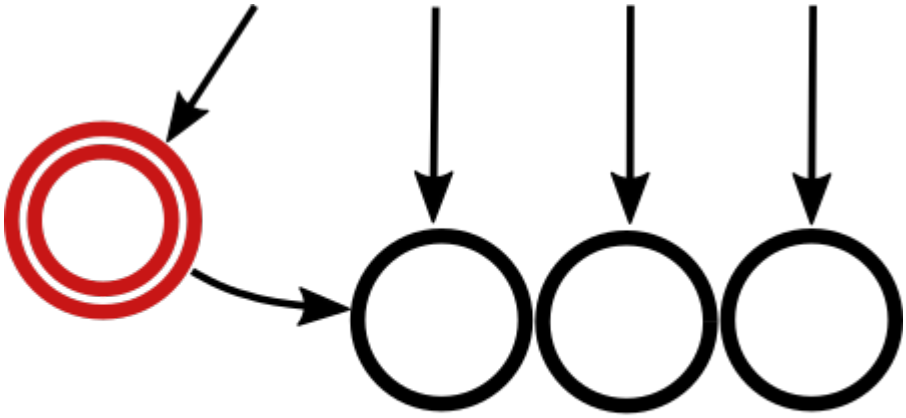

JustCause

justcause Documentation

Florian Wilhelm

Mar 23, 2020

Contents

Evaluating causal inference methods in a scientifically thorough way is a cumbersome and error-prone task. To foster good scientific practice **JustCause** provides a framework to easily:

1. evaluate your method using common data sets like IHDP, IBM ACIC, and others;
2. create synthetic data sets with a generic but standardized approach;
3. benchmark your method against several baseline and state-of-the-art methods.

Our *cause* is to develop a framework that allows you to compare methods for causal inference in a fair and *just* way. JustCause is a work in progress and new contributors are always welcome.

The reasons for creating a library like JustCause are laid out in the thesis `A Systematic Review of Machine Learning Estimators for Causal Effects` of Maximilian Franz. Therein, it is shown that many publications about causality:

- lack reproducibility,
- use different versions of the seemingly same data set,
- fail to state that some theoretical conditions in the data set are not met,
- miss several state of the art methods in their comparison.

A more standardised approach, as offered by JustCause, is able to improve these points.

CHAPTER 1

Installation

Install JustCause with:

```
pip install justcause
```

but consider using conda to set up an isolated environment beforehand. This can be done with:

```
conda env create -f environment.yaml
conda activate justcause
```

with the following `environment.yaml`.

CHAPTER 2

Quickstart

For a minimal example we are going to load the IHDP (Infant Health and Development Program) data set, do a train/test split, apply a basic learner on each replication and display some metrics:

```
>>> from justcause.data.sets import load_ihdp
>>> from justcause.learners import SLearner
>>> from justcause.learners.propensity import estimate_propensities
>>> from justcause.metrics import pehe_score, mean_absolute
>>> from justcause.evaluation import calc_scores

>>> from sklearn.model_selection import train_test_split
>>> from sklearn.linear_model import LinearRegression

>>> import pandas as pd

>>> replications = load_ihdp(select_rep=[0, 1, 2])
>>> slearner = SLearner(LinearRegression())
>>> metrics = [pehe_score, mean_absolute]
>>> scores = []

>>> for rep in replications:
>>>     train, test = train_test_split(rep, train_size=0.8)
>>>     p = estimate_propensities(train.np.X, train.np.t)
>>>     slearner.fit(train.np.X, train.np.t, train.np.y, weights=1/p)
>>>     pred_ite = slearner.predict_ite(test.np.X, test.np.t, test.np.y)
>>>     scores.append(calc_scores(test.np.ite, pred_ite, metrics))

>>> pd.DataFrame(scores)
   pehe_score  mean_absolute
0    0.998388       0.149710
1    0.790441       0.119423
2    0.894113       0.151275
```

CHAPTER 3

Contents

3.1 Usage

3.1.1 Quick Overview

To get a quick overview of what JustCause has to offer, let's take a look at the package structure:

```
├── contrib        <- third party methods not meant to be accessed directly
├── data
│   ├── generators <- data generators for full synthetic data sets
│   ├── sets       <- empirical data sets like IHDP, Twins, etc.
│   ├── frames     <- provides the CausalFrame
│   ├── transport  <- functionality to download data sets
│   └── utils      <- generic helper functions for data sets
├── learners
│   ├── ate        <- average treatment effect estimators
│   ├── meta       <- meta learners working with the help of classical estimators
│   ├── propensity <- functionality estimate propensity scores
│   └── utils      <- generic helper functions for learners
├── evaluation     <- helper functions for evaluation
├── metrics        <- various metrics to compare a result to the ground truth
└── utils          <- most generic helper functions not related to data and learners
```

Most commonly you will deal with `data.generators` and `data.sets` to generate or fetch a data set and apply some basic learners within `learners`. To evaluate your results you can use `metrics` and `evaluation`. All methods within `contrib` are not meant to be accessed directly and are wrapped within `learners`.

Note: Since version 0.4 the `learners` is greatly reduced to focus the attention and efforts of JustCause on evaluation. See section 'Implement

3.1.2 The Reason for DGPs

Due to the so called Fundamental Problem of Causal Inference, there is no ground truth for any real treatment effect dataset. In order to be able to evaluate methods, we thus need to resort to semi- or fully-synthetic data. The process of generating such a dataset is called a data generating process (DGP). We distinguish between i) an Empirical Monte Carlo Study [1] which uses real covariates - the features of real instances (e.g. patients, ...) - and generates a synthetic potential outcome on top of it and ii) a fully synthetic approach, where covariates are sampled from some distribution.

Briefly, a reference data set following our convention contains these columns with special meaning:

- `t`: binary treatment indicator
- `y`: observed outcome
- `y_cf`: counterfactual outcome
- `y_0`: untreated potential outcome with possible noise
- `y_1`: treated potential outcome with possible noise
- `mu_0`: true untreated potential outcome without noise
- `mu_1`: true treated potential outcome without noise
- `ite`: true individual treatment effect

For those columns the following relationships hold:

- `y = t*y_1 + (1-t)*y_0`
- `y_cf = (1-t)*y_1 + t*y_0` (*counterfactual* of `y`)
- `y = y_0 if t == 0 else y_1`
- `y_0 = mu_0 + e` and `y_1 = mu_1 + e` where e (read epsilon) is some random noise or 0
- `ite = mu_1 - mu_0`

Besides these columns, there are covariates (also called features) and optionally other columns for managing meta information like datetime or an id of sample. Within the provided data sets covariates are called `x_0`, `x_1`, etc. by default. However, named covariates from the IBM or Twins studies are kept in their format (e.g. `dob_mm` for date-of-birth month) for clarity. Accordingly you can also use any name if you use your own data set as explained below. The matrix of all covariates is `X := [x_0, x_1, ..., x_n]` and its usage is explained below. Besides covariates, the provided data sets have a column `sample_id` to easily identify one sample in different replications.

Replications

Since most DGPs are based on some form of random sampling, usually researchers use multiple so called replications of the same data to avoid a large influence of the randomness underlying the distributions. A replication is generated by sampling from the probability distributions that define the data. In the case of IHDP 1000 replications of the same data are used for a full evaluation, thus ensuring robust evaluation results.

The concept of replications is build into JustCause by design to encourage robust comparisons.

3.1.3 Handling Data

JustCause uses a generalization of a Pandas `DataFrame` for managing your data named `CausalFrame`. A CausalFrame encompasses all the functionality of a Pandas DataFrame but additionally keeps track which columns, besides the ones with special meanings like explained above, are covariates or others. This allows to easily access them in a programmatic way.

All data sets provided by JustCause are provided as lists of CausalFrames, i.e. for each replication one CausalFrame. Thus, we get a single CausalFrame `cf` from one of the provided data sets by:

```
>>> from justcause.data.sets import load_ihdp

>>> cf = load_ihdp(select_rep=0)[0]  # select replication 0
>>> type(cf)
justcause.data.frames.CausalFrame
```

As usual, `cf.columns` would list the names of all columns. To find out which of these columns are *covariates* or *others*, we can use the attribute accessor `names`:

```
>>> cf.names.covariates
['x_0', 'x_1', 'x_2', ..., 'x_22', 'x_23', 'x_24']
>>> cf.names.others
['sample_id']
```

This allows us to easily apply transformations for instance only to covariates. In general, this leads to more robust code since the API of a CausalFrame enforces the differentiation between covariates, columns with special meaning, e.g. outcome `y`, treatment `t` and other columns such as metadata like a datetime or an id of an observation, e.g. `sample_id`.

If we want to construct a CausalFrame, we do that just in the same way as with a DataFrame but have to specify covariate columns:

```
>>> import justcause as jc
>>> from numpy.random import rand, randint
>>> import numpy as np
>>> import pandas as pd

>>> N = 10
>>> mu_0 = np.zeros(N)
>>> mu_1 = np.zeros(N)
>>> ite = mu_1 - mu_0
>>> y_0 = mu_0 + 0.1*rand(N)
>>> y_1 = mu_1 + 0.1*rand(N)
>>> t = randint(2, size=N)
>>> y = np.where(t, y_1, y_0)
>>> y_cf = np.where(t, y_0, y_1)

>>> dates = pd.date_range('2020-01-01', periods=N)
>>> cf = jc.CausalFrame({'c1': rand(N),
>>>                      'c2': rand(N),
>>>                      'date': dates,
>>>                      't': t,
>>>                      'y': y,
>>>                      'y_cf': y_cf,
>>>                      'y_0': y_0,
>>>                      'y_1': y_1,
>>>                      'mu_0': mu_0,
>>>                      'mu_1': mu_1,
>>>                      'ite': ite
>>>                      },
>>>                      covariates=['c1', 'c2'])
```

All columns that are neither covariates nor columns with special meaning like `t` and `y` are treated as *others*:

```
>>> cf.names.others
['date']
```

3.1.4 Working with Learners

Within the PyData stack, Numpy surely is the lowest common denominator and is thus used by a lot of libraries. Since JustCause mainly wraps third-party libraries for causal methods under a common API, the decision was taken to only allow passing Numpy arrays to the learners, i.e. causal methods, within JustCause. This allows for more flexibility and keeps the abstraction layer to the original method much smaller.

The `fit` method of a learner takes at least the parameters `X` for the covariate matrix, `t` for the treatment and `y` for the outcome, i.e. target, vector as Numpy arrays. In order to bridge the gap between rich CausalFrames and plain arrays, a `CausalFrame` provides the attribute accessor `np` (for *numpy*). Using it, we can easily pass the covariates `X`, treatment `t` and outcome `y` to a learner:

```
>>> from sklearn.ensemble import RandomForestRegressor

>>> reg = RandomForestRegressor()
>>> learner = jc.learners.SLearner(reg)
>>> learner.fit(cf.np.X, cf.np.t, cf.np.y)
```

3.1.5 Evaluating Methods

The central element of JustCause is evaluation. We want to score learners on various datasets using common metrics. This can either be done manually, or using predefined standard routines (`evaluate_ite()`). JustCause allows you to do both.

Quickstart

The simplest and fastest evaluation is using standard datasets and the methods provided by JustCause:

```
from justcause.learners import SLearner, TLearner
from justcause.metrics import pehe_score, mean_absolute
from justcause.data.sets import load_ihdp

replications = load_ihdp(select_rep=np.arange(100))
metrics = [pehe_score, mean_absolute]
train_size = 0.8
random_state = 42
methods = [basic_slearner, weighted_slearner]

# All in standard configuration
methods = [SLearner(), TLearner()]
result = evaluate_ite(replications,
                      methods,
                      metrics,
                      train_size=train_size,
                      random_state=random_state)
```

Here, we use two methods `basic_slearner` and `weighted_slearner` that haven't been defined yet. To better understand what's happening inside and how to customize, let us take a look at an evaluation loop in more detail.

Evaluating Learners

Let's implement a simple evaluation of two learners - a weighted SLearner vs. a standard SLearner. The standard SLearner is already provided in `SLearner`, while the weighted SLearner requires a slight adaption. We define a callable, which takes train and test data, fits a weighted model and predicts ITE for both train and test samples:

```
from justcause.learners import SLearner
from justcause.learners.propensity import estimate_propensities
from sklearn.linear_model import LinearRegression

def weighted_slearner(train, test):
    """
    Custom method that takes 'train' and 'test' CausalFrames (see causal_frames.ipynb)
    and returns ITE predictions for both after training on 'train'.

    Implement your own method in a similar fashion to evaluate them within the␣
↪framework!
    """
    train_X, train_t, train_y = train.np.X, train.np.t, train.np.y
    test_X, test_t, test_y = test.np.X, test.np.t, test.np.y


    # Get calibrated propensity estimates
    p = estimate_propensities(train_X, train_t)

    # Make sure the supplied learner is able to use `sample_weights` in the fit()␣
↪method
    slearner = SLearner(LinearRegression())

    # Weight with inverse probability of treatment (inverse propensity)
    slearner.fit(train_X, train_t, train_y, weights=1/p)
    return (
        slearner.predict_ite(train_X, train_t, train_y),
        slearner.predict_ite(test_X, test_t, test_y)
    )


def basic_slearner(train, test):
    """Basic SLearner callable"""
    train_X, train_t, train_y = train.np.X, train.np.t, train.np.y
    test_X, test_t, test_y = test.np.X, test.np.t, test.np.y

    slearner = SLearner(LinearRegression())
    slearner.fit(train_X, train_t, train_y)
    return (
        slearner.predict_ite(train_X, train_t, train_y),
        slearner.predict_ite(test_X, test_t, test_y)
    )
```

Note: Another way to add new learners is to implement them as a class similiar to the implementations in `learners` (for example `SLearner`) providing at least the methods `fit(x, t, y)` and `predict_ite(x, t, y)`. See the section *Implementing New Learners* for more.

Custom Evaluation Loop

Given the two functions defined above, we can go ahead and write our own simple evaluation loop:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from justcause.data import Col
from justcause.data.sets import load_ihdp
from justcause.metrics import pehe_score, mean_absolute
from justcause.evaluation import calc_scores, summarize_scores

replications = load_ihdp(select_rep=np.arange(100))
metrics = [pehe_score, mean_absolute]
train_size = 0.8
random_state = 42
methods = [basic_slearner, weighted_slearner]

results = list()

for method in methods:

    test_scores = list()
    train_scores = list()

    for rep in replications:

        train, test = train_test_split(
            rep, train_size=train_size, random_state=random_state
        )

        # REPLACE this with the function you implemented and want to evaluate
        train_ite, test_ite = method(train, test)

        # Calculate the scores and append them to a dataframe
        test_scores.append(calc_scores(test[Col.ite], test_ite, metrics))
        train_scores.append(calc_scores(train[Col.ite], train_ite, metrics))

    # Summarize the scores and save them in a dataframe
    train_result, test_result = summarize_scores(train_scores), summarize_scores(test_
↪scores)
    train_result.update({'method': method.__name__, 'train': True})
    test_result.update({'method': method.__name__, 'train': False})

    results.append(train_result)
    results.append(test_result)
```

Finally, we can compare the results:

method	train	pehe_score-mean	...	mean_absolute-std
basic_slearner	True	5.633659795888	...	1.4932757697867
basic_slearner	False	5.625971000721	...	2.4746034286861
weighted_slearner	True	5.592355721307	...	0.5243953093767
weighted_slearner	False	5.493401193725	...	0.9419412237398

Understanding Scores and Results

In the above evaluation loop, `train_scores` contains the scores of ITE prediction on the train set for each replication. To better understand what's happening inside, let's take a look at these intermediate scores:

```
>>> pd.DataFrame(train_scores) # for better visualization

#   pehe_score      mean_absolute
0   0.893524        0.074874
1   0.826433        0.200816
2   0.909720        0.080099
3   1.945077        0.091223
4   2.671555        0.466394
... ...             ...
95  2.194153        0.180240
96  2.161083        0.087108
97  13.238825       1.218813
98  3.917264        0.054858
99  2.538756        0.654481
```

And we then summarize these scores using different formats (like `np.mean`, `np.std`, ...):

```
>>> train_result = summarize_scores(train_scores)
>>> pd.DataFrame([train_result])
```

which yields:

pehe_score-mean	pehe_score-median	pehe_score-std	...	mean_absolute-std
5.592356	2.569472	8.248291	...	0.524395

Simplifying Evaluation

There's two things we can make a lot simpler using JustCause:

1. Standard methods can be used as-is
2. Standard evaluation is pre-implemented

Using the standard evaluation looks like this:

```
from justcause.evaluation import evaluate_ite
result = evaluate_ite(replications,
                      methods,
                      metrics,
                      train_size=train_size,
                      random_state=random_state)
```

And, we can also get rid of `basic_slearner` since that is the default usage of a learner: fit on train, predict on train and test without special settings or parameters. Instead, we simply pass the instantiation of the `SLearner` along to the methods parameter.

> # All in standard configuration methods = [SLearner(), weighted_slearner] result = evaluate_ite(replications,
>
> methods, metrics, train_size=train_size, random_state=random_state)

Note: Note that the Meta Learners use a default setting to determine which regression to use when none is provided.

3.1.6 Implementing New Data

JustCause provides some of the most common reference dataset, but is open for extension. You can either provide fixed reference datasets or define a parametric data generation process (DGP) that generates new data.

Providing Datasets

In the JustCause Data Repository we provide datasets in the `.parquet` format, which is highly efficient and can easily be read by Pandas. In order to avoid duplicate data we store covariates and outcomes in separate files and only join them upon loading. This is to say that usually we have a fixed set of covariates for a number of instances. In the outcomes file we define factual outcomes and counterfactual for these instances for one or multiple replications. ext:rst

Note: If you have a new reference dataset or a useful set of covariates and want to allow others to use it, feel free to submit a Pull Request in the JustCause Data Repository and this repo. See `data.sets.ihdp` for an example. In `data.transport` we provide useful methods for fetching data. You only have to add the top level access and the respective directory in `justcause-data`.

If you only want to use your data once, you can simply load it directly into a CausalFrame as shown in section *Handling Data*.

Parametric DGPs

Another way to generate data is by defining the functions of the potential outcomes based on the covariates. This allows to sample as many replications as one requires. Let's walk through an example to see what parts are required. Let's assume with work with the covariates from the IHDP study provided in `data.sets.ihdp`.

Note: For a fully fledged DGP we need:

1. Covariates
2. Potential Outcomes with and without noise
3. Treatment Assignment

Covariates

We simply access the covariates with:

```
>>> from justcause.data.sets.ihdp import get_ihdp_covariates

>>> covariates = get_ihdp_covariates()
>>> covariates.shape
    (747, 25)
```

Outcomes

Let's define the outcome based on the following function:

$$y_0 = N(0, 0.2)$$
$$y_1 = y_0 + \tau$$

where

$$c = \mathbb{I}(sigmoid(X_8) > 0.5)$$
$$\tau = N(3 * c + (1 - c), 0.1).$$

To implement that as a DGP in JustCause we define the outcome function as follows:

```
from sklearn.utils import check_random_state  # ensures usable random state
from justcause.data.utils import generate_data
from scipy.special import expit

def outcome(covariates, *, random_state: RandomState, **kwargs):
    random_state = check_random_state(random_state)

    # define tau
    prob = expit(covariates[:, 7]) > 0.5
    tau = random_state.normal((3 * prob) + 1 * (1 - prob), 0.1)

    y_0 = random_state.normal(0, 0.2, size=len(covariates))
    y_1 = y_0 + tau
    mu_0, mu_1 = y_0, y_1  # no noise for this example
    return mu_0, mu_1, y_0, y_1
```

Hint: Every outcome function you want to use with JustCause must take a `covariates` parameter and a `random_state`. Using the `**kwargs`, you can pass further parameters to the outcome and treatment assignemnt function. The outcome function must return all four potential outcomes (with and without noise).

Treatment

In order to get a confounded example, we assign treatment based on the covariates `X_8` that was already used to define the strength of the treatment effect.

$$t = \text{BERN}[sigmoid(X_8)]$$

As a function this is simply:

```
def treatment(covariates, *, random_state: RandomState, **kwargs):
    random_state = check_random_state(random_state)
    return random_state.binomial(1, p=expit(covariates[:, 7]))
```

Hint: The treatment function also has to accept `covariates` and `random_state` arguments and should return the treatment indicator vector for the given covariates. Again, `**kwargs` can be used to pass further parameters

Plugging it Together

With the covariates we fetched and the outcomes we defined, we can now sample data from that DGP using the powerful `generate_data()`:

```
>>> replications = generate_data(
    covariates,
    treatment,
    outcome,
    n_samples=747,  # Optional but 747 is the maximum available with IHDP covariates
    n_replications=100,
    random_state=0,  # Fix random_state for replicability
    **kwargs=None,  # Use if further parameters are required
)
```

A standardized Terminology

By using `generate_data()` we encourage a consistent terminology for DGPs. This is important as we've found that different researchers use different formalizations that are technically identical. However, assuring that they are actually the same requires one to transform the notation.

Take for example the synthetic example studies in the RLearner Paper where outcomes are defined as

$$y_i = b(X) + (W - 0.5) \cdot \tau(X) + \sigma\epsilon(X).$$

That is to say, they start from a base value $b(X)$ and add or substract half the treatment effect $\tau(X)$ depending on the treatment. This can be defined equivalently in our terminology as:

$$\begin{aligned}
\mu_0 &= b(X) - \frac{1}{2} \cdot \tau, \\
\mu_1 &= b(X) + \frac{1}{2} \cdot \tau, \\
y_0 &= \mu_0 + \sigma\epsilon(X), \\
y_1 &= \mu_1 + \sigma\epsilon(X).
\end{aligned}$$

We encourage users of JustCause to start their considerations with the terminology introduced at the top of this document.

3.1.7 Implementing New Learners

As of JustCause 0.4 only very basic learners - namely the S- and T-Learner are provided in `learners`. Here, we clarify how to implement and use more learners with JustCause. The consideration behind removing all the learners was, that all the different packages out there (e.g. causalML) sport different APIs for there learners and are changing quickly. Instead of exerting efforts on trying to unify these APIs with the one proposed in JustCause, we provide two ways of adapting whatever methods you have at hand to work with Justcause.

1. Implementation as a method (See *Evaluating Learners*)
2. Implementation as a class

For recurring use of a learner within the JustCause package it might be favorable to wrap a learner in a class. For example, the RLearner from causalML can be wrapped in the way it was done it JustCause 0.3.2

```python
"""Wrapper of the python RLearner implemented in the ``causalml`` package"""
from typing import Optional, Union

import numpy as np
from numpy.random import RandomState
from sklearn.linear_model import LinearRegression
from sklearn.utils import check_random_state

from ..propensity import estimate_propensities

StateType = Optional[Union[int, RandomState]]


class RLearner:
    """A wrapper of the BaseRRegressor from ``causalml``
    Defaults to LassoLars regression as a base learner if not specified otherwise.
    Allows to either specify one learner for both tasks or two distinct learners
    for the task outcome and effect learning.

    """

    def __init__(
        self,
        learner=None,
        outcome_learner=None,
        effect_learner=None,
        random_state: StateType = None,
    ):
        """Setup an RLearner with defaults
        Args:
            learner: default learner for both outcome and effect
            outcome_learner: specific learner for outcome
            effect_learner: specific learner for effect
            random_state: RandomState or int to be used for K-fold splitting. NOT used
                in the learners, this has to be done by the user.
        """
        from causalml.inference.meta import BaseRRegressor

        if learner is None and (outcome_learner is None and effect_learner is None):
            learner = LinearRegression()

        self.random_state = check_random_state(random_state)
        self.model = BaseRRegressor(
            learner, outcome_learner, effect_learner, random_state=random_state
        )

    def fit(self, x: np.array, t: np.array, y: np.array, p: np.array = None) -> None:
        """Fits the RLearner on given samples.
        Defaults to `justcause.learners.propensities.estimate_propensities`
        for ``p`` if not given explicitly, in order to allow a generic call
        to the fit() method
        Args:
            x: covariate matrix of shape (num_instances, num_features)
            t: treatment indicator vector, shape (num_instances)
            y: factual outcomes, (num_instances)
            p: propensities, shape (num_instances)
        """
```

(continues on next page)

(continued from previous page)

```
        if p is None:
            # Propensity is needed by CausalML, so we estimate it,
            # if it was not provided
            p = estimate_propensities(x, t)

        self.model.fit(x, p, t, y)

    def predict_ite(self, x: np.array, *args) -> np.array:
        """Predicts ITE for given samples; ignores the factual outcome and treatment
        Args:
            x: covariates used for precition
            *args: NOT USED but kept to work with the standard ``fit(x, t, y)`` call
        """

        # assert t is None and y is None, "The R-Learner does not use factual outcomes
↪"
        return self.model.predict(x).flatten()

    def estimate_ate(
        self, x: np.array, t: np.array, y: np.array, p: Optional[np.array] = None
    ) -> float:
        """Estimate the average treatment effect (ATE) by fit and predict on given␣
↪data
        Estimates the ATE as the mean of ITE predictions on the given data.
        Args:
            x: covariates of shape (num_samples, num_covariates)
            t: treatment indicator vector, shape (num_instances)
            y: factual outcomes, (num_instances)
            p: propensities, shape (num_instances)
        Returns:
            the average treatment effect estimate
        """
        self.fit(x, t, y, p)
        ite = self.predict_ite(x, t, y)
        return float(np.mean(ite))
```

In the code above, we've used the internal functionality of the causalml class `BaseRRegressor` and have wrapped in our API definition that works directly within the JustCause evaluation. Having implemented that once, we can used it in the prototypical evalaution just like the `SLearner`

```
>>> methods = [SLearner(), RLearner()]
>>> result = evaluate_ite(replications,
                  methods,
                  metrics,
                  train_size=train_size,
                  random_state=random_state)
```

Similarly, we could wrap the RLearner in a function, like it was done in *Evaluating Learners* for the SLearner.

3.2 Best Practices

3.2.1 Dependency Management & Reproducibility

1. Always keep your abstract (unpinned) dependencies updated in `environment.yaml` and eventually in `setup.cfg` if you want to ship and install your package via `pip` later on.
2. Create concrete dependencies as `environment.lock.yaml` for the exact reproduction of your environment with:

   ```
   conda env export -n justcause -f environment.lock.yaml
   ```

   For multi-OS development, consider using `--no-builds` during the export.
3. Update your current environment with respect to a new `environment.lock.yaml` using:

   ```
   conda env update -f environment.lock.yaml --prune
   ```

3.2.2 Organization, Logging and Reproduction of Experiments

Sacred is a wonderful tool to do reproducible experimentation. From their documentation:

> Sacred is a tool to configure, organize, log and reproduce computational experiments. It is designed to introduce only minimal overhead, while encouraging modularity and configurability of experiments.

We thus recommend to use it together with JustCause and will add some more details here soon.

3.3 License

3.4 Contributors

- Maximilian Franz <Maximilian.Franz@inovex.de>
- Florian Wilhelm <Florian.Wilhelm@inovex.de>

- Tanmay Kulkarni <tanmaygk97@gmail.com>

3.5 Changelog

3.5.1 Version 0.4

- removed dependency to `causalml` thus x- and t- learner were removed
- removed dependency to `rpy` thus CausalForest method was removed
- removed dependency to `tensorflow` thus ganite and dragonnet were removed
- added missing `requests` library in dependencies

3.5.2 Version 0.3.2

- some fixes in usage documentation
- fixed persistent "future annotation"-bug that prevent importing justcause on Python 3.6
- `generate_data` handles now CausaFrames as covariates correctly

3.5.3 Version 0.3.1

- bugfixes in data generator based on IHDP (wrong covariate was used)
- bugfix in evaluation (train/test results were in fact the same)
- Support for lower Python versions (type annotations as used in JustCause were only available for Python >=3.7)

3.5.4 Version 0.3

- data sets and generators now return a list of CausalFrames instead of iterators
- treatment column is always `t` and outcome column always `y`
- improved and extended documentation

3.5.5 Version 0.2

- Complete overhaul of everything that was done before in order to have:
  - a distributable Python package
  - unit tests
  - a consistent API
  - some documentation
  - and much more ;-)

3.5.6 Version 0.1

- Reflecting the state of the finished bachelor thesis

3.6 Contributing

3.6.1 Issue Reports

If you experience bugs or in general issues with JustCause, please file an issue report on our issue tracker.

3.6.2 Code Contributions

Submit an issue

Before you work on any non-trivial code contribution it's best to first create an issue report to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Clone the repository

1. Create a Gitub account if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone this copy to your local disk:

```
git clone git@github.com:YourLogin/justcause.git
```

4. Create an environment `justcause` with the help of Miniconda and activate it:

```
conda env create -f environment.yaml
conda activate justcause
```

5. Install `justcause` with:

```
python setup.py develop
```

6. Install `pre-commit`:

```
pip install pre-commit
pre-commit install
```

   JustCause comes with a lot of hooks configured to automatically help you with providing clean code.

7. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work on the master branch!

8. Start your work on this branch. When you're done editing, do:

```
git add modified_files
git commit
```

   to record your changes in Git, then push them to GitHub with:

```
git push -u origin my-feature
```

9. Please check that your changes don't break any unit tests with:

```
py.test
```

Don't forget to also add unit tests in case your contribution adds an additional feature and is not just a bugfix.

10. Add yourself to the list of contributors in `AUTHORS.rst`.
11. Go to the web page of your JustCause fork, and click "Create pull request" to send your changes to the maintainers for review. Find more detailed information creating a PR.

3.6.3 Release

As a JustCause maintainer following steps are needed to release a new version:

1. Make sure all unit tests on Cirrus-CI are green.
2. Update the `CHANGELOG.rst` file.
3. Tag the current commit on the master branch with a release tag, e.g. `v1.2.3`.
4. Clean up the `dist` and `build` folders with `rm -rf dist build` to avoid confusion with old builds and Sphinx docs.
5. Run `python setup.py dists` and check that the files in `dist` have the correct version (no `.dirty` or Git hash) according to the Git tag. Also sizes of the distributions should be less than 500KB (for bdist), otherwise unwanted clutter may have been included.
6. Make sure you uploaded the new tag to Github, run `git push origin --tags`.
7. Run `twine upload dist/*` and check that everything was uploaded to PyPI correctly.

3.7 justcause

3.7.1 justcause package

Subpackages

justcause.data package

Subpackages

justcause.data.generators package

Submodules

justcause.data.generators.ihdp module

justcause.data.generators.rlearner module

justcause.data.generators.toy module

justcause.data.sets package

justcause.metrics module

justcause.utils module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search