CFRL: A Python library for counterfactually fair offline reinforcement learning using data preprocessing

Several Different Contributors[1¶]

1 Several Different Departments, University of Michigan ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Reinforcement learning (RL) algorithms aim to learn a sequential decision-making rule, often referred to as a "policy", that maximizes some pre-specified benefit in an environment across multiple or even infinite time steps. It has been widely applied to fields such as healthcare, banking, and autonomous driving. Despite their usefulness, the decisions made by RL algorithms might exhibit systematic bias due to bias in the training data. For example, when using an RL algorithm to assign treatment to patients over time, the algorithm might consistently assign treatment resources to patients of some races at the expense of patients of other races. Concerns have been raised that the deployment of such biased algorithms could exacerbate the discrimination faced by socioeconomically disadvantaged groups.

To address this problem, Wang et al. (2025) extended the concept of single-stage counterfactual fairness (Kusner et al. 2017) to the multi-stage setting and proposed a data preprocessing algorithm that ensures counterfactual fairness in offline reinforcement learning. An RL policy is counterfactually fair if, at every time step, it would assign the same decisions with the same probability for an individual had the individual belong to a different subgroup defined by some sensitive attribute (such as race and gender). At its core, counterfactual fairness views the observed states and rewards as biased proxies of the (unobserved) true underlying states and rewards, where the bias can often be seen as a result of the observed sensitive attribute. In this light, the data preprocessing algorithm ensures counterfactual fairness by removing this bias from the input offline trajectories.

The `CFRL` library is built upon this definition of RL counterfactual fairness introduced in Wang et al. (2025). It implements the data preprocessing algorithm proposed by Wang et al. (2025) and provides a set of tools to evaluate the value and counterfactual fairness achieved by a given policy. In particular, it takes in an offline RL trajectory and outputs a preprocessed, bias-free trajectory, which could be passed to any off-the-shelf offline RL algorithms to learn a counterfactually fair policy. Additionally, it could also take in an RL policy and return its value and counterfactual fairness metric.

Statement of Need

Many existing Python libraries implement algorithms that ensure fairness in machine learning. For example, `Fairlearn` and `aif360` focus on mitigating bias in single-stage machine learning predictions under statistical assocoiation-based fairness criterion such as demographic parity and equal opportunity (). However, they do not accommodate counterfactual fairness, which defines fairness from a causal perspective, and they cannot be easily extended to the reinforcement learning setting in general. Additionally, `ml-fairness-gym` provides tools to simulate unfairness in sequential decision-making, but it neither implement algorithms that reduce unfairness nor address counterfactual fairness (). To our current knowledge, Wang et al. (2025) is the first work to study counterfactual fairness in reinforcement learning. Correspondingly, `CFRL` is also the first code library to address counterfactual fairness in the reinforcement learning setting.

The contribution of CFRL is two-fold. First, it implements a data preprocessing algorithm that removes bias from offline RL training data. For each individual in the data, the preprocessing algorithm estimates the counterfactual states under different sensitive attribute values and concatenates all of the individual's counterfactual states into a new state variable. The preprocessed data can then be directly used by existing RL algorithms for policy learning, and the learned policy should be approximately counterfactually fair. Second, it provides a platform for evaluating RL policies based on counterfactual fairness. After passing in a policy and a trajectory dataset from the target environment, users can assess how well the policy performs in the target environment in terms of the discounted cumulative reward and a counterfactual fairness metric. This not only allows stakeholders to test their fair RL policies before deployment but also offers RL researchers a hands-on tool to evaluate newly developed counterfactually fair RL algorithms.

High-level Design

The `CFRL` library is composed of 5 major modules. The functionalities of the modules are summarized in the table below.

Module	Functionalities
`reader`	Implements functions that read tabular trajectory data from either a `.csv` file or a `pandas.Dataframe` into a format required by `CFRL`. Also implements functions that export trajectory data to either a `.csv` file or a `pandas.Dataframe`.
`preprocessor`	Implements the data preprocessing algorithm introduced in Wang et al. (2025).
`agents`	Implements a fitted Q-iteration (FQI) algorithm, which learns RL policies and makes decisions based on the learned policy. Users can also pass a preprocessor to the FQI; in this case, the FQI will be able to take in unpreprocessed trajectories, internally preprocess the input trajectories, and directly output counterfactually fair policies.
`environment`	Implements a synthetic environment that produces synthetic data as well as a simulated environment that simulates the transition dynamics of the environment underlying some real-world RL trajectory data. Also implements functions for sampling trajectories from the synthetic and simulated environments.
`fqe`	Implements a fitted Q-evaluation (FQE) algorithm, which can be used to evaluate the value of a policy.
`evaluation`	Implements functions that evaluate the value and counterfactual fairness of a policy. Depending on the user's needs, the evaluation can be done either in a synthetic environment or in a simulated environment.

A general CFRL workflow is as follows: First, simulate a trajectory using `environment` or read in a trajectory using `reader`. Then, train a preprocessor using `preprocessor` to remove the bias in the trajectory data. After that, pass the preprocessed trajectory into the FQI algorithm in `agents` to learn a counterfactually fair policy. Finally, use functions in `evaluation` to evaluate the value and counterfactual fairness of the trained policy.

Data Example

We provide a data example to demontrate how CFRL uses real data to learn a counterfactually fair policy and evaluate the value and counterfactual fairness of the learned policy. This is only one of the many workflows that CFRL can perform. We refer interested readers to the "Example Workflows" section of the CFRL documentation for more workflow examples.

Data Loading

In this demonstration, we use an offline trajectory generated from a SyntheticEnvironment using some pre-specified transition rules. Although it is actually synthesized, we treat it as if it is from some unknown environment for pedagogical convenience.

The trajectory contains 500 individuals (i.e. $N = 500$) and 10 transitions (i.e. $T = 10$). The sensitive attribute variable and the state variable are both univariate. The actions are binary (0 or 1) and were sampled using a random policy that selects 0 or 1 randomly with equal probability. The trajectory is stored in a tabular format in a .csv file. We load the trajectory from the tabular form into the array format required by CFRL.

```
zs, states, actions, rewards, ids = read_trajectory_from_dataframe(
                                        path='../data/sample_data_large_uni.csv',
                                        z_labels=['z1'],
                                        state_labels=['state1'],
                                        action_label='action',
                                        reward_label='reward',
                                        id_label='ID',
                                        T=10)
```

We split the trajectory data into a training set (80%) and a testing set (20%). The training set is used to train the policy, while the testing set is used to evaluate the value and counterfactual fairness metric achieved by the policy.

```
(
    zs_train, zs_test,
    states_train, states_test,
    actions_train, actions_test,
    rewards_train, rewards_test
) = train_test_split(zs, states, actions, rewards, test_size=0.2)
```

Preprocessor Training & Trajectory Preprocessing

We now train the preprocessor and preprocess the trajectory. Instead of performing train-test split again, we set cross_folds=5 to save data and avoid overfitting. In this case, train_preprocessor() will internally divide the training data into 5 folds, and each fold is preprocessed using a model that is trained on the other folds. We initialize the SequentialPreprocessor, and train_preprocessor() will take care of both preprocessor training and trajectory preprocessing.

```
sp = SequentialPreprocessor(z_space=[[0], [1]], num_actions=2, cross_folds=5,
                            mode='single', reg_model='nn')
states_tilde, rewards_tilde = sp.train_preprocessor(
    zs=zs_train, xs=states_train, actions=actions_train, rewards=rewards_train)
```

Policy Learning

Now we train a policy using FQI and the preprocessed data with sp as its internal preprocessor. Note that the training data state_tilde and rewards_tilde are already preprocessed. Thus, we set `preprocess=False` during training so that the input trajectory will not be preprocessed again by the internal preprocessor (i.e. `sp`).

```
agent = FQI(num_actions=2, model_type='nn', preprocessor=sp)
agent.train(zs=zs_train, xs=states_tilde, actions=actions_train,
            rewards=rewards_tilde, max_iter=100, preprocess=False)
```

`SimulatedEnvironment` Training

Before moving on to the evaluation stage, there is one more thing to do: We need to train a `SimulatedEnvironment` that mimics the transition rules of the true environment that generated the training trajectory, which will be used by the evaluation functions to simulate the true data-generating environment. To do so, we initialize a `SimulatedEnvironment` and train it on the whole trajectory data (i.e. training set and testing set combined).

```
env = SimulatedEnvironment(num_actions=2,
                           state_model_type='nn',
                           reward_model_type='nn')
env.fit(zs=zs, states=states, actions=actions, rewards=rewards)
```

Value and Counterfactual Fairness Evaluation

We now estimate the value and counterfactual fairness achieved by the trained policy when interacting with the target environment using `evaluate_value_through_fqe()` and `evaluate_fairness_through_model()`, respectively. The counterfactual fairness is represented by a metric from 0 to 1, with 0 representing perfect fairness and 1 indicating complete unfairness. We use the testing set for evaluation.

```
value = evaluate_reward_through_fqe(zs=zs_test, states=states_test,
                                    actions=actions_test, rewards=rewards_test,
                                    policy=agent, model_type='nn')
cf_metric = evaluate_fairness_through_model(env=env, zs=zs_test, states=states_test,
                                            actions=actions_test, policy=agent)
```

The output `value` is `7.3576775` and `cf_metric` is `0.04181818181818181824`, which indicates our policy is close to being perfectly counterfactually fair. Indeed, the CF metric should be exactly 0 if we know the true underlying environment; the reason why it is not exactly 0 here is because we need to estimate the true underlying environment during preprocessing, which can introduce errors.

Bonus: Assessing a Fairness-through-unawareness Policy

Fairness-through-unawareness proposes to ensure fairness by excluding the sensitive attribute from the agent's decision-making. However, it might still be unfair because of the indirect bias in the states and rewards. In this section, we use the same trajectory data train a policy following fairness-through-unawareness and estimate its value and counterfactual fairness.

```
agent_unaware = FQI(num_actions=2, model_type='nn', preprocessor=None)
agent_unaware.train(zs=zs_train, xs=states_train, actions=actions_train,
                    rewards=rewards_train, max_iter=100, preprocess=False)
value_unaware = evaluate_reward_through_fqe(
              zs=zs_test, states=states_test, actions=actions_test,
              rewards=rewards_test, policy=agent_unaware, model_type='nn')
cf_metric_unaware = evaluate_fairness_through_model(
              env=env, zs=zs_test, states=states_test,
              actions=actions_test, policy=agent_unaware)
```

The output `value` is `8.588442` and `cf_metric` is `0.44636363636363635`. The fairness-through-unawareness policy is much less fair than the policy learned using the preprocessed trajectory.

This suggests that the preprocessing method likely reduced the bias in the training trajectory effectively.

Conclusions

CFRL is a Python library that empowers counterfactually fair reinforcement learning using data preprocessing. It also provides tools to evaluate the value and counterfactual fairness of a given policy. As far as we know, it is the first library to address counterfactual fairness problems in the context of reinforcement learning. Nevertheless, despite this, CFRL also admits a few limitations. For example, the current CFRL implementation requires every episode in the offline dataset to have the same number of time steps. Extending the library to accommodate variable-length episodes can improve its flexibility and usefulness. Besides, CFRL could also be made more well-rounded by integrating the preprocessor with established offline RL algorithm libraries such as `d3rlpy`, or connecting the evaluation functions with popular RL environment libraries such as `gym`. We leave these extensions to future updates.

Acknowledgements

This is the acknowledgements.