Graph Summarization with Graph Neural Networks

Maximilian Blasi
Universität Ulm
Ulm, Germany
maximilian.blasi@uni-ulm.de

Manuel Freudenreich
Universität Ulm
Ulm, Germany
manuel.freudenreich@uni-ulm.de

Johannes Horvath
Universität Ulm
Ulm, Germany
johannes.horvath@uni-ulm.de

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

Ansgar Scherp
Universität Ulm
Ulm, 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 pre-defined features of a graph's vertex within a k-hop neighborhood such as the vertex labels and edge labels. Based on this 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 pre-defined 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 (GNN) 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 GraphMLP model outperforms the other GNNs. The performance of the standard MLP is extraordinary good, especially in the presence of many classes. Finally, 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 shows an interaction effect between the number of classes for the summarization task and the chosen method.

KEYWORDS

neural networks, graph neural networks, graph summary, rdf-graph

1 INTRODUCTION

Graph summaries provide a condensed representation of an input graph. The goal is to preserve pre-defined features that are relevant for specific tasks such as queries on the summary [4]. Graph summaries are used for tasks such as estimating cardinalities [25], data search [21], and data visualization [10]. In this paper, we consider graph summaries based on equivalence classes [4]. Such summaries are lossless in regards to the considered features such as the vertices' edges, the vertex neighbors, and others. With the recent rise of graph neural networks (GNN), we wanted to investigate how well GNNs are suited to computing graph summaries.

In general, a graph summary 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 M considers different properties of neighbouring 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.

Each equivalence class is characterized based on the features defined in the summary model M. The aggregation of the information from a vertex's subgraph is required to be isomorphism-invariant (i. e., it must depend on the structure of the graph itself, and not, for example, on the order in which the edges are listed in the database), which can be seen in Figure 1.

The calculation of the equivalence class can be done using deterministic computations like a hash function [22]. A hash function is used to aggregate the information in a vertex v's neighborhood [21]. The considered properties of the neighbouring vertices have to be sorted before hashing to fulfill the isomorphism-invariance requirement, since hash functions are inherently permutation-variant.

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 [13], which also provides the desired effect for this case. The application of neural networks for hash functions is mainly researched under a security perspective [23, 32]. Their 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 2.3.

We consider different GNN models, both based on the popular message-passing architecture, namely graph convolutional network (GCN), GraphSAGE, and GraphSAINT and an approach that

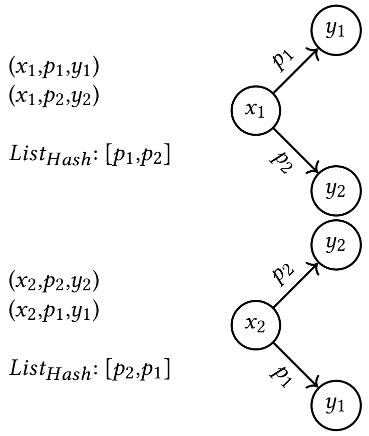


Figure 1: Left-hand side shows a representation of the graph on the right-hand side by a set of triples. While the two graphs are actually identical, the order of triples in the lists differ. In a naive approach of hashing the corresponding lists of triples, different hash values and thus different equivalence 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 3.1).

does not use message-passing, GraphMLP. GCN applies a message-passing technique based on the vertex embedding and edges for vertex classification. 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. GraphMLP removes the message passing technique and replaces it with a precomputation and MLP-like network. To be able 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 multi-layer perceptron (MLP) as baseline model on a binary encoded feature matrix (one-hot encoded labels). We compare a Bloom filter approach and measure its performance to compute graph summaries. The Bloom filter is used as a non-neural baseline and also circumvents the need to sort the relevant feature values needed for summarizing vertices. Overall, we evaluate six different methods: four GNNs, one MLP, and the Bloom filter.

We use the DyLDO dataset [19] to train and test the different classifier models. The dataset consists of a subset of Linked Data which is generated by crawling from a set of seed-URIs. The DyLDO dataset is chosen as the base dataset because the data is stored in a standardized way and it is representative of real-world use of graphs on the web, but its size is still manageable. We apply four different popular graph summary models 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 vertex's neighboring vertices' labels [4]. We use the term *node* for the use in the neural network context and *vertex* in the graph context. In summary, our research questions are:

- **Q 1:** Can we produce a graph summary via a vertex classification based on GNNs? How accurate is a graph summary calculated by a GNN?
- **Q 2:** How do different classifiers perform against each other at calculating a graph summary? How do more sophisticated models perform against a baseline?
- **Q 3:** 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 [31]. 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 GraphMLP outperforms the message-passing GNNs. The performance of the standard MLP is close to GNNs and ranked third and fourth in three of the experiments. Particularly in the presence of many classes, the standard MLP is on par with the GNNs. We run ablation studies on the DyLDO dataset, where singleton class occurrences are removed, which improved the results but interestingly not for all models. We also 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 in order to answer the research questions in Section 3. The experimental apparatus, Section 4, contains an introduction into the dataset, experimental procedure, implementation, and evaluation metrics. The results of our experiments are reported in Section 5. We discuss the results in Section 6, before we further investigate our experimental results with ablation studies in Section 7.

2 RELATED WORK

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

2.1 Graph Summarization

Graph summaries are a good structural representation of large graph datasets [4]. Example use cases for graph summaries are data exploration [3], data visualization [11], and vocabulary term recommendations [29].

Graph. The structure of a graph $G = (V, E, R)$ can be described as a combination of vertices $v_i, v_j \in V$, labeled edges $(v_i, r, v_j) \in E$, and the relation type $r \in R$, which lead to a description of the data and their relationships.

RDF Graphs and `rdf:type`. In order to answer the research questions, Resource Description Framework (RDF) datasets were used. The framework is a way to conceptually describe or model information as a graph. It is structured in triples of *subject*, *predicate*, and *object* as (s, p, o)-triples [1]. 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 [6]. A special predicate in the RDF standard is the `rdf:type`. The triple $(s_i,$ `rdf:type`, $o_j)$ simulates that the vertex s_i has a vertex label o_j. An edge predicate $p \neq$ `rdf:type` is called an RDF property.

Graph Summary. The idea of graph summarization is to generate a condensed representation SG from an input graph G [4]. The result is a summarized graph that preserves selected structural information. To this end, the neighborhood of vertices is gathered and combined in a specific way defined by the summary model M. The summary model M defines what features of vertices and/or neighborhoods are gathered. These features are "compressed" with the help of a hash function and this results in a equivalence class, which is used as a class label. The label is attached to the so called root vertex, which is the starting point for the described neighborhood. The advantage of a graph summary is that the execution time for different tasks is much faster on the summary than on the original graph [4], e. g., counting vertices which have the same information in the neighborhood.

FLUID. A good way to calculate graph summaries is to use FLUID [4]. 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.

2.2 Graph Neural Networks for Vertex Classification

For the following section, we define $\boldsymbol{A}$ as an adjacency matrix and $\boldsymbol{D}$ as 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 $\tilde{\boldsymbol{A}}$.

GCN. To run a classification task on graphs, Kipf et al. [20] suggest a convolutional network 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 $G = (V, E)$ using message-passing. Here, $\boldsymbol{X}$ denotes a matrix containing vectors of node features, $\hat{\boldsymbol{A}}$ is the normalized symmetric Laplacian of $\tilde{\boldsymbol{A}}$, and $\boldsymbol{W}^{(i)}$ denotes the weight matrix of the hidden layer i. The symmetric Laplacian is calculated by $\hat{\boldsymbol{A}} = \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$. The weights are symmetrically normalized by the vertex degree. For message-passing, each vertex generates a message based on its embedding and sends this message to all its neighbours. Each vertex then aggregates the received messages in an permutation invariant manner and updates its own embedding with the aggregation result. To be able to handle semi-supervised training data, the function $f(\boldsymbol{X}, \boldsymbol{A})$ is conditioned through the loss function on the vertices which have labels. But during training and inference there are also vertices present that do not have any label. Kipf et al. [20] show that a two layer GCN results in a 2-hop aggregation.

R-GCN. Schlichtkrull et al. [30] 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.

SimpleGCN. Another approach is proposed by Wu et al. [34]. Their goal is to reduce the complexity of GCN by removing nonlinearities and by condensing the weight matrices of single layers into a combined one. The result is a simplification of Equation 2.

GCNs are limited by their usage of memory and because batching is not possible, because the weight matrices $\boldsymbol{W}$ and node feature matrices $\boldsymbol{X}$ are dimensioned on the full adjacency matrix $\boldsymbol{A}$. This means the whole graph has to be loaded into the memory. Thus, only a transductive training is possible. These shortcomings can be mitigated by GraphSAINT and GraphSAGE.

GraphSAINT. Zeng et al. [37] suggest using 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 utilizing 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.

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

Graph Sampling and Aggregate (GraphSAGE). GraphSAGE [14] aggregates the information in a k-neighborhood into a vertex embedding of the current root vertex of the neighborhood using a so-called $aggregation\,function/aggregator$. Hamilton et al. [14] 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

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

There are also further sampling-based models: Chen et al. [7] 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. [18] 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 use of such models and further works on GNNs for future experimentation.

GraphMLP. Hu et al. [17] 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.

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

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

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

For feature transformation, a neighbouring 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 r^{th} power of the normalized adjacency matrix $\hat{\boldsymbol{A}}^r$ and a cosine similarity with a temperature parameter τ. To the output layer $\boldsymbol{Y}$, cf. 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 $loss_{total} = loss_{CE} + \alpha \cdot loss_{NC}$.

2.3 Neural Networks as Hash Functions

The goal of a graph summary is the partitioning of the vertices into equivalence classes, where each class can be represented using a class label. These class labels can be generated by using a hash function. The application of neural networks as a hash function is researched mainly in the context of security. Turčaník et al. [32] initialize a neural network with random weights and biases. Lian et al. [23] 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, those security aspects as discussed in Lian et al. [23] 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.

2.4 Bloom Filter

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 [28]. 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*. In order to query a membership on these Bloom filters, 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].

3 MODELS

First, we explain how to compute graph summaries and provide an overview of the different 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.

3.1 Computing Graph Summaries

The main goal of this work is the calculation of the *SG* using neural network techniques. For training and evaluation, it is essential to determine the class of a vertex. The class depends on the specified summary model M_{i}. The class labels are calculated in a "traditional", lossless way to establish a ground truth and to use them later as target labels for neural network based approaches.

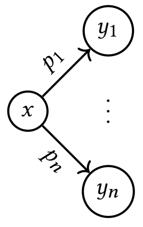
The information of vertex v is defined through all the (s, p, o)-triples with $s = v$. The summary model M_{i} defines which triples are considered for each subject. The vertex information is gathered in a list (see Figure 1) and sorted. Based on that list, the equivalence class for each vertex is determined. 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 [22].

The sorting algorithm timsort [16] 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 [27]. Note, one could have used a generic framework like FLUID, see Appendix B.

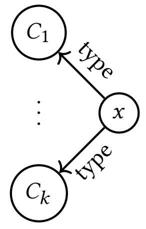
For the graph summary model Attribute Collection M_{AC} (see Figure 2a), the information used as described above is the property set. The property set of a vertex is its set of outgoing edge labels, excluding `rdf:type`. The Class Collection M_{CC} (see Figure 2b) can be seen as the other side of the 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_{SEX} can be described as a combination of summary model specifications of multiple M_{i}. It is shown in Figure 2d. For M_{SEX}, 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 correct neighbouring vertices into a single list. The calculation of the hash for the equivalence class is then the same as for the other M.

3.2 GNNs

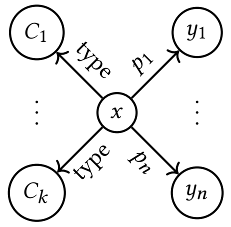
We give a more detailed description of the application of the selected GNN models chosen from Section 2.2. We use a ReLU-activation function for all our hidden layers, while a softmax function is used



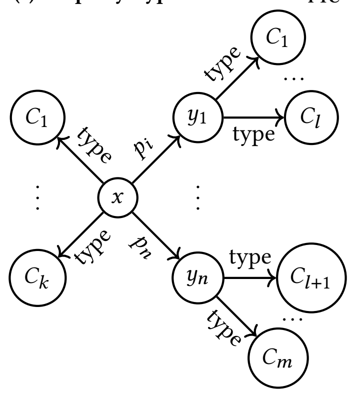
(a) Attribute Collection M_{AC}



(b) Class Collection M_{CC}



(c) Property Type Collection M_{PTC}



(d) SchemEX M_{SEX}

Figure 2: The summary models M_{AC}, M_{CC}, M_{PTC}, and M_{SEX} considered in this work. (x: root vertex, y: object connected to x through property set, c: object connected to x or y through type set)

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 [13, Chapter 5.2]. This issue can be solved by applying normalization. But we do not normalize prior to the aggregation function since the goal of graph summaries is to preserve the graph structure and therefore 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 have to apply a sampler on our extensive dataset. Here, we are following the proposition of 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 in a semi-supervised transductive training fashion [13, 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.

GCN. To be able 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.

R-GCN. Compared to the datasets used in the original paper [30], the DyLDO dataset is about 7 times larger vertex-wise, 20 times larger edge-wise, 110 times larger edge type-wise, and 10k times larger class-wise. Since the complete model has to be present during training in our setup and R-GCN has additional weights per edge type, the R-GCN model exceeds our hardware limitations massively, even with weight sharing techniques like basis-decomposition [30].

SimpleGCN. In a few pre-experiments on a smaller dataset SimpleGCN 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 DyLDO dataset the number of classes increased, and now the SimpleGCN model exceeded the GPU memory. Because of the two mentioned reasons we decided to exclude SimpleGCN from our experiments. For more details see Appendix C.

Graph Sampling and Aggregate (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 [14] show, this limitation can be ignored.

GraphSAINT. We replace GraphSAINT's vertex sampler by one based on the inverse class distribution. We apply a GraphSAINT network with 2 layers with jumping knowledge or 1 layer of Weisfeiler–Leman kernels.

GraphMLP. We use GraphMLP [17], but again we remove the normalization layer, for the same reason as stated above. We use GraphMLP with a modification. 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 GraphMLP to the k-hop characteristic of each graph summary model. The specific r-value is appended to the model name.

We report the results as *GNN-k* with *GNN* corresponding to the neural model and k to the number of hops. 2-hop models consist of two layers and with dropout in between.

3.3 Baselines

3.3.1 MLP. As a baseline, we use a standard multi-layer perceptron (MLP) with two hidden layer and ReLU-activation function with dropout. This is motivated by the work on the performance of MLPs for text classification by Galke et al. [9].

3.3.2 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. In order 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, this new approach 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 possible 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^{-7}, 10^{-3} and 10^{-1}. These combinations result in different numbers of hash functions and bits in the array. Details can be found in Appendix A.1.

4 EXPERIMENTAL APPARATUS

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

4.1 Dataset

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

4.1.1 Dataset and Basic Statistics. Dynamic Linked Data Observatory (DyLDO) [19] is a framework for monitoring Linked Data over time. It takes frequent snapshots of a subset of Linked Data in order to capture raw data about the dynamics of Linked Data. DyLDO takes a sampling-based approach of seed-URIs, which are also representative for the vertices of the graph. One weekly snapshot has about 100 million triples. The dataset is populated by crawling from a list of seed-URIs. The crawling of these seed URIs is restricted to a crawling depth of two hops [19]. We decided to use this as our dataset due to its relatively manageable size and the possibility of using different, even smaller, snapshots of the same dataset.

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 [5] shows that it contains 7,093,011 vertices with 15,017 overall edge properties. From these edge properties, `rdf:type` occurs $5.4M$ times. Each vertex has on average 17 outgoing properties.

DyLDO-6_499. As M_{SEX} is the only graph summary model requiring a second hop, there is a huge increase in observed vertices and consequently there is also an increase 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 root vertices with a class occurrence of < 6, as this leaves ~15% of the dataset which fits our hard drive restrictions. Also only subgraphs with ≤ 499 vertices are kept, because in our sampling process during inference, we use this as a guard condition as a maximum mini-batch size. We call this dataset DyLDO-*6_499* and likewise we call M_{SEX} on this dataset $M_{\mathrm{SEX}\text{-}6\_499}$. This dataset has $16.7M$ 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.

4.1.2 Datasets Class Distribution. By executing the M_{i} introduced in Section 3.1 on our dataset, we generate different datasets specific for the graph summary models. We can now see how many equivalence classes each M_{i} generates. This can be seen in Table 1.

In Figure 3, we plotted the class distribution for each M_{i} 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 $M_{\mathrm{SEX}\text{-}6\_499}$ (Figure 3d) all show a skewed distribution due to the majority of classes appearing only once. For reference, the class distribution of M_{SEX} can be seen in Appendix A.1.

4.2 Hyperparameter Optimization

We applied a hyperparameter search to the learning rate over the values in $\{0.1, 0.01, 0.001\}$ of the optimizer, dropout in $\{0.0, 0.2, 0.5\}$, and the hidden layer size $\{32, 64\}$ for the GNNs with 2-hops. For GraphMLP, 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 for this purpose. For the search we trained our models for 30 epochs. We manually confirmed the optimization process by inspecting the loss curve. We ensured that the models properly converged. The number of training epochs for our experiments also were based on convergence.

The validation accuracy is the mean over 15 mini-batches from the validation fold. For our MLP, we also look at a hidden layer size of 1,024 following Galke and Scherp [9]. For GraphMLP, we use the authors' recommended dropout of 0.6 without any hyperparameter optimization of the dropout value. We present the optimal GNNs' hyperparameters for the different M_{i} in Appendix A.1 and Appendix A.2.1. Interestingly, some models work best without any dropout and the dropout values are overall low.

An overview over all parameter counts for the competing models is given in Table 2. The parameter count depends per neural network and graph summary model on the chosen hidden layer hyperparameter.

4.3 Procedure

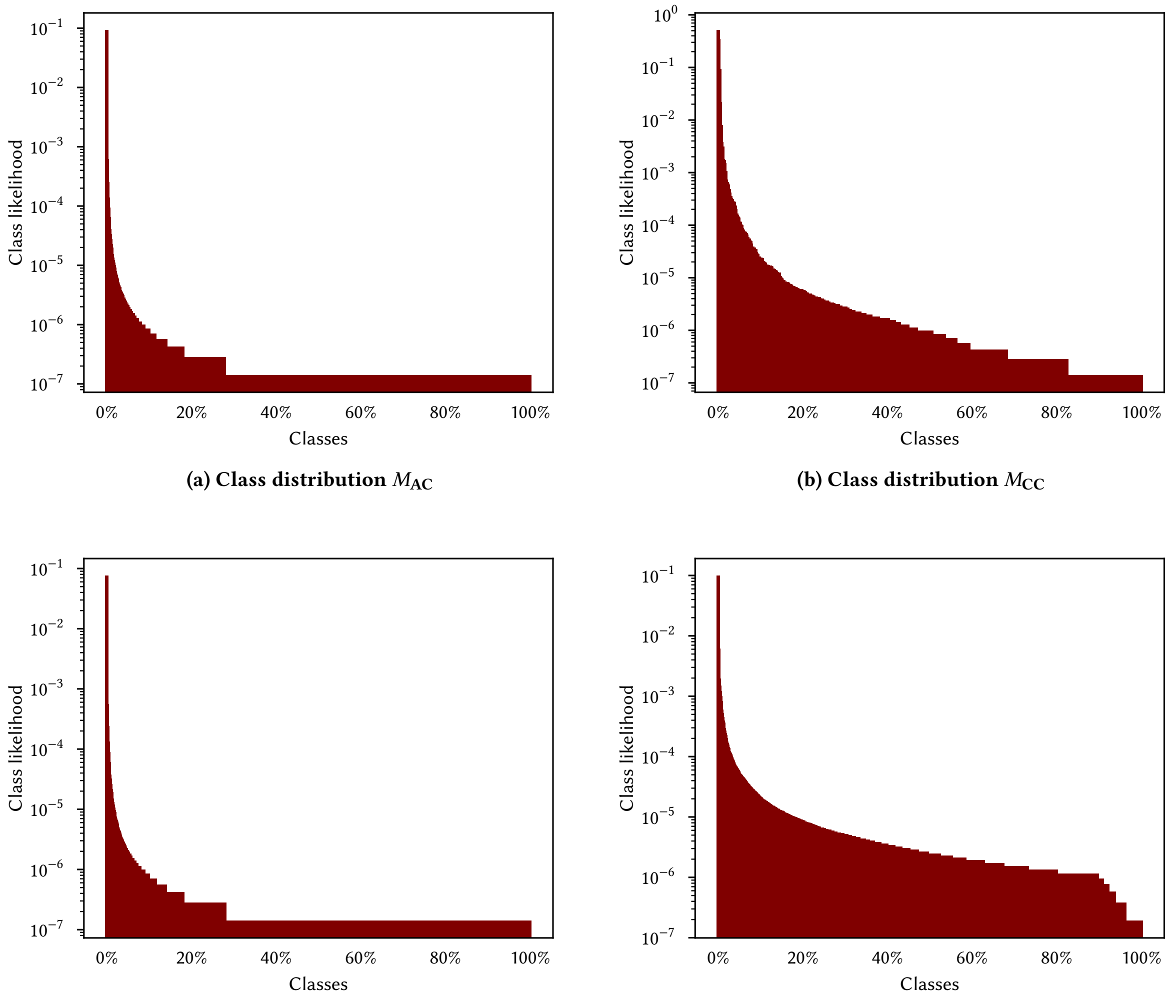
Our goal is to use the equivalence classes of the graph summaries as labels for our classification task. Given our big dataset, we also have to apply a sampling method and train our model on mini-batches. To be able to apply the GNNs on our data, we have to apply the following pre-processing steps. Afterwards, we explain the training and the evaluation process.

Pre-processing. 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 (see *SubjectInformation* in Algorithm 1). In the structure *SubjectInformation*, the equivalence class label, all the neighbours, and the fold number of a subject vertex is stored.

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

	M_{AC}	M_{CC}	M_{PTC}	M_{SEX}	$M_{\text{SEX-}\mathit{6\_499}}$
Number of Classes	162,521	576	178,472	335,608	18,314

Table 1: Number of equivalence classes in the DyLDO dataset according to M_i. The column $M_{\text{SEX-}\mathit{6\_499}}$ denotes the number of M_{SEX} classes in the DyLDO-*6_499* dataset.



(a) Class distribution M_{AC}

(b) Class distribution M_{CC}

(c) Class distribution M_{PTC}

(d) Class distribution $M_{\text{SEX-}\mathit{6\_499}}$

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

ten folds by assigning each subject vertex randomly to a fold. So a fold consists of the subgraphs of its subjects. The data structure for a subgraph is outlined as *GraphData* in Algorithm 1. 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. For that, we apply the Algorithm 2 to generate data for training, validation, and test based on subgraphs using *Subject-Information* for each subject in the dictionary. Here, we take into account the different hop characteristics of the graph summary models.

Training Machine Learning Models. After the pre-processing, the neural networks are trained on mini-batches from the training folds

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SEX-}6\_499}$
MLP-2	11,525,017	15,968,832	12,561,832	34,149,257
sample-based 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
GraphMLP-1	45,678,809	4,058,944	49,778,216	–
GraphMLP-2	–	–	–	8,617,353

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

Algorithm 1 SubjectInformation and GraphData

```
struct {
  array[num_nodes, num_features]
     feature matrix;
  list[num_nodes] class label;
  list[num_edges] edges e=(s,o);
  list[num_edges] edge type p of (s,p,o);
  int num_nodes; //reference to the number
                   of nodes in GraphData
} GraphData;

struct {
  int ClassLabel;
  list(p,o) Edges;
  int fold;
} SubjectInformation;
```

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 $data \leftarrow \emptyset$
5:              $SI_s \leftarrow$ 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:                      $SI_o \leftarrow$ GET SubjectInformation of $o$
10:                     transform $SI_o$ into GraphData and append to $data$
11:                 end for
12:             end if
13:             Store $data$ to disk
14:         end for
15:     end for
```

and then tested on the mini-batches from the test fold. The training process is monitored and evaluated on the validation accuracy.

To generate the mini-batches with a constant mini-batch size *guard*, we apply Algorithm 3 with a set of folds *Foldset* and the *guard* condition representing the desired batch size. In order 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.

Algorithm 3 Create mini-batch for training from the foldset

```
1: procedure CreateMiniBatch(f: foldset, $guard$: integer)
2:     GraphData $data \leftarrow \emptyset$
3:     Draw subgraph sample $data_{sample}$ from f
4:     while $data.num\_nodes + data_{sample}.num\_nodes < guard$ do
5:         Append $data_{sample}$ to $data$
6:         Draw next subgraph sample $data_{sample}$ from f
7:     end while
8:     return $data$
```

Only the root vertices of the subgraphs are considered in the loss and accuracy calculation, ignoring the object neighbours 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 4.2, were used. The models are trained for 30 epochs, except for GraphMLP-1 and M_{CC} which were trained for 25 epochs.

Evaluation Procedure. 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 then tested it on 75 mini-batches of 1 test fold to report a mean testing accuracy per run.

4.4 Implementation

We implemented our experiment setup in Python. For the GNNs we used the already implemented functionality provided by the Pytorch-library [26] and the PyTorch Geometric extension [8] for graphs. For GraphMLP we adapted the original PyTorch-code, distributed by the authors, to fit our data representation of PyTorch Geometric.

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 be able to utilize the most out of every machine. On machine 1 all the pre-processing 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.

4.5 Metrics

To evaluate our models for the different graph summaries, we use an accuracy rating on the classification result. The test and validation accuracy are calculated via the mean over multiple sampled batches on the corresponding fold to gain more stability. We report our results as 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 3.1 as ground truth and the hash values calculated by the Bloom filter method (see Section 3.3.2) 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 with size $|\Omega| = N_\Omega$. Each cluster $\omega \in \Omega$ can be clustered again based on the ground truth values of each subject vertex, resulting in the clusters C and $|C| = N_C$. 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 [36] and the probability $p_{ij} = \frac{m_{ij}}{|\omega_i|}$ with m_{ij}, the number of objects from C_j in ω_i, see Equation 6a. The final impurity value, Equation 6b, is calculated via the sum over all the clusters weighted by their relative frequency $\frac{|\omega_i|}{N}$.

$$Q_g(\omega_i) = 1 - \sum_{j=1}^{N_C} p_{ij}^2 \tag{6a}$$

$$Q_g(\mathcal{R}) = \sum_{i=1}^{N_\Omega} \frac{|\omega_i|}{N} Q_g(\omega_i) \tag{6b}$$

5 RESULTS

For our results, we applied 10-fold cross-validation for all our models and all M_i. The only exception where no cross-validation was applied is Bloom filter, as Bloom filter is a deterministic algorithm and would yield the exact same results each time. The results can be seen in Table 3.

The Bloom filter outperforms all tested neural models, by a margin that depends on the model M_i. The second best result for M_AC is 0.2333 lower (GraphMLP-1), M_PTC is 0.2256 lower (GraphMLP-1) and $M_{\text{SEX-}\mathit{6\_499}}$ is 0.2278 lower (GraphMLP-2). For M_CC the Bloom filter and sample-based GCN-2 performance is similar. The results for neural network based models of M_AC is within 0.0372, M_CC is within 0.0158, M_PTC is within 0.0521, and $M_{\text{SEX-}\mathit{6\_499}}$ 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 $M_{\text{SEX-}\mathit{6\_499}}$. The lowest standard error is 0.0025 for sample-based GCN-2 and M_CC.

Table 4 shows the impurity metric for the differently parameterized Bloom filter.

The Bloom filter accuracy results for every parameter combination within a single M_i differ always at most by 0.0237 from each other (see Table 3). 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.

6 DISCUSSION

Main Results. Our experiment uses real-world datasets. Thus, the number of classes is much higher than in commonly used datasets [35], 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.

Bloom filters deliver overall the best and most consistent results. The choice of parameters does not seem to have a huge 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 ~0.8563 for M_AC, ~0.8562 for M_CC, ~0.8224 for M_PTC, and ~0.7767 for $M_{\text{SEX-}\mathit{6\_499}}$. The impurity results converge at ~0.2018 for M_AC, ~0.2423 for M_CC, ~0.2517 for M_PTC, and ~0.2198 for $M_{\text{SEX-}\mathit{6\_499}}$. This strong performance can be explained because Bloom filters approximate set memberships 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.

The different M_i have different numbers of class labels (see Table 1). This means that with an increasing number of equivalence classes, the complexity also increases because the classification requires more differentiation. The influence of complexity can be observed in two ways. The accuracy results of our different neural network based models are worse for M_i with a higher class count and these results differ more within an M_i 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 lowest number of classes by a large margin (see Table 1). An exception to this observation is the results for $M_{\text{SEX-}\mathit{6\_499}}$. 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.

Since every M_i is computed best by a different neural model, we cannot make a decisive conclusion as to which neural network model is the best. This is a known observation from [31]. However, we observe that GraphSAGE always outperforms GraphSAINT. The good performance of GraphSAGE (among the top-3 NN 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 [31]. The random neighborhood sampling and the learning

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SEX-6\_499}}$
Bloom filter ($n = 4$, $p = 10^{-1}$)	0.8376	**0.8562**	0.7987	0.7598
Bloom filter ($n = 4$, $p = 10^{-3}$)	0.8529	0.8562	0.8186	0.7757
Bloom filter ($n = 4$, $p = 10^{-7}$)	0.8547	0.8562	0.8206	0.7764
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 ($n = 15$, $p = 10^{-7}$)	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
Bloom filter ($n = 60$, $p = 10^{-7}$)	0.8563	0.8562	0.8224	0.7767
MLP-2	**0.6033** ± 0.0048	0.8496 ± 0.0067	**0.5706** ± 0.0030	**0.5470** ± 0.0127
sample-based GCN-2	0.6007 ± 0.0060	**0.8559** ± 0.0025	0.5611 ± 0.0047	**0.5473** ± 0.0052
GraphSAINT-2	0.5858 ± 0.0047	**0.8538** ± 0.0026	0.5447 ± 0.0062	0.5080 ± 0.0076
GraphSAGE-2	**0.6138** ± 0.0052	**0.8541** ± 0.0027	**0.5652** ± 0.0079	0.5354 ± 0.0053
GraphMLP-1	**0.6230** ± 0.0057	0.8401 ± 0.0031	**0.5968** ± 0.0037	–
GraphMLP-2	–	–	–	**0.5489** ± 0.0066

Table 3: 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 75*th*, 95*th*, and 99*th* 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 (see Table 10). Best viewed in color.

	M_{AC}	M_{CC}	M_{PTC}	$M_{\text{SEX-6\_499}}$
$n = 4$, $p = 10^{-1}$	0.2232	0.2423	0.2793	0.2363
$n = 4$, $p = 10^{-3}$	0.2057	0.2423	0.2560	0.2207
$n = 4$, $p = 10^{-7}$	0.2035	0.2423	0.2536	0.2202
$n = 15$, $p = 10^{-1}$	0.2026	0.2423	0.2526	0.2202
$n = 15$, $p = 10^{-3}$	0.2019	0.2423	0.2518	0.2198
$n = 15$, $p = 10^{-7}$	0.2018	0.2423	0.2517	0.2198
$n = 60$, $p = 10^{-1}$	0.2018	0.2423	0.2517	0.2198
$n = 60$, $p = 10^{-3}$	0.2018	0.2423	0.2517	0.2198
$n = 60$, $p = 10^{-7}$	0.2018	0.2423	0.2517	0.2198

Table 4: Impurity measure for Bloom filter using the gini-index with expected input items n and false positive probability p (lower impurity better).

of the aggregration 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 may be surprising, but MLP has performed well in other classification tasks [33]. The GraphMLP model, which does not use message passing and introduces a MLP-like structure, was only tested on smaller datasets in the original paper [17]. Thus, for the first time it has been applied on large graphs with large numbers of classes. Our results show that it outperforms the other neural models on these larger datasets and that it is the best neural model in three out of four graph summary experiments.

It is noticeable that for M_{PTC} GraphSAINT-2 has the lowest hidden layer size and also the lowest accuracy score. Comparing the results, Table 3, and the hyperparameter values, Appendix A.1, also shows that a higher hidden layer size seems to lead to a higher standard error.

Threats to Validity. Since the standard errors are low and consistent within an M_{i}, one can trust in the results. A threat to the generalization could be that we only used one dataset. However, we investigated four different graph summary models that generate different graphs.

The applied GNN models cannot handle directed edges. This limitation might impact the classification performance for our task, because graph summaries are sensitive to the edge direction. However, as discussed in Section 3.2, the graph sampler we applied considers the direction of the edges.

The exact same results for the Bloom filter for M_{CC} would suggest some kind of error in the procedure. These results have been double checked. Furthermore, we executed the same procedure for M_{CC} as for the other M_{i}. This confirms the correctness of these results.

7 ABLATION STUDIES

In this section, we provide further studies to our topic. First, we study the impact of removing the singleton classes from the graphs. Then, we 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. [9].

7.1 No Singletons

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 M_{i} with removed singletons $M_{\text{AC-NS}}$, $M_{\text{CC-NS}}$, and $M_{\text{PTC-NS}}$. On average 46% of classes remain, see Table 5. In Appendix A.2.1 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. The used parameters can be seen in Appendix A.2.1. We did not include SchemEX, because in the dataset used for $M_{\text{SEX-}\mathit{6\_499}}$, DyLDO-*6_499*, we are already filtering with a *min_support* higher than 1.

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

Table 5: Number of equivalence classes in the DyLDO dataset according to M_i after the removal of classes that only occur once , the percentage of remaining classes (cf. Table 1), the number of the original 7,093,011 subgraphs remaining, and the percentage of the remaining subgraphs.

The results are reported via the average and standard error of the test accuracy for a 10-fold cross-validation, see Table 6, after training for 40 epochs. There is an increase in accuracy for $M_{\text{AC-NS}}$ and $M_{\text{PTC-NS}}$ by ~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 ~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 and also since a three-way dataset split is applied, further studies could be done on a dataset with *min_support* > 2 to consider this data split.

7.2 1-hop GNNs for 1-hop Graph Summaries

In our main experiment 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 subsequently also lower. The increase in size of GNN-1 models is caused by the missing hidden layer. But most of our M_i 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 M_i we investigate if neural models with a matching hop-number show a better performance or any other behaviour.

So, we investigate this by choosing the two best performing GNN-2 models per graph summary from Table 3 to run a 10-fold cross-validation as an ablation study on their 1-hop counterparts. For comparison we also run this study with GraphMLP-1. For a 1-hop GCN we reduce Equation 2 to Equation 7:

$$f_{\text{GCN}}(\boldsymbol{X}, \boldsymbol{A}) = \text{softmax}\left(\hat{\boldsymbol{A}} \cdot (\boldsymbol{X}\boldsymbol{W}^{(0)})\right). \tag{7}$$

Due to the substantially higher computational costs of these GNN-1 models we could not apply all M_i as described in our main experiment. Thus we used 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%. Thus we used DyLDO-25% for the graph summary models and applied $M_{\text{AC-25}}$, $M_{\text{CC-25}}$ and $M_{\text{PTC-25}}$. The number of equivalence classes of these models can be seen in Table 7 and their class distributions in Appendix A.2.2. Also we excluded M_{SEX} since it is a 2-hop graph summary.

We use the learning rates we found during the hyperparameter optimization, Section 4.2, of the GNN-2 models. We can reuse the hyperparameter values since the structure of the datasets are the same. In Table 8 the chosen models per graph summary are listed.

We run the experiments with an 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.

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 ~2% lower. The biggest difference in accuracy are ~10% for GraphSAGE-1 and $M_{\text{PTC-25}}$.

GraphMLP-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 GraphMLP-1 models have a higher standard error.

We found that the 1-hop GNN models do not improve the overall results. We explain this with the massively increased parameter count of the models.

7.3 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. [9]. For this we, take the MLP-2 model from our experiments and increase the hidden layer size to 2,048 and 4,096, respectively. This leads to results that can be seen in Table 9. The accuracy does only 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 GraphMLP-1 with $\alpha = 0$ corresponds to a normal MLP-2, the comparison of MLP and GraphMLP (see Table 3) shows the importance of a contrastive loss.

8 FUTURE WORK

Future studies should be conducted on the topic on a dataset where not only singletons are removed, but the minimum support of the class occurrence is matched to the number of data splits. 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

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

Table 6: 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. The parameter count for these models can be found in Appendix A.2.1.

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

Table 7: Number of equivalence classes for $M_{\text{AC-25}}$, $M_{\text{CC-25}}$ and $M_{\text{PTC-25}}$ in the DyLDO-25% dataset.

not remove single class occurrence nodes but which maps the class onto the next best-fitting class [12].

We suggest considering more large-scale and real-world datasets in the experiments for proposing new GNN models and for surveys on models. The frequently used datasets are smaller ones like citation graphs [35]. The biggest dataset in [30] has 1.67M vertices, with 1,000 vertices being labeled and having 11 classes and 133 relations. Our dataset has orders of magnitude more classes and relations (see Section 4). While trying to apply the R-GCN model to our datasets, we ran into memory limitations. 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 [15] or the LOD Laundromat [2].

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 SimpleGCN.

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 $M_{\text{SEX-}6\_499}$.

Another future study could be a runtime analysis. Sorting and hashing operations for the calculation of a lossless graph summary are replaced by permutation-invariant, but lossy, GNN models. This could be extended into a cost-benefit analysis for the application of those methods comparing the computation performance gain to the accuracy loss.

9 CONCLUSION

We methodically compared the effectiveness of computing graph summaries with established and recent graph neural network models. Bloom filters outperformed the GNN models by a large margin in three out of four cases. The GNNs performed well for the M_{CC} graph summary model. The differences between the considered GNN models are rather small and this explains the different rankings of the best performing graph models. The GraphMLP model without a classical message-passing pipeline is the best GNN model in three of the four cases. 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 an accuracy drop with increasing the numbers of equivalence classes. The performance of our non-neural baseline, the Bloom filter, seemed to be independent of this number while the performance of the GNNs dropped with larger numbers of classes.

ACKNOWLEDGMENTS

We thank the data science paper reading club of the University Ulm 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.

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

REFERENCES

[1] David Beckett. 2014. RDF 1.1 N-TRIPLES. https://www.w3.org/TR/n-triples/. Accessed: 2021-04-17.
[2] Wouter Beek, Laurens Rietveld, Hamid R Bazoobandi, Jan Wielemaker, and Stefan Schlobach. 2014. LOD laundromat: a uniform way of publishing other people's dirty data. In *International semantic web conference*. Springer, 213–228.
[3] Fabio Benedetti, Sonia Bergamaschi, and Laura Po. 2015. Exposing the Underlying Schema of LOD Sources. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, Vol. 1. 301–304. https://doi.org/10.1109/WI-IAT.2015.99
[4] Till Blume, David Richerby, and Ansgar Scherp. 2021. FLUID: A common model for semantic structural graph summaries based on equivalence relations. *Theoretical Computer Science* 854 (2021), 136 – 158. https://doi.org/10.1016/j.tcs.2020.12.019
[5] Till Blume and Ansgar Scherp. 2020. Indexing Data on the Web: A Comparison of Schema-Level Indices for Data Search. In *International Conference on Database and Expert Systems Applications*. Springer, 277–286.
[6] Gavin Carothers, Lex Machina, Inc. 2014. RDF 1.1 N-Quads. https://www.w3.org/TR/n-quads/. Accessed: 2021-04-17.
[7] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. arXiv:1710.10568 [stat.ML]
[8] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. http://arxiv.org/abs/1903.02428 cite arxiv:1903.02428.

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
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
sample-based 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
GraphMLP-1	0.6202 ± 0.0044	0.7945 ± 0.0102	0.5796 ± 0.0101

Table 8: Results for 10-fold cross-validation of the 1-hop for the message-passing GNNs reported by mean accuracy with standard error on the test fold (higher accuracy, lower standard error better). 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 5. The parameter count for these models can be found in Appendix A.2.2. The impurity results for the Bloom filter can also be found in Appendix A.2.2.

	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 9: 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 Appendix A.2.3.

[9] Lukas Galke and Ansgar Scherp. 2021. Forget me not: A Gentle Reminder to Mind the Simple Multi-Layer Perceptron Baseline for Text Classification. arXiv:2109.03777 [cs.CL]

[10] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. 2020. RDF graph summarization for first-sight structure discovery. *VLDB J.* 29, 5 (2020), 1191–1218. https://doi.org/10.1007/s00778-020-00611-y

[11] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. 2020. RDF graph summarization for first-sight structure discovery. *The VLDB Journal* 29, 5 (April 2020), 1191–1218. https://doi.org/10.1007/s00778-020-00611-y

[12] Thomas Gottron, Ansgar Scherp, Bastian Krayer, and Arne Peters. 2013. LODatio: using a schema-level index to support users infinding relevant sources of linked data. In *Proceedings of the 7th International Conference on Knowledge Capture, K-CAP 2013, Banff, Canada, June 23-26, 2013*, V. Richard Benjamins, Mathieu d'Aquin, and Andrew Gordon (Eds.). ACM, 105–108. https://doi.org/10.1145/2479832.2479841

[13] William L Hamilton. 2020. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14, 3 (2020), 1–159.

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI]

[15] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. 2019. BTC-2019: The 2019 Billion Triple Challenge Dataset. In *International Semantic Web Conference*. Springer, 163–180.

[16] Angela Heumann. 2018. The magic behind the sort algorithm in Python. https://medium.com/ub-women-data-scholars/the-magic-behind-the-sort-algorithm-in-python-1cb9515294b5 Accessed: 2021-06-18.

[17] Yang Hu, Haoxuan You, Zhecan Wang, Zhicheng Wang, Erjin Zhou, and Yue Gao. 2021. Graph-MLP: Node Classification without Message Passing in Graph. *arXiv preprint arXiv:2106.04051* (2021).

[18] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. *CoRR* abs/1809.05343 (2018). arXiv:1809.05343 http://arxiv.org/abs/1809.05343

[19] Tobias Käfer, Jürgen Umbrich, Aidan Hogan, and Axel Polleres. 2012. Towards a dynamic linked data observatory. *LDOW at WWW* (2012).

[20] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[21] M. Konrath, T. Gottron, S. Staab, and A. Scherp. 2012. SchemEX - Efficient construction of a data catalogue by stream-based indexing of linked data. *J. Web Semant.* 16 (2012), 52–58. https://doi.org/10.1016/j.websem.2012.06.002

[22] Mathias Konrath, Thomas Gottron, Steffen Staab, and Ansgar Scherp. 2012. SchemEX: Efficient Construction of a Data Catalogue by Stream-Based Indexing of Linked Data. *SSRN Electronic Journal* (01 2012). https://doi.org/10.2139/ssrn.3198981

[23] Shiguo Lian, Jinsheng Sun, and Zhiquan Wang. 2007. One-way hash function based on neural network. *arXiv preprint arXiv:0707.4032* (2007).

[24] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.* Cambridge university press.

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

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[27] Python. 2021. https://github.com/python/cpython/blob/bb3e0c240bc60fe08d332ff5955d54197f79751c/Python/pyhash.c#L245 Accessed: 2021-06-18.

[28] Ori Rottenstreich and Isaac Keslassy. 2014. The bloom paradox: When not to use a bloom filter. *IEEE/ACM Transactions on Networking* 23, 3 (2014), 703–716.

[29] Johann Schaible, Thomas Gottron, and Ansgar Scherp. 2016. 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*, Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange (Eds.). Springer International Publishing, Cham, 101–117.

[30] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. arXiv:1703.06103 [stat.ML]

[31] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Pitfalls of Graph Neural Network Evaluation. *CoRR* abs/1811.05868 (2018). arXiv:1811.05868 http://arxiv.org/abs/1811.05868

[32] Michal Turčaník and Martin Javurek. 2016. Hash function generation by neural network. In *2016 New Trends in Signal Processing (NTSP)*. IEEE, 1–5.

[33] Ishwar Venugopal, Jessica Töllich, Michael Fairbank, and Ansgar Scherp. 2021. A Comparison of Deep-Learning Methods for Analysing and Predicting Business Processes. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*. IEEE, 1–8. https://doi.org/10.1109/IJCNN52387.2021.9533742

[34] Felix Wu, Tianyi Zhang, Amauri H. Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. *CoRR* abs/1902.07153 (2019). arXiv:1902.07153 http://arxiv.org/abs/1902.07153

[35] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. 2016. Revisiting Semi-Supervised Learning with Graph Embeddings. *CoRR* abs/1603.08861 (2016). arXiv:1603.08861 http://arxiv.org/abs/1603.08861

[36] Ye Yuan, Liji Wu, and Xiangmin Zhang. 2021. Gini-Impurity Index Analysis. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3154–3169. https://doi.org/10.1109/TIFS.2021.3076932

[37] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2019. GraphSAINT: Graph Sampling Based Inductive Learning Method. *CoRR* abs/1907.04931 (2019). arXiv:1907.04931 http://arxiv.org/abs/1907.04931

APPENDIX

A ADDITIONAL STATISTICS

This section is a collection of additional tables and figures that were too specific for the main paper.

A.1 Main Experiment

Table 10 shows the number of hash functions and the bits of the Bloom filter array for our used Bloom filter parameters.

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 10: 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

In Figure 4 a comparison of the class distribution of M_{SEX} and $M_{\text{SEX-}6\_499}$ can be seen.

A detailed summary of our used hyperparameters for the specific M_{i} and neural network models can be found in Table 11.

A.2 Ablation Studies

A.2.1 No Singletons. Figure 5 shows the class distribution of different M_{i} on the DyLDO dataset after removal of the singleton classes. In Table 12 a detailed overview of the used hyperparameters for the no singletons ablation study can be found.

Table 13 shows the parameter count for the no singleton ablation study.

A.2.2 1-hop GNNs for 1-hop Graph Summaries. The class distribution of the different M_{i} on DyLDO-25% can be seen in Figure 6.

Similarly the parameter count for the 1-hop ablation study can be found in Table 14.

Also the Bloom filter impurity results for DyLDO-25% can be seen in Table 15.

A.2.3 Widened MLP. And Table 16 shows the parameter count for the widened MLP ablation study.

B FLUID-FRAMEWORK

The FLUID-framework uses an algorithm based on hash maps and creates its own data structures to be able to calculate all the different graph summary models in an efficient way. Because we only implemented a selection of them the graph summary calculation is based on that but simplified.

C DETAIL DISCUSSION OF SIMPLE-GCN

At the start we also considered using SimpleGCN. In a few pre-experiments on the DyLDO dataset with only 30M edges, we call this dataset DyLDO-30M, SimpleGCN scored considerably lower than the other GNNs, see Table 17, 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_{SEX} which in turn increased the number of parameters for our models. Now the SimpleGCN 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.

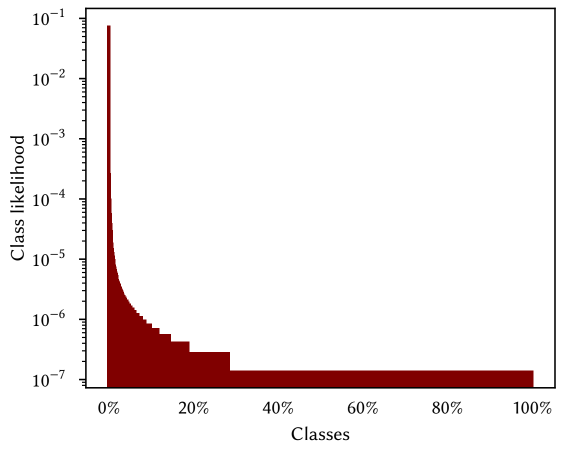
Related Work. Another approach is proposed by Wu et al. [34]. Their goal is to reduce the complexity of GCN by removing nonlinearities and by condensing the weight matrices of single layers into a combined one. This results in the following simplification of Equation 2

$$f_{\text{SGC}}(\boldsymbol{X}, \boldsymbol{A}) = \text{softmax}\Big(\hat{\boldsymbol{A}}^k \cdot (\boldsymbol{X}\boldsymbol{\Theta})\Big) \tag{8}$$

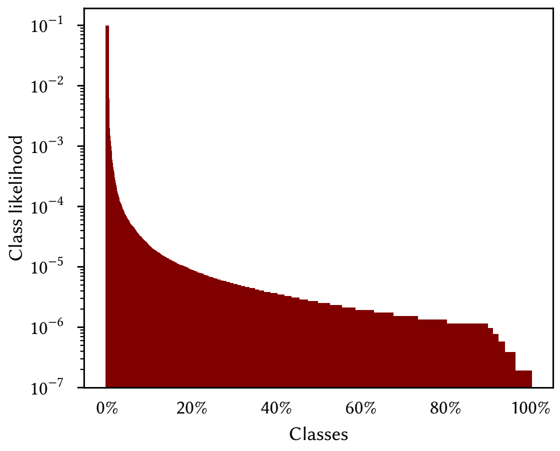
with $\boldsymbol{\Theta} = \boldsymbol{W}^{(0)}\boldsymbol{W}^{(1)} \dots \boldsymbol{W}^{(k)}$ for k-hops.

Model. We are using a vanilla SimpleGCN model without any adaption. The 1-hop SimpleGCN is equivalent to the 1-hop GCN (see Equation 7 and Equation 8 for $k = 1$ and 1-layer GCN).

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



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



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

Figure 4: Class distribution of M_{SEX} (left) vs $M_{\text{SEX-}6\_499}$ (right)

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	64	0.0
sample-based 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
sample-based 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
sample-based 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}

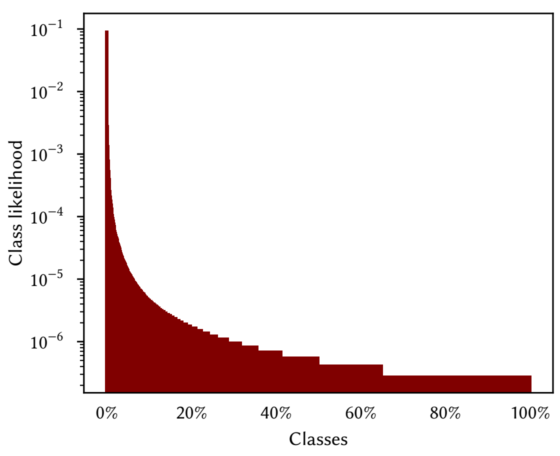
	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.5
sample-based GCN-2	0.1	64	0.2
GraphSAINT-2	0.1	32	0.0
GraphSAGE-2	0.1	64	0.2

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

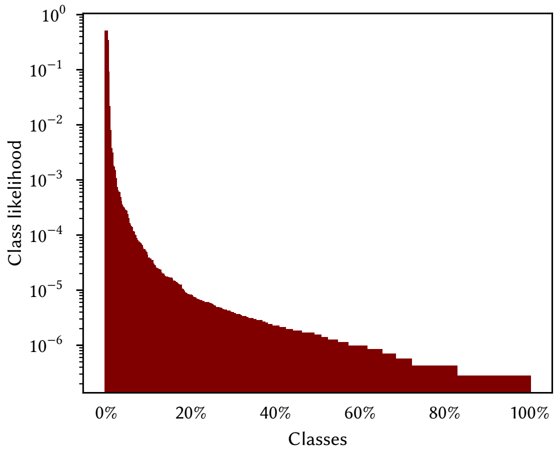
	learning rate μ	hidden layer size	temperature τ	weighting coefficient α
M_{AC}	0.01	256	1.0	100
M_{CC}	0.01	256	1.0	100
M_{PTC}	0.01	256	2.0	10
$M_{\text{SEX-}6\_499}$	0.01	256	2.0	100

(e) Best hyperparameter values for the different summary models for GraphMLP.

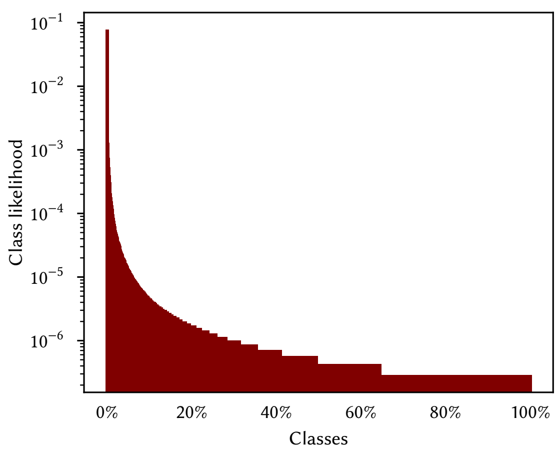
Table 11: Best hyperparameter values for the different graph summary models and for GraphMLP.



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



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



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

Figure 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.

	learning rate μ	hidden layer size	dropout
MLP-2	0.1	1024	0.5
sample-based 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
sample-based 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
sample-based 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 GraphMLP without singleton classes.

Table 12: 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
sample-based 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
GraphMLP-1	15,526,798	996,250	16,688,695

Table 13: Parameter count of the different graph summary models for the no singleton ablation study.

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

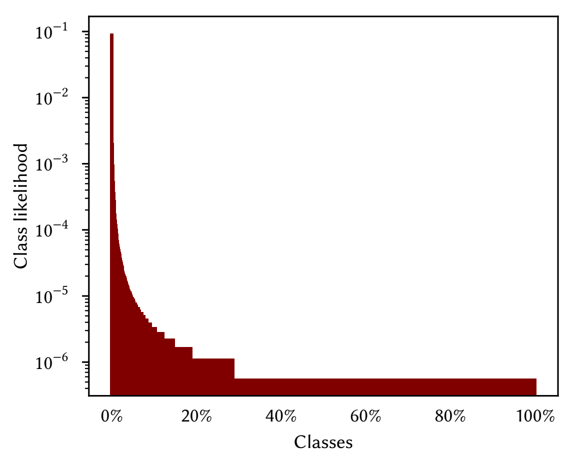
Table 14: Parameter count of the different graph summary models for the 1-hop ablation study.

	$M_{\text{AC-25}}$	$M_{\text{CC-25}}$	$M_{\text{PTC-25}}$
$n = 4, p = 10^{-7}$	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

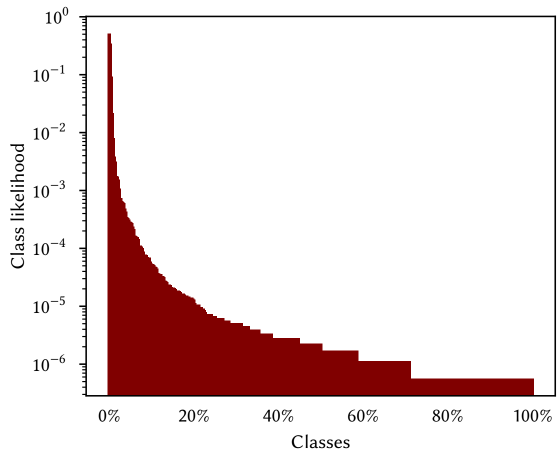
Table 15: Impurity measure for Bloom filter using the gini-index on DyLDO-25% with expected input items n and false positive probability p (lower impurity better).

	M_{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

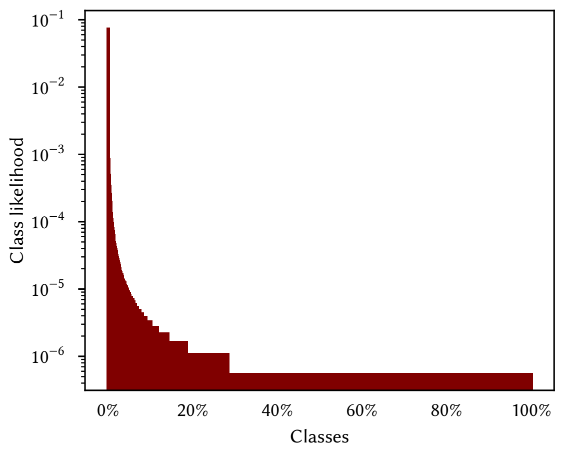
Table 16: Parameter count of the different graph summary models for the widened MLP ablation study.



(a) Class distribution $M_{\text{AC-25}}$



(b) Class distribution $M_{\text{CC-25}}$



(c) Class distribution $M_{\text{PTC-25}}$

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

	$M_{\text{AC-30M}}$	$M_{\text{CC-30M}}$	$M_{\text{PTC-30M}}$
SimpleGCN-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 17: 10-fold cross-validation results for SimpleGCN (higher accuracy, lower standard error better).