Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] . As we state in the abstract, our goal is to provide a Python framework for benchmarking counterfactual explanation methods. Users can easily evaluate our results by accessing our Github repository, where we host our Python framework and our benchmarking results.

   (b) Did you describe the limitations of your work? [Yes] . In Section 6, we discuss the current limitations of our approach. The counterfactual explanation methods are based on the original implementation of the respective research groups. Researchers mostly implement their experiments and models for specific ML frameworks and data sets. For example, some explanation methods are restricted to Tensorflow and are not applicable to Pytorch models.

   (c) Did you discuss any potential negative societal impacts of your work? [N/A] . We discuss the broader impact of our benchmarking library in Section 6; we mainly see positive impacts on the literature of algorithmic recourse.

   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] . We have read the ethics review guidelines and attest that our paper conforms to the guidelines.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] . We did not provide theoretical results.

   (b) Did you include complete proofs of all theoretical results? [N/A] . We did not provide theoretical results.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] . Details of implementations, data sets and instructions can be found here: Appendices A, C, E, and our Github repository.

   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] . Please see Appendices E and C.

   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] . Error bars have been reported for our cost comparisons in terms of the 25th and 75ht percentiles of the cost distribution, see for example Figure 3.

   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] . All models are evaluated on an i7-8550U CPU with 16 Gb RAM, running on Windows 10.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
    (a) If your work uses existing assets, did you cite the creators? [Yes] . The data sets, which are publicly available are appropriately cited in Section 5. We cite and link to any additional code used, for example [3].
    (b) Did you mention the license of the assets? [Yes] . All assets are publicly available and attributed.
    (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] . Our implementation and code is accessible through our Github repository.
    (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] . We use publicly available data sets without any personal identifying information.
    (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] . We use publicly available data sets without any personal identifying information.
5. If you used crowdsourcing or conducted research with human subjects...
    (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] . We did not use crowdsourcing or conduct research with human subjects.
    (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] . We did not use crowdsourcing or conduct research with human subjects.
    (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] . We did not use crowdsourcing or conduct research with human subjects.

A `CARLA`’s Software Interface

In the following, we introduce our open-source benchmarking software `CARLA`. we describe the architecture in more detail and provide examples of different use-cases and their implementation.

A.1 `CARLA`’s High Level Software Architecture

The purpose of this Python library is to provide a simple and standardized framework to allow users to apply different state-of-the-art recourse methods to arbitrary data sets and black-box-models. It is possible to compare different approaches and save the evaluation results, as described in Section 4.2. For research groups, `CARLA` provides an implementation interface to integrate new recourse methods in an easy-to-use way, which allows to compare their method to already existing methods.

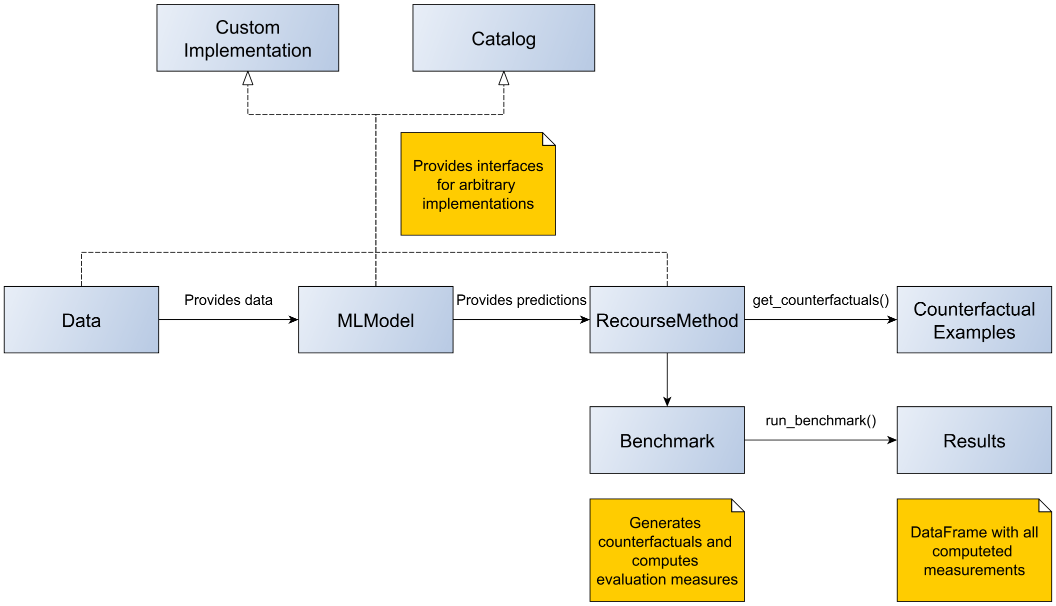

Figure 4: Architecture of the `CARLA` python library. The silver boxes show the individual objects that will be created to generate counterfactual explanations and evaluate recourse methods. Useful explanations to specific processes are illustrated as yellow notes. The dashed arrows are showing the different implementation possibilities; either use pre-defined catalog objects or provide custom implementation. All dependencies between these objects are visualised by solid arrows with an additional description.

A simplified visualization of the `CARLA` software architecture is depicted in Figure 4. For every component (*Data*, *MLModel*, and *RecourseMethod*) the library provides the possibility to use existing methods from our catalog, or extend the users custom methods and implementations. The components represent an interface to the key parts in the process of generating counterfactual explanations. *Data* provides a common way to access the data across the software and maintains information about the features. *MLModel* wraps each black-box model and stores details on the encoding, scaling and feature order specific to the model. The primary purpose of *RecourseMethod* is to provide a common interface to easily generate counterfactual examples.

Besides the possibility to use pretrained black-box-models and preprocessed data, `CARLA` provides an easy way to load and define own data sets and model structures independent of their framework (e.g., Pytorch, Tensorflow, sklearn). The following sections will give an overview and provide example implementations of different use cases.

A.2 `CARLA` for Research Groups

One of the most exciting features of `CARLA` is the *RecourseMethod*-wrapper which enables researchers to implement their own method to generate counterfactual explanations. This opens up a way of standardized and consistent comparisons between different recourse methods. Strong and weak points of new algorithms can be stated, benchmarked and analysed in future work.

In Figure 5, we show how an implementation of a custom recourse method can be structured. After defining the recourse method, it can be used with the library to generate counterfactuals for a given data set and benchmark its results against other methods. Researchers have the choice to do this using our provided catalog of data sets, recourse methods and black-box models (Figure 6) or use their own models and data sets (see Figures 7 and 8).

```python

from carla import RecourseMethod

# Custom recourse implementations need to
# inherit from the RecourseMethod interface
class MyRecourseMethod(RecourseMethod):
    def __init__(self, mlmodel):
        super().__init__(mlmodel)

    # Generate and return encoded and
    # scaled counterfactual examples
    def get_counterfactuals(self, factuals: pd.DataFrame):
    [...]
    return counterfactual_examples

```

Figure 5: Pseudo-implementation of the `CARLA` recourse method wrapper

A.3 `CARLA` as a Recourse Library

A common usage of the package is to generate counterfactual explanations. This can be done by loading black-box-models and data sets from our provided catalogs, or by user-defined models and data sets via integration with the defined interfaces. Figure 6 shows an implementation example of a simple use-case, applying a recourse method to a pre-defined data set and model from our catalog. After importing both catalogs, the only necessary step is to describe the data set name (e.g., adult, give me some credit, or compas) and the model type (e.g., ann, or linear) the user wants to load. Every recourse method contains the same properties to generate counterfactual explanations.

```python

from carla import DataCatalog, MLModelCatalog
from carla.recourse_methods import GrowingSpheres

# 1. Load data set from the DataCatalog
data_name = "adult"
dataset = DataCatalog(data_name)

# 2. Load pre-trained black-box model from the MLModelCatalog
model = MLModelCatalog(dataset, "ann")

# 3. Load recourse model with model specific hyperparameters
gs = GrowingSpheres(model)

# 4. Generate counterfactual examples
factuals = dataset.raw.sample(10)
counterfactuals = gs.get_counterfactuals(factuals)

```

Figure 6: Example implementation of `CARLA`, using the data and model catalog.

To give users the possiblity to explore their own black-box-model on a custom data set, we provide an easy-to-use interface, that is able to wrap every possible model or data set. These interfaces specify particular properties users have to implement, to be able to work with the library. Figure 7 shows an example implementation of the data wrapper, and Figure 8 depicts the same for an arbitrary black-box-model. After defining data set and black-box model classes, users simply need to call the canonical methods and generate counterfactual examples, similar to the process in Figure 6.

```python

from carla import Data
from carla.recourse_methods import GrowingSpheres

# Custom data set implementations need to inherit from the Data interface
class MyOwnDataSet(Data):
    def __init__(self):
        # The data set can e.g. be loaded in the constructor
        self._dataset = load_dataset_from_disk()

    # List of all categorical features
    def categoricals(self):
        return [...]

    # List of all continous features
    def continous(self):
        return [...]

    # List of all immutable features which
    # should not be changed by the recourse method
    def immutables(self):
        return [...]

    # Feature name of the target column
    def target(self):
        return "label"

    # Non-encoded and  non-normalized, raw data set
    def raw(self):
        return self._dataset

```

Figure 7: Pseudo-implementation of the `CARLA` data wrapper

A.4 Benchmarking Recourse Methods

Besides the generation of counterfactual explanations, the focus of `CARLA` lies on benchmarking recourse methods. Users are able to compute evaluation measures to make qualitative statements about usability and applicability.

All measurements, which are described in Section 4.2, are implemented in the *Benchmarking* class of `CARLA` and can be used for every wrapped recourse method. Figure 9 shows an example implementation of a benchmarking process based on the variables of Figure 6.

B Additional Experimental Results

In this Section, we depict the missing experiments from the COMPAS data set in Figure 10 and Table 4. These results underline the trends that we have already highlighted in Section 5.

C ML Classifiers

In this section, we describe how the black–box models f were fitted. `CARLA` supports different ML libraries to estimate these models (e.g., Pytorch, Tensorflow) as the implementations of the various explanation methods work with a particular ML library. We note that the various explanation methods rely on different binary feature encodings. `DICE`, for example, requires that binary inputs are supplied as one–hot vectors, while `FACE` needs binary features encoded in a single column. If this was the case, we fitted two ML models, using the same hyperparameters, and generated CEs with respect to the same set of samples.

To ensure similar behavior between the different ML libraries and encoding variations, each black-box model type has the same structure (e.g., number of hidden layer, number of neurons), and training parameters (e.g., learning rate, epochs, etc.).

```python
from carla import MLModel

# Custom black-box models need to inherit from
# the MLModel interface
class MyOwnModel(MLModel):
    def __init__(self, data):
        super().__init__(data)
        # The constructor can be used to load or build an
        # arbitrary black-box-model
        self._mymodel = load_model()

        # Define a fitted sklearn scaler to normalize input data
        self.scaler = MySklearnScaler().fit()

        # Define a fitted sklearn encoder for binary input data
        self.encoder = MySklearnEncoder.fit()

    # List of the feature order the ml model was trained on
    def feature_input_order(self):
        return [...]

    # The ML framework the model was trained on
    def backend(self):
        return "pytorch"

    # The black-box model object
    def raw_model(self):
        return self._mymodel

    # The predict function outputs
    # the continous prediction of the model
    def predict(self, x):
        return self._mymodel.predict(x)

    # The predict_proba method outputs
    # the prediction as class probabilities
    def predict_proba(self, x):
        return self._mymodel.predict_proba(x)
```

Figure 8: Pseudo-implementation of the `CARLA` black-box-model wrapper

The first model is a multi-layer perceptron, consisting of three hidden layers with 18, 9 and 3 neurons, respectively. We use ReLu activation functions and binary cross entropy to calculate class probabilities. Optimization of the loss function is done by RMSProp [54] using a learning rate of 0.002 for every data set. By performing 25 epochs on COMPAS and 10 epochs on Adult and GMC we reached acceptable performance. Further increasing epochs gave rise to very marginal performance increases. For Adult we use a batch–size of 1024, for COMPAS 25 and for GMC 2048.

To allow a more extensive comparison between CE methods, we choose linear models as the second black–box model category for which we evaluate the CE methods. Again, we optimized these models with RMSProp using a binary cross entropy loss. For Adult, we used 100 epochs and a batch–size of 2048, for COMPAS we choose 25 epochs and batch–size of 128, and for GMC we chose 10 epochs with a batch–size of 2048. The learning rate on every data set is set 0.002. Table 5 provides an overview of the model's classification accuracies.

```python
from carla import Benchmark

# 1. Initilize the benchmarking class by passing
# black-box-model, recourse method, and factuals into it
benchmark = Benchmark(model, gs, factuals)

# 2. Either only compute the distance measures
distances = benchmark.compute_distances()

# 3. Or run all implemented measurements and create a
# DataFrame which consists of all results
results = benchmark.run_benchmark()
```

Figure 9: Pseudo-implementation of the `CARLA` recourse method wrapper

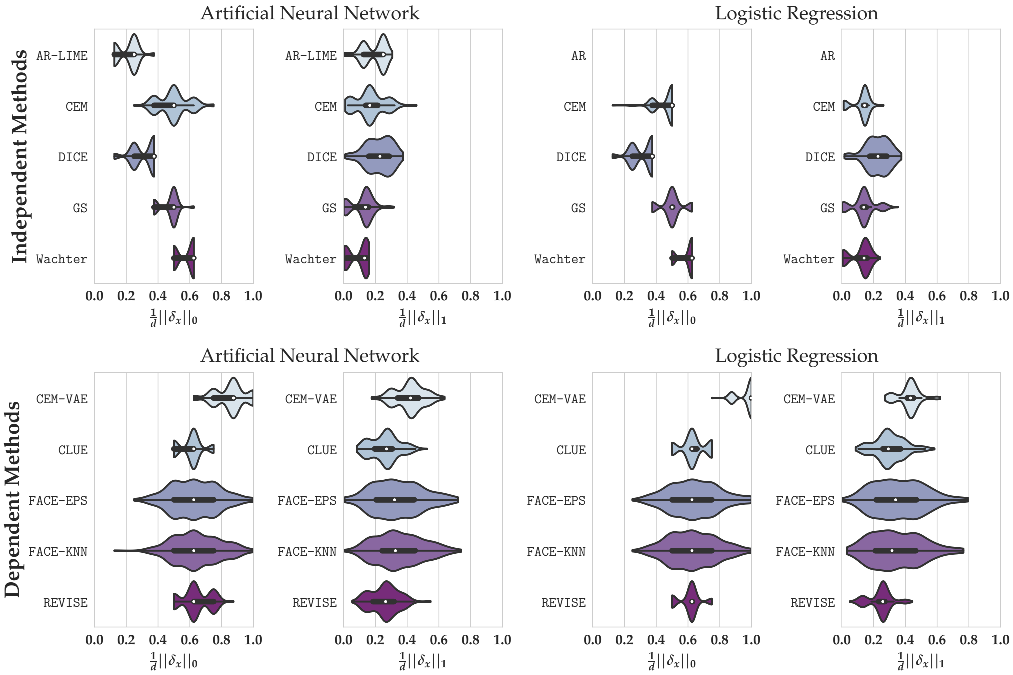

Figure 10: COMPAS Data

Figure 11: Evaluating the distribution of costs of counterfactual explanations on the COMPAS dataset. For all instances with a negative prediction ($\{x \in \mathcal{D} : f(x) < \theta\}$), we plot the distribution of ℓ_0 and ℓ_1 costs of algorithmic recourse as defined in (1) for a logistic regression and an artificial neural network classifier. The white dots indicate the medians (lower is better), and the black boxes indicate the interquartile ranges. We distinguish between independence based and dependence based methods.

	Adult	COMPAS	Give Me Credit
Logistic Regression	0.83	0.84	0.92
Neural Network	0.84	0.85	0.93

Table 5: Performance of classification models used for generating algortihmic recourse.

		Artificial Neural Network					Logistic Regression				
Data Set	Method	*yNN*	redund.	violation	success	$\bar{t}(s)$	*yNN*	redund.	violation	success	$\bar{t}(s)$
COMPAS	`AR(-LIME)`	**0.91**	**0.00**	**0.02**	0.53	0.06	–	–	–	0.00	**0.01**
	`CEM`	**0.98**	2.29	0.43	**1.00**	0.89	0.93	1.88	0.99	**1.00**	0.86
	`DICE`	0.89	0.88	1.03	**1.00**	0.09	**0.95**	0.94	0.90	**1.00**	0.09
	`GS`	0.44	0.97	0.03	**1.00**	**0.01**	0.60	**0.64**	**0.02**	**1.00**	**0.01**
	`Wachter`	0.56	1.77	0.74	0.66	10.90	0.50	1.21	0.79	**1.00**	0.02

(a) Independence based methods

		Artificial Neural Network					Logistic Regression				
Data Set	Method	*yNN*	redund.	violation	success	$\bar{t}(s)$	*yNN*	redund.	violation	success	$\bar{t}(s)$
COMPAS	`CEM-VAE`	**1.00**	5.59	1.98	**1.00**	0.89	1.00	6.91	2.14	**1.00**	0.87
	`CLUE`	0.99	4.06	**1.08**	**1.00**	2.03	**1.00**	4.62	1.25	**1.00**	1.88
	`FACE-EPS`	0.94	3.71	1.55	0.99	0.45	0.97	3.94	1.62	0.99	0.45
	`FACE-KNN`	0.94	3.83	1.63	**1.00**	**0.44**	0.97	3.86	1.57	**1.00**	**0.44**
	`REVISE`	**1.00**	**3.29**	1.29	**1.00**	6.06	0.92	**3.15**	**1.03**	**1.00**	5.35

(b) Dependence based methods

Table 4: Summary of COMPAS results for independence and dependence based methods. For all instances with a negative prediction ($\{x \in \mathcal{D} : f(x) < \theta\}$), we compute counterfactual explanations for which we then measure yNN (higher is better), redundancy (lower is better), violation (lower is better), success rate (higher is better) and time (lower is better). We distinguish between a logistic regression and an artificial neural network classifier. Detailed descriptions of these measures can be found in Section 4. The results are discussed in Appendix B.

D COMPAS Data Set Description

The **COMPAS** data set [21] contains data for more than 10,000 criminal defendants in Florida. It is used by the jurisdiction to score defendant's likelihood of reoffending. We kept a small part of the raw data as features like *name*, *id*, *casenumbers* or *date-time* were dropped. The classification task consists of classifying an instance into high risk of recidivism (*score_text* is high). By converting the feature *race* into *white* and *non-white*, we keep the categorical input binary. Similar to Adult, the immutable features for COMPAS are *age, sex* and *race*.

E Hyperparameter Search for the Counterfactual Explanation and Recourse Methods

We generated counterfactual explanations for instances from H^-, the set of factuals with negative class predictions.

`AR` and `AR-LIME` It frequently occurred that the action with the lowest cost did not flip the prediction of the black-box classifier. To overcome this problem, we let `AR` compute a flipset of 150 actions per instance, and subsequently search this set for low–cost CEs. For `AR-LIME`, we used `LIME` [49] and required `sampling around the instance` to make sure that the coefficients at x were truly local.

`CEM` After performing grid search, we set the ℓ_1 weight to 0.9 and the ℓ_2 weight to 0.1, yielding the strongest performance. For `CEM-VAE` we set the ℓ_2 weight to 0.1, and the VAE–weight to 0.9.

`CLUE` We use the default hyperparameters from [3], which are set as a function of the data set dimension d. Performing hyperparameter search did not yield results that were improving distances while keeping the same success rate.

`DICE` Since `DICE` is able to compute a set of counterfactuals for a given instance, we only chose to generate one CE per input instance. We use a *grid search* for the *proximity* and *diversity* weights.

`FACE` To determine the strongest hyperparameters for the graph size we conducted a *grid search*. We found that values of $k_{FACE} = 50$ gave rise to the best balance of success rate and costs. For the epsilon graph, a radius of 0.25 yields the strongest results to balance between high yNN and low cost.

`GS` We chose 0.02 as the step size with which the sphere is grown. Lower values yield similar results at the costs of higher computational time, while higher values gave worse results.

`REVISE` The *grid search* to find an acceptable learning rate and similarity weight λ yielded $\eta = 0.1$ and $\lambda = 0.5$ for about 1500 iterations.

`Wachter` For the target loss, we choose the Binary Cross Entropy with a learning rate of 0.01 and an initial λ of 0.01. For the distance loss, we use the ℓ_1- norm to measure the similarity between the factual input and the counterfactual point $\check{x}$.