Does not show a single digram of input-and-output for the given problem
No algorithm, no architecture

Graph Summarization as Vertex Classification Task using Graph Neural Networks vs. Bloom Filter

M. Blasi, M. Freudenreich, J. Horvath
Ulm University, Germany
{maximilian.blasi,manuel.freudenreich, johannes.horvath}@uni-ulm.de

D. Richerby
University of Essex, UK
david.richerby@essex.ac.uk

A. Scherp
Ulm University, Germany
ansgar.scherp@uni-ulm.de

***Abstract*—The goal of graph summarization is to represent large graphs in a structured and compact way. A graph summary based on equivalence classes preserves predefined features of a graph's vertex within a k-hop neighborhood such as the vertex labels and edge labels. Based on these neighborhood characteristics, the vertex is assigned to an equivalence class. The calculation of the assigned equivalence class must be a permutation invariant operation on the predefined features. This is achieved by sorting on the feature values, e. g., the edge labels, which is computationally expensive, and subsequently hashing the result. Graph Neural Networks (GNNs) fulfill the permutation invariance requirement. We formulate the problem of graph summarization as a subgraph classification task on the root vertex of the k-hop neighborhood. We adapt different GNN architectures, both based on the popular message-passing protocol and alternative approaches, to perform the structural graph summarization task. We compare different GNNs with a standard multi-layer perceptron (MLP) and Bloom filter as non-neural method. For our experiments, we consider four popular graph summary models on a large web graph. This resembles challenging multi-class vertex classification tasks with the numbers of classes ranging from 576 to multiple hundreds of thousands. Our results show that the performance of GNNs are close to each other. In three out of four experiments, the non-message-passing Graph-MLP model outperforms the other GNNs. The performance of a standard-MLP baseline is on par with the GNNs. However, the Bloom filter outperforms all neural architectures by a large margin, except for the dataset with the fewest number of 576 classes. This is an interesting result, since it sheds light on how well and in which contexts GNNs are suited for graph summarization. Furthermore, it demonstrates the need for considering strong non-neural baselines for standard GNN tasks such as vertex classification.**

Our source code is available at https://github.com/johorvath/Graph_Summarization_with_Graph_Neural_Networks

***Index Terms*—Graph learning, RDF, Bloom Filter**

I. Introduction

Graph summaries provide a condensed representation of an input graph. The goal is to preserve predefined features that are relevant for specific tasks such as queries on the summary [1]. Graph summaries are used for tasks such as estimating cardinalities [2]; data search [3], exploration [4], and visualization [5]; and vocabulary term recommendations [6]. Lossless summaries preserve all information needed for the desired task, allowing it to be computed exactly from the summary. In this paper, we consider graph summaries based on equivalence classes [1]. These classify vertices based on features such as vertices labels, edges, and neighbors to produce a lossless summary. With the recent rise of graph neural networks (GNNs), our contribution is to understand how well and in which context GNNs are suited to computing approximations of lossless graph summaries, and how they compare to Bloom filters, as a non-neural baseline.

In general, graph summarization can be understood as a function $f_{M,G}$ that assigns each vertex in the graph G to an equivalence class:

$$f_{M,G}\colon V(G) \mapsto V(SG)\,, \tag{1}$$

where $V(G)$ is the vertex set of G, and SG is the summary graph, whose vertices correspond to equivalence classes. The content of these equivalence classes is determined by the graph summary model M. Each model M considers different properties of neighboring vertices for the calculation. So the summary function maps a set of information gathered in the subgraph around a vertex v to a concise representation corresponding to the equivalence class. The aggregation of the information from a vertex's subgraph is required to be isomorphism-invariant, i. e., it must depend only on the structure of the graph itself, and not on, e. g., the order in which the edges are listed in the database. Figure 1 illustrates this requirement for isomorphism-invariance.

The calculation of the equivalence class can be done using deterministic computations like a hash function [3]. Because the hash value resulting from $f_{M,G}$ corresponds to an equivalence class, we can interpret it as a vertex label and apply a neural network such as a GNN for the classification. In other words, we cast the problem of graph summarization as a vertex classification task in GNNs. This is possible because a key design feature for GNNs is that they should be permutation invariant or at least equivariant [7], as required for this application. The use of neural networks as hash functions has so far mainly been researched in the context of security [8], [9]. There, the goal is to provide a one-way function that ensures big changes in the resulting hash, even when there are only small differences in the input. We discuss this in more detail in Section II-C.

We consider different GNN models, both based on the popular message-passing architecture, namely the graph convolutional network (GCN) [10], GraphSAGE [11], and GraphSAINT [12], and an approach that does not use message-

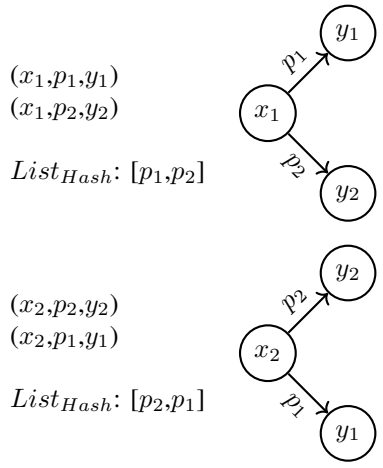


Fig. 1: The left-hand side shows two list-based representations of the two graphs on the right-hand side. While the two graphs are actually identical, the order of triples in the lists differ. In a naive approach of hashing the corresponding lists of triples, different hash values and thus different equivalence classes of the vertex x_1 and x_2 are computed. However, the two subgraphs should yield the same equivalence class for x_1 and x_2 based on the graph summary model Attribute Collection (see Section III-A) used in this example.

passing, namely Graph-MLP [13]. GCN applies a message-passing technique for vertex classification, based on the vertex embedding and edges. GraphSAGE applies functions with trainable weights for vertex classification instead of trainable weights directly on the vertex embeddings. GraphSAINT uses a sampler for batching and then a GCN defined on the dimension of the smaller batch. Graph-MLP removes the message passing technique and replaces it with a specific loss function and multi-layer perceptron (MLP) network. To apply the GNNs on large-scale datasets and avoid the general memory limitations of GNNs, we implemented a vertex sampler to generate mini-batches. This is similar to the NodeSampler used in GraphSAINT, but our sampler takes the class distribution into account by using a distribution inverse to the number of occurrences of classes. We use a vanilla MLP as baseline model on a binary encoded feature matrix (one-hot encoded labels). We compare a Bloom filter approach and measure its performance when computing graph summaries. The Bloom filter is used as a non-neural baseline. A specific characteristic of the filter is that it does not require the relevant feature values to be sorted for summarizing vertices. Overall, we evaluate six different methods: four GNNs, one MLP, and the Bloom filter.

We apply four different popular graph summary models [1] based on features that consider the edge labels (see Figure 1), vertex labels, their combination (edge labels plus vertex labels), as well as a model based on a the labels of a vertex's neighbors. We compute four graph summaries using these models for the DyLDO dataset [14], which is a subset of linked data crawled from a set of seed-URIs. We randomly split the DyLDO graph into training and test sets to evaluate different classifier models on the four summaries. The DyLDO dataset was chosen because the data is stored in a standardized way and because it is representative of real-world use of graphs on the web, but its size is still manageable.

In summary, our research questions are:

- Can we produce a graph summary via a vertex classification based on GNNs? How accurate is a graph summary calculated by a GNN?
- How do different classifiers perform against each other at calculating a graph summary? How do more sophisticated models perform against a baseline?
- Do the classifiers perform differently depending on the summary model M?

The GNNs are evaluated by a 10-fold cross-validation, following the recommendation of [15]. In our experiments, the Bloom filter outperforms all GNNs and the MLP baseline. The results for the GNNs are close to each other, and in three out of four graph summary models the non-message-passing Graph-MLP outperforms the message-passing GNNs. The performance of the standard MLP is close to GNNs and ranked third or fourth in three of the experiments. Particularly in the presence of many classes, the standard MLP is on par with the GNNs. We run an ablation study on the DyLDO dataset, where singleton class occurrences are removed, reducing the complexity of the classification task. The experiments show that the results improved but interestingly not for all models. Further ablation studies are reported in our extended report [16]. First, we applied 1-hop GNN models on 1-hop graph summaries, which reduced the standard error but overall did not improve the results. Finally, we investigated if widening the MLP's hidden layer, i. e., increasing its capacity, improves the result. Doubling the number of hidden nodes to 2,048 slightly improves the results but cannot reach the performance of the other models.

Below, we discuss the related works. We describe models we use to answer the research questions in Section III. The experimental apparatus, Section IV, contains an introduction into the dataset, experimental procedure, implementation, and evaluation metrics. The results of our experiments are reported in Section V. We discuss the results in Section VI, before we conclude. Note that we use the term *node* for the use in the neural network context and *vertex* in the graph context.

II. Related Work

First, we summarize the state of the art for graph summarization. Thereafter, we review different neural network models suited for the vertex classification task. We continue discussing hash functions in the context of neural networks. Finally, an introduction to Bloom filters is given.

A. Graph Summarization

Graph summaries are a good approach to provide a structural representation of large graph datasets [1]. Below, we define our notion of graphs and the task of graph summarization.

a) Graph: A graph $G = (V, E, R)$ consists of a set V of vertices, a set R of relation types and a set $E \subseteq V \times R \times V$ of labeled edges. Vertices typically represent entities, and edges describe the relationships between them.

b) RDF Graphs and `rdf:type`: The Resource Description Framework (RDF) is a W3C standard that models graphs as *subject–predicate–object* triples (s, p, o) [17]. For our purposes, we assume that $s, o \in V$, $p \in R$, and $(s, p, o) \in E$. It can also be expanded from triples to quads, where an optional fourth value describing the *graph* in the dataset the triple belongs to, is added to the tuple [18]. A special predicate in the RDF standard is `rdf:type`. The triple (s_i, `rdf:type`, o_j) denotes that s_i has the vertex label o_j. Predicates $p \neq$ `rdf:type` are called RDF properties.

c) Graph Summary: As discussed above, the idea of graph summarization is to generate a condensed representation SG of an input graph G [1]. The result is a summarized graph which is typically much smaller than G but which preserves structural information necessary for a given task. This compression allows tasks to be computed on the graph summary much faster than on the original graph [1], e. g., counting vertices that have the same information in the neighborhood. To this end, information about each vertex's neighborhood in the graph G is gathered and combined in a specific way defined by the summary model M. Vertices are classed as equivalent based on the properties defined by M. For example, the Attribute Collection summary model determines vertices to be equivalent if they have the same set of RDF properties [1]. The summary graph SG has one vertex per equivalence class and edges between classes corresponding to the neighborhood structure identified by the summary model.

The neighborhood features extracted by the summary model can be "compressed" by a hash function to give a class label for the equivalence class. This labeling can be seen as a vertex classification task, and we investigate the extent to which this task can be performed by neural networks.

d) FLUID: A good way to calculate graph summaries is to use FLUID [1]. It is a language and generic algorithm for flexibly defining and adapting graph summaries and is able to define all existing lossless structural graph summaries. However the computation time increases for larger graphs when using the FLUID framework because of the usage of sorting and hashing operations. The runtime of those operations depends on the graph size, resulting in long computation times for large graphs.

B. Graph Neural Networks for Vertex Classification

For the following section, we denote by $\boldsymbol{A}$ an adjacency matrix and by $\boldsymbol{D}$ the corresponding degree matrix. The adjacency matrix with self loops is calculated by $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$ with the identity matrix $\boldsymbol{I}_N$. $\tilde{\boldsymbol{D}}$ corresponds to the degree matrix of $\tilde{\boldsymbol{A}}$.

a) GCN and Variants: To run a classification task on graphs, Kipf et al. [10] suggest a graph convolutional network (GCN) model

$$f_{\text{GCN}}(\boldsymbol{X}, \boldsymbol{A}) = \text{softmax}\big(\hat{\boldsymbol{A}} \cdot \text{ReLU}\big(\hat{\boldsymbol{A}} \cdot (\boldsymbol{X}\boldsymbol{W}^{(0)})\big) \cdot \boldsymbol{W}^{(1)}\big) \tag{2}$$

applied directly on the graph G using message-passing. Here, $\boldsymbol{X}$ denotes a matrix containing vectors of node features, $\hat{\boldsymbol{A}} = \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$ is the symmetric Laplacian of $\tilde{\boldsymbol{A}}$ (symmetrically normalized by vertex degree), and $\boldsymbol{W}^{(i)}$ denotes the weight matrix of the hidden layer i. For message-passing, each vertex generates a message based on its embedding and sends this message to all its neighbors. Each vertex then aggregates the received messages in a permutation invariant manner and updates its own embedding with the aggregation result. To handle semi-supervised training data, the function $f_{\text{GCN}}(\boldsymbol{X}, \boldsymbol{A})$ is conditioned through the loss function on the vertices that have labels. But during training and inference there are also vertices present that do not have any label. Kipf et al. [10] show that a two-layer GCN results in a 2-hop aggregation.

Schlichtkrull et al. [19] propose relational GCN (R-GCN) that extends the GCN model by defining a directed graph $G = (V, E, R)$. The model adds a trainable weight matrix for every relation type $r \in R$. Now a GCN can also be applied for entity classification and link prediction on relational data. Another approach is Simple Graph Convolution (SGC) proposed by Wu et al. [20]. Their goal is to reduce the complexity of GCN by removing non-linearities and by condensing the weight matrices of single layers into a combined one.

b) Sampling-based GNNs: GCNs are limited by their memory requirements. The weight matrices $\boldsymbol{W}$ and node feature matrices $\boldsymbol{X}$ are dimensioned on the full adjacency matrix $\boldsymbol{A}$. This means that the whole graph has to be loaded into memory. Thus, batching is not possible. Furthermore, only a transductive training is possible. These shortcomings are mitigated by sampling-based models like GraphSAINT and GraphSAGE.

GraphSAINT [12] uses a sampler (vertex-, edge-, or random walk-based) to generate smaller batches that are applied to a classic GCN model. The GCN model is dimensioned to fit the sampled graph $G_s = (V_s, E_s)$ and so it is greatly reduced in size. The weights of the smaller model have to be transferred from the complete model and then later updated. The activation h_v^k of each vertex v in layer k is calculated via Equation 3, using an activation function σ, the normalized and sampled adjacency matrix $\tilde{\boldsymbol{A}}_s$, and the weight matrix $\boldsymbol{W}_s$, containing all necessary weights for the sampled model.

$$h_v^k = \sigma\Bigg(\sum_{u \in \mathcal{V}_s} \tilde{A}_s[v, u](\boldsymbol{W}_s^{k-1})^T h_u^{(k-1)}\Bigg). \tag{3}$$

GraphSAGE [11] aggregates the information in a k-neighborhood into a vertex embedding of the current root vertex of the neighborhood. Its authors explored mean, pre-trained LSTM, and pooling aggregators, which are permutation invariant. Training those aggregators instead of the weights of the vertices' embeddings, like in GCN, allows an inductive training process. The node activation h_v^k for $v \in \mathcal{V}$ and the neighborhood $\mathcal{N}$ of v in layer k is calculated by

$$h_v^k = \sigma\Big(\boldsymbol{W} \cdot \text{MEAN}\big(\{h_{\text{v}}^{k-1}\} \cup \{h_u^{k-1} \mid u \in \mathcal{N}(\text{v})\}\big)\Big). \tag{4}$$

There are further sampling-based models [21], [22]. Chen et al. [21] propose a StochasticGCN model. It uses a control variate-based sampler to reduce the receptive field of a node. An activation history is kept per node and then used in the stochastic training process. An adaptive sampling method is proposed by Huan et al. [22] to increase the convergence speed of a model. They apply a layerwise sampling approach which results in a linear growth of nodes. We leave the consideration of these models for future work.

c) Graph-MLP: Hu et al. [13] introduce an MLP-based GNN without the message-passing mechanism used in conventional GNNs. The model can be split into a two-layer MLP followed by a classifier layer. In the MLP, a GELU-activation function σ, layer-normalization, and dropout is applied (see Equations 5a and 5b, where $\boldsymbol{X}^{(i)}$ and $\boldsymbol{W}^{(i)}$ denote the node feature matrix $\boldsymbol{X}$ and the weight matrix $\boldsymbol{W}$ of the i-th layer). Finally, a linear classifier layer is applied as shown in Equation 5c.

$$\boldsymbol{X}^{(1)} = \text{Dropout}\big(LN\big(\sigma(\boldsymbol{X}\boldsymbol{W}^{(0)})\big)\big) \tag{5a}$$

$$\boldsymbol{Z} = \boldsymbol{X}^{(1)}\boldsymbol{W}^{(1)} \tag{5b}$$

$$\boldsymbol{Y} = \boldsymbol{Z}\boldsymbol{W}^{(2)} \tag{5c}$$

For feature transformation, a neighboring contrastive (NContrastive) loss $loss_{NC}$ is introduced by applying it to the output layer of the MLP $\boldsymbol{Z}$ (see Equation 5b). In the NContrastive loss, all vertices inside the r-hop neighborhood for each vertex are counted as positive samples and vertices outside this neighborhood as negative ones. The loss is calculated on the base of the rth power $\hat{\boldsymbol{A}}^r$ of the normalized adjacency matrix and a cosine similarity with a temperature parameter τ. To the output layer $\boldsymbol{Y}$ (see Equation 5c), a standard cross-entropy loss $loss_{CE}$ is applied for the classification objective. The NContrastive loss is weighted by coefficient α resulting in the $loss_{total} = loss_{CE} + \alpha \cdot loss_{NC}$.

C. Neural Networks as Hash Functions

The goal of a graph summary is the partitioning of the vertices into equivalence classes. Each class can be represented using a class label, which can be generated using a hash function. Until now, the application of neural networks as a hash function has been considered mainly in the context of security. Turčaník et al. [8] initialize a neural network with random weights and biases. Lian et al. [9] apply a chaotic map function to the weights and biases. The goal of those neural networks is to provide a one-way function and they aim to ensure that small changes in the input sequence create big changes in the resulting hash value.

Our goal is not a general purpose hash function but a classifier. Therefore, the security aspects discussed in Lian et al. [9] are not relevant in our application. Nevertheless, since the result of the hash function can be interpreted as a class label for the graph summary's equivalence class, we can apply a neural network for this classification task.

D. Bloom Filters

A Bloom filter is a probabilistic data structure used to test whether an element is a member of a set. False positive matches can occur with a low probability, but false negatives results are impossible [23]. A Bloom filter uses a Boolean array with a predefined size, each entry is initiated with *FALSE*. When adding an element to the Bloom filter, the element is hashed by multiple predefined hash functions, resulting in multiple indices of the Bloom filter array and the values at these array indexes are then set to *TRUE*. To query a membership in a Bloom filter, the element just needs to be hashed by the predefined hash functions and then checked whether the resulting array indices return *TRUE*. If all of them do so, the element is most likely in the set, with a small chance of false positives, and if any of the returned values is *FALSE*, the element is definitely not in the set [24].

III. Models

First, we explain how to compute graph summaries and provide an overview of the summary models considered in our work. Thereafter, we describe the graph neural networks used in our experiments. Finally, we explain the multi-layer perceptron used as a neural baseline and Bloom filters as a non-neural baseline.

A. Computing Graph Summaries with Neural Networks

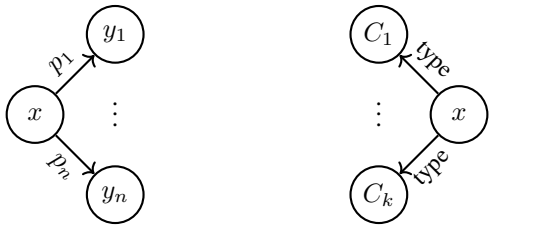
The main goal of this work is the calculation of graph summaries using neural network techniques. For training and evaluation, it is essential to determine the true class of each vertex. The class depends on the specified summary model M. The class labels are calculated in a "traditional", lossless way to establish a ground truth and to use them later as target labels for training the neural networks.

The information of vertex v is defined through all the (s, p, o)-triples with $s = v$. The summary model defines which triples are considered for each subject. The vertex information is gathered in a list (see Figure 1) and sorted. Based on that list, the equivalence class for each vertex is determined. In more detail, we collect the information as strings in a list, sort the list, concatenate the strings into a single string, and then apply a hash function. The resulting hash can then be used as the equivalence class. This approach is based on SchemEX [3].

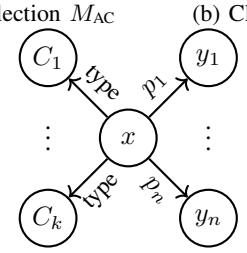
The sorting algorithm timsort [25] is used, which is a combination of insertion sort and merge sort. For small input sizes, insertion sort is used with a complexity of $O(n^2)$ and for big sizes, merge sort with a complexity of $O(n \log n)$. This leads to timsort's worst-case and average complexity of $O(n \log n)$, and best-case complexity $O(n)$. For strings, timsort has the property that it uses fewer comparisons than other $\Theta(n \log n)$ sorting algorithms. This is advantageous for us, since it is our use case. We use the default hashing function for strings in Python, which is an implementation of the modified Fowler–Noll–Vo algorithm [26].

B. Considered Graph Summary Models

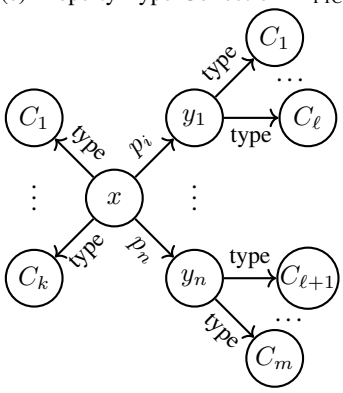
For the graph summary model Attribute Collection M_{AC} (see Figure 2a), the information used as described above is the property set. The property set of a vertex is its set of outgoing edge labels, excluding `rdf:type`. The Class Collection M_{CC} (see Figure 2b) can be seen as the other side of the Attribute Collection. The information used is called the type set and consists only of the RDF types. A combination of both is the Property Type Collection M_{PTC} shown in Figure 2c. With the gathered knowledge, SchemEX M_{SX} can be described as a combination of the specifications of multiple summary models. It is shown in Figure 2d. For M_{SX}, the equivalence class is calculated by the aggregation of the M_{CC} information on the root vertex and the M_{CC} information of a neighborhood. The neighborhood is defined by the M_{AC} specification of the root vertex. Thus, we are aggregating the M_{CC} information of the root vertex and the M_{CC} information of the neighboring vertices into a single list. The calculation of the hash for the equivalence class is then the same as for the other models.



(a) Attribute Collection M_{AC} (b) Class Collection M_{CC}



(c) Property Type Collection M_{PTC}



(d) SchemEX M_{SX}

Fig. 2: The summary models M_{AC}, M_{CC}, M_{PTC}, and M_{SX} considered in this work. Here, x is the root vertex that is to be summarized, y is an object connected to x through an attribute set, and C is an object connected to x or y connected via an `rdf:type` edge.

C. Graph Neural Networks Selected for the Experiments

We describe the graph neural networks (GNNs) we selected from Section II-B that we use in our experiments. These are GCN, GraphSAGE, GraphSAINT, and Graph-MLP. For all GNNs, we use a ReLU-activation function and dropout for all our hidden layers, while a softmax function is used on the output layer. One common issue with GNN approaches is that they are highly sensitive to the vertex degrees, which can lead to numerical instabilities and problems during training [7, Chapter 5.2]. This can usually be solved by normalizing w.r.t. degree prior to aggregation. However, we cannot do this, as the goal of graph summaries is to preserve the graph structure and the vertex degree is an important structural feature.

The optimization objective during training is a negative log likelihood loss function. Given the high memory requirements of GNNs, we apply a sampler on our extensive dataset. Here, we are following GraphSAINT's subgraph sampling strategy. We use a directed-edge sampler to reduce the batch size and then apply a GNN on a batch of root-vertex centered subgraphs. This is done via semi-supervised transductive training [7, Chapter 6.1.1]: during inference all vertices are present, while the loss function is only calculated on the labeled vertices, which are the root vertices.

We report the results per GNN model as *GNN-k* with *GNN* corresponding to the neural model and k to the number of hops.

a) GCN: To apply classic GCNs, we chose to create the models on the full adjacency matrix but only feed batches created by our sampler through the models. We do not shrink the adjacency matrix to the batch size, because of the high computational complexity of reintegrating those smaller graphs into the complete model.

b) GraphSAGE: In our application, we use a vanilla GraphSAGE with a mean-aggregator. One limitation of GraphSAGE is that it assumes that nodes in a neighborhood belong to the same equivalence class, which is not guaranteed in our datasets. But as different applications [11] show, this limitation can be ignored.

c) GraphSAINT: We replace GraphSAINT's vertex sampler by one based on the inverse class distribution. We apply a GraphSAINT network using Weisfeiler–Leman kernels. If the network has two layers then we also apply jumping knowledge.

d) Graph-MLP: We use Graph-MLP [13] and remove the normalization layer as above. The normalizing technique, applied by the authors, is again skipped because of the characteristics of our application. For our experiments we align the r-hop-parameter of Graph-MLP to the k-hop characteristic of each graph summary model. The specific r-value is appended to the model name.

e) Discussion of Further Models: We also considered using R-GCN and SGC. But both models could not be used on the large numbers of classes of our datasets. Our DyLDO dataset has about 7 times more vertices, 20 times more edges, 110 times more edge types, and $10,000$ times more classes than the datasets used in the original R-GCN paper

by Schlichtkrull et al. [19]. Since the complete R-GCN model must be present during training and R-GCN has additional weights per edge type, the model massively exceeds the available GPU memory, even with weight sharing techniques such as basis decomposition [19]. We ran pre-experiments for SGC on a smaller dataset. SGC scored considerably lower than the other GNNs and was also the biggest model, which pushed us to the limits of our GPU memory. In our main experiments on the larger DyLDO dataset, the SGC model exceeded the GPU memory. Thus, we do no use R-GCN and SGC in our experiments. For more details see Section K.

D. Baselines

We consider a simple but effective neural baseline and a strong non-neural baseline.

a) MLP: We use a standard multi-layer perceptron (MLP) with two hidden layers and ReLU-activation function with dropout. This is motivated by the strong performance of MLPs for text classification [27].

b) Bloom Filter: As a non-neural baseline, we use a Bloom filter. For each vertex, the graph summary specific information is added into an empty Bloom filter array. To calculate the equivalence class of the vertex, the Bloom filter array is hashed. This approach avoids the sorting normally required for graph summarization. However, it comes with a cost in terms of accuracy as the Bloom filter is susceptible to false positives, meaning that some originally different equivalence classes result in the same Bloom filter array and thereby are wrongly assigned to the same equivalence class.

We define our Bloom filter with 4, 15, and 60 input items, corresponding to the 75th, 95th, and 99th percentile of the node degree distribution of the DyLDO dataset and false positive probabilities of 10^{-3} and 10^{-1}. These combinations result in different numbers of hash functions and bits in the array. Results of more configurations are documented in Appendix A.

IV. Experimental Apparatus

We introduce the dataset, hyperparameter optimization, the experimental procedure, important information regarding the implementation, and the evaluation metrics.

A. Datasets

This section gives an overview over the considered RDF-dataset and the class distributions generated by the different graph summary models.

1) Basic Statistics: The Dynamic Linked Data Observatory (DyLDO) [14] is a framework for monitoring linked data. It takes snapshots of a subset of linked data, to capture its dynamics. DyLDO takes a sampling-based approach of seed-URIs, which are also representative for the vertices of the graph. Each weekly snapshot has about 100 million triples. The dataset is populated by crawling from a list of seed-URIs. Crawling from the seed URIs is restricted to a depth of two hops [14]. We use this dataset because of its reasonable size, but it still reflects real-world linked data.

a) DyLDO: The main snapshot used for this paper is the very first one, as it is with $127M$ triples the largest one. Analysis by Blume and Scherp [28] shows that it contains 7,093,011 vertices with 15,017 overall edge properties. From these edge properties, `rdf:type` occurs 5.4 M times. Each vertex has on average 17 outgoing properties.

b) DyLDO-6_499: The SchemEX graph summary model M_{SX} is the only one we consider that requires a second hop, so there is a huge increase in observed vertices and consequently in the size of each subgraph. Hence, generating and using all 2-hop subgraphs during inference would have exceeded our disk space and memory capacity. Therefore, a smaller portion of the dataset was used to reduce the computational requirements. This smaller dataset was built by removing vertices with a class occurrence of < 6, which leaves around 15% of the dataset and fits our hard drive restrictions. Also only subgraphs with fewer than 500 vertices are kept. In our sampling process (during inference), we use this number as guard condition for the maximum size of the mini-batch (see Section IV-C0b). We call this dataset DyLDO-*6_499* and likewise we call M_{SX} on this dataset $M_{\text{SX-6\_499}}$. This dataset has 16.7 M triples, 5,312,991 vertices, and 8,102 edge properties. From these edge properties, `rdf:type` occurs $3.7M$ times. Each vertex has on average 3 outgoing properties.

2) Class Distributions: Computing the summaries for the four summary models introduced in Section III-A, generates four different datasets on which to train the neural networks. These four graph datasets differ in the number of equivalence classes, which correspond to the number of classes for the graph neural networks. Table I shows the number of classes for each summary model.

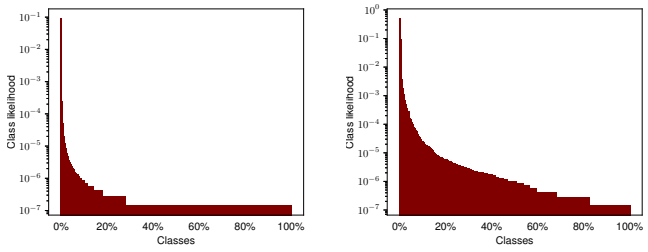
M_{AC}	M_{CC}	M_{PTC}	M_{SX}	$M_{\text{SX-6\_499}}$
162,521	576	178,472	335,608	18,314

TABLE I: Number of equivalence classes in the DyLDO dataset for each summary model. $M_{\text{SX-6\_499}}$ denotes the number of M_{SX} classes in the DyLDO-*6_499* dataset.

In Figure 3, we plot the class distribution for each summary model as the likelihood of a specific class appearing. The class distributions of M_{AC} (Figure 3a), M_{CC} (Figure 3b), M_{PTC} (Figure 3c), and $M_{\text{SX-6\_499}}$ (Figure 3d) all show a skewed distribution due to the majority of classes appearing only once. For reference, the class distribution of the full M_{SX} can be seen in Appendix I.

B. Hyperparameter Optimization

We applied a hyperparameter search to the learning rate over the values in $\{0.1, 0.01, 0.001\}$ of the Adam optimizer, dropout in $\{0.0, 0.2, 0.5\}$, and the hidden layer size $\{32, 64\}$ for the GNNs with 2-hops. For Graph-MLP, we apply the hyperparameter search onto the weighting coefficient $\alpha \in \{1.0, 10.0, 100.0\}$, the temperature $\tau \in \{0.5, 1.0, 2.0\}$, learning rate in $\{0.1, 0.01\}$, and the hidden layer size $\{64, 256\}$. The validation accuracy and training loss are monitored and evaluated on the details of convergence speed and stability



(a) Model M_{AC} (b) Model M_{CC}

(c) Model M_{PTC} (d) Model $M_{\text{SX-6\_499}}$

Fig. 3: The class distributions of the different graph summary models on the DyLDO dataset. The horizontal axis shows the classes sorted in descending likelihood.

for this purpose. For the search, we trained our models for 30 epochs. We manually confirmed the optimization process by inspecting the loss curve. This ensured that the models properly converged. The validation accuracy is the mean over 15 mini-batches from the validation fold.

For our MLP, we use a hidden layer size of 1,024 following prior work [27]. For Graph-MLP, we use the recommended dropout of 0.6 without any hyperparameter optimization of the dropout value. For the GNNs, the dropout values are overall low. Interestingly, some of our GNN models work best without any dropout. The optimal hyperparameter values of the GNNs for the different summary models are shown in Appendix B. An overview over all parameter counts for the competing models is given in Appendix C.

C. Procedure

Our goal is to use the equivalence classes of the graph summaries as labels for our classification task. Given our large dataset, we had to apply a sampling method and train our models on mini-batches. To apply the GNNs on our data, we have to use the following preprocessing steps. Afterwards, we explain the training and the evaluation process.

a) Preprocessing: In the first step, all triples (s, p, o) in G are preprocessed into a dictionary. This dictionary consists of the subject URI as key and the corresponding information structure for each subject vertex as value. The latter contains the equivalence class label, all the neighbors, and the fold number of a subject vertex. Based on this dictionary, the chosen graph summary model is calculated. The calculated hash values (equivalence classes) are transformed into class labels $\boldsymbol{y}$, which annotate the corresponding vertices and are set in *SubjectInformation*. To ensure the separation of training, validation, and test data, we split our data into ten folds by assigning each subject vertex randomly to a fold. So a fold consists of the subgraphs of its subjects. The pseudocode of the algorithm to generate the training data for the different summary models is shown in Appendix D.

The structure for each subgraph consists of a feature matrix, a list of class labels for each node, an adjacency list, and a list containing the edge types, which corresponds to the adjacency list element. The feature matrix one-hot encodes each node in the subgraph. Because of RAM limitations on the GPU, we apply a sampler to generate mini-batches (see below).

b) Training the Neural Networks: After the preprocessing, the neural networks are trained on mini-batches from the training folds and then tested on the mini-batches from the test fold. The training process is monitored and evaluated on the validation accuracy.

Algorithm 1 Create mini-batch for training from the foldset

```
1: procedure CreateMiniBatch(f: foldset, guard: integer)
2:     GraphData batch ← ∅
3:     Draw subgraph sample from f
4:     while batch.#nodes + sample.#nodes < guard do
5:         Append sample to batch
6:         Draw next subgraph sample from f
7:     end while
8:     return data
```

Our sampler differs from the vertex-, edge-, and random walk-based samplers used in GraphSAINT [12]. Independent of the specific sampling method, GraphSAINT determines the vertices involved by the sample (e. g., the edges' source and target vertices) and adds the subgraphs induced by those vertices to the batch. This can cause a huge variance in the number of vertices and edges, i. e., huge differences in the batch sizes. In practice, this lead to out of memory problems on the GPU, so we restrict the size of the batch with the help of a guard.

To generate the mini-batches with a constant mini-batch size $guard$, we apply Algorithm 1 with a set of folds f and the $guard$ condition representing the desired batch size. To maintain a constant mini-batch size, the mini-batches are padded with additional *dummy* vertices that are single nodes without any edges and a class label of -1. This is done until reaching the $guard$ limit. We define $guard = 500$. This restriction of the constant mini-batch size is based on the limitation of the GPU memory. The goal is to fit the largest possible mini-batch into memory for all GNNs. As an optimizer Adam is applied for all GNNs.

Only the root vertices of the subgraphs are considered in the loss and accuracy calculation, ignoring the object neighbors and the dummy vertices. To compensate for the skewed class distribution, the subgraphs are randomly sampled for the training process by a probability which is inversely proportional to the frequency of the class occurrences. This results in a uniform distribution of sampled subgraphs to increase stability

and speed of training. In the last step, those mini-batches can be used in the training process or in validation or test of the model. For the training of the models, the results of the hyperparameter optimization, Section IV-B, were used. The models are trained for 30 epochs, except for Graph-MLP-1 and M_{CC}, which were trained for 25 epochs.

c) Evaluation: To evaluate our methods, we run a full 10-fold cross-validation. For each step, we trained our GNN models on the data of 8 folds, evaluated the training progress on 1 validation fold, and tested it on 75 mini-batches of 1 test fold to report a mean testing accuracy per run.

D. Metrics

We use accuracy to measure the classification performance of the GNNs. We report the mean and standard error of the test accuracy on a 10-fold cross-validation.

The Bloom filter is evaluated based on accuracy and impurity. We compare between the hash values calculated via the graph summarization method from Section III-A as ground truth and the hash values calculated by the Bloom filter method (see Section III-D) of all N subject vertices, which we call the root set $\mathcal{R}$. We cluster the vertices in $\mathcal{R}$ via their Bloom filter hash values, resulting in the clustering Ω. Each cluster $\omega \in \Omega$ can be clustered again based on the ground truth values of each subject vertex, resulting in the clusters $\mathcal{C}$. We compute the accuracy based on the assumption that the vertices in the largest category per cluster are the true positives. This follows the fundamentals of the impurity definition. Per cluster $\omega \in \Omega$, the impurity measure Q is calculated based on the Gini-index resulting in Q_g [29] and the probability $p_{ij} = \frac{m_{ij}}{|\omega_i|}$ with m_{ij}, the number of objects from C_j in ω_i. The final impurity value $Q_g(\mathcal{R})$ is calculated via the sum over all the clusters weighted by their relative frequency $\frac{|\omega_i|}{N}$ with

$$Q_g(\mathcal{R}) = \sum_{i=1}^{|\Omega|} \frac{|\omega_i|}{N} Q_g(\omega_i), \quad Q_g(\omega_i) = 1 - \sum_{j=1}^{|\mathcal{C}|} p_{ij}^2 \,.$$

V. Results

For our results, we applied 10-fold cross-validation for all our models except the Bloom filter, and all summary models. We did not cross-validate Bloom filters, as they are deterministic and would give the same result each time. The results can be seen in Table II.

The Bloom filter outperforms all tested neural models, by a margin that depends on the summary model. The second-best result for M_{AC} is 0.2333 lower (Graph-MLP-1), M_{PTC} is 0.2256 lower (Graph-MLP-1) and $M_{\text{SX-6\_499}}$ is 0.2278 lower (Graph-MLP-2). For M_{CC} the Bloom filter and sample-based GCN-2 (short: GCN-2) performance is similar. The results for the neural network models of M_{AC} are within 0.0372, M_{CC} is within 0.0158, M_{PTC} is within 0.0521, and $M_{\text{SX-6\_499}}$ is within 0.0409 of the reported accuracy. The standard error for all GNN models is below 0.0080, with the exception of MLP-2 and $M_{\text{SX-6\_499}}$. The lowest standard error is 0.0025 for sample-based GCN-2 and M_{CC}.

Table III shows the impurity metric for the differently parameterized Bloom filters. The Bloom filter accuracy results for every parameter combination within a single summary model differ at most by 0.0237 from each other (see Table II). For the impurity measures, a similar result can be observed. The average scores are very close to each other. The higher the chosen n and the lower the chosen p, the better the accuracy and impurity results. All Bloom filter accuracies and impurities for M_{CC} are equivalent.

VI. Discussion

a) Main Results: Our experiments show that graph summarization via vertex classification is a challenging task. The models have to deal from a few hundred to hundreds of thousands of classes, depending on the summary model. Bloom filters deliver overall the best and most consistent results. The graph neural networks are on par with Bloom filter in one of the four summary models, namely M_{CC}. The number of classes for the M_{CC} model is rather low (576). Generally, the performance of the graph neural networks are close to each other. For each summary model, the difference between the best- and worst-performing GNN is less than 4.1 points.

Our results show that Graph-MLP outperforms the other neural models in three out of four graph summary experiments. However, it is difficult to make a decisive conclusion as to which graph neural network model is the best, as Shchur et al. [15] have shown that minor changes in the model parameters, training data, etc. of GNNs have a significant effect on the ranking of the models. The Graph-MLP model, which does not use message passing and introduces an MLP-like structure, was only tested on smaller datasets in the original paper [13]. Thus, for the first time, it has been applied on large graphs with large numbers of classes. Furthermore, GraphSAGE outperforms GraphSAINT in all cases. The good performance of GraphSAGE (among the top-3 neural network models in three out of four graph summary experiments) on a classification task where the labels are dependent on the neighborhood structure might be surprising, but is consistent with findings in previous works [15]. The random neighborhood sampling and the learning of the aggregation function of GraphSAGE seem to be strong regularizers.

Our baseline MLP is quite strong but does not outperform graph-based models. Such a strong performance of a standard MLP may be surprising, but MLP has performed well in other classification tasks [27], [30].

b) Further Discussion: The different summary models have different numbers of class labels (see Table I). This means that with an increasing number of equivalence classes, the complexity of classification also increases because it requires more differentiation. The influence of complexity can be observed in two ways. The accuracy results of our different neural network models are worse for models with a higher class count and these results differ more within models with a higher class count. This can be best observed in the good and very similar results for M_{CC}, as M_{CC} has the

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SX-6\_499}}$
Bloom filter ($n = 4$, $p = 10^{-1}$)	0.8376	**0.8562**	0.7987	0.7598
Bloom filter ($n = 15$, $p = 10^{-1}$)	0.8555	0.8562	0.8216	0.7764
Bloom filter ($n = 15$, $p = 10^{-3}$)	0.8562	0.8562	0.8223	**0.7767**
Bloom filter ($n = 60$, $p = 10^{-1}$)	0.8562	0.8562	**0.8224**	0.7767
Bloom filter ($n = 60$, $p = 10^{-3}$)	**0.8563**	0.8562	0.8224	0.7767
MLP-2	**0.6033** $\pm$ 0.0048	0.8496 $\pm$ 0.0067	**0.5706** $\pm$ 0.0030	**0.5470** $\pm$ 0.0127
GCN-2	0.6007 $\pm$ 0.0060	**0.8559** $\pm$ 0.0025	0.5611 $\pm$ 0.0047	**0.5473** $\pm$ 0.0052
GraphSAINT-2	0.5858 $\pm$ 0.0047	**0.8538** $\pm$ 0.0026	0.5447 $\pm$ 0.0062	0.5080 $\pm$ 0.0076
GraphSAGE-2	**0.6138** $\pm$ 0.0052	**0.8541** $\pm$ 0.0027	**0.5652** $\pm$ 0.0079	0.5354 $\pm$ 0.0053
Graph-MLP-1	**0.6230** $\pm$ 0.0057	0.8401 $\pm$ 0.0031	**0.5968** $\pm$ 0.0037	–
Graph-MLP-2	–	–	–	**0.5489** $\pm$ 0.0066

TABLE II: Results for 10-fold cross-validation on DyLDO's first snapshot reported by mean accuracy with standard error (higher accuracy, lower standard error better). The chosen n-values for Bloom filter describe the 75th, 95th, and 99th percentile of the node degree distribution. The top four models are highlighted per graph summary: **first**, **second**, **third**, and **fourth** place. For Bloom filter the model with the highest accuracy score, with the smallest number of hash functions k, and the smallest number of bits in the array m is marked as best. Best viewed in color.

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SX-6\_499}}$
$n = 4$, $p = 10^{-1}$	0.2232	0.2423	0.2793	0.2363
$n = 15$, $p = 10^{-1}$	0.2026	0.2423	0.2526	0.2202
$n = 60$, $p = 10^{-1}$	0.2018	0.2423	0.2517	0.2198

TABLE III: Impurity metric for Bloom filter using the Gini-index with expected input items n and false positive probability p (lower impurity better).

lowest number of classes by a large margin (see Table I). An exception to this observation is the results for $M_{\text{SX-6\_499}}$. We suspect that this deviation is caused by the 2-hop property of SchemEX in conjunction with the lack of normalization. The 2-hop characteristic increases the complexity significantly by aggregating additional information from the second hop into the root node.

The choice of parameters for the Bloom filter does not seem to have much impact. Furthermore there seem to be diminishing returns when higher n and the lower p are chosen. The Bloom filter accuracy results seem to converge at $\sim$0.8563 for M_{AC}, $\sim$0.8562 for M_{CC}, $\sim$0.8224 for M_{PTC}, and $\sim$0.7767 for $M_{\text{SX-6\_499}}$. The impurity results converge at $\sim$0.2018 for M_{AC}, $\sim$0.2423 for M_{CC}, $\sim$0.2517 for M_{PTC}, and $\sim$0.2198 for $M_{\text{SX-6\_499}}$. This strong performance can be explained because Bloom filters approximate set membership with an one-sided error, which is required by various summary models. The high number of bits and the resulting high number of potential representation of classes can also contribute the the strong performance.

c) Threats to Validity and Future Work: It is known that experiments on GNNs can suffer from parameter choices [15]. We address these challenges by running a 10-fold cross validation and provide the standard errors of our GNN results. As shown in Table II, the standard error over our folds is very low and consistent within each summary model. Another observation is that the Bloom filter scores for M_{CC} are identical. This may have suggested some error in our procedure. However, we manually checked the results. Furthermore, we executed the same experimental procedure (code, evaluation etc) for M_{CC} as for the other summary models, which further supports the correctness of the M_{CC} results.

One may note that the applied GNN models consider directed edges as undirected, which is commonly the case in this domain. This limitation might have impacted our classification performance, because graph summaries are sensitive to the edge direction. However, as discussed in Section III-C our graph sampler applied to feed the GNNs considers the direction of the edges based on the applied summary models. Thus, the applied GNNs also properly consider the edge directions when computing the classifications of the vertices.

We experiment with real-world datasets. Thus, the number of classes is much higher than in commonly used datasets [31], which are based on citations. Approximately 54% of our classes only occur once in the dataset. This leads to an extremely skewed class distribution, as can be seen in Figure 3. Thus, overall one can consider the results on such skewed datasets as impressive and further improvements could be made by, e. g., using hybrid models of GNNs combined with label propagation [32].

With our datasets we already consider graphs that are orders of magnitude larger in terms of classes and relations (see Section IV). The biggest dataset for R-GCN [19] has $1.67M$ vertices, with $1,000$ vertices being labeled and having 11 classes and 133 relations. While trying to apply the R-GCN model to our datasets, we ran into memory limitations. In the future work, we like to experiment with further datasets such as citation graphs [31].

To gain further insight into the performance and applicability of GNNs for graph summarization on larger datasets, our experiments should be repeated on further datasets, like the Billion Triple Challenge [33] or the LOD Laundromat [34].

A huge performance improvement would be to follow the proposed method of GraphSAINT, where only the partial model with the required nodes of the mini-batch are present. This would reduce the model's memory footprint and would allow a higher mini-batch size. For this, a method is required which manages the parameter transfer from the complete model, tracking the whole process, to the partial model, on

which inference and back-propagation can be run, and vice versa. This improvement would allow to apply models, like R-GCN and SGC.

We ran our experiments without normalization, because the class labels are dependent on the structural information of the neighborhood. Normalization and regularization could also be the subject of further research. This is motivated by the deviating results we received for the $M_{\text{SX-6\_499}}$. Another future study could be a runtime analysis in which we perform a cost-benefit analysis of using GNN methods with the involved the accuracy loss compared to the computation performance of lossless computations.

VII. Ablation Study: No Singleton Classes

We provide further insights into our experimental results by analyzing the impact of removing the singleton classes from the graphs on the models' performance. We run additional ablations and analyses, which are reported in Appendix H.

To check the effect of the skewed class distribution on the classification accuracy, we run an ablation study removing all singleton class occurrences from our graphs (min-support > 1). We call the models with removed singletons $M_{\text{AC-NS}}$, $M_{\text{CC-NS}}$, and $M_{\text{PTC-NS}}$. On average, 46% of the classes remain (see Table IV). In Appendix E, the new class distribution can be seen, in comparison to the original class distribution. Because the data distribution changes in this procedure, another hyperparameter optimization is applied. We did not include SchemEX, because in the dataset used for $M_{\text{SX-6\_499}}$, DyLDO-*6_499*, we are already filtering with a min-support higher than 1. The optimal hyperparameter values can be seen in Appendix E. The parameter counts for these models without singletons can be found in Appendix F.

	$M_{\text{AC-NS}}$	$M_{\text{CC-NS}}$	$M_{\text{PTC-NS}}$
Remaining Classes	45,198	474	49,719
Class Percentage	$\sim$28%	$\sim$82%	$\sim$28%
Remaining Subgraphs	6,975,688	7,092,909	6,807,122
Subgraph Percentage	$\sim$98%	$\sim$100%	$\sim$96%

TABLE IV: Number of equivalence classes in the DyLDO dataset by summary model, after the removal of classes that only occur once, the percentage of remaining classes (cf. Table I), the number of the original $7,093,011$ subgraphs remaining, and the percentage of the remaining subgraphs.

The results are reported in Table V via the average and standard error of the test accuracy for a 10-fold cross-validation after training for 40 epochs. There is an increase in accuracy for $M_{\text{AC-NS}}$ and $M_{\text{PTC-NS}}$ by $\sim$3–6%. Also the standard error for $M_{\text{AC-NS}}$ is lower except for MLP-2. However, $M_{\text{CC-NS}}$ gets worse by $\sim$2–4%. The singleton class occurrences make the problem harder. Comparing the results of this ablation study and the experiment, the difference is actually smaller than expected. From the results it is noticeable that the models with the lowest hidden layer size also have the lowest accuracy: for $M_{\text{AC-NS}}$, it is GCN-2 and GraphSAINT-2 for $M_{\text{PTC-NS}}$. This was also a finding in the main experiment. Even without the singletons, we still have a very skewed class distribution. Since we have a train–validate–test split on the data, further studies could be done on a dataset with min-support higher than 2 to consider the random distribution of classes in the data split.

Also, the lower performance of GNNs on the dataset without singletons for the $M_{\text{CC-NS}}$ should be researched as it is the only result that contradicted our expectation. To combat the skewed class distribution another less invasive measure could also be developed, which does not remove single class occurrence nodes but which maps the class onto the next best-fitting class [35].

VIII. Conclusion

We methodically compared the effectiveness of computing graph summaries, using established and recent graph neural network models and Bloom filters. Our main result is that Bloom filters outperformed the GNN models by a large margin in three out of four cases and had equal performance on the fourth, which had by far the least number of classes.

The Graph-MLP model without a classical message-passing pipeline is the best GNN model in three of the four cases. The GNNs performed well for the M_{CC} graph summary model. The ranking of the GNN models was not consistent across graph summary models, but the differences between them were rather small. A classical 2-layer MLP comes close to the results of the GNNs, being within 3% of the best graph model.

In our experiments, we noticed that the accuracy of the neural network models decreases with increasing numbers of equivalence classes. The performance of our non-neural baseline, the Bloom filter, seemed to be independent of the number of classes.

Acknowledgements

We thank the data science paper reading club of Ulm University and especially Lukas Galke for the input on implementation details on graph neural networks. Additional thanks go to Till Blume for the original proposal to investigate Bloom filters for graph summarization.

This research is the result of a Master module "Project Data Science" taught at Ulm University in 2020–2022. The last two authors are the supervisors of the student group.

References

[1] T. Blume, D. Richerby, and A. Scherp, "Fluid: A common model for semantic structural graph summaries based on equivalence relations," *Theoretical Computer Science*, vol. 854, pp. 136 – 158, 2021. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397520307234

[2] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *Int. Conf. on Data Engineering (ICDE)*. IEEE, 2011, pp. 984–994. [Online]. Available: https://doi.org/10.1109/ICDE.2011.5767868

[3] M. Konrath, T. Gottron, S. Staab, and A. Scherp, "SchemEX – efficient construction of a data catalogue by stream-based indexing of linked data," *J. Web Semant.*, vol. 16, pp. 52–58, 2012. [Online]. Available: https://doi.org/10.1016/j.websem.2012.06.002

[4] F. Benedetti, S. Bergamaschi, and L. Po, "Exposing the underlying schema of LOD sources," in *IEEE/WIC/ACM Intl. Conf. on Web Intelligence and Intelligent Agent Technology (WI-IAT 2015)*, vol. 1, 2015, pp. 301–304.

	$M_{\text{AC-NS}}$	$M_{\text{CC-NS}}$	$M_{\text{PTC-NS}}$
MLP-2	**0.6501** $\pm$ 0.0059	**0.8298** $\pm$ 0.0022	0.6050 $\pm$ 0.0078
GCN-2	0.6298 $\pm$ 0.0031	0.8129 $\pm$ 0.0069	**0.6154** $\pm$ 0.0029
GraphSAINT-2	0.6426 $\pm$ 0.0038	**0.8234** $\pm$ 0.0027	0.5952 $\pm$ 0.0067
GraphSAGE-2	**0.6442** $\pm$ 0.0039	**0.8300** $\pm$ 0.0022	**0.6141** $\pm$ 0.0022
Graph-MLP-1	**0.6627** $\pm$ 0.0039	0.8224 $\pm$ 0.0024	**0.6213** $\pm$ 0.0042

TABLE V: Results for 10-fold cross-validation for the different models after the removal of singleton classes reported by mean accuracy and standard error (higher accuracy, lower standard error better). The top three models are highlighted per graph summary: **first**, **second** and **third** place.

[5] F. Goasdoué, P. Guzewicz, and I. Manolescu, "RDF graph summarization for first-sight structure discovery," *VLDB J.*, vol. 29, no. 5, pp. 1191–1218, 2020. [Online]. Available: https://doi.org/10.1007/s00778-020-00611-y

[6] J. Schaible, T. Gottron, and A. Scherp, "TermPicker: Enabling the reuse of vocabulary terms by exploiting data from the linked open data cloud," in *The Semantic Web. Latest Advances and New Domains*. Springer, 2016, pp. 101–117.

[7] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[8] M. Turčaník and M. Javurek, "Hash function generation by neural network," in *New Trends in Signal Processing (NTSP 2016)*. IEEE, 2016, pp. 1–5.

[9] S. Lian, J. Sun, and Z. Wang, "One-way hash function based on neural network," *arXiv preprint arXiv:0707.4032*, 2007.

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

[11] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *30th Annual Conf. on Neural Information Processing Systems*, 2017, pp. 1024–1034. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[12] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "GraphSAINT: Graph sampling based inductive learning method," *CoRR*, vol. abs/1907.04931, 2019. [Online]. Available: http://arxiv.org/abs/1907.04931

[13] Y. Hu, H. You, Z. Wang, Z. Wang, E. Zhou, and Y. Gao, "Graph-MLP: Node classification without message passing in graph," *arXiv preprint arXiv:2106.04051*, 2021.

[14] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres, "Dyldo: Towards a dynamic linked data observatory," in *Workshop on Linked Data on the Web (WWW 2012)*, ser. CEUR Workshop Proceedings, vol. 937. CEUR-WS.org, 2012. [Online]. Available: http://ceur-ws.org/Vol-937/ldow2012-paper-14.pdf

[15] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *CoRR*, vol. abs/1811.05868, 2018. [Online]. Available: http://arxiv.org/abs/1811.05868

[16] M. Blasi, M. Freudenreich, J. Horvath, D. Richerby, and A. Scherp, "Graph summarization with graph neural networks," *CoRR*, vol. abs/2203.05919, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.05919

[17] D. Beckett, "Rdf 1.1 n-triples," 2014, accessed: 2021-04-17. [Online]. Available: https://www.w3.org/TR/n-triples/.

[18] G. Carothers, Lex Machina, Inc, "Rdf 1.1 n-quads," 2014, accessed: 2021-04-17. [Online]. Available: https://www.w3.org/TR/n-quads/.

[19] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web – 15th Intl. Conf., (ESWC 2018)*, ser. Lecture Notes in Computer Science, vol. 10843. Springer, 2018, pp. 593–607. [Online]. Available: https://doi.org/10.1007/978-3-319-93417-4_38

[20] F. Wu, T. Zhang, A. H. Souza Jr., C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," *CoRR*, vol. abs/1902.07153, 2019. [Online]. Available: http://arxiv.org/abs/1902.07153

[21] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proc. 35th Intl. Conf. on Machine Learning, ICML*, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 941–949. [Online]. Available: http://proceedings.mlr.press/v80/chen18p.html

[22] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," *CoRR*, vol. abs/1809.05343, 2018. [Online]. Available: http://arxiv.org/abs/1809.05343

[23] O. Rottenstreich and I. Keslassy, "The Bloom paradox: When not to use a Bloom filter," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 703–716, 2014.

[24] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge University Press, 2017.

[25] A. Heumann, "The magic behind the sort algorithm in Python," 07 2018, accessed: 2021-06-18. [Online]. Available: https://medium.com/ub-women-data-scholars/the-magic-behind-the-sort-algorithm-in-p

[26] Python, 04 2021, accessed: 2021-06-18. [Online]. Available: https://github.com/python/cpython/blob/bb3e0c240bc60fe08d332ff5955d54197f79751c/

[27] L. Galke and A. Scherp, "Forget me not: A gentle reminder to mind the simple multi-layer perceptron baseline for text classification," 2021. [Online]. Available: https://arxiv.org/abs/2109.03777

[28] T. Blume and A. Scherp, "Indexing data on the web: A comparison of schema-level indices for data search," in *Intl. Conf. on Database and Expert Systems Applications*. Springer, 2020, pp. 277–286.

[29] Y. Yuan, L. Wu, and X. Zhang, "Gini-impurity index analysis," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3154–3169, 2021.

[30] I. Venugopal, J. Töllich, M. Fairbank, and A. Scherp, "A comparison of deep-learning methods for analysing and predicting business processes," in *Intl. Joint Conf. on Neural Networks (IJCNN 2021)*. IEEE, 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IJCNN52387.2021.9533742

[31] Z. Yang, W. W. Cohen, and R. Salakhutdinov, "Revisiting semi-supervised learning with graph embeddings," *CoRR*, vol. abs/1603.08861, 2016. [Online]. Available: http://arxiv.org/abs/1603.08861

[32] H. Wang and J. Leskovec, "Combining graph convolutional neural networks and label propagation," *ACM Trans. Inf. Syst.*, vol. 40, no. 4, nov 2021. [Online]. Available: https://doi.org/10.1145/3490478

[33] J.-M. Herrera, A. Hogan, and T. Käfer, "BTC-2019: The 2019 billion triple challenge dataset," in *Intl. Semantic Web Conf.* Springer, 2019, pp. 163–180.

[34] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach, "LOD Laundromat: a uniform way of publishing other people's dirty data," in *Intl. Semantic Web Conf.* Springer, 2014, pp. 213–228.

[35] T. Gottron, A. Scherp, B. Krayer, and A. Peters, "Lodatio: using a schema-level index to support users in finding relevant sources of linked data," in *Proc. 7th Intl. Conf. on Knowledge Capture (K-CAP 2013)*. ACM, 2013, pp. 105–108. [Online]. Available: https://doi.org/10.1145/2479832.2479841

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, and et al., "PyTorch: An imperative style, high-performance deep learning library." Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep

[37] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," 2019, cite arxiv:1903.02428. [Online]. Available: http://arxiv.org/abs/1903.02428

APPENDIX

A. Bloom Filter Hash Functions and Bits

Table VI shows the number of hash functions and the bits of the Bloom filter array for the Bloom filter parameters that we used.

Bloom filter	k	m
$n = 4$, $p = 10^{-1}$	3	20
$n = 4$, $p = 10^{-3}$	10	58
$n = 4$, $p = 10^{-7}$	23	135
$n = 15$, $p = 10^{-1}$	3	72
$n = 15$, $p = 10^{-3}$	10	216
$n = 15$, $p = 10^{-7}$	23	504
$n = 60$, $p = 10^{-1}$	3	288
$n = 60$, $p = 10^{-3}$	10	863
$n = 60$, $p = 10^{-7}$	23	2,013

TABLE VI: The number of hash functions k and the bits in the Bloom filter array m according to our expected input items n and false positive probability p.

B. Optimal Hyperparameter Values

A detailed summary of the hyperparameters that we used for the specific summary models and neural network models can be found in Table VII.

C. Parameter Count

The parameter counts for the competing models are given in Table VIII. The parameter counts depend on the neural network, given graph summary model, and the chosen hidden layer size.

D. Generating Training Data for the Summary Models

We apply the Algorithm 2 to generate data for training, validation, and test based on subgraphs using *SubjectInformation* for each subject in the dictionary. Here, we take into account the different hop characteristics of the graph summary models.

Algorithm 2 Data conversion from *SubjectInformation* dictionary to subgraph GraphData

```
1:  procedure DataConversion(k-folds: integer, M: graph summary model)
2:      for every fold f in k-folds do
3:          for every subject vertex s in f do
4:              GraphData data ← ∅
5:              SI_s ← GET SubjectInformation of s
6:              Transform SI_s into GraphData and append to data
7:              if M requires 2-hop then
8:                  for Every (p, o) in SI_s.Edges do
9:                      SI_o ← GET SubjectInformation of o
10:                     append SI_o to data
11:                 end for
12:             end if
13:             Store data to disk
14:         end for
15:     end for
```

E. Optimal Hyperparameter Values for No-Singleton

A detailed overview of the hyperparameters used for the no-singletons ablation study is in Table IX.

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	64	0.0
GCN-2	0.1	64	0.0
GraphSAINT-2	0.1	64	0.2
GraphSAGE-2	0.1	64	0.2

(a) Best hyperparameter values for M_{AC}

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.0
GCN-2	0.1	64	0.5
GraphSAINT-2	0.1	64	0.2
GraphSAGE-2	0.1	64	0.5

(b) Best hyperparameter values for M_{CC}

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	64	0.2
GCN-2	0.1	64	0.5
GraphSAINT-2	0.1	32	0.0
GraphSAGE-2	0.1	64	0.2

(c) Best hyperparameter values for M_{PTC}

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.5
GCN-2	0.1	64	0.2
GraphSAINT-2	0.1	32	0.0
GraphSAGE-2	0.1	64	0.2

(d) Best hyperparameter values for $M_{\text{SX-6\_499}}$

	learning rate μ	hidden layer size	temperature τ	weighting coefficient α
M_{AC}	0.01	256	1.0	100
M_{CC}	0.01	256	1.0	100
M_{PTC}	0.01	256	2.0	10
$M_{\text{SX-6\_499}}$	0.01	256	2.0	100

(e) Best hyperparameter values for the different summary models for Graph-MLP.

TABLE VII: Best hyperparameter values for the different graph summary models and for Graph-MLP.

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SX-6\_499}}$
MLP-2	11,525,017	15,968,832	12,561,832	34,149,257
GCN-2	11,525,017	998,592	12,561,832	2,151,497
GraphSAINT-2	22,895,705	2,004,800	12,563,880	2,153,545
GraphSAGE-2	22,887,449	1,996,544	24,945,128	4,284,746
Graph-MLP-1	45,678,809	4,058,944	49,778,216	–
Graph-MLP-2	–	–	–	8,617,353

TABLE VIII: Parameter count of MLP-2 and the different graph neural network models in PyTorch Geometric.

F. Parameter Count for No-Singleton

Table X shows the parameter count for the no-singleton ablation study.

G. Implementation Details

For the GNNs, we used the already implemented functionality provided by the PyTorch library [36] and the PyTorch Geometric extension [37] for graphs. For Graph-MLP we adapted the original PyTorch code, distributed by its authors, to fit our data representation of PyTorch Geometric.

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.5
GCN-2	0.1	32	0.2
GraphSAINT-2	0.1	64	0.2
GraphSAGE-2	0.1	64	0.0

(a) Best hyperparameter values for $M_{\text{AC-NS}}$ without singleton classes.

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	64	0.5
GCN-2	0.1	64	0.0
GraphSAINT-2	0.1	64	0.2
GraphSAGE-2	0.1	64	0.0

(b) Best hyperparameter values for $M_{\text{CC-NS}}$ without singleton classes.

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.0
GCN-2	0.1	64	0.2
GraphSAINT-2	0.1	32	0.0
GraphSAGE-2	0.1	64	0.0

(c) Best hyperparameter values for $M_{\text{PTC-NS}}$ without singleton classes.

	learning rate μ	hidden layer size	temperature τ	weighting coefficient α
M_{AC}	0.01	256	2.0	1.0
M_{CC}	0.01	64	1.0	10.0
M_{PTC}	0.01	256	1.0	1.0

(d) Best hyperparameter values for the different summary models for Graph-MLP without singleton classes.

TABLE IX: Best hyperparameter values for the different graph summary models after the removal of singleton classes.

	$M_{\text{AC-NS}}$	$M_{\text{CC-NS}}$	$M_{\text{PTC-NS}}$
MLP-2	61,706,382	991,962	66,340,407
GCN-2	1,972,110	991,962	4,192,887
GraphSAINT-2	7,761,038	1,991,642	4,194,935
GraphSAGE-2	7,752,782	1,983,386	8,335,991
Graph-MLP-1	15,526,798	996,250	16,688,695

TABLE X: Parameter count of the different graph summary models for the no singleton ablation study.

To run our experiments we used machine 1 (CPU: AMD EPYC 7F32 3.89 GHz; 1.96TB RAM) and machine 2 (CPU: AMD EPYC 7302 3.297 GHz; GPU: 4x NVidia A100-SXM4-40GB; 504GB RAM). Both machines were used for different tasks to utilize the most out of every machine. On machine 1 all the preprocessing to generate the data, the Bloom filter evaluation, creation of plots and statistics were done using the faster CPU cores. Since the higher parallelization capability of GPUs allow faster training and inference times we used the GPUs of machine 2 for the hyperparameter search, GNN training, and the cross-validation.

H. Further Ablation Studies

We provide further insights by ablations and more detailed analyses. Section VII reports the impact of removing the singleton classes from the graphs. Below, we additionally investigate the differences occurring when using 1-hop GNNs instead of 2-hop. Finally, we widen our MLP to test the impact of a bigger hidden layer size, as suggested by Galke et al. [27].

1) 1-hop GNNs for 1-hop Graph Summaries: In our main experiments, we only apply 2-hop graph neural network models. Because of the smaller parameter count than their 1-hop counterpart, the computational cost to train them is also lower. The increase in size of GNN-1 models is caused by the missing hidden layer. But most of our summary models consider only the information contained in their direct neighbours, in other words they consider 1-hop of information. By applying 1-hop neural models to those summaries, we investigate if neural models with a matching hop-number show a better performance or any other behavior.

We investigate this by choosing the two best performing GNN-2 models per graph summary from Table II to run a 10-fold cross-validation as an ablation study on their 1-hop counterparts. For comparison, we also run this study with Graph-MLP-1. For a 1-hop GCN, we reduce Equation 2 to

$$f_{\text{GCN}}(\boldsymbol{X}, \boldsymbol{A}) = \text{softmax}\big(\hat{\boldsymbol{A}} \cdot (\boldsymbol{X}\boldsymbol{W}^{(0)})\big)\,. \tag{6}$$

Due to the substantially higher computational costs of these GNN-1 models we could not apply all summary models as described in our main experiment. We use a smaller portion of the dataset to reduce the computational requirements. This smaller dataset was built by using 25% of the root vertices of the whole dataset and therefore we call this dataset DyLDO-25%. We use the DyLDO-25% dataset for the graph summary models $M_{\text{AC-25}}$, $M_{\text{CC-25}}$ and $M_{\text{PTC-25}}$. The number of equivalence classes of these models can be seen in Table XI and their class distributions in Section I. Also, we excluded M_{SX} since it is a 2-hop graph summary.

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
Number of Classes	60,024	411	66,184

TABLE XI: Number of equivalence classes for $M_{\text{AC-25}}$, $M_{\text{CC-25}}$ and $M_{\text{PTC-25}}$ in the DyLDO-25% dataset.

We use the learning rates, we found during the hyperparameter optimization, Section IV-B, of the GNN-2 models. We can reuse the hyperparameter values since the structure of the datasets is the same. The parameter count for the 1-hop ablation study can be found in Table XII. We run the experiments with the Adam optimizer for 75 epochs, except for GraphSAGE-1 and $M_{\text{PTC-25}}$. For the latter, we use SGD and 400 epochs, which was necessary because of the higher memory usage of the Adam optimizer.

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
GCN-1	901,440,432	6,172,398	993,951,312
GraphSAGE-1	1,802,820,840	12,344,385	1,987,836,440

TABLE XII: Parameter count of the different graph summary models for the 1-hop ablation study.

In Table XIII, the results of the models per graph summary are listed. All Bloom filter results are within a range of 0.0018. $M_{\text{AC-25}}$ and $M_{\text{PTC-25}}$ show a larger result range than

$M_{\text{CC-25}}$. In contrast to our main experiment it could not be observed that a higher n and lower p always lead to a higher accuracy, but the results only differ at the fourth decimal digit. Overall the standard errors are lower compared to their 2-hop counterparts. The GCN-1 results for $M_{\text{AC-25}}$ and $M_{\text{PTC-25}}$ have increased accuracy measures. For the graph summary $M_{\text{CC-25}}$, the accuracy measures are $\sim$2% lower. The biggest difference in accuracy are $\sim$10% for GraphSAGE-1 and $M_{\text{PTC-25}}$. Graph-MLP-1 is the top performing GNN model for $M_{\text{AC-25}}$ and $M_{\text{PTC-25}}$ and the worst performer for $M_{\text{CC-25}}$, as in our main experiment. Also, for $M_{\text{CC-25}}$ and $M_{\text{PTC-25}}$ the Graph-MLP-1 models have a higher standard error.

We found that the 1-hop GNN models do not improve the overall results. This is likely due to the massively increased parameter count of the models.

2) Widened MLP: In our last ablation study, we explore the MLP-baseline model by widening it to check if the accuracy increases. This is motivated by Galke et al. [27]. For this, we take the MLP-2 model from our experiments and increase the hidden layer size to 2,048 and 4,096, respectively. Table XIV shows the parameter count for the widened MLP ablation study.

The results can be seen in Table XV. The accuracy does slightly improve for 2,048, but does not outperform the other graph-based models. However, for 4,096 hidden nodes, the scores actually decrease for M_{AC} and M_{PTC}. The standard error also increases considerably for those graph summaries.

Since this ablation study shows that widening the MLP-2s did not improve the accuracy scores significantly, we conclude that it is not possible to improve the MLPs any further. Also since Graph-MLP-1 with $\alpha = 0$ corresponds to a normal MLP-2, the comparison of MLP and Graph-MLP (see Table II) shows the importance of the contrastive loss function.

I. Further Class Distributions

In Figure 4 a comparison of the class distribution of M_{SX} and $M_{\text{SX-6\_499}}$ can be seen. Figure 5 shows the class distribution of the summary models on the DyLDO dataset after removal of the singleton classes. The class distribution of the summary models on DyLDO-25% can be seen in Figure 6.

J. Bloom Filter Impurity Results

The Bloom filter impurity results for DyLDO-25% can be seen in Table XVI.

K. Detailed Discussion of Simple-GCN

We also considered using the SGC model by Wu et al. [20]. Their goal is to reduce the complexity of GCN by removing non-linearities and by condensing the weight matrices of single layers into a combined one. This results in the following simplification of Equation 2

$$f_{\text{SGC}}(\boldsymbol{X}, \boldsymbol{A}) = \text{softmax}\Big(\hat{\boldsymbol{A}}^k \cdot (\boldsymbol{X}\boldsymbol{\Theta})\Big) \tag{7}$$

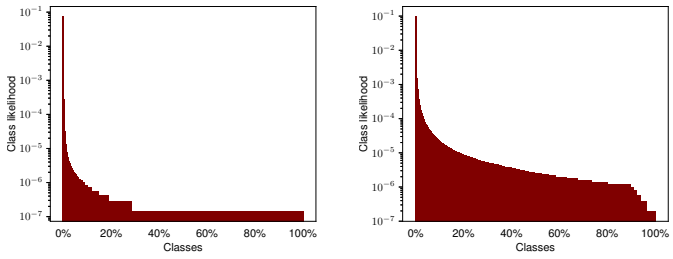
with $\boldsymbol{\Theta} = \boldsymbol{W}^{(0)}\boldsymbol{W}^{(1)}\dots\boldsymbol{W}^{(k)}$ for k-hops. The 1-hop SGC is equivalent to the 1-hop GCN (see Equation 6 and Equation 7 for $k = 1$ and 1-layer GCN).

In a few pre-experiments on the DyLDO dataset with only $30M$ edges, we call this dataset DyLDO-30M, SGC scored considerably lower than the other GNNs (see Table XVII), and was also the biggest model, which pushed us to the limits of our GPU memory. In our main experiments on the DyLDO dataset, the number of classes increased for M_{AC}, M_{PTC} and M_{SX} which in turn increased the number of parameters for our models. Now the SGC model exceeded the GPU memory. Because it also had a low performance in our pre-experiments, we decided to exclude this from our main paper.

For our experiments we report the results for the summary models on DyLDO-30M as $M_{\text{AC-30M}}$, $M_{\text{CC-30M}}$ and $M_{\text{PTC-30M}}$.

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
Bloom filter ($n = 4$, $p = 10^{-7}$)	0.8566	0.8566	0.8229
Bloom filter ($n = 15$, $p = 10^{-7}$)	0.8582	0.8562	0.8244
Bloom filter ($n = 60$, $p = 10^{-7}$)	0.8581	0.8561	0.8247
GCN-1	0.6066 ± 0.0042	0.8315 ± 0.0024	0.5709 ± 0.0017
GraphSAGE-1	0.6064 ± 0.0032	0.8340 ± 0.0025	0.4647 ± 0.0036
Graph-MLP-1	0.6202 ± 0.0044	0.7945 ± 0.0102	0.5796 ± 0.0101

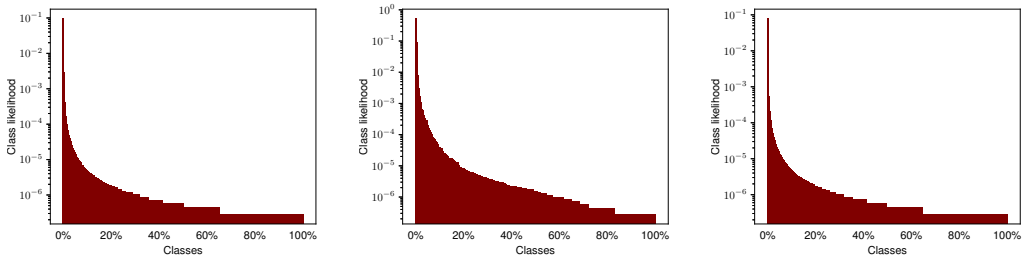
TABLE XIII: Results for 10-fold cross-validation of the 1-hop message-passing GNNs and Graph-MLP-1 reported by mean accuracy with standard error on the test fold (higher accuracy, lower standard error better). For direct comparison, the Bloom filter results are shown, too. The n-values for Bloom filter are chosen based on the 75th, 95th and 99th percentile of the node degree distribution. There was no cross-validation applied to the Bloom filter results for the same reason as stated in Section V.



(a) Class distribution M_{SX} on the whole DyLDO dataset.

(b) Class distribution $M_{\text{SX-}\textit{6\_499}}$ on the DyLDO-*6_499* dataset.

Fig. 4: Class distribution of M_{SX} (left) vs $M_{\text{SX-}\textit{6\_499}}$ (right)



(a) Class distribution $M_{\text{AC-NS}}$

(b) Class distribution $M_{\text{CC-NS}}$

(c) Class distribution $M_{\text{PTC-NS}}$

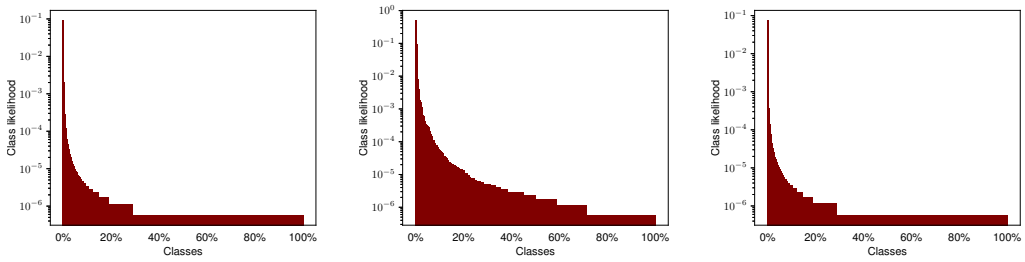
Fig. 5: The class distributions of the different graph summary models on the DyLDO dataset after removal of the singleton classes. The x-axis shows the classes sorted in descending likelihood.

	M_{AC}	M_{CC}	M_{PTC}
MLP-2(original)	11,525,017	15,968,832	12,561,832
MLP-2-2048	363,762,393	31,937,088	396,445,992
MLP-2-4096	727,362,265	63,873,600	792,713,512

TABLE XIV: Parameter count of the different graph summary models for the widened MLP ablation study.

	M_{AC}	M_{CC}	M_{PTC}
MLP-2(original)	0.6033 ± 0.0048	0.8496 ± 0.0067	0.5706 ± 0.0030
MLP-2-2048	0.6052 ± 0.0079	0.8510 ± 0.0040	0.5902 ± 0.0061
MLP-2-4096	0.6012 ± 0.0067	0.8307 ± 0.0128	0.5713 ± 0.0039

TABLE XV: Results for 10-fold cross-validation for wider hidden layer sizes MLP (higher accuracy, lower standard error better). The model names are composed by adding to MLP-2 the respective number of hidden vertices. The parameter count for these models can be found in Section H2.



(a) Class distribution $M_{\text{AC-25}}$ (b) Class distribution $M_{\text{CC-25}}$ (c) Class distribution $M_{\text{PTC-25}}$

Fig. 6: Class distributions of $M_{\text{AC-25}}$ (top-left), $M_{\text{CC-25}}$ (top-right) and $M_{\text{PTC-25}}$ (bottom)

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
$n = 4$, $p = 10^{-7}$	0.0502	0.0604	0.0626
$n = 15$, $p = 10^{-7}$	0.0498	0.0606	0.0622
$n = 60$, $p = 10^{-7}$	0.0498	0.0606	0.0621

TABLE XVI: Impurity measure for Bloom filter using the Gini-index on DyLDO-25% with expected input items n and false positive probability p (lower impurity better).

	$M_{\text{AC-30M}}$	$M_{\text{CC-30M}}$	$M_{\text{PTC-30M}}$
SGC-2	0.2079 ± 0.0027	0.2542 ± 0.0036	0.0818 ± 0.0012
GraphSAGE-2	0.6929 ± 0.0044	0.8771 ± 0.0023	0.6389 ± 0.0041

TABLE XVII: 10-fold cross-validation results for SGC (higher accuracy, lower standard error better).