Graph Summarization with Graph Neural Networks - Technical Report

Maximilian Blasi, Manuel Freudenreich, Johannes Horvath

March 9, 2022

Contents

1 Introduction

This technical report belongs to the paper *Graph Summarization with Graph Neural Networks* by Johannes Horvath, Maximilian Blasi and Manuel Freudenreich, in which the application of neural networks for the purpose of graph summaries, in comparison to non-neural baselines, is tested. This document describes the technical details of the experiments in order to ensure verifiability and correctness of the experiments. In Section 2, we explain the principal architecture and features of our experimental setup. Section 3 describes how we implemented it and in Section 4 we illustrate how to use our setup.

2 Application Design

We first introduce our code structure as listing of the important code files in alphanumerical order. In Section 2.2 we list then the structure of input, artifact, and result files. The listed files are only the key files required to run our experiments. Utils and debugging scripts, which are not required to replicate our results are, not listed.

2.1 Code Structure

- **lib**
  - **config_utils**
    - config_util.py: wrapper functions for the modules *configparser* and *pathlib*.
  - **dyldo_rdflib**
    - file_filter.py: filters invalid RDF-Quads.
  - **graph_summary_generator**
    - graph_summary_generator.py: generates internal data structure as so called graph information, saves and loads graph information, generates graph summaries out of graph information and save as folds, functions as getter for information like features, classes, and nodes, and plots the class distribution.
  - **neural_model**
    - gcn.py: GCN model based on torch_geometric.nn.GCNConv layers.
    - graphmlp.py: GraphMLP model from [1] and adapted by us to the pytorch_geometric API.
    - mlp.py: MLP model based on GCNConv layers.
    - **obsolete**: contains obsolete models which were not used for the final experiment
      - rgcn.py: RGCN model based on torch_geometric.nn.RGCNConv layers (listed for completion purposes, not part of our main experiments because of computation limitations).
      - sgcn.py: SimpleGCN model based on torch_geometric.nn.SGConv layers.
    - sage.py: GraphSAGE model based on torch_geometric.nn.SAGEConv layers.
    - saint.py: a model based on GraphSAINT, which is based on torch_geometric.nn.GraphConv layers.
    - utils.py: wrapper functions for the modules *torch* and *pandas*.
  - **utils**
    - bloom_metrics.py: utility functions to calculate the metrics for the Bloom filter.
    - plot_class_distribution.py: plotting function for element dictionary (key=element, value=count of element).
- **src**
  - **cross_validation**

    - config_cross_validation.ini: config for cross_validation.py.
    - config_cross_validation_graphmlp.ini: config for cross_validation_graphmlp.py.
    - cross_validation.py: create a model (GCN, GraphSAGE, GraphSAINT and MLP) and run a k-fold crossvalidation. The results are stored in a csv-file.
    - cross_validation_graphmlp.py: create a GraphMLP model and run a k-fold cross-validation. The results are stored in a csv-file.
- **evaluation**
    - cross_eval.py: calculate mean and std_dev over the values given in argument –file.
    - config_eval_bloomfilter.ini config for eval_bloomfilter.py
    - config_eval_bloomfilter_reduced_schemex.ini config for eval_bloomfilter_reduced_schemex.py.
    - eval_bloomfilter.py: evaluate the Bloom filter results.
    - eval_bloomfilter_reduced_schemex.py: evaluate the Bloom filter results for the reduced SchemEX dataset.
    - plot_val_acc_epoch.py: load TensorBoard summaries and parse them; store the results in dictionary and plot chosen results from the dictionary.
- **gnn_training**
    - config_run_training.ini: config for run_training.py.
    - config_run_training_graphmlp.ini: config for run_graphmlp_training.py.
    - run_training.py: create a model (MLP, GCN, GraphSAGE, GraphSAINT) and train it. Can be used for the hyperparameter search.
    - run_graphmlp_training.py: create a GraphMLP model and train it.
- **preprocess**
    - config_preprocess_bloomfilter.ini: config for run_preprocessing.py.
    - config_preprocessing_bloomfilter_schemex.ini: config for run_preprocessing_schemex.py.
    - config_preprocess_filter.ini: config for run_preprocessing.py.
    - config_preprocess_generate_data.ini: config for run_preprocessing.py.
    - config_preprocessing_schemex.ini: config for run_preprocessing_schemex.py.
    - config_reduce_class_labels.ini: config for reduce_class_labels.py.
    - reduce_class_labels.py: load the panda dataframe from the description files, update the class labels to the new label range and save the changes inplace
    - run_preprocessing.py: run complete preprocessing pipeline of RDF-filter, graph-summary calculation, information generation, and subgraph extraction.
    - run_preprocessing_schemex.py: run complete preprocessing pipeline of RDF-filter, graphsummary calculation, information generation and subgraph concatenation based on preexisting subgraphs of a 1-hop model with filters on the class occurrences and number of vertices in the total subgraph.
- **utils**
    - config_remove_singletons.ini: config for remove_singletons.py.
    - config_stats_schemex.ini: config for stats_schemex.py.
    - graph_info.py: load all description.csv's from the given list and calculate the mean, median, given percentiles and sum of degrees.
    - plot_class_distribution_graph_information.ini: config for plot_class_distribution_graph_information.py.
    - plot_class_distribution_graph_information.py: plot the class distribution of the given graph subject information dict.
    - plot_class_distribution_subgraph.ini: config for fileplot_class_distribution_subgraph.py.

    - plot_class_distribution_subgraph.py: plot the class distribution of the given folds by collecting the information from the description files.
    - remove_singletons.py: remove subjects from the data distribution for classes which do not meet the minimum support class occurrences and update the description file into a new one.
    - stats_schemex.py: print the required statistics for our custom reduced dataset for $M_{\text{SEX-}6\_499}$.
- README.md
- .gitignore

2.2 Data File Structure / Artifacts

base_dir: root dir of all input, artifacts and output *Param[in] DataExchange.basedir*
load_dir: child folder of base_dir *Param[in] DataExchange.loaddir*
run_dir: child folder of base_dir *Param[in] DataExchange.rundir*
save_dir: child folder of base_dir *Param[in] DataExchange.savedir*

DyLDO-Filter Iterate over all n-quads and check if their format is valid [1]. Filter out invalid quads.

- **base_dir**
  - raw_datafile [.nq.gz]: input file of Dyldo *Param[in] Dyldo.raw_datafile*.
  - filtered_datafile [.nq.gz]: output file with remaining valid quads *Param[out] Dyldo.filtered_datafile*.
  - trashed_datafile [.nq.gz]: output file with invalid quads *Param[out] Dyldo.filtered_datafile*.

GraphSummary Calculates the graph summary (*Param[in] GraphSummary.GS-Model*) based on the quads from *Param[in] Dyldo.filtered_datafile* and stores the intermediate results in a dictionary (*Param[in] save_file*) with *key-value*-pairs of *subject-subject_information*. The subject_information contains edges, hash, Bloom Filter hash.

- filtered_datafile [.nq.gz]: input file with valid RDF-quads *Param[in] Dyldo.filtered_datafile*.
- pickled[2] dictionary *Param[out] save_file*

Subgraph Extraction Extract from the RDF-Graph all subjects with their neighbours considering the hop-characteristic of the GraphSummary model and save them as single subgraphs as pytorch geometric data object. The subgraphs are separated by their assigning to the k-folds. Per fold, also a pandadataframe with the columns subject_index.pt, degree, inverse class weight, and class label is stored as a csv file.

- pickled dictionary *Param[in] save_file*
- **run_dir**
  - **GraphSummaryIndex** Output folder *Param[out] GraphSummary.GS-Model*
    - **000** corresponding k-fold of child subgraphs
      - subject_index.pt: pytorch geometric data object representing the subgraph of the root vertex $subjectindex$

[1] N-Quad Standard of W3C at `https://www.w3.org/TR/n-quads/`
[2] We use the adjective "pickled" to express a serialization via the Python module "pickle"

        - ⋮
        - description.csv
    - **001**
        - ⋮
        - description.csv
    - ⋮
    - **k** (with leading zeros till 3 digits)
        - ⋮
        - description.csv

- **load_dir** containing the subgraphs of a 1-hop graph summary which are stitched together for SchemEX.

GNN results For training the training accuracy, loss, validation accuracy and test accuracy per epoch are logged in a **summary**-folder. The last model is saved into a checkpoint-file. The current date is given in %b%d_%H-%M-%Sformat. For the results of the **gnn_training**, the model_checkpoint-files and the **summary**-folders are stored following this naming convention:

```
${current_date}_{model_checkpoint|summary}_${summary_model}_${gnn}_
    ${learning_rate}_${hidden_layer_size}_${dropout}
example: Oct14_00-00-43_model_checkpoint_04_gcn_0-1_64_0-2
```

For the **cross_validation**, the model_checkpoint-files and the **summary**-folders are following the same naming convention as **gnn_training**, but *_${k-fold_iteration}* is appended at the end. Additionally, in a result-file the test accuracies are stored after each iteration of the k-fold cross-validation (example: **Oct13_20-36-59_summary_04_gcn_0-1_64_0-2_5**). For the cross-validation the test accuracies are written into a result-file following the naming convention:

```
result_k_cross_${graphsummary}_${gnn}[.csv]
example: result_k_cross_03_graphsaint.csv
```

- **save_dir**
    - **runs**
        - **summary**
        - ⋮
    - checkpoint
    - ⋮
    - **k-folds**
        - result

3 Implementation

3.1 Dependencies

We implemented our pipeline in Python 3.6.13. Our most important libraries were:

- **PyTorch 1.8.1:** Machine learning framework.
- **PyTorch Geometric 1.6.3:** Extension of PyTorch for GNN and structured data.

3.2 Preprocessing

Our complete preprocessing pipeline is called via src/preprocess/run_preprocessing.py. This pipeline consists of the following three steps: dataset processing, graph summary calculation and subgraph extraction.

3.2.1 Dataset

If there are some non RDF-conform quads in the DyLDO dataset these need to be filtered in order to process the data correctly. This is done in lib/dyldo_rdflib/file_filter.py. We started the implementation using **rdflib 5.0.0**, but through a bug only ∼25% n-quads were loaded, so we replaced the rdflib with an own file reader. The snapshot file containing the DyLDO data is loaded. Every line in the file contains its own quad. Each of these quads is then independently validated by trying to parse it into a valid n-quad. Faulty quads get saved to a trash-file and the correct ones get saved to a filtered-file, giving as a preprocessed RDF-dataset. There is also an option to manually manage "blank nodes" in these quads if needed, this option is activated with the boolean "preskolemize".

3.2.2 Graph Summary Calculation

As the name suggests, the preprocessed RDF-dataset is used to calculate the defined graph summary, via lib/graph_summary_generator/graph_summary_generator.py. We implemented an internal data class for handling the information and data. It is also used to calculate the graph summary and if chosen also using the Bloom filter. The calculation of the graph summary is done by first collecting all features important for the defined graph summary in a list as strings. Those strings are then sorted and concatenated to be able to calculate a hash-value. The hash-value is then used as the label for the equivalence class. In order to reuse the already calculated data the whole object of the internal data class is saved and can be loaded.

Bloom Filter: For the hash calculation of the Bloom filter, we use the library *bloom-filter2*. The size of the filter is defined by the maximum number of features and an error rate of 0.1. The same features used by the normal graph summary calculation are added into a new Bloom filter. For the calculation of the equivalence class the internal bit array of the Bloom filter is hashed.

3.2.3 Subgraph Extraction

As we run into memory limitations because of the dataset size, we need to split the dataset into subgraphs. We extract a subgraph for each subject containing all the neighbours of the root vertex based on the graph summary hop characteristics, following our graph definition including `rdf:type`, in lib/graph_summary_generator/ graph_summary_generator.py. Only the root vertex of the subgraph has a meaningful class label, because the other vertices are objects. The subgraphs are stored as torch_geometric.Data objects and saved following the file structure explained in Section 2.2. For the 1-hop graph summaries, the subgraphs are equal, except the class labels. For SchemEX, we stitch the previously extracted 1-hop subgraphs together to reduce calculations.

3.3 GNN Models

All GNN models are built using standard layers from pytorch geometric. The implementation followed examples and tutorials from pytorch and pytorch geometric. For GraphMLP we adapted the source code from `https://github.com/yanghu819/Graph-MLP` to fit the pytorch geometric API.

3.4 Cross-validation

For the cross-validation, we shift the folds k-times in a circle: the first fold corresponds to the validation data, the second to the test data, and the remaining folds are concatenated into the

training data.

3.5 Evaluation

3.5.1 Bloom Filter Evaluation

As evaluation metrics for the Bloom filter, we calculate an accuracy and the impurity value. These metrics are implemented as described in Section 4.5 in *Graph Summarization with Graph Neural Networks* and are run by executing src/evaluation/eval_bloomfilter.py. To evaluate the Bloom filter results for $M_{\text{SEX-}6\_499}$, which uses a custom reduction, src/evaluation/eval_bloomfilter_reduced_schemex.py can be run. This script applies the same reduction to the dataset as the subgraph extraction during pre-processing, before evaluating the remaining data.

3.5.2 Summary Plots

There are two ways to plot the class distribution of the graph summaries: The first relies on the pickled dictionary we save after the graph summary calculation; the second one utilizes and concatenates the description files from the folds. Based on the loaded data, a histogram of class label and class occurrence count is calculated. The histogram is then sorted by the class count in descending order and transformed into a likelihood. The number of classes which are plotted are reduced to a maximum of $1,000$ classes, because more can not be plotted based on the maximum count of displayable pixels. The likelihood is plotted on a log-scale. The plots are saved as pgfs through the matplotlib.

4 Manual

In this section, an explanation is given how to use our setup to perform an experiment starting from generating the data, to training neural networks and obtaining the results. The first part will cover the data acquisition and pre-processing of the data. After that, we explain how to determine the best hyperparameters for the neural networks. Then we show how to execute the cross-validation and how to evaluate the results. And finally, we give some details and tips that should ease the use of our experimental setup. All the python and config files can be found in our github repository, `https://bit.ly/3Fx2f5u`. The config files have to be adapted to the local environment.

4.1 Pre-processing

To be able to run our experiments, the first step is to download the first ever published DyLDO snapshot, published on 2012-05-06, from `http://km.aifb.kit.edu/projects/dyldo/data/`. To be able to build up our own graph from the data, a filter step is needed to remove all malformed or broken triples. The filter functionality is included in the "run_preprocessing.py" and needs to be executed before the generation of the pickled dictionary and the folds.

```
python run_preprocessing.py --config config_preprocess_filter.ini
```

After filtering the file, we also need to remove existing duplicates which can be done as following.

```
awk '!seen[\$0]++' data-filtered.nq $>$ data-filtered-no-duplicates.nq
```

For our approach of using 10-folds, we combined the generation of the pickled dictionary and the generation of the folds into one config file.

```
python run_preprocessing.py --config config_preprocess_generate_data.ini
```

After executing all of these commands, everything should be ready for the hyperparameter search and the cross-validation for M_{AC}, M_{CC}, and M_{PTC}. The *Param gs* parameter defines for which

of the graph summary models the data is generated. Values for the parameter are $01 = M_{\mathrm{AC}}$, $02 = M_{\mathrm{CC}}$, $03 = M_{\mathrm{SEX}}$, or $04 = M_{\mathrm{PTC}}$.

To save storage space and runtime, we introduced a description file which links to the general generated small subgraphs inside the folds. So for example the folds can be generated for one graph summary and for further graph summaries only new description files need to be created. Therefore the *Param extract_subgraph* is needed to be set to "True" or "False" in combination with *Param calc_graph_summary* also "True" or "False".

For the 1-hop ablation study, we needed to reduce the data so a *Param save_fold_percentage* is introduced which can be defined in a range of $[0, 1]$. The defined percentage of subgraphs are then selected by random sampling from all possible subgraphs inside each fold.

To be able to generate the data for M_{SEX} a few things need to be considered. The main problem is the increased disk space usage, because a lot more data is generated through the second hop. Also the calculation for the subgraphs needs more time. For that, we introduced multiple tricks to be able to work with M_{SEX}. It can be seen that it is just a combination of the smaller already calculated subgraphs. Therefore, we introduced the *Param load_desc_file* to load already generated description files as a base. The subgraphs of those description files will be used to generate the combined subgraphs for M_{SEX}. Because of the limited disk space, the data needs to be filtered. Therefore *Param min_support_classes*, *Param edge_sampling_percentage* and *Param mini_batch_size* were introduced. *Param min_support_classes* is used to not consider all classes of M_{SEX} with occurrence in the data less then the defined value. *Param edge_sampling_percentage* defines a value in the range of $[0, 1]$ for sampling the edges of the first hop not considering the second hop of that edge. *Param mini_batch_size* defines the maximum number of vertices we are able to consider because of the GPU memory limitation.

```
python run_preprocessing_schemex.py --config config_preprocess_schemex.ini
```

For another ablation study, we needed the functionality to remove singleton classes for which we implemented another script. It removes all subgraphs, whose classes have a single occurrence in the data. It is applied to the description files and new description files are generated without the singletons.

```
python remove_singletons.py --config config_remove_singletons.ini
```

Since we use the class index as a label, all class indices for reduced datasets have to be updated after the generation to fit the reduced class index domain. Therefore, another script is needed to fix those indices.

```
python reduce_class_labels.py --config config_reduce_class_labels.ini
```

4.2 Hyperparameter Search and GNN Training

After the data generation, the next step is a hyperparameter search to determine the best parameters for the cross-validation. We used different config-files and executed them all in parallel or in sequence.

```
python run_training.py --config config_run_training.ini
```

The results are different logs of the validation accuracy and of the loss function. The training process is also logged using the TensorBoard summary writer. The TensorBoard tool can also be used to visualize the logs.

4.3 Cross-validation

To gain a deeper understanding of the results and the distribution of the results, we applied a 10-fold cross validation.

```
python cross_validation.py --config config_cross_validation.ini
```

The evaluation of the cross-validation is printed at the end of the script via the mean and standard error over the 10 test accuarcies. The single results are also saved in a .csv-file, which can be evaluated seperately.

Note Sometimes the cross-validation script runs out of memory because of pytorch internal operations. Should this happen, a workaround is to run the run_training.py script, used by the hyperparameter search, 10 times where in the config the folds are shifted manually for a 10-fold cross-validation.

4.4 Evaluation Bloom Filter

We also used Bloom filter as a comparison to our chosen neural networks. To be able to do that two steps are necessary. First, the graph summary needs to be calculated with Bloom filter support on the whole dataset, then the Bloom filter needs to be evaluated. For the generation of the Bloom filter data, it is important to define the *Param[in] max_items*, *Param[in] error_rate* and *Param[in] bloomfilter*. An explanation of the parameters can be found here `https://hur.st/bloomfilter/` with a visualization of the correlation of the parameters.

```
python run_preprocessing.py --config config_preprocess_bloomfilter.ini
```

For the evaluation of the Bloom filter, a few important things need to be considered and defined in the configuration files. *Param[in] eval* should be set to either "impurity" or "acc". The Gini-index based impurity and the accuracy are always calculated either way.

If for the extraction of the subgraphs a *Param[in] save_fold_percentage* < 1 is used, then for the evaluation of the Bloom filter this parameter must be also set to the same value. One of its use-cases was the creation of the DyLDO-25% dataset for our 1-hop graph summaries ablation study.

```
python eval_bloomfilter.py --config config_eval_bloomfilter.ini
```

For SchemEX, we needed the ability to reduce the dataset even more, so *Param[in] min_support_classes* and *Param[in] mini_batch* are also introduced for the Bloom filter evaluation.

```
python eval_bloomfilter_reduced_schemex.py --config
    config_eval_bloomfilter_reduced_schemex.ini
```

4.5 Plots

There are two scripts to generate the class distribution plots. Both scripts create the same plots, but use different input data: the first script uses the pickled graph summary dictionary file as input, whereas the second script loads the different description files from the fold folders. For the plot_class_distribution_graph_information.py the *Param[in] save_file* defines the graph summary file to plot from.

```
python plot_class_distribution_graph_information.py --config
    plot_class_distribution_graph_information.ini
```

For the plot_class_distribution_subgraph.py the *Param[in] folds*, *Param[in] description_file*, and the *Param[in] run_dir* is used to generate the plots from the given data inside the folds.

```
python plot_class_distribution_subgraph.py --config
    plot_class_distribution_subgraph.ini
```

4.6 Stats

Because we needed to change the dataset to be able to work with SchemEX, the dataset statistics (number of vertices, number of edges, unique edges and amount of rdf-type occurrences) will change for the reduced dataset. Therefore the following script extracts all necessary statistics.

```
python stats_schemex.py --config config_stats_schemex.ini
```

4.7 Details and Tips

TensorBoard summaries are saved during the training process, which can be viewed by using the TensorBoard tool `https://www.tensorflow.org/tensorboard/`.

The *Param[in] hidden_layer* can be used to define the **width** of a hidden layer used in the neural network. If that parameter is set to 0, the neural network is considered as a 1-hop model without a hidden layer. For values above 0 it is a 2-hop model, where the defined number is the width of the hidden layer.

A *screen*[3] session can be used to prevent a process from being terminated when the connection to the server is closed or lost. Since this session has no scrollback, looking at the past output is not possible. To solve this problem, *tee*[4] can be used to duplicate the application output onto standard out and into a log file.

```
python -u foo.py --config config_foo.ini 2>&1 |tee foo.log
```

If GPU memory problems occur with the ADAM optimizer, the SGD optimizer can be used.

[3] `https://linux.die.net/man/1/screen`
[4] `https://linux.die.net/man/1/tee`

References

[1] Yang Hu. Implementation of Graph-MLP. https://github.com/yanghu819/Graph-MLP, 2021. [Online; accessed 18-February-2022].