1. Implementation

In order to be able to evaluate the available graph embedding methods and the proposed modifications, we have implemented the approaches and several testing scenarios within a program capable of creating, evaluating and comparing embedding methods. In the following chapter, we will focus on presenting the project, guide the future developers through its modules and explain the possible extensions. The second part of the chapter provides a documentation for the potential users of our application, explain the available configurations and the use cases. Finally, the third section of the chapter offers a guide for installing and lauching the project in several supported environments.

1.1 Technical Documentation

To provide a brief, structured overview of the project, we will start by specifying the used libraries and frameworks. We will then continue by describing the modules of our application, the functionality they provide and finally, we will quickly introduce our implementation.

1.1.1 Languages and Libraries

The whole project is implemented in ***Python*** [1] (v3.6.7). However, some of the used libraries like *NumPy* or *TensorFlow* provide a faster implementation (*C*, *C++* or others) within the available libraries.

1. To handle work with array-like objects and numerical computations, we have opted for a widely used python library ***NumPy*** [2] (v 1.16.3).

2. ***SciPy*** [3] (v 1.2.1) provides an implementation of sparse matrices and scientific computations.

3. ***TensorFlow*** [4] (v 1.13.1) manages the construction, training and evaluation of neural networks.

[1] `https://www.python.org/`
[2] `https://www.numpy.org/`
[3] `https://www.scipy.org/`
[4] `https://www.tensorflow.org/`

4. The ***Networkx*** [5] (v 2.3) package handles the majority of graph manipulations from loading the networks, storing edge weights and node labels to creating sparse adjacency matrices.

5. The standard machine learning models required within our project are implemented with the help of ***scikit-learn*** [6] (0.20.3) library.

6. Recall, that the SDNE approach requires Restricted Boltzmann Machines in order to implement the Deep Belief Networks used to pre-train its deep autoencoder. We have selected a Tensorflow implementation of Restricted Boltzmann Machine by Egor Malykh (a.k.a.*meownoid*), available at `www.github.com/meownoid/tensorfow-rbm` (v 0.0.2). The solution is published under The MIT License (MIT) available at `www.github.com/meownoid/tensorfow-rbm/blob/master/LICENSE`.

7. And finally, the ***matplotlib*** [7] and its *pyplot* package (v 3.0.3) provide a Matlab-like plotting framework, which was used to create all charts within this thesis.

1.1.2 Dependencies

Requirements for our project are managed via both ***pip*** [8] *requirements.txt* files and the ***pipenv*** [9] *Pipfile*. The Pipfile is located directly at the project's root, while the requirements.txt files are located in *gpu-docker/* and *cpu-docker/* subdirectories and are meant primarily to be used with docker-based installations. For installing the dependencies and running the solution, please refer to the Installation Guide in the third part of this chapter.

1.1.3 Packages

The full project is considerably complex and is, therefore, divided into multiple python packages. Although not providing a full in-depth technical documentation of the solution, the following section focuses on introducing the individual packages and their functionality.

[5] `https://networkx.github.io/`
[6] `https://scikit-learn.org/stable/`
[7] `https://matplotlib.org/`
[8] `https://pypi.org/project/pip/`
[9] `https://pipenv.readthedocs.io/en/latest/`

Graphs

The package *graphs* contains code for loading and modifying graphs before they are used to create embeddings. The main module of this package is *graph_factory.py*, which connects the supplied configurations with the available parsing methods to load a graph and prepare it for future use within the application. During its run, it uses Graph Parsers and the NodeMapper:

Graph Parsers – to allow loading graphs in multiple input formats, our solution allows creating user-defined graph parsers, defined in module *graph_parsers.py*. Currently, all our graphs are stored as lists of edges, specified by triplets (source node, target node, weight) and are loaded using the *edgelist* parser included within the solution. However, the future users can introduce further parsers by simply implementing them into the python module.

As the arguments for parsers are dataset-specific, they are loaded from a configuration file *ds_config.json*. Its structure is described in Section 1.2.1.

NodeMapper – the second module within the package is *node_mapper.py*, which contains the NodeMapper class. As its name suggests, this module maps the original IDs of the nodes within the graph, which can be either numbers or generally any strings such as usernames and random hashes, to integers from 1 to $|V|$. This functionality is crucial for the performance of our application, as it provides a convenient way of accessing and iterating through the vertices of the graph.

Edge Hiding - besides the graph loading functionality, the *edge_hiding.py* module adds functionality for hiding a given portion of the graph edges, which is necessary for the task of link prediction described in **??**.

Normalization

The *normalization* package provides the functionality for normalizing the edge weights. The package consists of *init_norm.py* module, containing the initialization norms and the *eval_ns_norm.py* module, which provides the negative sampling-based evaluation normalization as described in **??**.

Embeddings

The *embeddings* package contains the implementation of all graph embeddings used within this thesis. The main class of the module is *Embedding-*

Factory, which connects the dataset-specific embedding configurations (described in Section 1.2.1) with the available graph embedding methods.

Like other parts of the solution, this package supports adding new graph embedding methods. We can implement such methods by using the *EmbeddingBase* as our base class and overriding its abstract methods *learn* and *estimate_weights*. Finally, we need to register the new method by creating a static function for the created embedding in class *Embeddings* within the *graph_factory.py* module.

Reconstruction

The next package within our solution provides various means of reconstructing the original graph, as well as predicting the non-observed edges. The functionality of the modules corresponds to the evaluation methods proposed in **??** and is frequently used in evaluation tasks in **??** and **??**.

Evaluations

All evaluation methods used in the experiments throughout this thesis are implemented in the evaluations module. Currently, the module provides modules for evaluating Visualisation (**??**), Graph Reconstruction (**??**) and Link Prediction (**??**).

This package also enables adding new evaluation methods. Like the *embeddings* package, a new evaluation can be implemented by inheriting the *EvaluationBase* class, overriding its abstract methods *run* and *show*, and finally registering the evaluation by creating a static method in the *Evaluations* class within the *evaluation_factory.py* module.

Similarly to the other modules within this thesis, this package uses the factory-based design, where the *EvaluationFactory* handles both dataset-specific configurations, as well as the available methods and provides a convenient way of calling the specific evaluations. We will describe the structure of the configuration file *ev_config.json* in Section 1.2.1.

Run

In order to provide a convenient way of implementing experiments as well as the main functionality of our solution, we have created the run package including the *RunStages* class, which connects all factories and helper classes of this solution into a single, organised facade which covers most of the use cases of our solution. The class provides methods for loading the graph,

creating and evaluating the embedding together with the means of storing the results along the way.

Experiments

Finally, all experiments, evaluations and comparisons presented within this thesis are implemented within the experiments folder. Thanks to the modularity of our solution, most of the experiments contain a short *evaluate.py* module for running the experiment and *make_chart.py* module for creating the plots.

1.2 User Documentation

The second part of this chapter focuses on providing user documentation for our application. At the beginning of the section, we will present the configurations of the solution. The second part of this section aims to introduce a user to the main script, its use cases and the available parameters.

1.2.1 Configurations

As our solution offers a variety of graph embedding methods and evaluations, many of the functionalities require parameters and configurations in order to work properly.

In order to avoid large amounts of the parameters within the main script, we have attempted to isolate the dataset-specific configuration into *json* files, located within the *config* folder of our project.

Dataset Configuration

As our solution can work with multiple datasets, we have included a convenient way of switching between the evaluated datasets. First, all of the evaluated datasets must have an entry in the *ds_config.json* file, located in the *config/* folder. This configuration contains all the necessary knowledge about the dataset, including a path to the source file, information about the graph edges (whether they are directed and weighted) and which parser should be used to load the graph into our solution.

To add a new configuration entry, please find a sample configuration within the cofiguration file itself and modify it according to your needs.

Embedding Configuration

The second type of configuration files within our project is the embedding configuration files, or *em_config.json*. As any datasets may require a specific set up of the embedding parameters, we can create a dataset-specific embedding configuration file by placing it into the *config/[dataset entry name]* folder.

The embedding configuration file contains all parameters of the generated embedding with the exception of the dimension of the final embeddings. Consequently, to add an entry for a new embedding method, we can simply add a new entry with a dictionary of key-value pairs for each of the parameters of the newly created embedding method.

Evaluation Configuration

Like the embedding configuration, each of the available evaluations has its own, possibly dataset-specific configuration file named *ev_config.json*. As in the previous case, a configuration for a new dataset can be added by creating *config/[dataset entry name]* folder with the required evaluation configuration within that directory.

Configurations of the evaluations currently include options for tweaking the generated visualisations, followed by the settings of the maximum amount of predicted edges for the tasks of graph reconstruction and link prediction.

1.2.2 Running the Project

As described in the installation guide above, we can launch the project by running the *main.py* file. Overall, the script can execute the following 7 steps:

1. Load the Graph
2. [*Optional*] Normalise the Graph
3. [*Optional*] Hide Edges
4. Load or Learn Embeddings
5. [*Optional*] Save Embeddings
6. [*Optional*] Evaluate Embeddings
7. [*Optional*] Store the Evaluated Results

We can show the parameters of the script and their descriptions via *-h* or *--help* argument. Notice, that the optional stages of the application run can be excluded by not specifying the corresponding of the parameters – for example, not specifying the *--embed_file* parameter will not save the embeddings.

Although we will not include all parameters of the solution, some of the most important arguments of the script are:

- *--dataset* - specifies the dataset entry within the *ds_config.json* file.
- *--embed_method* - specifies which embedding method will be used.
- *--eval_method* - specifies which evaluation method will be used. Not specifying this parameter will skip the evaluation phase.
- *--embed_dim* - Sets the dimension of the generated embedding.

1.3 Installation Guide

To allow as much flexibility as possible, we have included two options for installing dependencies and running our project. We can either run the solution directly within our operating system and install the dependencies via *Pipenv*, or deploy the solution to the server and launch the project within *Docker*, which is currently one of the most popular tools for sharing and deploying applications.

The following commands should work from ***Linux BASH***, ***GIT BASH*** or ***Windows Subsystem for Linux*** *(untested)*.

1.3.1 Setup

We will set up the project structure and download its dependencies by running the following script from the project root:

```
chmod u+x install.sh
./install.sh
```

Note, that the install script creates a directory *ds/*, which is the default location for the testing datasets. It is, therefore, recommended to copy all of the evaluated datasets into this folder.

1.3.2 Install and Run

Firstly, as mentioned above, the project can run directly on a local PC without virtualisation. Secondly, we have also included the configuration files required for running our solution within Docker containers. Note, that due to the complexity of the CUDA installation, running our applications with GPU acceleration is available only via Docker.

Again, the following commands should work from ***Linux BASH***, ***GIT BASH*** or ***Windows Subsystem for Linux*** *(untested)*.

All scripts should be launched ***from the project root***.

Pipenv [CPU]

1. Install Requirements:

   - ***python*** v3.6 with ***pip***, available at
     `https://www.python.org/downloads/`
   - ***pipenv***, available at
     `https://docs.pipenv.org/en/latest/install/#installing-pipenv`

2. Install depenencies:

   ```
   pipenv install --skip-lock
   ```

   Note, that the *matplotlib* package furthermore requires *tkinter*, which is not always installed with python. The library can be installed using the guide at
   `https://tkdocs.com/tutorial/install.html`

3. Run the project environment:

   ```
   pipenv shell
   ```

4. Launch the project:

   ```
   cd ./src
   python3 ./main.py #[optional: arguments or -h for help]
   ```

Docker [CPU]

1. Install Requirements:

   - ***docker***, available at `https://www.docker.com/`

2. Install depenencies:

```
cd ./cpu-docker
docker build -t gem-docker-cpu .
cd ..
```

3. Run the project environment:

```
docker run --rm -it \
    -v $(pwd):/usr/local/gem \
    -u $(id -u):$(id -g) \
    -w /usr/local/gem \
    gem-docker-cpu bash
```

4. Launch the project:

```
cd /usr/local/gem/src
python3 ./main.py #[optional: arguments or -h for help]
```

Docker [GPU]

1. Install Requirements:

   - ***nvidia-docker***, available at `https://github.com/NVIDIA/nvidia-docker`

2. Install depenencies:

```
cd ./gpu-docker
docker build -t gem-docker-gpu .
cd ..
```

3. Run the project environment:

   (a) One-GPU Server:

```
nvidia-docker run --rm -it \
    -v $(pwd):/usr/local/gem \
    -u $(id -u):$(id -g) \
    -w /usr/local/gem \
    gem-docker-gpu bash
```

(b) Multiple-GPU Server:

```
GPU=0 # or replace 0 with an available GPU
NV_GPU=${GPU} nvidia-docker run --rm -it \
    -v $(pwd):/usr/local/gem \
    -u $(id -u):$(id -g) \
    -w /usr/local/gem \
    gem-docker-gpu bash
```

4. Launch the project:

```
cd /usr/local/gem/src
python3 ./main.py #[optional: arguments or -h for help]
```

1.4 Conclusion

After this chapter, we should be familiar with the project and its modules, we are capable of understanding, modifying and adding further configurations and running the solution with the required parameters. Following this chapter, we can now focus on using our solution to evaluate the presented graph embedding methods in the next chapter.