Developing and Evaluating Graph Counterfactual Explanation with GRETEL

Mario Alfonso Prado-Romero
Gran Sasso Science Institute
L'Aquila, Italy
marioalfonso.prado@gssi.it

Bardh Prenkaj
Sapienza University of Rome
Rome, Italy
prenkaj@di.uniroma1.it

Giovanni Stilo
University of L'Aquila
L'Aquila, Italy
giovanni.stilo@univaq.it

ABSTRACT

The black-box nature and the lack of interpretability detract from constant improvements in Graph Neural Networks (GNNs) performance in social network tasks like friendship prediction and community detection. Graph Counterfactual Explanation (GCE) methods aid in understanding the prediction of GNNs by generating counterfactual examples that promote trustworthiness, debiasing, and privacy in social networks. Alas, the literature on GCE lacks standardised definitions, explainers, datasets, and evaluation metrics. To bridge the gap between the performance and interpretability of GNNs in social networks, we discuss GRETEL, a unified framework for GCE methods development and evaluation. We demonstrate how GRETEL comes with fully extensible built-in components that allow users to define ad-hoc explainer methods, generate synthetic datasets, implement custom evaluation metrics, and integrate state-of-the-art prediction models.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Machine learning**; • **General and reference** → **Evaluation**.

KEYWORDS

Machine Learning, Graph Neural Networks, Explainable AI, Counterfactual, Evaluation Framework

ACM Reference Format:
Mario Alfonso Prado-Romero, Bardh Prenkaj, and Giovanni Stilo. 2023. Developing and Evaluating Graph Counterfactual Explanation with GRETEL. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining (WSDM '23), February 27-March 3, 2023, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3539597.3573026

1 INTRODUCTION

Nowadays, social network usage accounts for a whopping 1200 petabytes[1] consumed in user information exchanges only in 2022. This enormous data pool has given sprout to ingenious deep-learning strategies that solve real-world problems. Social network tasks such as community detection [7], user friendship prediction [5], and session-based recommendations [6] have recently had significant performance improvement due to the advent of graph neural networks (GNNs) [4]. Despite their promising performance in these tasks, GNNs suffer from the so-called black-box problem, which hinders interpreting what happens *"under the hood"*. According to the European Commission [1], interpretability creates safer and innovation-friendly digital environments for users, developers, and business stakeholders, while encouraging privacy, trustworthiness, and fairness. To address these critical aspects of social networks, graph counterfactual explanation (GCE) methods answer the question *"how should the social network, or the actors therein, change to obtain a different outcome?"*. We refer the reader to [2] for the formal definition of GCE. A GCE method enables domain laymen to interpret the predicted outcome by inspecting which portion of the social network contributed the most (trustworthiness). Additionally, the usage of GCE methods helps emerge intrinsic bias/unfairness and privacy violations from the underlying prediction model, guaranteeing a reinforced and secure space for users. The literature on GCE lacks standardised explainers, datasets, and evaluation measurements, thus leading to a poor benchmarking system that hinders the promotion of the previous three aspects. To provide a cornerstone for researchers and practitioners (both of academia and private companies) in explainable AI, here we discuss GRETEL [3]. GRETEL is a unified framework that develops and systematically evaluates GCE methods and promotes Open Science and FAIR principles. Additionally, it provides a set of mechanisms to easily integrate and manage real and synthetic datasets, prediction models, explanation techniques, and evaluation measures. GRETEL's explanation workflow for a *new* instance is depicted in Figure 1.

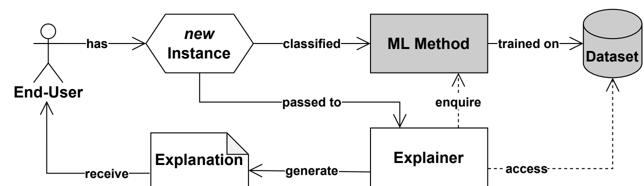

Figure 1: Typical workflow for the explanation of an instance classified by a black-box model.

To summarise what this demonstration covers, we provide the reader with the following conceptual map:

1. **Addressed problem** - Interpreting GNNs' predictions on social network tasks via counterfactual explanations. Here we focus on the evaluation of GCE scenarios.
2. **Target audience** - Academic researchers and industry practitioners in social networks, data mining, and explainable AI.

[1] https://earthweb.com/how-much-data-is-created-every-day/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WSDM '23, February 27-March 3, 2023, Singapore, Singapore
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9407-9/23/02…$15.00
https://doi.org/10.1145/3539597.3573026

(3) **Solution demonstration** - We show how the end-user can extend GRETEL to generate counterfactual explanations for their ad-hoc prediction scenarios.
(4) **Technical/commercial impact** - GRETEL is the first fully-extensible explainability framework for GNNs. Companies and researchers will benefit from GRETEL to uncover underlying biased predictors.

The rest of the paper is organised as follow. Section 2 briefly discusses the overview of GRETEL. Section 3 shows how to extend GRETEL to (i) evaluate the explainer on existing datasets and metrics (cf. 3.1), (ii) generate synthetic datasets (cf. 3.2), and (iii) define ad-hoc evaluation metrics (cf. 3.3). Section 4 concludes the paper.

2 GRETEL OVERVIEW

To provide the readers with a reference picture of GRETEL, we review its basic concepts and architecture. GRETEL was designed keeping in mind the point of view of a user (researcher or company employee) who wants to perform an exhaustive set of evaluations. The users can easily use and extend the framework in terms of datasets, explainers, metrics, and prediction models. The framework was designed using the *Object-Oriented* paradigm, where the framework's core is constituted mainly by abstract classes which need to be specialised in their implementations. To promote the framework's extensibility, the authors adopted the *"Factory Method"* design pattern and leveraged configuration files as constituting part of the running framework. Figure 2 illustrates the interaction between the main components of GRETEL.

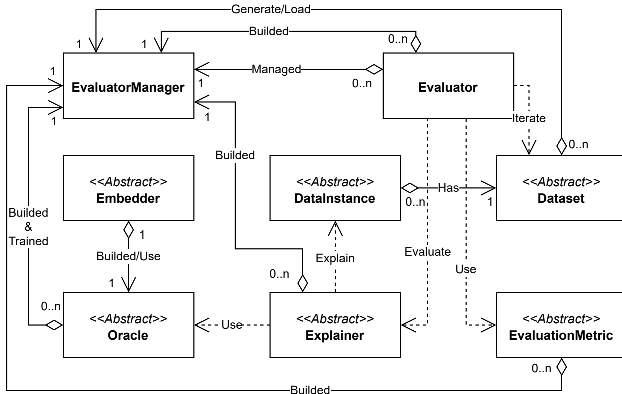

Figure 2: Overview of the main classes, of GRETEL Framework, and their relations.

The **Explainer** is the base class used by all explanation methods. The **Evaluator** evaluates a specific Explainer. The subclasses of **EvaluationMetric** define specific metrics that will be used to assess the quality of the Explainer. The **Oracle** provides a generic interface for interacting with the underlying prediction models. The **EvaluatorManager** facilitates running experiments specified by the configuration files and instantiating all the components needed to perform the different evaluations. The **DataInstance** class provides an abstract way to interact with data instances. The **Dataset** class manages all the details related to generating, reading, writing, and transforming the data.

To mitigate the efforts needed to evaluate new settings, GRETEL can generate synthetic datasets and train a model on the fly if it is not already stored and readily available. A more detailed description of the framework components can be found in [3].

3 DEMONSTRATION

Here, we show, by three fundamental use cases, how to extend GRETEL[2] to evaluate (i) a new Explainer on existing Datasets and Metrics (cf. 3.1), (ii) a new Synthetic Datasets on existing Explainers and Metrics (cf. 3.2), (iii) an existing Explainers and Datasets using new Metrics (cf. 3.3). The general extension workflow comprises of three-step:

(1) Create a custom implementation of one of the abstract classes (e.g. Explainer, Dataset, EvaluationMetric).
(2) Implement a custom factory class.
(3) Use the implementation to perform explanations/evaluations.

For the sake of space, we do not explicitly show how to use the **CustomDatasetFactory**, nor how to implement a **CustomEvaluationMetricFactory**, since their made can be easily inferred from the first use case of Section 3.1.

3.1 Creating a Custom Explainer

Here, we explain how to create a custom explanation method by extending the Explainer base class. Throughout this example, we assume that we use the datasets and oracles already available in GRETEL.

```
import Explainer

class DummyExplainer(Explainer):

    def __init__(self, id, config_dict=None) -> None:
        super().__init__(id, config_dict)
        self._name = 'DummyExplainer'

    def explain(self, instance, oracle, dataset):
        instance_label = oracle.predict(instance)
        counterfactual = instance

        for d_inst in dataset.instances:
            d_inst_label = oracle.predict(d_inst)

            if (instance_label != d_inst_label):
                counterfactual = d_inst
                return counterfactual

        return counterfactual
```

Listing 1: Minimal Code to implement a Custom Explainer

Listing 1 shows how to create a "dummy" explainer that searches for the first counterfactual example in the dataset. The code fragment 1 depicts the class **DummyExplainer** that inherits from the Explainer abstract class. The **__init__** method only assigns a name to the explainer and calls the same method from the parent class. Notice that the constructor of the explainer takes as input a configuration dictionary that contains the necessary parameters to build an explanation, including the storage path of the real/synthetic dataset.

[2] The source code and a video tutorial of the framework are available at https://github.com/MarioTheOne/GRETEL.

As shown in Figure 1, the **explain** method takes in input the instance to explain, the oracle to question, and the dataset to which the counterfactual explainer (might) want to access. This trivial explainer searches the dataset and returns the first instance - the counterfactual explanation - whose label differs from the one taken in the input. Notice that only one method must be implemented to create a custom explainer.

The second step, depicted by code fragment 2, explains how to create a **CustomExplainerFactory** that inherits from the **ExplainerFactory** base class. This new factory needs to access the DummyExplainer while maintaining access to other explainers already available in GRETEL. Thus, notice that CustomExplainerFactory has a new method called **get_dummy_explainer()** to create the DummyExplainer. Additionally, it must override the **get_explainer_by_name()** method of the parent class. Where checks if the explainer specified in the configuration file is an instance of DummyExplainer: in that case, the newly defined explainer is returned; otherwise, the same method in the parent class is called. Finally, it is possible to create the CustomExplainerFactory and pass it to the EvaluatorManager. The main part of fragment 2 shows how to create an EvaluatorManager, which uses the newly extended factory.

```
import ExplainerFactory, EvaluationMetricFactory

class CustomExplainerFactory(ExplainerFactory):

    def __init__(self, explainer_store_path):
        super().__init__(explainer_store_path)

    def get_explainer_by_name(self, explainer_dict,
        metric_factory): -> Explainer
        explainer_name = explainer_dict['name']

        if explainer_name == 'dummy_explainer':
            # Returning the explainer
            return self.get_dummy_explainer(explainer_dict)

        return super().get_explainer_by_name(explainer_dict,
                                             metric_factory)

    def get_dummy_explainer(self, config_dict=None):
        result = DummyExplainer(self._explainer_id_counter,
                                config_dict)

        self._explainer_id_counter += 1
        return result

if __name__ == '__main__':
    ex_factory = CustomExplainerFactory(ex_store_path)

    evm = EvaluatorManager(config_file_path, run_number=0,
                           explainer_factory=ex_factory)

    evm.create_evaluators()
    evm.evaluate()
```

Listing 2: Code to implement a Custom Explainer Factory

3.2 Creating a Custom Synthetic Dataset

In this scenario, we want to use an already available explanation method (or the custom one) on an ad-hoc synthetic dataset. The new custom dataset class must inherit from **Dataset**. The code fragment 3 shows how to create a new synthetic dataset where each instance is a cycle graph with either three (triangle) or four (square) vertices.

The **SquaresTrianglesDataset** class has a very simple initialisation method that creates just an empty list of instances. The method **create_cycle()** builds square and triangle graphs relying on the **networkx** library. The **generate_squares_triangles_dataset()** method creates the synthetic dataset choosing at random to populate it with a square or a triangle data instance. The methods **write_data()** and **read_data()**, useful to store and load the dataset respectively, are already provided by the parent class, and, thus, they do not need to be implemented in this child one. Once the new dataset class is implemented, we need to create a **CustomDatasetFactory** to make it available to the framework. Listing 4 shows how to perform this implementation.

```
import DataInstance, Dataset
import networkx as nx
import numpy as np

class SquaresTrianglesDataset(Dataset):

    def __init__(self, id, config_dict=None) -> None:
        super().__init__(id, config_dict)
        self.instances = []

    def create_cycle(self, cycle_size, role_label=1):
        # Creating an empty graph and adding the nodes
        graph = nx.Graph()
        graph.add_nodes_from(range(0, cycle_size))
        # Adding the edges  of the graph
        for i in range(cycle_size - 1):
            graph.add_edges_from([(i, i + 1)])
        graph.add_edges_from([(cycle_size - 1, 0)])
        # Creating the dictionary containing the node labels
        node_labels = {n:role_label for n in graph.nodes}
        # Creating the dictionary containing the edge labels
        edge_labels = {e:role_label for e in graph.edges}
        # Returning the cycle graph and the role labels
        return graph, node_labels, edge_labels

    def generate_squares_triangles_dataset(self, n_instances):
        self._name = f'st_ninst-{n_instances}'
        result = []
        for i in range(0, n_instances):
            is_triangle = np.random.randint(0,2)
            data_instance = DataInstance(
                self._instance_id_counter)
            self._instance_id_counter += 1
            i_name = f'g{i}'
            # Create the instance shape-specific properties
            cycle_size = 3 if is_triangle else 4
            role_label = 1 if is_triangle else 0
            # Create the triangle/square graph
            i_graph, i_node_labels, i_edge_labels =
                self.create_cycle(cycle_size=cycle_size,
                    role_label=role_label
            data_instance.graph_label = role_label
            # Creating the general instance properties
            data_instance.graph = i_graph
            data_instance.node_labels = i_node_labels
            data_instance.edge_labels = i_edge_labels
            data_instance.name = i_name
            result.append(data_instance)
        # return the set of instances
        self.instances = result
```

Listing 3: Code to implement a Custom Dataset

The CustomDatasetFactory follows the same main logic as the CustomExplainerFactory. In **get_dataset_by_name()** method is

possible to set the dataset generation parameters. Furthermore, it can be created the **get_squares_triangles_dataset()** method needed to generate the square and triangle dataset on the fly. The use of CustomDatasetFactory in the EvaluatorManager follows the same steps as in the previous use case (see 3.1).

```
import Dataset, DatasetFactory
import os, shutil

class CustomDatasetFactory(DatasetFactory):

    def __init__(self, data_store_path) -> None:
        self._data_store_path = data_store_path
        self._dataset_id_counter = 0

    def get_dataset_by_name(self, dataset_dict) -> Dataset:

        dataset_name = dataset_dict['name']
        params_dict = dataset_dict['parameters']

        # Check if the dataset is a squares-triangles dataset
        if dataset_name == 'squares-triangles':
            if not 'n_inst' in params_dict:
                raise ValueError('''"n_inst" parameter is
                    required for squares-triangles dataset''')

            return self.get_squares_triangles_dataset(params_dict['n_inst'],
                                                       False,
                                                       dataset_dict)
        else:
            # call the base method in to generate any of the originally
            # supported datasets
            return super().get_dataset_by_name(dataset_dict)


    def get_squares_triangles_dataset(self,
                                      n_instances,
                                      regenerate,
                                      config_dict) -> Dataset:

        result = SquaresTrianglesDataset(self._dataset_id_counter,
                                         config_dict)
        self._dataset_id_counter+=1

        ds_name = f'squares-triangles_instances-{n_instances}'
        ds_uri = os.path.join(self._data_store_path, ds_name)
        ds_exists = os.path.exists(ds_uri)

        if regenerate and ds_exists:
            shutil.rmtree(ds_uri)

        if(ds_exists):
            result.read_data(ds_uri)
        else:
            result.generate_squares_triangles_dataset(n_instances)
            result.generate_splits()
            result.write_data(self._data_store_path)

        return result
```

Listing 4: Code to implement a Custom Dataset Factory

3.3 Creating a Custom Metric

Finally, we illustrate how to implement a new bespoke metric that better represents the user's evaluation needs by leveraging GRETEL's components. In code fragment 5, we show how to create the *"Validity"* metric, which implements (for simplicity) the same logic as the built-in **Correctness** one. The straightforward implementation of the custom evaluation metric can be achieved by extending the EvaluationMetric base class and overriding the **evaluate** method. The process to create a custom **EvaluationMetricFactory** is similar to the one of the other previously explained factories. Hence, we do not provide the reader with the code fragment corresponding to CustomEvaluationMetricFactory. Note that this functionality is useful for future research since, typically, the set of used metrics tends to evolve over time while the field of research becomes more mature.

```
import EvaluationMetric

class ValidityMetric(EvaluationMetric):
    """Verifies that the class from the counterfactual example
    is different from that of the original instance"""
    def __init__(self, config_dict=None) -> None:
        super().__init__(config_dict)
        self._name = 'Validity'

    def evaluate(self, instance_1, instance_2, oracle):
        label_instance_1 = oracle.predict(instance_1)
        label_instance_2 = oracle.predict(instance_2)
        oracle._call_counter -= 2

        return int(label_instance_1 != label_instance_2)
```

Listing 5: Code to implement a new evaluation metric

4 CONCLUSIONS

We demonstrated how to use and extend GRETEL, the first unified framework for evaluating and developing graph counterfactual explanation (GCE) methods. We illustrated three use cases that provide users, coming from both academia and companies, with boilerplate material to implement custom evaluation and explanation scenarios. As already stated, GRETEL gives domain laymen the opportunity to extrapolate counterfactual examples from bespoke/built-in social network datasets while promoting trustworthiness in the generated explanations. Obliging under the Open Science movement and FAIR principles, GRETEL represents a cornerstone in future developments of explainable AI in social networks and, more in general, in graph-like data. Future directions on improving GRETEL encompass integrating privacy-preservation mechanisms on user profiles, friendship relationships, and other critical data according to specific social network ToS. Finally, we will incorporate metrics that measure the bias/unfairness of the explanations generated by the different explainers in GRETEL.

ACKNOWLEDGMENTS

The examples' tests have been conducted on the Caliban cluster of the DISIM Department of the University of L'Aquila. This work is partially supported by Territori Aperti a project funded by Fondo Territori Lavoro e Conoscenza CGIL CISL UIL, and by SoBigData-PlusPlus H2020-INFRAIA-2019-1 EU project, contract number 871042.

REFERENCES

[1] European Commission. 2020. *On Artificial Intelligence—A European Approach to Excellence and Trust.*
[2] M.A. Prado-Romero, B. Prenkaj, G. Stilo, and F. Giannotti. 2022. A Survey on Graph Counterfactual Explanations: Definitions, Methods, Evaluation. *arXiv preprint arXiv:2210.12089* (2022).
[3] M.A. Prado-Romero and G. Stilo. 2022. GRETEL: Graph Counterfactual Explanation Evaluation Framework. In *Proc. of the 31st ACM Int. Conf. on Inf. and Knowl. Management.*
[4] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2008. The graph neural network model. *IEEE trans. on neural networks* 20, 1 (2008), 61–80.
[5] X. Wei, Y. Liu, J. Sun, Y. Jiang, Q. Tang, and K. Yuan. 2022. Dual Subgraph-Based Graph Neural Network for Friendship Prediction in Location-Based Social Networks. *ACM Trans. Knowl. Discov. Data* (2022).
[6] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. 2019. Session-based recommendation with graph neural networks. In *Proc. of the AAAI Conf. on artificial intelligence*, Vol. 33. 346–353.
[7] X. Wu, Y. Xiong, Y. Zhang, Y. Jiao, C. Shan, Y. Sun, Y. Zhu, and P.S. Yu. 2022. CLARE: A Semi-Supervised Community Detection Algorithm. In *Proc. of the 28th ACM SIGKDD Conf. on Knowl. Discov. and Data Mining*. 2059–2069.