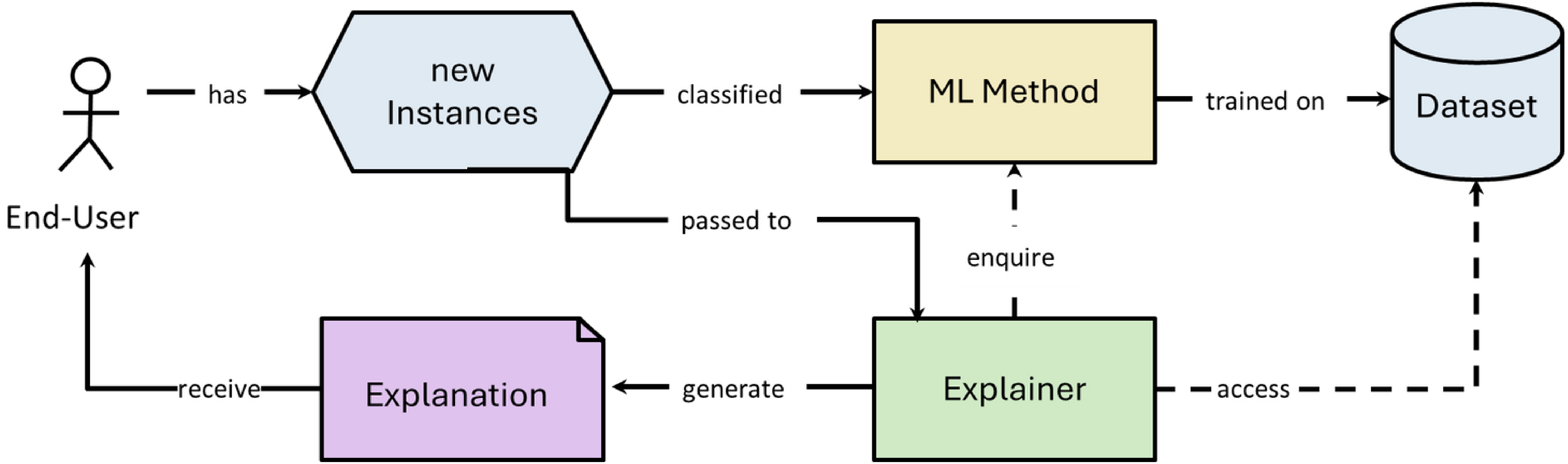
GRETEL: THE GOOD, THE BAD AND THE UGLY

Bardh Prenkaj
Post-doc in AI
prenkaj@di.uniroma1.it

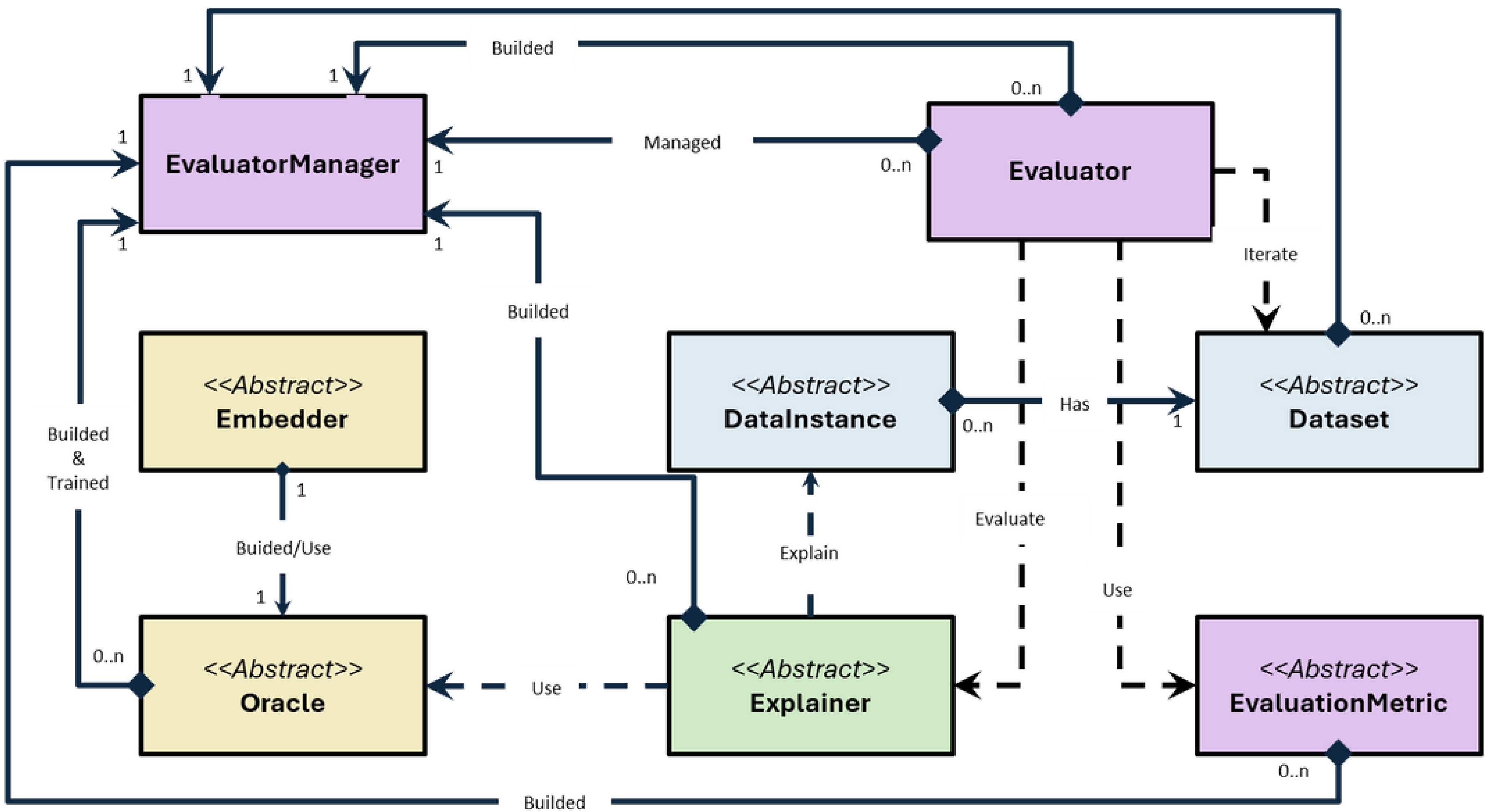
SAPIENZA
UNIVERSITÀ DI ROMA

aiim

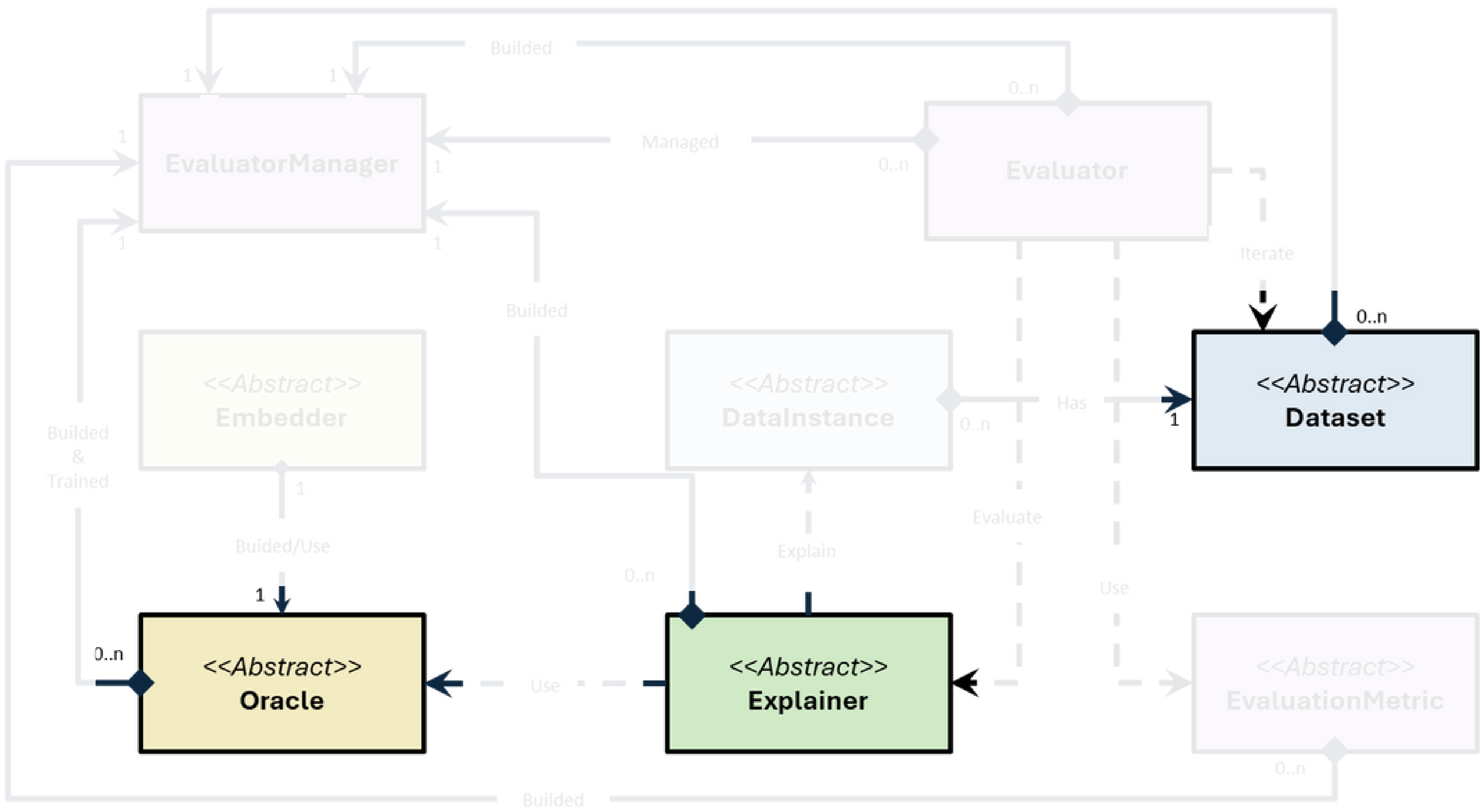
Typical Explainability Workflow



GRETEL


EvaluatorManager
Evaluator
<<Abstract>>
Embedder
<<Abstract>>
DataInstance
<<Abstract>>
Dataset
<<Abstract>>
Oracle
<<Abstract>>
Explainer
<<Abstract>>
EvaluationMetric
Builded
Managed
Iterate
Has
Builded & Trained
Buided/Use
Explain
Evaluate
Use
Use
Builded
Builded
1
0..n

GRETEL



Builded
1
1
0..n
1
EvaluatorManager
1
Managed
0..n
Evaluator
1
1
Iterate
Builded
0..n
<<Abstract>>
Embedder
<<Abstract>>
DataInstance
Has
0..n
1
<<Abstract>>
Dataset
Builded
&
Trained
1
Evaluate
Buided/Use
Explain
0..n
Use
1
0..n
<<Abstract>>
Oracle
Use
<<Abstract>>
Explainer
<<Abstract>>
EvaluationMetric
0..n
Builded

THE GOOD

What’s good in GRETEL?

- Highly modular framework
- **Core philosophy:** users can implement their explainers by only extending the Explainer class without having to worry about the entire XAI pipeline
- 80% of graph classification SoA methods are covered inside
- Configuration opportunities for already supported methods, datasets, evaluations

Example of good: Ad-hoc explainer

giovanni, 4 days ago | 1 author (giovanni)

```
class DCESearchExplainer(Explainer):
    """The Distribution Compliant Explanation Search Explainer performs a search of
    the minimum counterfactual instance in the original dataset instead of generating
    a new instance"""      giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        super().init()
        self._gd = GraphEditDistanceMetric()
        self.fold_id=-1
        self.dist_mat = np.full((len(self.dataset.instances), len(self.dataset.instances)), -1)
        self.cls_mat = np.full((len(self.dataset.instances), len(self.dataset.instances)), -1)


    def explain(self, instance):
        l_input_inst = self.oracle.predict(instance)

        # if the method does not find a counterfactual example returns the original graph
        min_counterfactual = instance

        min_counterfactual_dist = sys.float_info.max

```

Example of good: Ad-hoc explainer

```
giovanni, 4 days ago | 1 author (giovanni)
class DCESearchExplainer(Explainer):
    """The Distribution Compliant Explanation Search Explainer performs a search of
    the minimum counterfactual instance in the original dataset instead of generating
    a new instance"""       giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        super().init()
        self._gd = GraphEditDistanceMetric()
        self.fold_id=-1
        self.dist_mat = np.full((len(self.dataset.instances), len(self.dataset.instan
        self.cls_mat = np.full((len(self.dataset.instances), len(self.dataset.instances)), -1)


    def explain(self, instance):
        l_input_inst = self.oracle.predict(instance)

        # if the method does not find a counterfactual example returns the original graph
        min_counterfactual = instance

        min_counterfactual_dist = sys.float_info.max

```

Just extend the
abstract Explainer class

Example of good: Ad-hoc explainer

giovanni, 4 days ago | 1 author (giovanni)

```python
class DCESearchExplainer(Explainer):
    """The Distribution Compliant Explanation Search Explainer performs a search of
    the minimum counterfactual instance in the original dataset instead of generating
    a new instance"""       giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        super().init()
        self._gd = GraphEditDistanceMetric()
        self.fold_id=-1
        self.dist_mat = np.full((len(self.dataset.instances), len(self.dataset.instances)), -1)
        self.cls_mat = np.full((len(self.dataset.instances), len(self.dataset.instances)), -1)

    def explain(self, instance):
        l_input_inst = self.oracle.predict(instance)

        # if the method does not find a counterfactual example returns the original graph
        min_counterfactual = instance

        min_counterfactual_dist = sys.float_info.max
```

Define the abstract method **explain**

Example of good: Ad-hoc dataset

```
giovanni, 4 days ago | 1 author (giovanni)
class ADHD(Generator):        giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        base_path = self.local_config['parameters']['data_dir']
        # Path to the instances of the "Attention Deficit Hyperactivity Disorder class"
        self.adhd_class_path = join(base_path, 'adhd_dataset')
        self._td_file_path = join(base_path, 'td')
        self.generate_dataset()

    def get_num_instances(self):
        return len(self.dataset.instances)

    def generate_dataset(self):
        if not len(self.dataset.instances):
            self.read_adjacency_matrices()
```

Example of good: Ad-hoc dataset

```
giovanni, 4 days ago | 1 author (giovanni)
class ADHD(Generator):        giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        base_path = self.local_config['parameters']['data_dir']
        # Path to the instances of the "Attention Deficit Hyperactivity Disorder class"
        self.adhd_class_path = join(base_path, 'adhd_dataset')
        self._td_file_path = join(base_path, 'td')
        self.generate_dataset()

    def get_num_instances(self):
        return len(self.dataset.instances)

    def generate_dataset(self):
        if not len(self.dataset.instances):
            self.read_adjacency_matrices()
```

Just extend the **abstract Generator** class

Example of good: Ad-hoc dataset

```python
giovanni, 4 days ago | 1 author (giovanni)
class ADHD(Generator):        giovanni, 4 days ago • FIRST IMPORT GOLD

    def init(self):
        base_path = self.local_config['parameters']['data_dir']
        # Path to the instances of the "Attention Deficit Hyperactivity Disorder class"
        self.adhd_class_path = join(base_path, 'adhd_dataset')
        self._td_file_path = join(base_path, 'td')
        self.generate_dataset()

    def get_num_instances(self):
        return len(self.dataset.instances)

    def generate_dataset(self):
        if not len(self.dataset.instances):
            self.read_adjacency_matrices()
```

Implement the **generate_dataset** abstract method to contain the dataset creation logic

Example of good: Extend Oracle

giovanni, 4 days ago | 1 author (giovanni)

```python
class Oracle(Trainable,metaclass=ABCMeta):    giovanni, 4 days ago • FIRST IMPORT GOLD
    def __init__(self, context:Context, local_config) -> None:
        super().__init__(context, local_config)
        self._call_counter = 0

    @final
    def predict(self, data_instance):
        """predicts the label of a given data instance
        -------------
        INPUT:
            data_instance : The instance whose class is going to be predicted
        -------------
        OUTPUT:
            The predicted label for the data instance
        """
        self._call_counter += 1

        return self._real_predict(data_instance)

    @final
    def predict_proba(self, data_instance):
        """predicts the probability estimates for a given data instance
        -------------
```

\+

```python

    @abstractmethod
    def _real_predict(self, data_instance):
        pass

    @abstractmethod
    def _real_predict_proba(self, data_instance):
        pass

```

Example of good: Extend Oracle

```python
giovanni, 4 days ago | 1 author (giovanni)
class Oracle(Trainable,metaclass=ABCMeta):        giovanni, 4 days ago • FIRST IMPORT GOLD
    def __init__(self, context:Context, local_config) -> None:
        super().__init__(context, local_config)
        self._call_counter = 0

    @final
    def predict(self, data_instance):
        """predicts the label of a given data instance
        -------------
        INPUT:
            data_instance : The instance whose class is going to be predicted
        -------------
        OUTPUT:
            The predicted label for the data instance
        """
        self._call_counter += 1

        return self._real_predict(data_instance)

    @final
    def predict_proba(self, data_instance):
        """predicts the probability estimates for a given data instance
        -------------
```

+

```python

    @abstractmethod
    def _real_predict(self, data_instance):
        pass

    @abstractmethod
    def _real_predict_proba(self, data_instance):
        pass

```

When creating a **new** Oracle (predictor model) you need to **define** these abstract methods

Example of good: Extend Oracle

giovanni, 4 days ago | 1 author (giovanni)

```
class TreeCyclesOracle(Oracle):        giovanni, 4 days ago • FIRST IMPORT (

    def init(self):
        super().init()
        self.model = ""

    def real_fit(self):
        pass

    def _real_predict(self, data_instance):
        try:
            nx.find_cycle(data_instance.get_nx(), orientation='ignore')
            return 1
        except nx.exception.NetworkXNoCycle:
            return 0

    def _real_predict_proba(self, data_instance):
        # softmax-style probability predictions
        try:
            nx.find_cycle(data_instance.get_nx(), orientation='ignore')
            return np.array([0,1])
        except nx.exception.NetworkXNoCycle:
            return np.array([1,0])
```

THE
BAD

What's **bad** in GRETEL?

- High learning curve
- Only graph classification is supported for <u>now</u>
- Continuous development; you might find changes happening in real-time
- Some bad/ugly choices of using the same naming of built-in methods
- WILL NOT SUPPORT TENSORFLOW.... EVER

Example of bad: init()

```
giovanni, 4 days ago | 1 author (giovanni)
class TreeCyclesOracle(Oracle):        giovanni, 4 days ago • FIRST IMPORT C

    def init(self):
        super().init()
        self.model = ""

    def real_fit(self):
        pass

    def _real_predict(self, data_instance):
        try:
            nx.find_cycle(data_instance.get_nx(), orientation='ignore')
            return 1
        except nx.exception.NetworkXNoCycle:
            return 0

    def _real_predict_proba(self, data_instance):
        # softmax-style probability predictions
        try:
            nx.find_cycle(data_instance.get_nx(), orientation='ignore')
            return np.array([0,1])
        except nx.exception.NetworkXNoCycle:
            return np.array([1,0])
```

AND THE UGLY

What’s **ugly** in GRETEL?

- A complex project structure
- Configuration files can be daunting
- There’s a Context object passing around all the create objects dynamically; there’s a check_configuration method that can set default hyperparameters
- Some **__call__** methods are overridden to obfuscate the torch-like functionality to the end-user

Example of ugly: check_configuration()

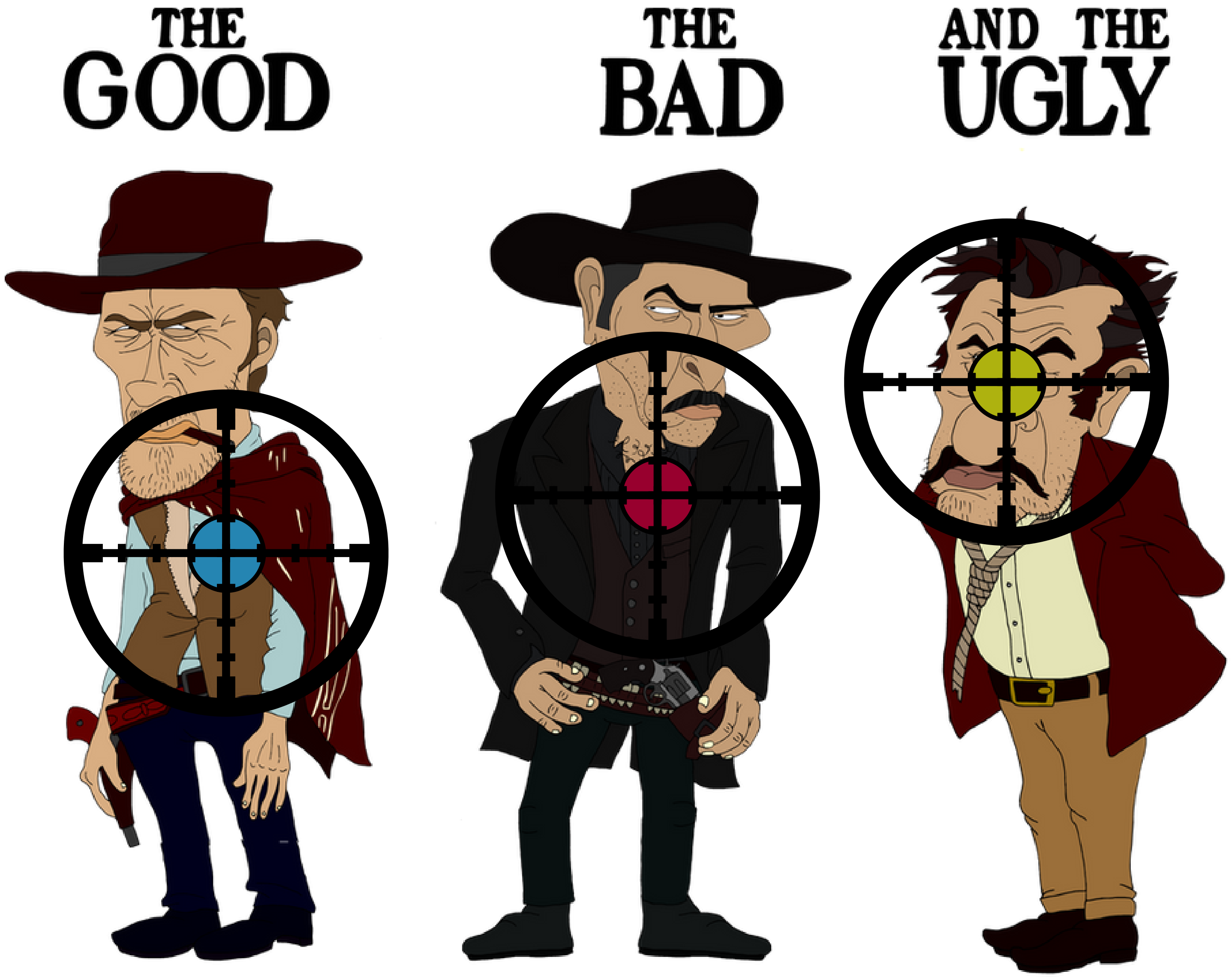
```python
    def check_configuration(self):
        super().check_configuration()
        self.local_config['parameters']['n_labels'] =  self.local_config['parameters'].get('n_labels', 2)
        self.local_config['parameters']['batch_size_ratio'] =  self.local_config['parameters'].get('batch_size_ratio', .1)
        self.local_config['parameters']['h_dim'] =  self.local_config['parameters'].get('h_dim', 10)
        self.local_config['parameters']['z_dim'] =  self.local_config['parameters'].get('z_dim', 10)
        self.local_config['parameters']['dropout'] =  self.local_config['parameters'].get('dropout', .1)
        self.local_config['parameters']['encoder_type'] =  self.local_config['parameters'].get('encoder_type', 'gcn')
        self.local_config['parameters']['graph_pool_type'] =  self.local_config['parameters'].get('graph_pool_type', 'mean')
        self.local_config['parameters']['disable_u'] =  self.local_config['parameters'].get('disable_u', False)
        self.local_config['parameters']['epochs'] =  self.local_config['parameters'].get('epochs', 200)
        self.local_config['parameters']['alpha'] =  self.local_config['parameters'].get('alpha', 5)
        self.local_config['parameters']['lr'] =  self.local_config['parameters'].get('lr', 1e-3)
        self.local_config['parameters']['weight_decay'] =  self.local_config['parameters'].get('weight_decay', 1e-5)
        self.local_config['parameters']['lambda_sim'] =  self.local_config['parameters'].get('lambda_sim', 1)
        self.local_config['parameters']['lambda_kl'] =  self.local_config['parameters'].get('lambda_kl', 1)
        self.local_config['parameters']['lambda_cfe'] =  self.local_config['parameters'].get('lambda_cfe', 1)
        self.local_config['parameters']['beta_x'] =  self.local_config['parameters'].get('beta_x', 10)
        self.local_config['parameters']['beta_adj'] =  self.local_config['parameters'].get('beta_adj', 10)

        n_nodes = self.local_config['parameters'].get('n_nodes', None)
        if not n_nodes:
            n_nodes = max([x.num_nodes for x in self.dataset.instances])
        self.local_config['parameters']['n_nodes'] = n_nodes

        self.local_config['parameters']['feature_dim'] = len(self.dataset.node_features_map)
```

Example of ugly: check_configuration()

```python
    def check_configuration(self):
        super().check_configuration()
        self.local_config['parameters']['n_labels'] =  self.local_config['parameters'].get('n_labels', 2)
        self.local_config['parameters']['batch_size_ratio'] =  self.local_config['parameters'].get('batch_size_ratio', .1)
        self.local_config['parameters']['h_dim'] =  self.local_config['parameters'].get('h_dim', 10)
        self.local_config['parameters']['z_dim'] =  self.local_config['parameters'].get('z_dim', 10)
        self.local_config['parameters']['dropout'] =  self.local_config['parameters'].get('dropout', .1)
        self.local_config['parameters']['encoder_type'] =  self.local_config['parameters'].get('encoder_type', 'gcn')
        self.local_config['parameters']['graph_pool_type'] =  self.local_config['parameters'].get('graph_pool_type', 'mean')
        self.local_config['parameters']['disable_u'] =  self.local_config['parameters'].get('disable_u', False)
        self.local_config['parameters']['epochs'] =  self.local_config['parameters'].get('epochs', 200)
        self.local_config['parameters']['alpha'] =  self.local_config['parameters'].get('alpha', 5)
        self.local_config['parameters']['lr'] =  self.local_config['parameters'].get('lr', 1e-3)
        self.local_config['parameters']['weight_decay'] =  self.local_config['parameters'].get('weight_decay', 1e-5)
        self.local_config['parameters']['lambda_sim'] =  self.local_config['parameters'].get('lambda_sim', 1)
        self.local_config['parameters']['lambda_kl'] =  self.local_config['parameters'].get('lambda_kl', 1)
        self.local_config['parameters']['lambda_cfe'] =  self.local_config['parameters'].get('lambda_cfe', 1)
        self.local_config['parameters']['beta_x'] =  self.local_config['parameters'].get('beta_x', 10)
        self.local_config['parameters']['beta_adj'] =  self.local_config['parameters'].get('beta_adj', 10)

        n_nodes = self.local_config['parameters'].get('n_nodes', None)
        if not n_nodes:
            n_nodes = max([x.num_nodes for x in self.dataset.instances])
        self.local_config['parameters']['n_nodes'] = n_nodes

        self.local_config['parameters']['feature_dim'] = len(self.dataset.node_features_map)
```

Example of ugly: __call__

```

    def __call__(self, *args: Tuple[GraphInstance], **kwds: Any) -> Any:
        torch_data = torch.from_numpy(args[0].data[None,None,:,:]).float()
        return self.generator(torch_data)      You, 22 hours ago • Uncommitted changes
```



Let’s code

Open research questions

- Are generative counterfactual explainers worth it?
- What’s the difference between counterfactual explanations and adversarial attacks?
- How sure are we about the counterfactuals generated? Can we incorporate uncertainty in them?
- How can we backtrack near the decision boundary once we overshoot on the other side to produce minimally perturbed counterfactuals?