Les Cahiers du GERAD ISSN: 0711–2440

Aggregator: Automating feature aggregation with graph neural networks

F.-X. Devailly,
X. Tan, G. Caporossi

G–2020–45

August 2020

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2020
– Bibliothèque et Archives Canada, 2020

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2020
– Library and Archives Canada, 2020

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Aggregator: Automating feature aggregation with graph neural networks

Francois-Xavier Devailly
Xinyue Tan
Gilles Caporossi

GERAD & Department of Decision Sciences,HEC Montréal, Montréal (Québec), Canada, H3T 2A7

francois-xavier.devailly@hec.ca
xinyue.tan@hec.ca
gilles.caporossi@hec.ca

August 2020
Les Cahiers du GERAD
G–2020–45

Abstract: `Aggregator` is an open-source python package which aims to facilitate the exploitation of relational datasets by automating feature aggregation.

Keywords: Aggregation learning, Graph Convolutional Networks, deep learning

Résumé : `Aggregator` est une extension en source libre visant à faciliter l'exploitation des jeux de données relationels en automatisant la création d'aggrégations.

Mots clés : Aggrégation apprise, réseaux de neurones à convolutions sur des graphs, apprentissage profond

Acknowledgments, feedbacks ans disclaimer: The development of `Aggregator` started in 2019.5 and was funded by The Desjardins Group and Mitacs. Any kind of feedback or criticism would be greatly appreciated (design, documentation, improvement ideas, spelling mistakes,bugs, etc.). Please feel free to contribute and send pull requests. This library is provided 'as is', without any guarantee made as to its suitability or fitness for any particular use. It may contain bugs, so use this tool at your own risk. Authors take no responsibility for any damage that may unintentionally be caused through its use.

1 Introduction

Given a machine learning task on a given relational dataset, the `Aggregator` package represents observations as networks and leverages the flexibility and inductive capabilities of Graph Convolutional Networks (GCNs) [4] to automate the learning of expressive aggregation functions.

The learnt aggregation functions are expressive non-linear transformations.

The automated learning of expressive aggregations functions has the following advantages:

- It may help save considerable time for a Data Scientist, as trying to identify relevant aggregations, especially when dealing with hierarchical aggregations, is a notoriously challenging and time consuming part of feature engineering.
- It may increase performance by enabling full granularity exploitation, as handcrafted aggregations are typically not sophisticated enough to isolate the signal from a subset of elements or even a single element which can be lost in a sum, average, minimum, maximum, and many more operations.
- It may significantly reduce the need for domain knowledge when addressing a given problem, as knowing which handcrafted aggregations loose the least signal given a task and a dataset remains critical in many cases to this day.

2 Background

Enabling machine learning algorithms to study complex relationships between objects in graphs has been a challenge. Graph representation learning, which aims to learn low-dimensional representations preserving both structural information of a network and the features of its nodes and edges, has recently received a lot of attention. Graph Convolutional Networks (GCNs) are one of the most recent and fruitful attempts to leverage neural networks and backpropagation to address the complexity of exploiting graph data.

2.1 Graph Convolutional Networks

In order to learn richer information in propagation of graphs, [5] proposed a stacked k convolutional layers as a neighborhood information aggregation framework. The framework exploits the graph structure and the node features by accumulating inputs from its k^{th} order neighborhood. At each layer of a GCN, model learns a representation for every node by aggregating communications sent to it by both its neighbors and itself following the equation:

$$n_i^l = f\left(\sum_{j \epsilon N(i) \cup i} C_{i,j} \cdot (W_l \cdot n_j^{l-1})\right) \tag{1}$$

where n_i^l is the node embedding of node i at layer l, $N(i)$ is the 1st order neighborhood of node i, $C_{i,j}$ is a normalization constant, W_l is the l^{th} layer's weight matrix and f is a non-linear differentiable function. In the version proposed by [5], weights are shared for both self and neighbor communications, and pairwise normalization constants are used. The model is trained end-to-end by backpropagation using a loss function based on embeddings and supervised, semi-supervised or unsupervised targets.

2.1.1 Relational Graph Convolutional Networks

Tasks on more general (heterogeneous) graphic structures can be addressed using different parameters (W) and normalization constraints (C) for different types of edges [8]. Adding self-loops as a relation type, the R-GCN propagation rule becomes:

$$n_i^{(l)} = f\left(\sum_{r \epsilon R, j \epsilon N_r(i)} C_{i,j,r} \cdot (W_{l_r} \cdot n_j^{(l-1)})\right) \tag{2}$$

where R is the set of all relation types, $N_r(i)$ is the 1st order neighborhood of node i in the graph of relation type r, $C_{i,j,r}$ is a relation-specific normalization constant, and W_{l_r} is the l^{th} layer's weight matrix for message propagation corresponding to a relation of type r.

2.1.2 Expressiveness through attention

Attention mechanisms [1] were popularized in natural language processing (NLP). They are now part of most of the state of the art deep learning architectures used in this field [9, 7, 11, 2]. [10] introduced the use of attention mechanisms on GCNs. For each layer, the nature of the information broadcast (features of the sending node) and the context in which it is received (features of the receiving node) are used to compute the importance/weighting assigned to a message by its receiving node with respect to the rest of this receiving node's incoming messages. Equation (1) can be slightly modified to obtain the new propagation rule. In fact, we simply need to replace $C_{i,j}$ with the following attention weight:

$$\alpha_{i,j} = \frac{exp\Big(f(a^l \cdot [W_l \cdot n_i^{(l-1)} || W_l \cdot n_j^{(l-1)}])\Big)}{\sum_{k \epsilon N(i)} exp\Big(f(a^l \cdot [W_l \cdot n_i^{(l-1)} || W_l \cdot n_k^{(l-1)}])\Big)} \tag{3}$$

where a^l is the l^th layer's attention weight vector and $||$ represents concatenation. For every layer, multiple attention heads can be used concomitantly, and their results can be concatenated or aggregated, to enable more expressiveness.

3 Model

3.1 Relational datasets as graphs

Before introducing the model, some terms must be defined:

main table: In relational datasets, the main table represents the final level at which we would like the aggregation to occur. If each table is represented by a node in a tree structure, the main table is denoted as the root node.

ID columns: ID columns refer to the keys used to identify observations and link tables in a relational dataset.

parent id: The key used to identify observations (i.e. rows in the main table) is called the parent ID.

`Aggregator` treats all observations in a dataset as independent graphs. Every graph is defined by:

- its nodes, their types and their respective initial representations (i.e. embeddings).
- its edges and their types.

Node and node types: Each observation (corresponding to a row from the main table) is represented as an individual graph, in which every node corresponds to a row, in a table, which is related to the given observation. Each table is assigned a unique type, which dictates its corresponding node type in each graph. A node's initial representation (i.e. embedding) is a vector composed of its corresponding row's features.

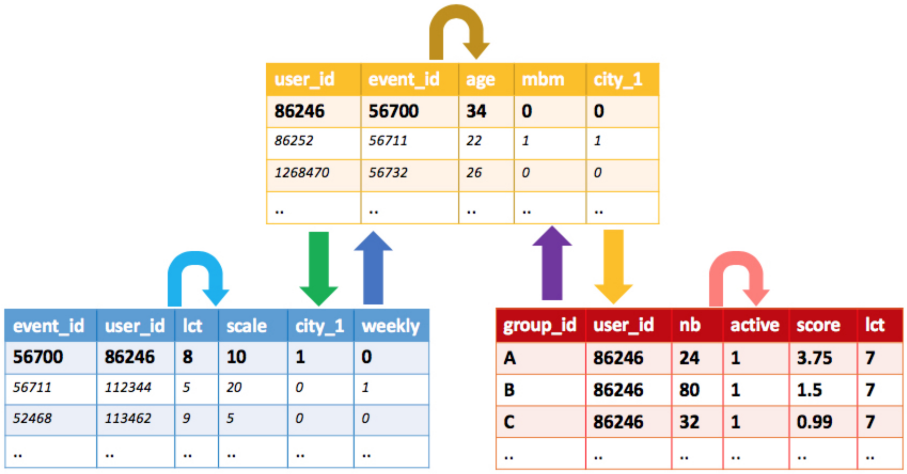
Edges and relation types: An edge is defined by its source node and destination node. The type of an edge is based on the types of both its source and destination nodes. Edges which are starting from a node of the same type and ending to a node of the same type are of the same type. Edges follow the structure of the relational dataset. For every layer of the GCN, every type of edge corresponds to a given set of parameters that is to be used for message propagation.

Bidirectional links and self connections: The user can choose to include bidirectional links during the creation of the graphs. In this case, new edges are simply added inverting source and destination nodes of existing edges. One can also choose to include self-connections. In this case edges are added to link all nodes to themselves.

3.1.1 Example: Converting a relational dataset to a set of graphs

In this section, a social network is used as an example to illustrate the graphs creation process. Let us consider three tables: `user`, `group` and `event`. The `user` table contains some information about the users. The `event` table contains events users took part in. Finally the `group` table indicates which group users have joined. The task is to predict whether a user will take part in a new event.

In this example, bidirectional edges and self-connections are included.



user_id	event_id	age	mbm	city_1
86246	56700	34	0	0
86252	56711	22	1	1
1268470	56732	26	0	0
..	..	..	..	..

event_id	user_id	lct	scale	city_1	weekly
56700	86246	8	10	1	0
56711	112344	5	20	0	1
52468	113462	9	5	0	0
..	..	..	..	..	..

group_id	user_id	nb	active	score	lct
A	86246	24	1	3.75	7
B	86246	80	1	1.5	7
C	86246	32	1	0.99	7
..	..	..	..	..	..

Figure 1: Step 1: Gathering user 8426's features. Different colors indicate different tables (node types) and different edge types. ID columns link all rows (feature vectors) related to a given observation across the dataset in order to gather them.

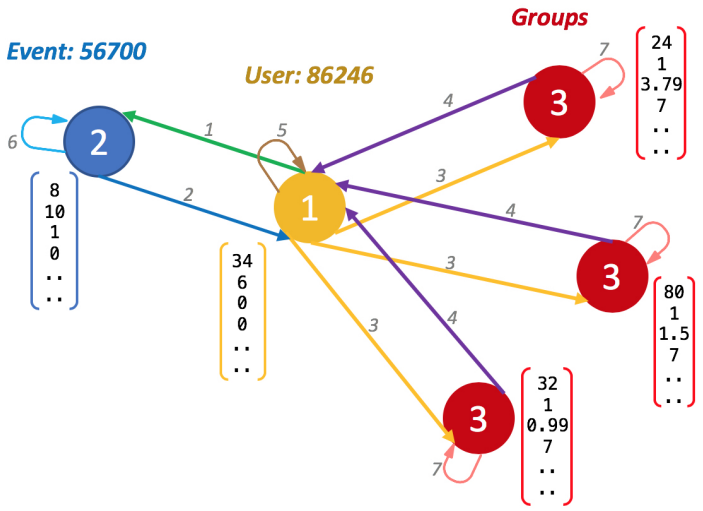


Figure 2: Step 2: Creating user 8426's graph. Different colors indicate different tables (node types) and different edge types. Nodes and edges are marked with their types. In this example, the 3 node types are: `user`: type 1; `event`: type 2; `groups`: type 3. The 7 edge types are: `user to event`: type 1, `event to user`: type 2, `user to groups`: type 3, `group to user`: type 4, `user self linkage`: type 5, `event self linkage`: type 6; `groups self linkage`: type 7. Each vector gathered in the previous step becomes the initial embedding of its corresponding node in the graph. To enable full parallelization, embeddings are padded with 0s to ensure they all have the same dimensionality.

The tree structure implied by the relational dataset is followed to create all graphs. Starting from the main table (`user` table), for each observation, all the related rows in all the linked tables (`event` and `groups`) are identified and written to the graph as described in Figure 1 and 2. Had there been additional tables related to `event` or `groups`, the previous process would be repeated iteratively to complete the graph.

`Aggregator` automatically saves all the graphs as pickle objects. Each file contains multiple graphs and is either saved in a train or test folder.

3.2 Graph sampling

As graphs scale up to hundreds of thousands (or more) of nodes and/or edges, dealing with full graphs might no longer be efficient or even feasible. `Aggregator` enables graph sampling to replace full graphs by sampled subgraphs for all included applications (e.g. training, scoring, embedding).

The graph sampling method used by `Aggregator` is based on neighbors sampling. Starting from the source node, a maximum number of 'children' nodes (neighbors belonging to the next layer and linked to the selected node) can be sampled. This process is repeated for every sampled node until sampling has been performed for all layers (i.e. leaf nodes are reached).

The expand_factor hyperparameter defines the maximum number of nodes which are sampled at every layer (see § 4.6.1).

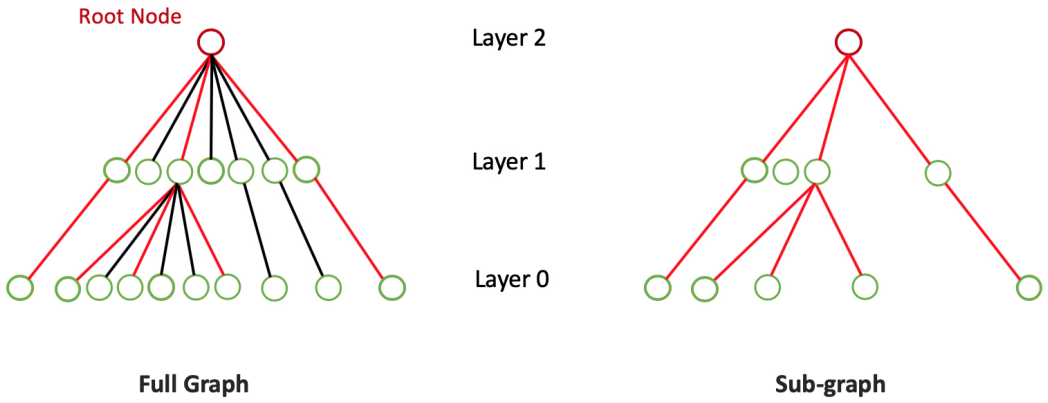


Figure 3: Graph Sampling. This figure illustrates the neighbor sampling process, when the expand_factor is 3 for all layers.

The sampling rules can also vary among layers. In `Aggregator` package, we use a list to define the rule where each element in the list represents the maximum number of sampled nodes in each layer.

3.3 Architecture

Our framework consists in two blocks: a GCN message passing block and a MLP (Multi-layer Perceptron) prediction block.

3.3.1 Leveraging the aggregation block

Our model can be used in two ways after being trained. One could directly use the entire architecture for prediction. One could also decide to discard the MLP and use only the GCN part to get aggregated representations of all observations. The extracted node embedding for the source nodes allow to replace the relational dataset by a single table which summarizes the original dataset with respect to the prediction task.

3.3.2 Parameter and architecture tuning

The `Aggregator` package enables flexibility in the creation of both the GCN block and the MLP block(s). This section covers some of the critical options and hyperparameters the user can experiment with when trying to identify a desired formulation for a given task.

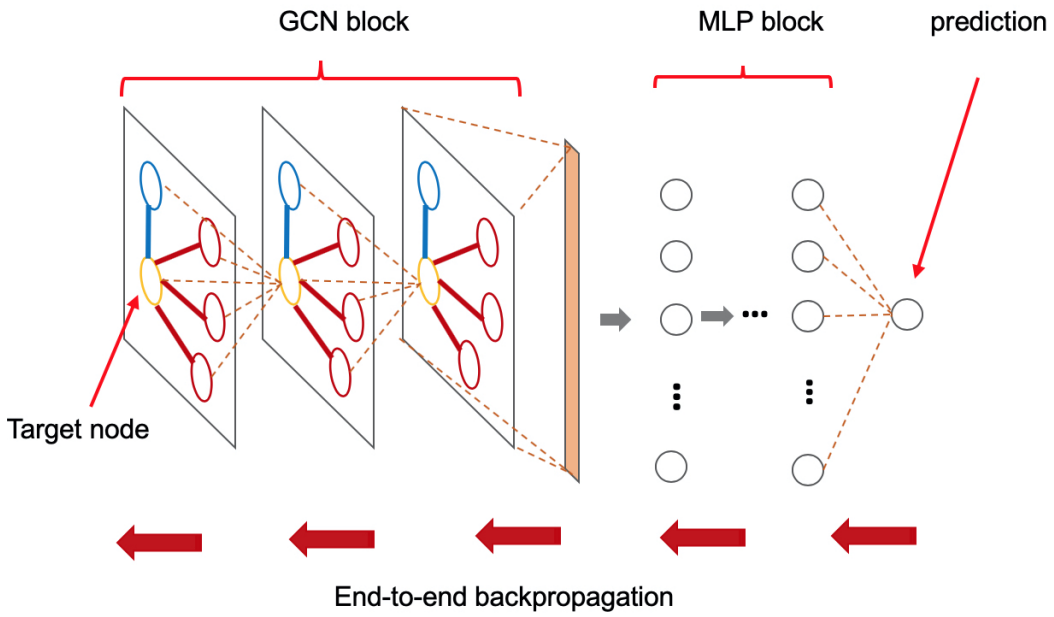


Figure 4: Model architecture. The GCN and the MLP are trained jointly (in a supervised fashion) via end to end back-propagation of the prediction error.

Attention type : Attentions mechanisms enable nodes to assign different importances/weightings to different incoming messages. Two types of attention are available: the *default/normal* attention which is explained in § 2.1.2 and the *scaled dot product* attention. The *scaled dot product* attention is proposed by Noam in [9][1].

Number of attention heads : Using a single attention mechanism leads to the use of a single weighting of messages. Alternatively, multiple attention mechanisms and their corresponding weightings can be used concomitantly. In this case, Messages are weighted and combined independently following each attention mechanism and all results are then concatenated. A final mapping links the concatenated vectors, which can be high dimensional, back into typically 'smaller' vectors which are to be used to update node embeddings.

GRU (Gated Recurrent Units) (optional): In order for a "vanilla" GCN to be able to aggregate information from nodes at a distance d, it requires at least d layers having different parameters. When d is large, this increase in the number of layers is likely to both require a long time to train, and result in vanishing or exploding gradients during backpropagation. One might decide to use the same parameters/layers for multiple, if not all propagations, using a recurrent neural network (RNN) architecture, but the vanishing/exploding gradients problem remains. Fortunately, Long-short term memory networks (LSTMs) and gated recurrent units networks (GRUs) both prevent gradients from degenerating through the use of gating mechanisms. [6, 3] showed that using GRUs as vertex update functions of the message-passing framework could improve the learning of long-distance dependencies. A gated graph convolutional networks can be used for an arbitrary number of layers/propagations and can even be used to deal with dynamic graphs and network time series (updating node embeddings with an arbitrary number of propagations at every time-step).

Residual (connections) (optional): When GRUs are used as vertex update function, a residual connection can be added as a shortcut to the node update. In other words, an "identity connection" ensures that the GRU component fits only a residual mapping(i.e. the difference between the new embedding and the previous one) rather than the desired under-laying mapping(i.e. the new embedding) directly.

[1]Please refer to the corresponding article for a detailed explanation

Attention standardization (optional): A softmax operation helps ensure that messages remain within a reasonable range. If this operation is included, the attention weightings of all messages leading to the same destination node always sum up to 1.

Message normalization (optional): With or without applying attention mechanism, if message normalization is used, a message is divided by the number of messages of the same type leading to the same node (in-degree corresponding to a specific edge type).

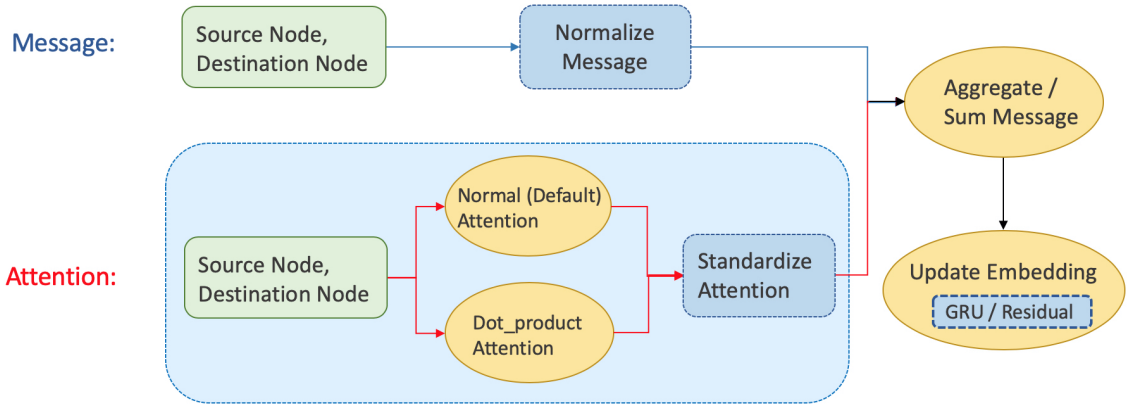


Figure 5: Flow chart of message-passing and vertex updating with corresponding options. This chart illustrates the process of computing messages and updating node embeddings in each message propagation of the GCN architecture. The steps in the doted square boxes are optional. Green boxes indicate that the user needs to select at least one of the included elements. Yellow ovals include essential parameters that cannot be omitted.The message on an edge are based on source and/or destination node(s)' embeddings, or both. Messages can then be normalized. The bottom part of the chart illustrates the attention mechanism. It is based on source and/or destination node(s)' embeddings. The user needs to select a type of attention: `normal`(default) or `dot-product` and a number of heads. Attention weightings can then be standardized. Messages are finally aggregated (optionally using attention weightings). Finally the destination node embedding is updated using the aggregated message. A GRU, which can include a residual operation, can be used as a vertex update function.

Detailing submodules architectures (optionnal): Both message and attention computations involve differentiable parameterized transformations. Each transformation can be seen as fully connected neural network. By default each of these neural networks consist in a unique transformation (i.e. a single layer). However, the user can decide to add hidden layers to these architectures using lists of layers' sizes when defining the model (see § 4.5 for more details). For instance, a user might choose to use a MLP including 3 hidden layers of 64 neurons to compute messages (i.e. message_hidden_layers = [64,64,64]) and a a MLP including 2 hidden layers of 32 neurons to compute 'default' attention (i.e. message_hidden_layers = [32,32]).

Defining final MLP(s) architecture(s) `Aggregator` can be trained using multiple targets. While the aggregation framework (GCN) is shared between all targets, every target will require its own final MLP. Every MLP can have its own architecture which should be represented as a list including:

- 1st element: The name of the corresponding output/prediction (string)
- 2nd element: The architecture of the corresponding fully connected neural network (as a list of hidden layers' sizes)
- 3rd element: The dimensionality of the *output* for regression; the *number of classes* for classification. Please note that the current version of `Aggregator` only supports binary classification.

`Aggregator` In the current example, we use a single MLP as the prediction is simply a score, which is a scalar, and we do not use any hidden layer. The corresponding framework can be defined as follows:

```
prediction_modules = []
prediction_modules.append(['repeater', [], 2])
```

The MLP of 3 hidden layers with 4, 3 neurons in each layer can be defined and visualized as:

```
prediction_modules = []
prediction_modules.append(['target', [4, 3], 1])
```

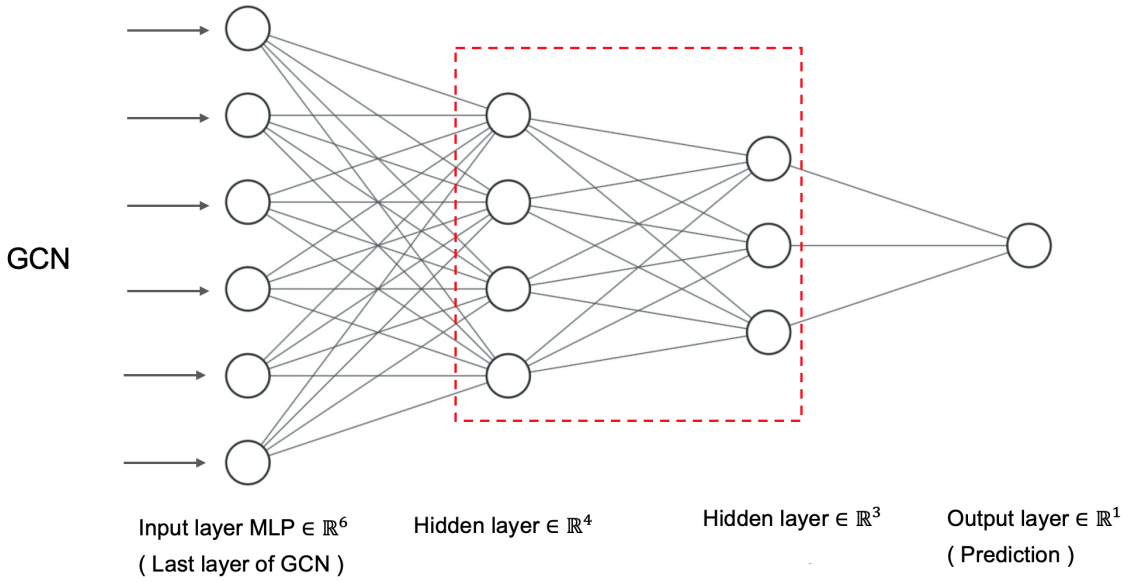


Figure 6: Framework of the prediction modules (MLP). The output of the GCN model (i.e. the final embedding of the source node) is fed to the prediction modules. The defined list of [4, 3] is the dimension size of each hidden layers of the MLP (in red dotted box).

4 Tutorial

The python package aims to ease the use of the `Aggregator` framework by abstracting away:

- The creation of the graphs from a given relational dataset
- The creation of the `Aggregator` architecture
- The training of an `Aggregator` instance based on a graphs dataset
- The optional writing of learnt aggregations, for all observations in the dataset, to a new file.

4.1 Get ready: Prerequisites

`Aggregator` is built with python version 3.6.8. Other versions have not been tested. Several libraries are required to use `Aggregator`. Versions of these packages which were used to build `Aggregator` are listed below. Please refer to the official documentations to install these libraries based on your own requirements:

- PyTorch (version 1.4.0)
- Deep Graph Library (DGL) (version 0.2)
- NetworkX (usually included with DGL) (version 2.3)

4.2 Get started

4.2.1 Example: Problem description

In this tutorial, we use a public transactional dataset as an example to illustrate the options provided by `Aggregator`. The task consists in predicting a loyalty score for a set of customers. The data comes from Kaggle, where a description is available:

https://www.kaggle.com/c/acquire-valued-shoppers-challenge/data

4.2.2 Find linkage and link ID between tables

Before dealing with the raw data, the user first needs to understand how tables belonging to a dataset of interest are linked together. In this example, there are 3 tables:

`train.csv, transactions.csv, offers.csv`.

The links between them are:

`train.csv` and `transactions.csv` are linked by `id`; `train.csv` and `offers.csv` are linked by `offer`.

The structure of the corresponding dataset is as follows:

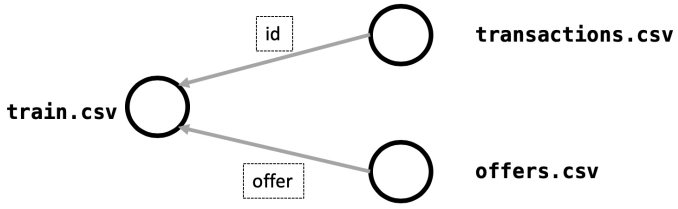


Figure 7: linkage between pairs of the tables. The logical structure followed to transform all observations into a set of graphs is a tree.

4.2.3 Pre-processing

`Aggregator` does not automate pre-processing of the raw dataset. The user should process the data from every table so as to be fed into a typical neural network architecture.

Pre-processing steps might include:

1. Dealing with missing values and outliers
2. Converting categorical variables into one-hot encodings
3. Normalizing/Standardizing data
4. Performing feature selection at the table level
5. Ensuring that all covariates are numerical except for the ID columns which define linkages between tables.

In our particular example, missing values are imputed, extreme values are replaced, categorical variables are represented as dummy variables, date features are transformed into a difference between dates. Except for the ID columns, all values should be converted to valid numerical value.

The fact that two tables are related does not mean every observation in one table has corresponding rows in the related table. This would result in, for instance, missing values in ID columns. In this case, the user needs to fill these missing values with a special indicator (such as '0'or 'N/A') to enable the package not to look for a non-existent relation.

4.3 Instantiating the `Aggregator`

```
from Aggregator.Aggregator import Aggregator
from Aggregator.functions import *
import os
import torch
```

After importing the `Aggregator` class, we can instantiate it. The instance holds the relational information between tables of a given dataset in the form of a tree structure which guides the creation of graphs. The table on which the prediction task is to be performed is the 'main' table. Every row belonging to this 'main' table becomes the root node of its own graph and observation.

4.3.1 Init method

Class: `Aggregator` (main_table, links, parent_id, target_variable = None, no_link_symbol = 'N/A')

Parameters

- **main_table**(str):
  The name of the table that has been identified as the 'main' table.
- **links**(list of tuples):
  Each tuple represents an existing relationship between two tables and takes the form: (`parent_table, child_table, linkage_key`) where parent and child are the names of the tables involved in the relationship. The parent table is always the one which is the closer to the main table in the relational tree. The ID is the name of the key which is used to link both tables.
- **parent_id**(str):
  The index or ID columns which indicate the observations in main 'table'.
- **target_variables**(str): default: None
  The name of the target variable in the main 'table'.
- **no_link_symbol**(int/str): default 'N/A'
  Indicator to be used for missing ID values (see § 4.2.3)
- **bidirectional**(bool): default False
  Bidirectional graphs allow messages to be sent bidirectionally between any pairs of related nodes at the cost of computational complexity. In non-bidirectional graph, messages are simply sent from child node to parent node.
- **self_loop**(bool): default False
  If `True`, the graph allows messages to be sent from each node to itself at the cost of computational complexity.

Returns

- **Aggregator object**
  `Aggregator` is the main class of the package. Its initialization method automatically defines a tree structure, node types, relation types, initial dimensions, etc., which will guide the transformation of all observations in the relational dataset into a set of graphs.

Code to create instance of the `Aggregator` class:

```
# Provide the links between paris of the tables
population = [('train.csv','offers.csv','offer'),
              ('train.csv','transactions.csv','id')]

# Create an instance of aggregator
aggregator = Aggregator(parent = 'train.csv',
                        links = population,
                        parent_id = 'id',
                        target_variables = ['repeater'],
                        no_link_symbol = 0,
                        bidirectional = True, # add bidirectional
                        self_loop = True) # add self connection
```

Note: At this point, no complete table has been read yet. Initializing the class is only meant to create the tree structure and create required nodes and edge types.

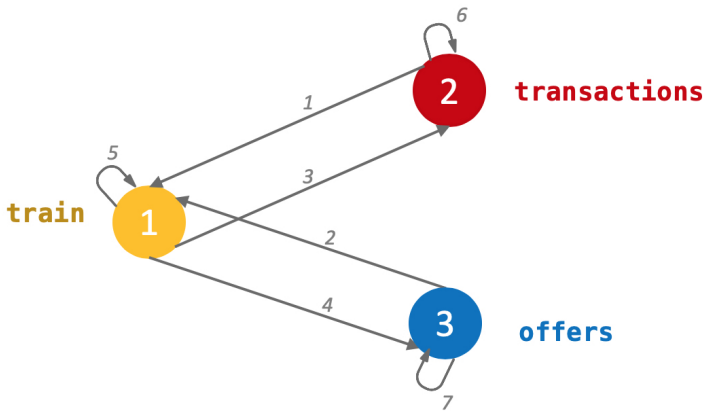


Figure 8: Graph defined based on tree.

Some useful attributes of `aggregator` instance can be printed to verify that the tree structure is correct:

```
print('number of type of the nodes:', aggregator.n_nodes)
print('number of type of the relations:', aggregator.n_rels)
print('initial embedding dimension:', aggregator.max_dim)
```

```
number of type of the nodes: 3
number of type of the relations: 7
initial embedding dimension: 41
```

```
# check assgined node types
aggregator.nodes_types
```

```
OrderedDict([('train.csv', 0), ('offers.csv', 1), ('transactions.csv', 2)])
```

The graph is well defined in terms of nodes, edges, dimensions.

4.4 Creating the graphs

4.4.1 create_graphs method

This method of the `Aggregator` class enables the creation of the graphs from the relational dataset and can be accelerated using parallelization.

During this step:

- Feature vectors related to all observations will be found and used to create graphs as illustrated in Figure 1 and 2.
- Every observation will be saved as a graph according to attributes of the `Aggregator` class.

Parameters

- **folder**: default = os.getcwd()
  This defines a path to set up folders in which graphs are to be saved. It will automatically create a folder named "graph" and two sub-folders "train" and "test" under it, if they do not already exist. When graphs are created, they are be automatically stored in 'train' and 'test' according to the argument `train percentage`.
- **n_workers** (int): default = 1
  Number of processes running in parallel to create and save graphs.
- **nb_graphs_per_file** (int): default = 10
  Number of graphs/observations one wants to save in every train and test file. Large values may lead to extreme memory requirements during both graphs creation and graph sampling during the training of the `Aggregator` model.
- **train_percentage**(float): default = 0.8
  Defines the percentage of observations to be included in the train folder. The rest is included in the test folder.
- **predict**(bool): default = False
  Set to True if the model is already trained and observations are to be transformed into graphs for scoring.

Code to create the graphs:

```
# Find linkage antomatically
# Create and save graphs using multiple prcesses
aggregator.create_all_graphs(folder = os.getcwd(),
                             n_workers = 20, # use 20 processes
                             nb_graphs_per_file = 400,
                             train_percentage = 0.8) # train validation split
```

To make predictions on the unseen data with a trained model, the user also needs to create graphs for observations in order to score them. In this case `'predict'` should be set to `'True'`:

```
# Use test dataset to create graphs
population = [('test.csv','offers.csv','offer'),
              ('test.csv','transactions.csv','id')]

aggregator = Aggregator(parent = 'test.csv', # main table is now the test set
                        links = population,
                        parent_id = 'id',
                        # no need to specify a target variable to score observations
                        target_variables = None,
                        no_link_symbol = 0,
                        bidirectional = True,
                        self_loop = True)

aggregator.create_all_graphs(folder = os.getcwd(),
                             n_workers = 20,
                             nb_graphs_per_file = 400,
                             # train_percentage is automatically set to 1 if predict is `True`
                             predict = True) # change to True
```

Notes:

- If the relational dataset involves a large number of observations, it may take a while to build all graphs. Given this example and the machine used to run it, 20 processes require 2 hours to build graphs for all 160,057 observations.
- The `folder()` argument calls an inner `setup_folders()` method, which may format previous records. If the creation of graphs is repeated, one might want to relocate a copy of the previously created instance(s).

It is also possible to verify the graph structure using visualization. The use can randomly choose one graph from a train or test file, then put the graph as the argument of the method `print_graph( )` to visualize it:

```
# find a small graph for visualization
import pickle
f = open('no_bidir/graph/train/1','rb')
graphs = pickle.load(f)
nodes = {}
for k,v in graphs.items():
    try:
        min_nodes
    except:
        min_nodes,g_id = v[0].number_of_nodes(),k
    if v[0].number_of_nodes() < min_nodes:
        min_nodes,g_id = v[0].number_of_nodes(),k
print(g_id,min_nodes)
```

```
475011192 24
```

The selected graph(s) can then be drawn:

```
import pickle
f = open('no_bidir/graph/train/1','rb')
graphs = pickle.load(f)
g = graphs[475011192][0] # select a graph
Aggregator.print_graph(aggregator,g)
```

```
OrderedDict([('train.csv', 0), ('offers.csv', 1), ('tansactions.csv', 2)])
```

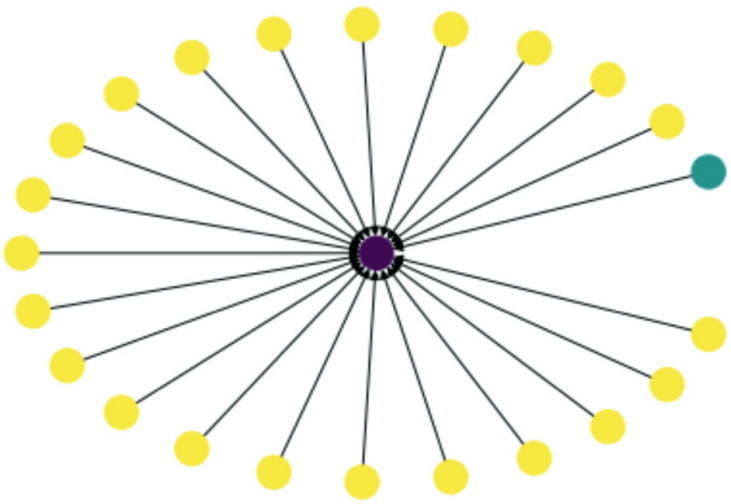


Figure 9: Visualizing a graph. The purple node in the center is the train node (source node). It is surrounded by and linked to yellow 'transactions' nodes and green 'offers' nodes.

4.5 Creating the model

The user is able to experiment with many options and hyperparameters in order to find a formulation of the aggregation framework which is relevant for a given task.

4.5.1 create_model method

The user can use this method to define the architecture of the entire model framework (GCN + MLP). See § 3.3.2 for more details about some of these options.

Parameters

- **prediction_modules**(list): default = [ ]
  Predictions frameworks defined in § 3.3.2.
- **use_attention**(bool): default = True
  Whether to include attention mechanism(s) in the GCN or not.
- **attention_type**(str, optional): default = 'normal'
  Two types of attention mechanism are provided: `'normal'` and `'scaled dot_product'`. The user can also specify if source nodes embeddings `'src'`, destination nodes embeddings`'dst'` or both `'src_dst'` are used when computing messages and attention. For instance, if the user wishes to choose scaled dot-product attention and only use source nodes embeddings to compute messages and attention, the attention type could be: `attention_type = "dot_product_src".` or `attention_type = "src_dot_product".`
- **use_gating**(bool): default = False
  If *True*, GRU(s) are used as a vertex update functions.
- **std_attention**(bool): default = True
  If *True*, weights outputted by each attention mechanisms are standardized (i.e. a softmax operator is applied). If False, each message is simply multiplied by its unstandardized attention weights.
- **node_embedding_size**(int): default = 32
  The size of the embeddings used for all nodes in the network (including the root node), at every layer. This size is the same for all nodes to enable full parallelization.
- **attention_heads**(int, optional): default = 5
  Number of concomitant attention mechanisms to be used. More attention mechanisms might lead to more expressiveness at an increased computational cost.
- **norm**(bool): default = True
  If *True*, messages are divided by the in-degrees (only including edges of the same type) of their respective destination nodes.
- **message_hidden_layers**(list): default = [ ]
  Architecture of the neural network computing the message from source node's embedding.
- **request_hidden_layers**(list): default = [ ]
  Architecture of the neural network used on the receiver/destination node's embedding to give context to the attention mechanism.
- **query_hidden_layers**(list, optional): default = [ ]
  Size of the query and key vectors. This is only used when the chosen attention type is 'scaled dot-product'.

Parameters

- **key_hidden_layers**(list, optional): default = [ ]
  Architecture of the neural network used to create the key vector of the attention mechanism. This is only used when the chosen attention type is 'scaled dot-product'.
- **q_size**(int, optional): default = 12
  The size of the query vector. This is only used when the chosen attention type is 'scaled dot-product'.
- **bidir**(bool): default = False
  If *True* and bidirectional edges were included during the creation of the graphs, each linked pair of nodes will send messages to each other. If *False* message will only be sent from parent nodes to the child nodes.
- **self_loop**(bool): default = False
  If *True* and self-connection edges were included during the creation of the graphs, nodes will send messages to themselves during message propagation.
- **full_graph_computation**(bool): default = False
  If *False*, messages are propagated layer by layer from leaf nodes until root nodes are reached. If *True*, for each layer, messages are sent along all edges and all nodes are updated.
- **residual**(bool, optional): default = False
  If *True* and if a GRU is used to update node embeddings, a residual connection is added. In other words, an "identity shortcut connection" ensures that the GRU component fits a residual mapping(i.e. the difference between the new embedding and the previous one) rather than the desired under-laying mapping(i.e. the new embedding) directly.
- **created_bidir**(bool): default = True
  Indicates whether bidirectional edges were added to the graphs during their creation.
- **created_self_loop**(bool): default = False
  Indicates whether self-connection edges were added to the graphs during their creation.

Note:

The number of convolutional layers to include in the GCN is automatically defined based on the number of levels of the tree structure (defined based on the relational dataset as illustrated in Figure 8. It determines the number of message-passing rounds. That number is equals to the maximum relational distance separating the main table from any other table. This ensures that all relevant features can reach the root node.

Graph creation and Model creation both involve parameters regarding self-connections and bidirectional connections. This is done so that the user can decide not to use bidirectional and/or self connections, even if they were included in the graphs during their creation.

```
aggregator.create_model(prediction_modules = prediction_modules,
                        use_attention = True,
                        attention_type = 'dot_product_src_dst',
                        std_attention = True,
                        node_embedding_size = 32,
                        attention_heads = 3,
                        norm = True,
                        message_hidden_layers = [],
                        request_hidden_layers = [],
                        attention_hidden_layers = [],
                        query_hidden_layers = [],
                        key_hidden_layers = [],
                        q_size = 32)
```

Figure 10: Create your Aggregator model.

4.6 Training the model

4.6.1 train_model method

This method is used to initiate the training of a model (whichever model the attribute 'model' of the `aggregator` instance refers to). Training and evaluation metrics are automatically saved to `result_folder` and visualized with `Tensorboard`.

Parameters

- **predicted_variables**(list):
  List of list. The aggregation framework can be trained using multiple tasks simultaneously. Every list refers to a target variable and should have the following structure: [target_variable,prediction_type, dimension]. Prediction_type should either be 'classification' (binary) or 'regression'. Multiple classification is not supported by the current version.
- **epochs**(int):
  Number of iterations. Here, an iteration consists in the computation of gradients using one batch of observations (i.e. graphs)
- **graphs_folder**(str):
  Folders where the saved graphs are stored There should be two sub-folders: *train* and *test*.
- **results_folder**(str):
  General folder used to save both training/evaluation results and the model itself(architecture and parameters). Two sub-folders *results* and *saved_model* are created.
- **alias**(str):
  Alias used to identify a particular run.
- **lr**(float): default = 1e-3
  Learning rate to be used by Adam optimizer during gradient descent.
- **weighted**(float): default = None
  For classification task only. Percentage of positive examples. If not None, the loss will be computed as weighted loss with an adjustment based on the number of the examples in each class.
- **batch_size**(int): default = 32
  Size of the batches from which gradients are computed.

Parameters

- **expand_factor**(int, list, or None): If *None*, no graph sampling is performed. An integer will determine the number of neighbours to sample at every layer. Alternatively a list of integers can be used to define a number of neighbors to sample per layer. See § 3.2
- **accumulation_steps**(int): default = 1
  Number of batches from which to average gradients before performing gradient descent. For instance, if gradient descent is to be performed on a batch of 32 but computational resources do not enable dealing with a batch larger than 16, the batch size could be set to 16 and the accumulation steps to 2.
- **save_frequency**(int): default = 100
  The frequency at which the model and the parameters are saved.
- **test_frequency**(int): default = 10
  Every test_frequency iterations, the model is evaluated on a 'test' batch composed of graphs coming from the 'test' folder. Gradients are not computed during these steps.
- **nb_batch_test**(int): default = 1
  The number of batches used to gather a representative average test evaluation metric.
- **device**(str): default = 'cuda' if torch.cuda.is_available(), else 'cpu'
  Defines on which device GCN computations are to be performed. The use of GPU(s) typically leads to substantial speedups.
- **tensorboard_port**: default = 6006
  The port to which Tensorboard 'broadcast' the training/evaluation results . The results (Root Mean Squared Error for regression and Are Under the Curve for classification) are logged and broadcast to the corresponding port automatically.

To illustrate hyperparameter tuning, we will experiment with several architectures:

```
# CREATE AND TRAIN FOR `model_1`
prediction_modules = []
# add a hidden layer for prediction module
prediction_modules.append(['repeater', [32], 2])

aggregator.create_model(prediction_modules = prediction_modules,
                        use_attention = True,
                        attention_type = 'normal_src_dst',
                        std_attention = True,
                        node_embedding_size = 64,
                        attention_heads = 5,
                        norm = False)

aggregator.train(predicted_variables = [['repeater', 'classification',1]],
                 n_epochs = 1e9,
                 graphs_folder = 'no_bidir/graph/',
                 results_folder = 'exp',
                 alias = 'model_1',
                 lr = 1e-4,
                 weighted = 0.27, # use the weighted loss
                 batch_size = 32,
                 expand_factor = 1000, # sample the graph
                 accumulation_steps = 1,
                 save_frequency = 100,
                 test_frequency = 10,
                 nb_batch_test = 10,
                 device = 'cuda:0' if torch.cuda.is_available() else 'cpu',
                 tensorboard_port = 6006)
```

Figure 11: Train your Aggregator model_1. In this formulation, we add a hidden layer for the MLP, use "normal" attention and use both source and destination nodes to compute messages and attention. Messages are standardized. The node embedding size is 64, and we use a weighted loss. to deal with class imbalance.

```
# CREATE AND TRAIN FOR `model_2`
prediction_modules = []
prediction_modules.append(['repeater', [32], 2])

aggregator.create_model(prediction_modules = prediction_modules,
                        use_attention = True,
                        attention_type = 'dot_product_src_dst',
                        std_attention = False,
                        node_embedding_size = 32,
                        attention_heads = 5,
                        norm = True,
                        message_hidden_layers = [16],
                        request_hidden_layers = [16],
                        attention_hidden_layers = [16],
                        query_hidden_layers = [16],
                        key_hidden_layers = [16],
                        q_size = 32)

aggregator.train(predicted_variables = [['repeater', 'classification',1]],
                 n_epochs = 1e9,
                 graphs_folder = 'no_bidir/graph/',
                 results_folder = 'exp',
                 alias = 'model_2',
                 lr = 1e-3,
                 weighted = 0.27,
                 batch_size = 16,
                 expand_factor = 2000,
                 accumulation_steps = 2,
                 save_frequency = 100,
                 test_frequency = 10,
                 nb_batch_test = 10,
                 device = 'cuda:1' if torch.cuda.is_available() else 'cpu',
                 tensorboard_port = 6006)
```

Figure 12: Train your Aggregator model_2. The second formulation is applied with 'scaled dot-product' attention, and includes hidden layers for computing the messages and the attention coefficients. The embedding size is set to 32. The learning rate is 1e-3. The batch size is 16 and 2 accumulation steps are performed before every gradient descent.

4.7 Visualizing the result and selecting your model

4.7.1 Results visualization

`Tensorboard` is used to monitor training and evaluation metrics. You can typically access the required port for Tensorboard from your browser: `localhost:6006`.

Replace '6006' by your chosen port if applicable. The user can select multiple results to compare performances and select an appropriate model. See § 4.6.1.

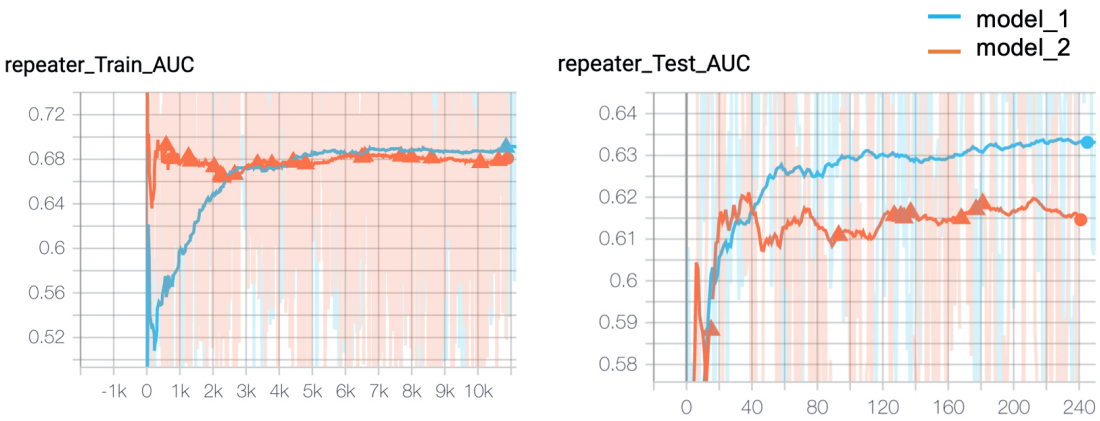


Figure 13: Visualization of the results. The AUC on test suggests that `model_1` provides a more efficient modeling with a better validation AUC. `model_2` has a slightly lower performance than `model_1`.

4.7.2 Using a pre-trained model

This method allows the user to load a pre-trained model in order to make continue training it or to score new observations.

- **load_model method:**

Parameters	• **path**(str): Path of the saved model which is to be loaded.

- **score method:**

Parameters	• **input_folders**(str): Folder path where the graphs of the observations to be scored are saved. • **predicted_variables**(list): List of list. Each list should have the following structure: [target variable, prediction type] • **write_path**(list): path to save the predictions to a new csv file. • **batch_size**(list): Number of predictions computed simultaneously (in parallel). • **expand_factor**(int, list, or None): If *None*, no graph sampling is performed. An integer will determine the number of neighbours to sample at every layer. Alternatively a list of integers can be used to define a number of neighbors to sample per layer. See § 3.2 • **device**(str): default = 'cuda' if torch.cuda.is_available(), else 'cpu'. Device used to compute predictions. Using a GPU typically leads to substantial speedups.

Here, the previously trained model is used to perform scoring. Predictions are automatically exported to the `write_path`. In this example, predictions will be saved as `"model_1.csv", "model_2.csv"`.

```
for model in ['model_1', 'model_2']:
    #load the model
    aggregator.load_model(path = 'exp/saved_model/'+ model +'/model_best.pt')
    # make prediction for for validation graphs
    aggregator.score(input_folders = ['no_bidir/graph/test/'],
                predicted_variables = [['repeater', 'classification']],
                write_path =  model + '.csv', # export result as csv
                batch_size = 16,
                expand_factor = None,
                device = 'cuda:0' if torch.cuda.is_available() else 'cpu')
```

Figure 14: Load your pre-trained model and make predictions.

The results can then be inspected by computing the AUC for the whole validation set using the predictions saved in the last step:

```python
def validation_AUC(file):
    # compute AUC for all validation set
    prediction_file = file
    labels = {}
    scores = {}
    la = []
    sc = []
    # get labels
    for e, line in enumerate(open('train.csv') ):
        if e != 0:
            labels[line.split(",")[0]] = int(line.split(",")[2])
    # get predictions
    for e, line in enumerate(open(prediction_file)):
        if e != 0:
            scores[line.split(",")[0]] = float(line.split(",")[1])
    # put labels and socres into lists
    for k,v in scores.items():
        sc.append(v)
        try:
            la.append(labels[k])
        except:
            print(k)
            break
    # use biuld in function to compute AUC
    auc = func_compute_AUC(la, sc)
    print(prediction_file ,':',auc)
result_files = ['model_1.csv','model_2.csv']
for file in result_files:
    validation_AUC(file)
```

```
model_1.csv : 0.6426082827613606
model_2.csv : 0.6175721663757263
```

Figure 15: Computing test AUC for these two models. AUC for model_1 and model_2 are: 64.26% and 61.76% respectively. model_1 still seems to be the preferable option based on this criterion.

4.8 Extracting learnt aggregations for later use

This section is completely optional. We show how to leverage the GCN block and combine it with traditional supervised learning approaches. The `create_embeddings method` exports, for each observation, the final embedding of its source node obtained using the selected model (whichever model the 'model' attribute refers to). Compared with the framework of the default model in Figure 4, here, the MLP block is discarded and replaced with a traditional machine learning algorithm such as gradient boosting, support vector machine, etc.

4.8.1 create_embeddings method

Parameters

- **input_folders**(list or str):
  List of folders containing the graphs of the observations from which embeddings are to be gathered.
- **write_path**(str):
  Path to the .csv file on which the embeddings are to be written.
- **batch_size**(int):
  Number of embeddings written simultaneously (in parallel).
- **device**(str): default = 'cuda' if torch.cuda.is_available(), else 'cpu'. Device used to compute the embeddings. Using a GPU typically leads to substantial speedups.

```
# Load the selected model
aggregator.load_model(path = 'exp/saved_model/model_1/model_best.pt')
# Extract embeddings for train data
aggregator.create_embeddings(input_folders = ['no_bidir/graph/train/'],
                             write_path = 'embed_train.csv',
                             batch_size = 32,
                             device = 'cuda:0' if torch.cuda.is_available() else 'cpu')
# Extract embeddings for test data
aggregator.create_embeddings(input_folders = ['no_bidir/graph/test/'],
                             write_path = 'embed_test.csv',
                             batch_size = 32,
                             device = 'cuda:0' if torch.cuda.is_available() else 'cpu')
```

Figure 16: Writing aggregations using a trained Aggregator model.

4.8.2 Feeding the aggregated observations to a traditional machine learning algorithm: Gradient boosting

```
import numpy as np
import pandas as pd
import lightgbm as lgb
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV

print('Loading data...')
# load or create your dataset
labels = pd.read_csv('train.csv',usecols=['id', 'repeater'])
df_train = pd.read_csv('embed_train.csv')
df_train = pd.merge(df_train,labels,on='id',how = 'inner')
df_test = pd.read_csv('embed_test.csv')
df_test = pd.merge(df_test,labels,on='id',how = 'inner')

y_train = df_train.repeater.values
y_test = df_test.repeater.values
X_train = df_train.drop(columns=['repeater']).values
X_test = df_test.drop(columns=['repeater']).values

print('Starting training...')
# train
gbm = lgb.LGBMRegressor(num_leaves=31,learning_rate=0.05,n_estimators=20)
gbm.fit(X_train, y_train, eval_set=[(X_test, y_test)],
        eval_metric='l1', early_stopping_rounds=5)

print('Starting predicting...')
# predict
y_pred = gbm.predict(X_test, num_iteration=gbm.best_iteration_)
# eval
#print('The rmse of prediction is:', mean_squared_error(y_test, y_pred) ** 0.5)
print('The AUC of prediction is:',func_compute_AUC(y_test, y_pred))
# feature importances
print('Feature importances:', list(gbm.feature_importances_))
```

```
Loading data...
Starting training...
Training until validation scores don't improve for 5 rounds
Starting predicting...
The AUC of prediction is: 0.642484141472248
```

The AUC on validation obtained on the test set is 64.24%, which is comparable to the performance obtained by the complete architecture (GCN+MLP).

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[3] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pages 1263–1272. JMLR. org, 2017.

[4] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems, pages 1024–1034, 2017.

[5] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

[6] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493, 2015.

[7] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[8] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In European Semantic Web Conference, pages 593–607. Springer, 2018.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

[10] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. arXiv preprint arXiv:1710.10903, 2017.

[11] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In Advances in neural information processing systems, pages 5753–5763, 2019.