth outliers in each dataset. The Recall@k is computed by the number true outliers among the top-k samples in the outlier ranking list divided by k. A higher Recall@k score indicates a better detection performance, and Recall@k equals 1 means the model perfectly ranks all outliers over the normal samples.

Average Precision (AP). AP summarizes the precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. AP is a metric that balances the effects of recall and precision, and a higher AP indicates the lower false-positive rate (FPR) and false-negative rate (FNR). For most outlier detection applications, FPR and FNR have equal importance as more misclassified normal samples could worsen legit users' experience.

Runtime. Due to the coverage of both classical algorithms and neural network methods, we consistently measure the model runtime as the duration between the experiment starts and ends, mimicking the real-world applications without specific differentiation between CPU and GPU time.

GPU Memory. Notably, GPU memory is often the bottleneck of machine learning algorithms due to its limitation in extension. In UNOD, we report the max active GPU memory for running an algorithm.

B Additional Experimental Settings and Details

Environment. The key libraries and their versions used in the experiment are as follows: Python=3.7, CUDA_version=11.1, torch=1.10, pytorch_geometric>=2.0.3, networkx=2.6.3, numpy=1.19.4, scipy=1.5.2, scikit-learn=0.22.1, pyod=1.0.1, pygod=0.3.0.

Hardware configuration. All the experiments were performed on A Linux server with a 3.50GHz Intel Core i5 CPU, 64GB RAM, and 1 NVIDIA GTX 1080 Ti GPU with 12GB memory.

More model implementation details. To include a large number of algorithms, we build Python Graph Outlier Detection (PyGOD)[1] [45], which provides more than 10 latest graph OD algorithms; all with unified APIs and optimizations. We tried our best to apply the same set of optimization techniques to each dataset. For Radar and

[1]PyGOD: `https://pygod.org/`

ANOMALOUS, we use gradient descent instead closed-form optimization provided in official implementation due to fairness and efficiency concern. For all deep algorithms, we implement sampling and minibatch training on large graphs (e.g., Flickr). See our library source code for more details. Meanwhile, we also include multiple non-graph baselines (LOF and IF) from our early work Python Outlier Detection (PyOD) [80].

Hyperparameter grid

The hyperparameter space is shown in Table 7. The candidates of hyperparameters are listed in square brackets. In each trial, a value is randomly chosen among candidates. The results (mean, std, max) are reported among 20 trials.

Due to the large graph size, full batch training on Flickr cannot fit in single GPU memory. Minibatch training and different batch size, sampling size, and the number of epochs are used on Flickr. Because of the complexity of real-world datasets (Weibo and Reddit), automated balancing by the standard deviation for weight alpha cannot balance well. Thus, three candidates are attempted. As Reddit has a lower feature dimension, we reduce hidden dimension values on Reddit.

Table 7: Hyperparameters in different algorithms. The values in "[]" are candidates. We present these common hyperparameters shared by multiple algorithms on the top, and also specify some algorithm-specific hyperparameters in the bottom. Refer to PyGOD doc for more details.

Algorithm	Hyperparameter	Cora	Amazon	Flickr	Weibo	Reddit	Gen
Common	*dropout*	[0, 0.1, 0.3]					
	learning rate	[0.1, 0.05, 0.01]					
	weight decay	0.01					
	batch size	full batch		64	full batch		
	sampling	all neigh.		3	all neigh.		
	epoch	300		2	300		
	alpha	auto			[0.8, 0.5, 0.2]		auto
	hid. dim.	[32, 64, 128, 256]				[32, 48, 64]	
SCAN	*eps*	[0.3, 0.5, 0.8]					
	mu	[2, 5, 10]					
AnomalyDAE	*theta*	[10, 40, 90]					
	eta	[3, 5, 8]					
GAAN	*noise dim.*	[8, 16, 32]					
GUIDE	*struct. hid.*	[4, 5, 6]					

How we determine the optimal performance in runtime comparison.

The optimal performance is determined by the ROC-AUC score. Taking the computational cost into account, we expect a reasonable score within as few training epochs as possible. Thus, when the score converges (i.e., the score increment of consequence epochs is less than 0.5%), we mark the current epoch as optimal.

C Additional Experimental Results

C.1 Additional Results on Real-world Datasets Detection Performance

Table 8: Average Precision (%) comparison among OD algorithms on five real-world benchmark datasets, where we show *the avg perf.* ± *the STD of perf.* (*max perf.*) of each. OOM denotes out of memory with regard to GPU (_G) and RAM (_R).

Algorithm	Cora	Amazon	Flickr	Weibo	Reddit
LOF	12.2±0.0 (12.2)	5.6±0.0 (5.6)	5.4±0.0 (5.4)	15.8±0.0 (15.8)	4.2±0.0 (4.2)
IF	10.1±0.7 (11.5)	6.2±1.3 (9.5)	8.2±0.6 (9.3)	12.9±2.6 (19.8)	2.8±0.1 (2.9)
SCAN	13.2±0.0 (13.2)	34.8±0.0 (34.9)	18.7±0.0 (18.7)	52.8±9.9 (64.5)	3.4±0.0 (3.4)
MLPAE	17.9±8.2 (34.0)	13.9±9.9 (33.6)	32.6±20.5 (50.7)	17.3±3.4 (20.5)	3.3±0.0 (3.3)
Radar	7.5±0.4 (7.9)	13.5±1.2 (14.2)	OOM_G	92.1±0.7 (92.9)	3.6±0.2 (3.9)
ANOMALOUS	7.1±2.2 (12.3)	11.8±1.7 (15.6)	OOM_G	92.1±0.7 (92.9)	4.0±0.6 (5.1)
GCNAE	13.2±0.0 (13.2)	34.8±0.0 (34.8)	16.8±4.3 (18.7)	70.8±5.0 (80.9)	3.4±0.0 (3.4)
DOMINANT	20.7±5.8 (24.1)	18.1±2.2 (19.5)	40.6±8.6 (44.8)	18.0±10.2 (36.2)	3.7±0.0 (3.8)
DONE	24.0±8.9 (40.9)	29.4±10.3 (51.2)	21.8±4.2 (25.8)	65.5±13.4 (77.3)	3.7±0.4 (4.5)
AdONE	19.3±4.8 (25.4)	25.5±11.4 (46.9)	18.6±4.7 (27.1)	62.9±9.5 (74.4)	3.3±0.4 (4.0)
AnomalyDAE	19.1±3.1 (25.2)	28.7±5.8 (34.7)	13.3±4.6 (19.0)	38.5±22.5 (77.3)	3.7±0.1 (3.8)
GAAN	14.9±0.1 (15.3)	35.2±0.4 (35.9)	18.6±0.1 (18.7)	80.3±0.2 (80.7)	3.7±0.1 (3.9)
GUIDE	14.0±0.2 (14.3)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	20.9±5.8 (24.2)	18.2±0.8 (19.4)	8.0±1.9 (9.7)	15.6±6.9 (31.7)	3.7±0.3 (4.6)

Table 9: Recall@k (%) comparison among OD algorithms on five real-world benchmark datasets, where we show *the avg perf.* ± *the STD of perf.* (*max perf.*) of each. k is set as the number of outliers in labels. OOM denotes out of memory with regard to GPU (_G) and RAM (_R).

Algorithm	Cora	Amazon	Flickr	Weibo	Reddit
LOF	15.2±0.0 (15.2)	5.2±0.0 (5.2)	8.4±0.0 (8.4)	22.0±0.0 (22.0)	4.4±0.0 (4.4)
IF	14.6±1.7 (18.1)	5.0±2.0 (9.9)	12.6±1.2 (14.7)	13.8±6.4 (24.3)	0.1±0.1 (0.3)
SCAN	18.8±0.0 (18.8)	45.0±0.0 (45.1)	28.0±0.1 (28.1)	48.9±11.0 (62.1)	3.0±0.0 (3.0)
MLPAE	27.0±11.5 (42.8)	18.2±13.1 (37.6)	35.4±22.1 (52.0)	23.8±7.0 (30.5)	2.7±0.3 (3.0)
Radar	3.1±0.3 (3.6)	21.7±3.7 (24.3)	OOM_G	86.4±0.8 (87.4)	2.1±0.8 (3.5)
ANOMALOUS	5.7±4.3 (15.9)	14.3±5.8 (25.7)	OOM_G	86.4±0.8 (87.4)	4.0±1.9 (7.9)
GCNAE	18.8±0.0 (18.8)	45.0±0.0 (45.1)	24.5±8.2 (28.1)	67.6±5.2 (77.3)	3.0±0.0 (3.0)
DOMINANT	23.5±7.4 (28.3)	22.1±3.4 (24.3)	49.8±10.1 (52.8)	19.7±13.8 (37.4)	0.9±0.4 (2.7)
DONE	28.1±10.8 (41.3)	33.6±11.7 (52.2)	25.2±4.9 (29.4)	65.4±12.4 (76.3)	2.8±1.6 (5.7)
AdONE	19.8±7.1 (31.2)	27.3±11.8 (52.7)	19.2±5.4 (28.6)	64.3±7.6 (74.3)	1.0±1.2 (3.8)
AnomalyDAE	22.1±3.1 (29.7)	36.5±9.4 (45.7)	17.1±9.9 (28.2)	42.2±23.7 (75.7)	0.9±0.5 (3.0)
GAAN	19.6±0.2 (20.3)	45.7±0.2 (46.2)	28.0±0.1 (28.2)	77.1±0.2 (77.4)	1.1±0.4 (2.2)
GUIDE	18.7±0.7 (19.6)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	24.8±6.7 (29.7)	22.8±0.9 (24.3)	10.7±4.3 (14.8)	20.3±13.3 (37.1)	1.3±1.6 (7.6)

C.2 Additional Results on Performance Variation under Different Types of Outliers

Table 10: Average Precision (%) comparison among OD algorithms on three datasets injected with contextual and structural outliers, where we show *the avg perf.* ± *the STD of perf.* (*max perf.*) of each. OOM denotes out of memory with regard to GPU (_G) and RAM (_R).

	Cora		**Amazon**		**Flickr**	
Algorithm	Contextual	Structural	Contextual	Structural	Contextual	Structural
LOF	12.2±0.0 (12.2)	3.1±0.0 (3.1)	3.2±0.0 (3.2)	2.6±0.0 (2.6)	3.5±0.0 (3.5)	2.5±0.0 (2.5)
IF	9.2±1.1 (11.7)	3.0±0.3 (3.6)	4.7±2.1 (10.5)	2.6±0.1 (2.7)	7.2±1.0 (9.1)	2.6±0.0 (2.6)
MLPAE	13.7±0.0 (13.7)	3.1±0.0 (3.1)	56.7±0.1 (56.9)	2.5±0.0 (2.5)	24.8±0.0 (24.9)	2.6±0.0 (2.6)
SCAN	2.6±0.0 (2.6)	27.8±16.9 (61.0)	2.5±0.0 (2.5)	19.8±20.1 (60.0)	2.5±0.0 (2.5)	58.3±41.2 (94.8)
Radar	2.5±0.0 (2.6)	6.0±0.8 (6.6)	13.0±2.1 (14.3)	3.4±0.1 (3.6)	OOM_G	OOM_G
ANOMALOUS	2.7±0.2 (3.3)	5.6±3.0 (14.0)	10.0±1.4 (13.7)	3.5±0.5 (4.5)	OOM_G	OOM_G
GCNAE	13.7±0.0 (13.7)	3.1±0.0 (3.1)	56.7±0.0 (56.8)	2.5±0.0 (2.5)	21.7±7.2 (24.9)	2.6±0.0 (2.6)
DOMINANT	6.3±1.8 (10.9)	18.1±5.9 (22.2)	4.8±0.4 (5.2)	16.5±2.5 (17.4)	4.6±0.5 (5.0)	59.3±13.9 (66.0)
DONE	6.8±1.8 (9.3)	23.2±11.9 (49.6)	11.1±7.1 (24.0)	26.1±19.9 (77.2)	14.9±2.7 (17.4)	9.5±2.1 (11.8)
AdONE	7.7±2.1 (12.0)	14.7±4.6 (23.1)	14.4±6.9 (25.1)	15.8±12.8 (50.0)	10.5±2.8 (17.1)	9.7±2.8 (12.8)
AnomalyDAE	11.2±2.3 (13.4)	10.6±6.0 (24.6)	27.0±17.5 (48.6)	10.3±5.5 (17.3)	13.8±7.5 (24.1)	3.1±0.3 (4.0)
GAAN	14.1±0.1 (14.2)	4.4±0.1 (4.6)	51.8±0.8 (52.9)	3.7±0.1 (3.9)	24.7±0.2 (24.9)	2.6±0.0 (2.6)
GUIDE	13.3±1.9 (14.0)	4.1±1.3 (9.8)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	6.1±1.2 (6.9)	18.8±6.0 (22.9)	4.7±0.3 (5.2)	16.8±0.7 (17.4)	3.8±0.8 (4.7)	4.5±1.2 (5.6)

Table 11: Recall@k (%) comparison among OD algorithms on three datasets injected with contextual and structural outliers, where we show *the avg perf.* ± *the STD of perf.* (*max perf.*) of each. k is set as the number of each types of outliers in labels. OOM denotes out of memory with regard to GPU (_G) and RAM (_R).

	Cora		**Amazon**		**Flickr**	
Algorithm	Contextual	Structural	Contextual	Structural	Contextual	Structural
LOF	12.9±0.0 (12.9)	5.7±0.0 (5.7)	1.1±0.0 (1.1)	3.7±0.0 (3.7)	9.0±0.0 (9.0)	2.2±0.0 (2.2)
IF	13.1±3.3 (20.0)	3.8±2.0 (8.6)	4.4±3.5 (13.4)	2.1±0.7 (3.1)	13.8±1.9 (17.7)	2.5±0.3 (3.0)
MLPAE	15.7±0.0 (15.7)	7.1±0.0 (7.1)	55.1±0.1 (55.1)	2.3±0.0 (2.3)	29.8±0.1 (30.0)	2.9±0.0 (2.9)
SCAN	2.4±1.1 (4.3)	32.4±18.3 (61.4)	2.3±0.4 (2.9)	24.7±26.0 (70.3)	2.8±0.2 (3.1)	59.0±41.7 (96.4)
Radar	1.4±0.0 (1.4)	0.4±0.6 (1.4)	16.0±5.7 (20.6)	1.7±1.0 (2.3)	OOM_G	OOM_G
ANOMALOUS	1.9±1.3 (4.3)	2.1±2.9 (11.4)	5.2±4.9 (19.7)	0.7±1.8 (6.6)	OOM_G	OOM_G
GCNAE	15.7±0.0 (15.7)	7.1±0.0 (7.1)	55.1±0.0 (55.1)	2.3±0.0 (2.3)	25.8±9.7 (30.0)	2.8±0.2 (2.9)
DOMINANT	8.4±3.4 (15.7)	15.9±4.6 (18.6)	5.0±0.8 (6.0)	10.0±1.7 (10.9)	3.6±0.3 (4.0)	69.3±15.8 (75.6)
DONE	9.5±3.1 (15.7)	22.8±13.4 (45.7)	11.4±8.0 (26.3)	26.4±20.5 (75.1)	22.2±5.1 (25.6)	5.2±1.4 (7.7)
AdONE	10.6±3.3 (17.1)	12.3±7.8 (28.6)	17.1±8.4 (33.1)	13.3±15.9 (57.1)	15.8±4.3 (24.6)	3.8±1.4 (6.9)
AnomalyDAE	14.0±3.0 (17.1)	12.4±5.4 (24.3)	29.4±21.3 (53.4)	8.4±3.2 (12.9)	16.2±12.0 (30.1)	3.6±1.3 (6.7)
GAAN	15.8±0.3 (17.1)	8.4±0.5 (8.6)	54.5±0.5 (56.0)	3.8±0.3 (4.3)	29.8±0.1 (30.0)	2.9±0.0 (2.9)
GUIDE	15.9±3.4 (17.1)	8.4±0.4 (8.6)	OOM_R	OOM_R	OOM_R	OOM_R
CONAD	7.7±3.1 (10.0)	16.1±4.8 (18.6)	4.9±0.5 (6.0)	10.6±1.0 (14.6)	5.6±2.5 (9.5)	6.5±2.9 (9.4)

C.3 Additional Results on Efficiency and Scalability Analysis

Table 12: Time consumption (s) comparison among OD algorithms on five different numbers of epochs. For non-iterative algorithms, i.e., LOF, IF, and SCAN, we report the total runtime.

Algorithm	10	100	200	300	400
LOF	0.10	0.10	0.10	0.10	0.10
IF	0.09	0.09	0.09	0.09	0.09
MLPAE	0.04	0.46	0.82	1.37	1.74
SCAN	0.02	0.02	0.02	0.02	0.02
Radar	0.02	0.10	0.19	0.34	0.36
ANOMALOUS	0.02	0.09	0.17	0.26	0.36
GCNAE	0.06	0.49	0.96	1.45	1.94
DOMINANT	0.08	0.70	1.41	2.10	2.79
DONE	0.08	0.77	1.53	2.30	3.08
AdONE	0.10	0.91	1.81	2.71	3.62
AnomalyDAE	0.43	0.64	1.28	1.92	2.55
GAAN	0.06	0.49	0.98	1.47	1.97
GUIDE	50.77	51.92	53.40	54.27	55.21
CONAD	0.11	1.04	2.07	3.07	4.10

Table 13: GPU memory consumption (MB) comparison among deep algorithms on five different graph size (number of nodes). Note that GPU memory measurement does not apply to algorithms like LOF, IF, and SCAN.

Algorithm	100	500	1000	5000	10000
MLPAE	0.66	2.10	3.90	19.41	38.52
GCNAE	0.92	3.40	6.38	31.18	62.11
GUIDE	0.96	3.46	6.47	31.45	62.60
Radar	0.49	9.32	36.73	871.32	3450.88
ANOMALOUS	0.44	5.03	20.50	482.47	2293.76
DOMINANT	1.09	7.58	27.15	591.74	2324.48
DONE	1.95	10.93	32.74	624.41	2385.92
AdONE	1.99	11.38	33.60	657.54	2447.36
AnomalyDAE	1.21	10.54	36.94	794.81	3112.96
GAAN	0.90	9.51	36.87	871.17	3450.88
CONAD	1.39	8.77	29.48	604.32	2344.96

D PyGOD Library

D.1 User Guidelines for `PyGOD`

Dependency. PyGOD builds for Python 3.6+, and depends on popular `Pytorch` [50] and `Pytorch Geometric (PyG)` [22] for effective graph learning on both CPUs and GPUs. Additionally, it uses `numpy` [26], `scipy` [63], and `scikit-learn` [51] for data manipulation.

API Design. Inspired by the API design of `scikit-learn` [6] and `PyOD` [80], all detection algorithms in PyGOD inherit from a base class with the same API interface: (*i*) `fit` trains the detection model, gets the outlier scores (the higher, the more outlying) and labels on the input data, and generates necessary statistics for prediction (in the inductive setting); (*ii*) `decision_function` leverages the trained model to predict the raw outlier scores of the incoming data (in the inductive setting); (*iii*) `predict` returns the binary prediction using the trained model (0 for normal samples and 1 for outliers); and (*iv*) `predict_proba` returns the probability of samples being outlier using the leading method [38]. Meanwhile, we also include the recent advance in providing the confidence score in outlier detection [54] for the above prediction methods. The usage of the above APIs is demonstrated in Code demo 1.

```
from pygod.utils import load_data            # import loader
data = load_data("inj_cora")                 # load data

from pygod.models import DOMINANT            # import the model
model = DOMINANT()                           # init. detection model
model.fit(data)                              # fit with data

pred_label = model.predict(data)             # predict binary labels
pred_score = model.decision_function(data)   # predict outlier scores
pred_proba = model.predict_proba(data)       # predict probabilities

from pygod.metrics import eval_roc_auc       # import eval. func.
eval_roc_auc(y, pred_score)                  # eval. by AUC
```

Code demo 1: Using Dominant [17] on Cora data [49]

Streamlined Graph Learning with PyG. We choose to develop PyGOD on top of the popular `PyG` Llibrary for multiple reasons. First, this reduces the complexity in processing graph data for users. That is, PyGOD only requires the input data to be in the standard graph data format in PyG[1]. Notably, different detection models need distinct information from a `PyG` graph. Within the implementation of each detection model, we design an abstract `process_graph` method to extract necessary information, e.g., the adjacency matrix, node, and edge attributes, etc., for the underlying detection algorithm. Second, most of the detection algorithms share common backbones (see Table 2) like graph convolutional neural networks (GCN) [36] and graph autoencoders [35], where PyG already provides optimized implementation. Third, PyG is the most popular GNN libraries with advanced functions like graph sampling and distributed training. Under the PyG framework, we enable and implement mini-batch and/or sampling for selected models to accommodate the learning with large graphs as shown in Table 2.

In addition to the detection models, a set of helpful utility functions is designed to facilitate graph outlier detection. Evaluation-wise, PyGOD provides common metrics for graph OD in the `metric` module. Data-wise, PyGOD offers outlier injection methods for both structural and contextual settings [17] in `generator`, as a solution to model evaluation and benchmarking.

D.2 Maintenance Plan

The maintenance plan of PyGOD will focus on multiple aspects: (i) including more algorithms and benchmark for different sub-tasks, e.g., outlier detection in edges and sub-graphs; (ii) collaborating with industries to make PyGOD more practical and tailor the needs of practitioners; (iii) optimizing its accessibility and scalability with the latest advancement in graph [31]; and (iv) incorporate automated machine learning to enable intelligent model selection and hyperparameter tuning [81].

[1] PyG data: `https://pytorch-geometric.readthedocs.io/en/latest/modules/data.html`